

TOSHIBA

TOSHIBA TX03 Peripheral Driver User Guide (TMPM384)

Ver 1
Sep, 2017

TOSHIBA ELECTRONIC DEVICES & STORAGE CORPORATION

RESTRICTIONS ON PRODUCT USE

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

Index

1. Introduction.....	1
2. Organization of TOSHIBA TX03 Peripheral Driver.....	1
3. ADC.....	2
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
3.2.4 Data Structure Description	12
4. CG.....	15
4.1 Overview.....	15
4.2 API Functions	16
4.2.1 Function List.....	16
4.2.2 Detailed Description	17
4.2.3 Function Documentation	17
4.2.4 Data Structure Description	33
5. DMAC.....	34
5.1 Overview.....	34
5.2 API Functions	34
5.2.1 Function List.....	34
5.2.2 Detailed Description	34
5.2.3 Function Documentation	35
5.2.4 Data Structure Description	43
6. FC.....	45
6.1 Overview.....	45
6.2 API Functions	45
6.2.1 Function List.....	45
6.2.2 Detailed Description	45
6.2.3 Function Documentation	46
6.2.4 Data Structure Description	50
7. GPIO.....	51
7.1 Overview.....	51
7.2 API Functions	51
7.2.1 Function List.....	51
7.2.2 Detailed Description	51
7.2.3 Function Documentation	52
7.2.4 Data Structure Description	64
8. OFD.....	66
8.1 Overview.....	66
8.2 API Functions	66
8.2.1 Function List.....	66
8.2.2 Detailed Description	66
8.2.3 Function Documentation	66
8.2.4 Data Structure Description	68
9. RMC.....	70
9.1 Overview.....	70
9.2 API Functions	70
9.2.1 Function List.....	70
9.2.2 Detailed Description	70
9.2.3 Function Documentation	71
9.2.4 Data Structure Description	79
10. RTC.....	81
10.1 Overview.....	81
10.2 API Functions	81
10.2.1 Function List.....	81
10.2.2 Detailed Description	82
10.2.3 Function Documentation	82
10.2.4 Data Structure Description	98
11. SBI.....	100

11.1	Overview.....	100
11.2	API Functions	100
11.2.1	Function List.....	100
11.2.2	Detailed Description	100
11.2.3	Function Documentation	101
11.2.4	Data Structure Description	106
12.	SSP.....	109
12.1	Overview.....	109
12.2	API Functions	109
12.2.1	Function List.....	109
12.2.2	Detailed Description	110
12.2.3	Function Documentation	110
12.2.4	Data Structure Description	119
13.	TMRB	121
13.1	Overview.....	121
13.2	API Functions	121
13.2.1	Function List.....	121
13.2.2	Detailed Description	122
13.2.3	Function Documentation	122
13.2.4	Data Structure Description	131
14.	SIO/UART.....	133
14.1	Overview.....	133
14.2	API Functions	133
14.2.1	Function List.....	133
14.2.2	Detailed Description	134
14.2.3	Function Documentation	134
14.2.4	Data Structure Description	147
15.	VLTD	150
15.1	Overview.....	150
15.2	API Functions	150
15.2.1	Function List.....	150
15.2.2	Detailed Description	150
15.2.3	Function Documentation	150
16.	WDT.....	153
16.1	Overview.....	153
16.2	API Functions	153
16.2.1	Function List.....	153
16.2.2	Detailed Description	153
16.2.3	Function Documentation	153
16.2.4	Data Structure Description	156
17.	ENC	157
17.1	Overview.....	157
17.2	API Functions	157
17.2.1	Function List.....	157
17.2.2	Detailed Description	157
17.2.3	Function Documentation	158
17.2.4	Data Structure Description	162
18.	PMD.....	164
18.1	Overview.....	164
18.2	API Functions	165
18.2.1	Function List.....	165
18.2.2	Detailed Description	166
18.2.3	Function Documentation	166
18.2.4	Data Structure Description	179

1. Introduction

TOSHIBA TX03 Peripheral Driver is a set of drivers for all peripherals found on the TOSHIBA TX03 series microcontrollers. TMPM384 Peripheral Driver is an important part of TOSHIBA TX03 Peripheral Driver, which are designed for TMPM384 series MCUs.

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

The design goals of TOSHIBA TMPM384 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 TMPM384 Peripheral Drivers.

/Libraries/ TX03_CMSIS

This folder contains the TMPM384 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 TMPM384 Peripheral Driver.

/Libraries/TX03_Periph_Driver/inc

This folder contains all the header files of TMPM384 Peripheral Drivers for each peripheral.

/Libraries/TX03_Periph_Driver/src

This folder contains all the source files of TMPM384 Peripheral Drivers for each peripheral.

/Project

This folder contains template project and examples for using TMPM384 Peripheral Driver.

/Project/Template

This folder contains template project of TOSHIBA TMPM384 Peripheral Driver.

/Project/Examples

This folder contains a set of examples for using TMPM384 Peripheral Driver

/Utilities/TMPM384-EVAL

This folder contains the configuration and driver files for hardware resources (e.g. led, key) on MCBTMPM384 boards.

3. ADC

3.1 Overview

TOSHIBA TMPM384 contains a 12bits/10bits(selectable) successive-approximation Analog-to-Digital Converter (ADC) with 22 channels (AIN0 to AIN21).

The ADC can start conversion by setting in Software Trigger Program or Constant Trigger Program. Meanwhile, it can also receiving trigger signal from PMD(MPT mode) or TMRB(interrupt).

- (1) It can select analog input and start AD conversion when receiving trigger signal from PMD(MPT) or TMRB(interrupt).
- (2) It can select analog input, in the Software Trigger Program and the Constant Trigger Program.
- (3) The ADC has twelve registers for AD conversion result.
- (4) The ADC generate interrupt signal at the end of the program which was started by PMD(MPT) trigger and TMRB trigger.
- (5) The ADC generate interrupt signal at the end of the program which are the Software Trigger Program and the Constant Trigger Program.
- (6) The ADC has the AD conversion monitoring function. When this function is enable, and interrupt is generated when a conversion result matches the specified comparison value.

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm384_adc.c, with /Libraries/TX03_Periph_Driver/inc/tmpm384_adc.h containing the macros, data types, structures and API definitions for use by applications.

3.2 API Functions

3.2.1 Function List

- ◆ void ADC_SetClk(uint32_t **Sample_HoldTime**, uint32_t **Prescaler_Output**);
- ◆ void ADC_SetResolution(ADC_Resolution **ADBits**);
- ◆ ADC_Resolution ADC_GetResolution(void);
- ◆ void ADC_Enable(void);
- ◆ void ADC_Disable(void);
- ◆ void ADC_Start(TrgType **Trg**);
- ◆ void ADC_StopConstantTrg(void);
- ◆ WorkState ADC_GetConvertState(TrgType **Trg**);
- ◆ void ADC_SetLowPowerMode(FunctionalState **NewState**);
- ◆ void ADC_SetMonitor(ADC_MonitorTypeDef * **Monitor**);
- ◆ void ADC_DisableMonitor(ADC_CMPCRx **CMPCRx**);
- ◆ ADC_ResultUnion ADC_GetConvertResult(ADC_REGx **ResultREGx**);
- ◆ void ADC_SelectPMDTrgProgNum(PMD_TRG_PROGRAM_SELx **SELx**, uint8_t **MacroProgNum**);
- ◆ void ADC_SetPMDTrgProgINT(PMD_TrgProgINTTypeDef * **TrgProgINT**);
- ◆ void ADC_SetTimerTrg(ADC_REGx **REGx**, uint8_t **MacroAINx**);
- ◆ void ADC_SetSWTrg(ADC_REGx **REGx**, uint8_t **MacroAINx**);
- ◆ void ADC_SetConstantTrg(ADC_REGx **REGx**, uint8_t **MacroAINx**);

3.2.2 Detailed Description

Functions listed above can be divided into four parts:

- 1) ADC setting by ADC_SetClk(), ADC_SetResolution(), ADC_SetMonitor(), ADC_DisableMonitor(), ADC_SelectPMDTrgProgNum(), ADC_SetPMDTrgProgINT(), ADC_SetTimerTrg(), ADC_SetSWTrg(), ADC_SetConstantTrg().
- 2) ADC function enable/disable & start/stop by ADC_Enable(), ADC_Disable(), ADC_Start(), ADC_StopConstantTrg().
- 3) ADC state or data read functions by ADC_GetResolution(), ADC_GetConvertState(), ADC_GetConvertResult().
- 4) ADC_SetLowPowerMode() handle other specified functions.

3.2.3 Function Documentation

3.2.3.1 ADC_SetClk

Set ADC sample hold time and prescaler output (SCLK).

Prototype:

void

ADC_SetClk(uint32_t **Sample_HoldTime**, uint32_t **Prescaler_Output**)

Parameters:

Sample_HoldTime: Select ADC sample hold time.

which can be set as:

- **ADC_HOLD_FIX:** write "1001b" to TSH<0:3>.

Prescaler_Output: Select ADC prescaler output.

which can be set as:

- **ADC_FC_DIVIDE_LEVEL_NONE:** fc

Description:

This function will set ADC sample hold time by **Sample_HoldTime** as

ADC_HOLD_FIX & select ADC prescaler output by **Prescaler_Output** as

ADC_FC_DIVIDE_LEVEL_NONE.

Return:

None

3.2.3.2 ADC_SetResolution

Set AD resolution.

Prototype:

void

ADC_SetResolution(ADC_Resolution **ADBits**)

Parameters:

ADBits: Set ADC resolution as 12bits or 10bits.

which can be set as:

- **ADC_10BITS:** 10bit
- **ADC_12BITS:** 12bit

Description:

This function will set ADC resolution by **ADBits** as **ADC_10BITS/ADC_12BITS**.

It should be called from beginning. When Power up reset, the ADC module is set in 12bits mode.

Return:
None

3.2.3.3 ADC_GetResolution

Get current AD resolution set.

Prototype:
ADC_Resolution
ADC_GetResolution(void)

Parameters:
None

Description:
This function will get ADC resolution.

Return:
ADC_Resolution type, the value means:
ADC_10BITS: 10bit
ADC_12BITS: 12bit

3.2.3.4 ADC_Enable

Enable ADC module before start ADC conversion.

Prototype:
void
ADC_Enable(void)

Parameters:
None

Description:
This function will enable ADC module before start ADC conversion.

Return:
None

3.2.3.5 ADC_Disable

Disable ADC module.

Prototype:
void
ADC_Disable(void)

Parameters:
None

Description:

This function will disable ADC module.

Return:

None

3.2.3.6 ADC_Start

Start ADC function when select Software or Constant trigger.

Prototype:

void
ADC_Start(TrgType *Trg*)

Parameters:

Trg: Set trigger type.

which can be set as:

- **TRG_SOFTWARE**: Software triggered conversion
- **TRG_CONSTANT**: Constant AD conversion

Description:

This function will start ADC by *Trg* as **TRG_SOFTWARE**, **TRG_CONSTANT**.

Return:

None

3.2.3.7 ADC_StopConstantTrg

Stop ADC function when select by Constant trigger.

Prototype:

void
ADC_StopConstantTrg(void)

Parameters:

None

Description:

This function will stop the constant ADC.

Return:

None

3.2.3.8 ADC_GetConvertState

Get the state of ADC conversion by input trigger information.

Prototype:

WorkState
ADC_GetConvertState(TrgType *Trg*)

Parameters:

Trg: Set trigger type.

which can be set as:

- **TRG_SOFTWARE:** Software triggered conversion
- **TRG_CONSTANT:** Constant AD conversion
- **TRG_TIMER:** Timer triggered conversion
- **TRG_PMD:** PMD triggered conversion

Description:

This function will get the state of ADC conversion as **BUSY/DONE**, when AD conversion is triggered set by **Trg** as **TRG_SOFTWARE**, **TRG_CONSTANT**, **TRG_TIMER**, **TRG_PMD**.

Return:

WorkState type, the value means:

BUSY: Conversion in progress

DONE: Conversion not in process

3.2.3.9 ADC_SetLowPowerMode

Set ADC module to run in Low Power mode or Normal mode.

Prototype:

void

ADC_SetLowPowerMode(FunctionalState **NewState**)

Parameters:

NewState: Specify ADC low power mode.

which can be set as:

- **DISABLE:** Exit low power mode to enter normal AD conversion
- **ENABLE:** Enter low power mode, AD conversion will not work.

Description:

This function will select ADC low power mode by **NewState** as **DISABLE**, or **ENABLE**.

When Power up reset, the ADC module is set in Low Power Mode so user must call **ADC_SetLowPowerMode(DISABLE)** before start AD conversion.

Return:

None

3.2.3.10 ADC_SetMonitor

Set and enable ADC monitor function.

Prototype:

void

ADC_SetMonitor(ADC_MonitorTypeDef * **Monitor**)

Parameters:

Monitor: It is a structure with detail as below:

```
typedef struct {  
    ADC_CMPCRx CMPCRx;  
    ADC_REGx ResultREGx;  
    uint32_t CmpTimes;  
    ADC_CmpCondition Condition;  
    uint32_t CmpValue;  
} ADC_MonitorTypeDef
```

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

Description:

This function will set AD conversion result monitoring function by ADC_MonitorTypeDef * **Monitor** and enable it.

Return:

None

3.2.3.11 ADC_DisableMonitor

Disable ADC monitor function.

Prototype:

void

ADC_DisableMonitor(ADC_CMPCR_x **CMPCR_x**)

Parameters:

CMPCR_x: Select compare control register.
which can be set as:

- **ADC_CMPCR_0**: ADCMPCR0
- **ADC_CMPCR_1**: ADCMPCR1

Description:

This function will disable ADC monitoring function by **CMPCR_x** as **ADC_CMPCR_0** or **ADC_CMPCR_1**.

Return:

None

3.2.3.12 ADC_GetConvertResult

Read ADC result storage flag, overrun state and value.

Prototype:

ADC_ResultUnion

ADC_GetConvertResult(ADC_REG_x **ResultREG_x**)

Parameters:

ResultREG_x: Set ADC result register.
which can be set as:

- **ADC_REG0**: ADREG0
- **ADC_REG1**: ADREG1
- **ADC_REG2**: ADREG2
- **ADC_REG3**: ADREG3
- **ADC_REG4**: ADREG4
- **ADC_REG5**: ADREG5
- **ADC_REG6**: ADREG6
- **ADC_REG7**: ADREG7
- **ADC_REG8**: ADREG8
- **ADC_REG9**: ADREG9
- **ADC_REG10**: ADREG10
- **ADC_REG11**: ADREG11

Description:

This function will read AD conversion result, overrun flag & AD conversion result storage flag by specified ADC result register by **ResultREGx** as **ADC_REG_0**, **ADC_REG_1**, **ADC_REG_2**, **ADC_REG_3**, **ADC_REG_4**, **ADC_REG_5**, **ADC_REG_6**, **ADC_REG_7**, **ADC_REG_8**, **ADC_REG_9**, **ADC_REG_10**, **ADC_REG_11**.

Return:

ADC_ResultUnion type:

--- ADC result value

--- Stored

--- OverRun

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

3.2.3.13 ADC_SelectPMDTrgProgNum

Select the program to be started (program 0 to 5) by each of trigger inputs PMD0 to PMD3.

Prototype:

void

ADC_SelectPMDTrgProgNum(PMD_TRG_PROGRAM_SELx **SELx**,
uint8_t **MacroProgNum**)

Parameters:

SELx: Specify the "trigger program number select register".

which can be set as:

- **PMD_TRG_PROGRAM_SEL0:** ADPSEL0
- **PMD_TRG_PROGRAM_SEL1:** ADPSEL1
- **PMD_TRG_PROGRAM_SEL2:** ADPSEL2
- **PMD_TRG_PROGRAM_SEL3:** ADPSEL3

MacroProgNum: Program number to be selected, together with its Enabled or Disabled setting through a macro of TRG_ENABLE(x)/TRG_DISABLE(x).

which can be set as:

- **TRG_ENABLE(PROGRAM0):** Enable PMD Trigger Program Select Register with Program 0
- **TRG_DISABLE(PROGRAM0):** Disable PMD Trigger Program Select Register with Program 0
- **TRG_ENABLE(PROGRAM1):** Enable PMD Trigger Program Select Register with Program 1
- **TRG_DISABLE(PROGRAM1):** Disable PMD Trigger Program Select Register with Program 1
- **TRG_ENABLE(PROGRAM2):** Enable PMD Trigger Program Select Register with Program 2
- **TRG_DISABLE(PROGRAM2):** Disable PMD Trigger Program Select Register with Program 2
- **TRG_ENABLE(PROGRAM3):** Enable PMD Trigger Program Select Register with Program 3
- **TRG_DISABLE(PROGRAM3):** Disable PMD Trigger Program Select Register with Program 3
- **TRG_ENABLE(PROGRAM4):** Enable PMD Trigger Program Select Register with Program 4
- **TRG_DISABLE(PROGRAM4):** Disable PMD Trigger Program Select Register with Program 4
- **TRG_ENABLE(PROGRAM5):** Enable PMD Trigger Program Select Register with Program 5

- **TRG_DISABLE(PROGRAM5):** Disable PMD Trigger Program Select Register with Program 5

Description:

This function will set the PMD Trigger Program Number Select Register by **SELx**, and enable/disable the specified register to be started the specified program by **MacroProgNum**.

Return:

None

3.2.3.14 ADC_SetPMDTrgProgINT

Select the interrupt to be generated for each of program 0 to 5.

Prototype:

void
ADC_SetPMDTrgProgINT (PMD_TrgProgINTTypeDef * **TrgProgINT**)

Parameters:

TrgProgINT: The structure containing interrupt configuration for all of PMD Trigger Program Numbers.

```
typedef struct {  
    PMD_INT_NAME IntProg0;  
    PMD_INT_NAME IntProg1;  
    PMD_INT_NAME IntProg2;  
    PMD_INT_NAME IntProg3;  
    PMD_INT_NAME IntProg4;  
    PMD_INT_NAME IntProg5;  
} PMD_TrgProgINTTypeDef
```

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

Description:

This function will set the interrupt to be generated for each of program 0 to 5 by **TrgProgINT**.

Return:

None

3.2.3.15 ADC_SetTimerTrg

Set Timer Trigger Program Register.

Prototype:

void
ADC_SetTimerTrg(ADC_REGx **REGx**, uint8_t **MacroAINx**)

Parameters:

ResultREGx: Set AD Conversion Result Register for programming timer triggers.

which can be set as:

- **ADC_REG0:** ADREG0
- **ADC_REG1:** ADREG1
- **ADC_REG2:** ADREG2
- **ADC_REG3:** ADREG3
- **ADC_REG4:** ADREG4
- **ADC_REG5:** ADREG5

- ADC_REG6: ADREG6
- ADC_REG7: ADREG7
- ADC_REG8: ADREG8
- ADC_REG9: ADREG9
- ADC_REG10: ADREG10
- ADC_REG11: ADREG11

MacroAINx: AIN pin to be selected, together with its Enabled or Disabled setting through a macro of TRG_ENABLE(x)/TRG_DISABLE(x).

which can be set as:

- TRG_ENABLE(AIN0): Enable REG with AIN0
- TRG_DISABLE(AIN0): Disable REG with AIN0
- TRG_ENABLE(AIN1): Enable REG with AIN1
- TRG_DISABLE(AIN1): Disable REG with AIN1
-
-
-
- TRG_ENABLE(AIN20): Enable REG with AIN20
- TRG_DISABLE(AIN20): Disable REG with AIN20
- TRG_ENABLE(AIN21): Enable REG with AIN21
- TRG_DISABLE(AIN21): Disable REG with AIN21

Description:

This function will set AD Conversion Result Register by **ResultREGx** as **ADC_REG_0, ADC_REG_1, ADC_REG_2, ADC_REG_3, ADC_REG_4, ADC_REG_5, ADC_REG_6, ADC_REG_7, ADC_REG_8, ADC_REG_9, ADC_REG_10, ADC_REG_11**, and enable/disable REG with AIN pin by **MacroAINx** in Timer Trigger Program Register.

Return:

None

3.2.3.16 ADC_SetSWTrg

Set Software Trigger Program Register.

Prototype:

void

ADC_SetSWTrg(ADC_REGx **REGx**, uint8_t **MacroAINx**)

Parameters:

ResultREGx: Set AD Conversion Result Register for programming software triggers.

which can be set as:

- ADC_REG0: ADREG0
- ADC_REG1: ADREG1
- ADC_REG2: ADREG2
- ADC_REG3: ADREG3
- ADC_REG4: ADREG4
- ADC_REG5: ADREG5
- ADC_REG6: ADREG6
- ADC_REG7: ADREG7
- ADC_REG8: ADREG8
- ADC_REG9: ADREG9
- ADC_REG10: ADREG10
- ADC_REG11: ADREG11

MacroAINx: AIN pin to be selected, together with its Enabled or Disabled setting through a macro of TRG_ENABLE(x)/TRG_DISABLE(x).

which can be set as:

- TRG_ENABLE(AIN0): Enable REG with AIN0
- TRG_DISABLE(AIN0): Disable REG with AIN0
- TRG_ENABLE(AIN1): Enable REG with AIN1
- TRG_DISABLE(AIN1): Disable REG with AIN1
-
-
-
- TRG_ENABLE(AIN20): Enable REG with AIN20
- TRG_DISABLE(AIN20): Disable REG with AIN20
- TRG_ENABLE(AIN21): Enable REG with AIN21
- TRG_DISABLE(AIN21): Disable REG with AIN21

Description:

This function will set AD Conversion Result Register by **ResultREGx** as **ADC_REG_0, ADC_REG_1, ADC_REG_2, ADC_REG_3, ADC_REG_4, ADC_REG_5, ADC_REG_6, ADC_REG_7, ADC_REG_8, ADC_REG_9, ADC_REG_10, ADC_REG_11**, and enable/disable REG with AIN pin by **MacroAINx** in Software Trigger Program Register.

Return:

None

3.2.3.17 ADC_SetConstantTrg

Set Constant Trigger Program Register.

Prototype:

void
ADC_SetConstantTrg(ADC_REGx **REGx**, uint8_t **MacroAINx**)

Parameters:

ResultREGx: Set AD Conversion Result Register for programming constant triggers.

which can be set as:

- ADC_REG0: ADREG0
- ADC_REG1: ADREG1
- ADC_REG2: ADREG2
- ADC_REG3: ADREG3
- ADC_REG4: ADREG4
- ADC_REG5: ADREG5
- ADC_REG6: ADREG6
- ADC_REG7: ADREG7
- ADC_REG8: ADREG8
- ADC_REG9: ADREG9
- ADC_REG10: ADREG10
- ADC_REG11: ADREG11

MacroAINx: AIN pin to be selected, together with its Enabled or Disabled setting through a macro of TRG_ENABLE(x)/TRG_DISABLE(x).

which can be set as:

- TRG_ENABLE(AIN0): Enable REG with AIN0
- TRG_DISABLE(AIN0): Disable REG with AIN0

- **TRG_ENABLE(AIN1):** Enable REG with AIN1
- **TRG_DISABLE(AIN1):** Disable REG with AIN1
- .
- .
- .
- **TRG_ENABLE(AIN20):** Enable REG with AIN20
- **TRG_DISABLE(AIN20):** Disable REG with AIN20
- **TRG_ENABLE(AIN21):** Enable REG with AIN21
- **TRG_DISABLE(AIN21):** Disable REG with AIN21

Description:

This function will set AD Conversion Result Register by **ResultREGx** as **ADC_REG_0, ADC_REG_1, ADC_REG_2, ADC_REG_3, ADC_REG_4, ADC_REG_5, ADC_REG_6, ADC_REG_7, ADC_REG_8, ADC_REG_9, ADC_REG_10, ADC_REG_11**, and enable/disable REG with AIN pin by **MacroAINx** in Constant Trigger Program Register.

Return:

None

3.2.4 Data Structure Description

3.2.4.1 ADC_MonitorTypeDef

Data Fields for this structure:

ADC_CMPCR_x

CMPCR_x Select Compare Control Register.

which can be:

- **ADC_CMPCR_0:** ADCMPCR0
- **ADC_CMPCR_1:** ADCMPCR0

ADC_REG_x

ResultREGx Select which ADC Result Register will be used.

which can be set as:

- **ADC_REG0:** ADREG0
- **ADC_REG1:** ADREG1
- **ADC_REG2:** ADREG2
- **ADC_REG3:** ADREG3
- **ADC_REG4:** ADREG4
- **ADC_REG5:** ADREG5
- **ADC_REG6:** ADREG6
- **ADC_REG7:** ADREG7
- **ADC_REG8:** ADREG8
- **ADC_REG9:** ADREG9
- **ADC_REG10:** ADREG10
- **ADC_REG11:** ADREG11

uint32_t

CmpTimes Define how many times will Comparison be counted.

which can be:

1 to 16

ADC_CmpCondition

Condition Conditon to compare ADREGx with ADCMPy.

(x = 0 to 11, y = 0 to 1)

which can be:

- **LARGER_THAN_COMPARE_REG**
- **SMALLER_THAN_COMPARE_REG**

uint32_t

CmpValue Comparison value to be set in ADCMP0 or ADCMP1.

which can be:

- **0 to 4095 for 12bits mode**
- **0 to 1023 for 10bits mode**

3.2.4.2 ADC_ResultUnion

Data Fields for this structure:

uint32_t

All: AD Conversion Result .

Bit

uint32_t

Stored: 1 '1' means AD result has been stored.

uint32_t

OverRun: 1 Overrun flag.

uint32_t

Reserved1: 2 Reserved1.

uint32_t

ADResult: 12 Store ADC result

uint32_t

Reserved2: 16 Reserved2.

3.2.4.3 PMD_TrgProgINTTypeDef

Data Fields for this structure:

PMD_INT_NAME

IntProg0 Select the interrupt to be generated for program 0.

which can be:

- **PMD_INTNONE: No interrupt output**
- **PMD_INTADPD0: INTADPD0 output**
- **PMD_INTADPD1: INTADPD1 output**

PMD_INT_NAME

IntProg1 Select the interrupt to be generated for program 1.

which can be:

- **PMD_INTNONE: No interrupt output**
- **PMD_INTADPD0: INTADPD0 output**
- **PMD_INTADPD1: INTADPD1 output**

PMD_INT_NAME

IntProg2 Select the interrupt to be generated for program 2.

which can be:

- **PMD_INTNONE: No interrupt output**
- **PMD_INTADPD0: INTADPD0 output**
- **PMD_INTADPD1: INTADPD1 output**

PMD_INT_NAME

IntProg3 Select the interrupt to be generated for program 3.
which can be:

- **PMD_INTNONE: No interrupt output**
- **PMD_INTADPD0: INTADPD0 output**
- **PMD_INTADPD1: INTADPD1 output**

PMD_INT_NAME

IntProg4 Select the interrupt to be generated for program 4.
which can be:

- **PMD_INTNONE: No interrupt output**
- **PMD_INTADPD0: INTADPD0 output**
- **PMD_INTADPD1: INTADPD1 output**

PMD_INT_NAME

IntProg5 Select the interrupt to be generated for program 5.
which can be:

- **PMD_INTNONE: No interrupt output**
- **PMD_INTADPD0: INTADPD0 output**
- **PMD_INTADPD1: INTADPD1 output**

3.2.4.4 PMD_TrgTypeDef

Data Fields for this structure:

PMD_PROGRAMx

ProgNum Select Program Number for ADPSETx (x = 0 to 5).
which can be:

- **PROGRAM0 to PROGRAM5**

uint8_t

Reg0_AINx elect Analog Input channel together with its enabled or disabled setting for REG0 in ADPSETx. It must be input with macro as the format below: TRG_ENABLE(y), TRG_DISABLE(y).

'y' above can be one of the following values:

- **AIN0 to AIN21**

uint8_t

Reg1_AINx elect Analog Input channel together with its enabled or disabled setting for REG1 in ADPSETx. Other information is same as above.

uint8_t

Reg2_AINx elect Analog Input channel together with its enabled or disabled setting for REG2 in ADPSETx. Other information is same as above.

uint8_t

Reg3_AINx elect Analog Input channel together with its enabled or disabled setting for REG3 in ADPSETx. Other information is same as above.

4. CG

4.1 Overview

The CG API provides a set of functions for using the TMPM384 CG modules as the following:

- Set up high-speed and low-speed oscillators, 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, Slow 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\tmpm384_cg.c, with TX03_Periph_Driver\inc\tmpm384_cg.h containing the API definitions for use by applications.

The following symbols *fosc*, *fs*, *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 input from the X1 and X2 pins.

fs: Clock input from the XT1 and XT2 (low-speed clock).

fppll: Clock quadrupled by PLL.

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

fgear: Clock specified by SYSCR1<GEAR2:0>.

fsys: Clock specified by CKSEL<SYSCK> (system clock).

fperiph: Clock specified by SYSCR1<FPSEL1:0>.

$\Phi T0$: Clock specified by CGSYSCR<PRCK2:0> (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_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**)
- ◆ Result CG_SetFs(FunctionalState **NewState**)
- ◆ FunctionalState CG_GetFsState(void)
- ◆ void CG_SetPortM(CG_PortMMode **Mode**)
- ◆ void CG_SetSTBYMode(CG_STBYMode **Mode**)
- ◆ CG_STBYMode CG_GetSTBYMode(void)
- ◆ void CG_SetExitStopModeFosc(FunctionalState **NewState**)
- ◆ FunctionalState CG_GetExitStopModeFoscState(void)
- ◆ void CG_SetExitStopModeFs(FunctionalState **NewState**)
- ◆ FunctionalState CG_GetExitStopModeFsState(void)
- ◆ void CG_SetPinStateInStopMode(FunctionalState **NewState**)
- ◆ FunctionalState CG_GetPinStateInStopMode(void)
- ◆ Result CG_SetFcSrc(CG_FcSrc **Source**)
- ◆ CG_FcSrc CG_GetFcSrc(void)
- ◆ Result CG_SetFsysSrc(CG_FsysSrc **Source**)
- ◆ CG_FsysSrc CG_GetFsysSrc(void)
- ◆ 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_SetPLL(), CG_GetPLLState(),
CG_SetFosc(), CG_SetFoscSrc(), CG_GetFoscSrc(), CG_GetFoscState(),
CG_SetFs(), CG_GetFsState(), CG_SetFcSrc(), CG_GetFcSrc(),
CG_SetFsysSrc(), CG_GetFsysSrc(), CG_SetPortM().
- 2) The 2nd group of APIs handle settings of standby modes:
CG_SetSTBYMode(), CG_GetSTBYMode(), CG_SetExitStopModeFosc(),
CG_GetExitStopModeFoscState(), CG_SetExitStopModeFs(),
CG_GetExitStopModeFsState(), CG_SetPinStateInStopMode(),
CG_GetPinStateInStopMode().
- 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**: fgear = fc/4
- **CG_DIVIDE_8**: fgear = fc/8
- **CG_DIVIDE_16**: fgear = fc/16

Description :

This function will set the dividing level between clock fgear and fc.

Return:

None

4.2.3.2 CG_GetFgearLevel

Get the dividing level between fgear and fc.

Prototype:

CG_DivideLevel
CG_GetFgearLevel (void)

Parameters:

None

Description:

This function will get the dividing level between fgear and fc.
If the value "Reserved" is read from the register, the API will return **CG_DIVIDE_UNKNOWN**.

Return:

The dividing level between clock fgear and fc.
The value returned can be one of the following values:

CG_DIVIDE_1: fgear = fc

CG_DIVIDE_2: fgear = fc/2

CG_DIVIDE_4: fgear = fc/4

CG_DIVIDE_8: fgear = fc/8

CG_DIVIDE_16: fgear = fc/16

CG_DIVIDE_UNKNOWN: invalid data is read

4.2.3.3 CG_SetPhiT0Src

Select the PhiT0(Φ T0) source between fperiph and fc or fperiph and fs.

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 fgear.
- **CG_PHIT0_SRC_FC** means PhiT0 source is fc.
- **CG_PHIT0_SRC_FS** means PhiT0 source is fs.

Description:

This function will select the PhiT0(Φ T0) source.

Return:

None

4.2.3.4 CG_GetPhiT0Src

Get the PhiT0 (Φ T0) source.

Prototype:

CG_PhiT0Src
CG_GetPhiT0Src (void)

Parameters:

None

Description:

This function will get the PhiT0(Φ T0) source.

Return:

CG_PHIT0_SRC_FGEAR means PhiT0 source is fgear.

CG_PHIT0_SRC_FC means PhiT0 source is fc.

CG_PHIT0_SRC_FS means PhiT0 source is fs.

4.2.3.5 CG_SetPhiT0Level

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

Prototype:

Result

CG_SetPhiT0Level (CG_DivideLevel ***DividePhiT0FromFc***)

Parameters:

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

This parameter can be one of the following values:

- **CG_DIVIDE_1**: $\Phi T0 = fc$ or fs
- **CG_DIVIDE_2**: $\Phi T0 = fc/2$ or $fs/2$
- **CG_DIVIDE_4**: $\Phi T0 = fc/4$ or $fs/2$
- **CG_DIVIDE_8**: $\Phi T0 = fc/8$ or $fs/8$
- **CG_DIVIDE_16**: $\Phi T0 = fc/16$ or $fs/16$
- **CG_DIVIDE_32**: $\Phi T0 = fc/32$ or $fs/32$
- **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 or $\Phi T0$ and fs.

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 fc or $\Phi T0$ and fs, the value will be one of the following:

- CG_DIVIDE_1**: $\Phi T0 = fc$ or fs
- CG_DIVIDE_2**: $\Phi T0 = fc/2$ or $fs/2$
- CG_DIVIDE_4**: $\Phi T0 = fc/4$ or $fs/4$
- CG_DIVIDE_8**: $\Phi T0 = fc/8$ or $fs/8$
- CG_DIVIDE_16**: $\Phi T0 = fc/16$ or $fs/16$

CG_DIVIDE_32: $\Phi T0 = fc/32$ or $fs/32$
CG_DIVIDE_64: $\Phi T0 = fc/64$
CG_DIVIDE_128: $\Phi T0 = fc/128$
CG_DIVIDE_256: $\Phi T0 = fc/256$
CG_DIVIDE_512: $\Phi T0 = fc/512$
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_FS:** SCOUT source is set to fs.
- **CG_SCOUT_SRC_HALF_FSYS:** SCOUT source is set to fsys/2.
- **CG_SCOUT_SRC_FSYS:** SCOUT source is set to fsys.
- **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_FS: SCOUT source is fs
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 $\Phi T0$

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_OSC1:** fosc1 is selected as timer source,
- **CG_WARM_UP_SRC_OSC2:** fosc2 is selected as timer source,
- **CG_WARM_UP_SRC_XT1:** fs is selected as timer source.

Time: If **Source** is **CG_WARM_UP_SRC_OSC1** or

CG_WARM_UP_SRC_OSC2, Time value range is 0U to 0x1000U.

If **Source** is **CG_WARM_UP_SRC_XT1**, Time value range is 0U to 0x4000U.

Description:

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

$$\text{Setting_value} = ((\text{warm-up time}) / (\text{input cycle time by frequency}))/16$$

Example of calculating the register value for warm-up time:

```
/* set up warm time 100us, input cycle by frequency is 8M */  
So value = 100*10E(-6)/(1/(8*10E(6)))/16=0x0320>>4=0x32
```

Return:

None.

4.2.3.10 CG_StartWarmUp

Start warm up timer.

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 that warm-up operation is in middle or completed.

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_OSC1, 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_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.

If the PLL is selected as fc, it can't be disabled; in that case the API will return **ERROR**.

Return:

SUCCESS: operation is finished successfully.

ERROR: operation is not done.

4.2.3.13 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.

4.2.3.14 CG_SetFosc

Enable or disable the high-speed oscillator (osc1 or osc2).

Prototype:

Result

CG_SetFosc(CG_FoscSrc **Source**,
FunctionalState **NewState**)

Parameters:

Source: select source for fosc.

- **CG_FOSC_OSC1**: fosc1 is selected,
- **CG_FOSC_OSC2**: fosc2 is selected.

NewState: select source for fosc.

- **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.

When fgear is selected as system clock (fsys), the high-speed oscillator (fosc) can't be disabled; in this case the API will return **ERROR**.

Return:

SUCCESS: operation is finished successfully.

ERROR: operation is not done.

4.2.3.15 CG_SetFoscSrc

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

Prototype:

void

CG_SetFoscSrc(CG_FoscSrc **Source**)

Parameters:

Source: select source for fosc.

- **CG_FOSC_OSC1**: fosc1 is selected,
- **CG_FOSC_OSC2**: fosc2 is selected.

Description:

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

Return:

None

4.2.3.16 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_OSC1: fosc1 is selected.

CG_FOSC_OSC2: fosc2 is selected.

4.2.3.17 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_OSC1**: fosc1 is selected,

➤ **CG_FOSC_OSC2**: fosc2 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.

4.2.3.18 CG_SetFs

Enable or disable the low-speed oscillator (XT1).

Prototype:

Result

CG_SetFs(FunctionalState **NewState**)

Parameters:

NewState:

➤ **ENABLE**: to enable the low-speed oscillator.

➤ **DISABLE**: to disable the low-speed oscillator.

Description:

This function will enable or disable the low-speed oscillator (XT1).

When fs is selected as system clock (fsys), the low-speed oscillator (XT1) can't be disabled, in that case the API will return **ERROR**.

Return:

SUCCESS: operation is finished successfully.

ERROR: operation is not done.

4.2.3.19 CG_GetFsState

Get the state of the low-speed oscillator (XT1)

Prototype:

FunctionalState

CG_GetFsState (void)

Parameters:

None

Description:

This function will get the state of the low-speed oscillator (XT1).

Return:

The state of XT1

ENABLE: XT1 is enabled.

DISABLE: XT1 is disabled.

4.2.3.20 CG_SetPortM

Set portM for X1/X2 or general port.

Prototype:

void

CG_SetPortM(CG_PortMMode **Mode**)

Parameters:**Mode:**

- **CG_PORTM_AS_GPIO** : to set port M as general port,
- **CG_PORTM_AS_HOSC**: to set port M as Hosc.

Description:

This function will set port M as general port when **Mode** is

CG_PORTM_AS_GPIO and set port M as Hosc when **Mode** is **CG_PORTM_AS_HOSC**.

Return:

None

4.2.3.21 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_STOP**: STOP mode. All the internal circuits including the internal oscillator are brought to a stop.
- **CG_STBY_MODE_SLEEP**: SLEEP mode. The internal low-speed oscillator, real time clock and RMC can operate.
- **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

4.2.3.22 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_STOP: STOP mode.

CG_STBY_MODE_SLEEP: SLEEP mode

CG_STBY_MODE_IDLE: IDLE mode

CG_STBY_MODE_UNKNOWN: Invalid data is read.

4.2.3.23 CG_SetExitStopModeFosc

Enable or disable fosc after releasing stop mode

Prototype:

void

CG_SetExitStopModeFosc(FunctionalState **NewState**)

Parameters:

NewState:

- **ENABLE** : enable X1 after releasing stop mode
- **DISABLE** : do not enable X1 after releasing stop mode

Description:

This function will enable or disable X1 after releasing stop mode.

Return:

None

4.2.3.24 CG_GetExitStopModeFoscState

Get the state of X1 after releasing stop mode

Prototype:

FunctionalState

CG_GetExitStopModeFoscState (void)

Parameters:

None

Description:

This function will get the state of fosc after releasing stop mode

Return:

ENABLE: enable X1 after releasing stop mode

DISABLE: do not enable X1 after releasing stop mode

4.2.3.25 CG_SetExitStopModeFs

Enable or disable XT1 after releasing stop mode

Prototype:

void

CG_SetExitStopModeFs (FunctionalState **NewState**)

Parameters:

NewState:

- **ENABLE :** enable XT1 after releasing stop mode
- **DISABLE:** do not enable XT1 after releasing stop mode

Description:

This function will enable or disable XT1 after releasing stop mode

Return:

None

4.2.3.26 CG_GetExitStopModeFsState

Get the state of XT1 after releasing stop mode

Prototype:

FunctionalState

CG_GetExitStopModeFsState (void)

Parameters:

None

Description:

This function will get the state of XT1 after releasing stop mode.

Return:

ENABLE: enable XT1 after releasing stop mode

DISABLE: do not enable XT1 after releasing stop mode

4.2.3.27 CG_SetPinStateInStopMode

Specify the pin status in stop mode

Prototype:

void

CG_SetPinStateInStopMode (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 STOP Mode" in the datasheet.

Description:

This function will specify the pin status in stop mode.

Return:

None

4.2.3.28 CG_GetPinStateInStopMode

Get the pin status in stop mode

Prototype:

FunctionalState

CG_GetPinStateInStopMode (void)

Parameters:

None

Description:

This function will get the pin status in stop mode.

Return:

The pin state in stop mode

DISABLE: <DRVE>=0

ENABLE: <DRVE>=1

4.2.3.29 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_FPLL** : fc source will be set to fpll

Description:

This function will set the clock source of fc.

The following conditions should be matched before calling this API

- a) high-speed oscillator is set to on
- b) If the input for parameter **Source** is **CG_FC_SRC_FPLL**, PLL circuit must be enabled earlier (by calling “**CG_SetPLL(ENABLE)**”) together with condition a) matched.

Otherwise, calling of this API will return **ERROR**

Return:

SUCCESS: set clock source for fc successfully

ERROR: clock source of fc is not changed.

4.2.3.30 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_FPLL: fc source is set to fpll.

4.2.3.31 CG_SetFsysSrc

Set the clock source of fsys.

Prototype:

Result

CG_SetFsysSrc (CG_FsysSrc **Source**)

Parameters:

Source: select the source of system clock (fsys)

This parameter can be one of the following values:

- **CG_FSYS_SRC_FGEAR:** source of fsys will be set to fgear
- **CG_FSYS_SRC_FS:** source of fsys will be set to fs.

Description:

This function will set the clock source of system clock (fsys).

If **CG_FSYS_SRC_FGEAR** is specified, the high-speed oscillator (X1) should be enabled earlier; if **CG_FSYS_SRC_FS** is specified, the low-speed oscillator (XT1) should be enabled earlier; otherwise, calling of this API will return **ERROR**.

Return:

SUCCESS: set clock source for fsys successfully

ERROR: the clock source of fsys is not changed

4.2.3.32 CG_GetFsysSrc

Get the clock source of fsys

Prototype:

CG_FsysSrc

CG_GetFsysSrc (void)

Parameters:

None

Description:

This function will get the source of system clock (fsys)

Return:

Source of fsys

The value returned can be one of the following values:

CG_FSYS_SRC_FGEAR : source of fsys is set to fgear
CG_FSYS_SRC_FS : source of fsys is set to fs.

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_A** : INT10
- **CG_INT_SRC_B** : INT11
- **CG_INT_SRC_C** : INT12
- **CG_INT_SRC_D** : INT13
- **CG_INT_SRC_E** : INT14
- **CG_INT_SRC_F** : INT15
- **CG_INT_SRC_RTC**: RTC interrupt.
- **CG_INT_SRC_RMC_RX**: receptions interrupt of RMC.

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.

For **CG_INT_SRC_RMC_RX**, only "rising" state

(**CG_INT_ACTIVE_STATE_RISING**) will be set to the register, no matter what the value of **ActiveState** is. For **CG_INT_SRC_RTC**, only "falling" state

(**CG_INT_ACTIVE_STATE_FALLING**) will be set to the register, no matter what the value of **ActiveState** is.

Note:

CG_INT_SRC_6, CG_INT_SRC_7, CG_INT_SRC_9, CG_INT_SRC_A, CG_INT_SRC_B, CG_INT_SRC_C, CG_INT_SRC_D, CG_INT_SRC_E are invalid for M382.

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_A, CG_INT_SRC_B, CG_INT_SRC_C, CG_INT_SRC_D, CG_INT_SRC_E, CG_INT_SRC_F, CG_INT_SRC_RTC, CG_INT_SRC_RMC_RX.

Description:

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

Note:

CG_INT_SRC_6, CG_INT_SRC_7, CG_INT_SRC_9, CG_INT_SRC_A, CG_INT_SRC_B, CG_INT_SRC_C, CG_INT_SRC_D, CG_INT_SRC_E are invalid for M382.

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 INT request for releasing standby mode.

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_A, CG_INT_SRC_B,
CG_INT_SRC_C, CG_INT_SRC_D, CG_INT_SRC_E, CG_INT_SRC_F,
CG_INT_SRC_RTC, CG_INT_SRC_RMC_RX.

Description:

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

Note:

CG_INT_SRC_6, CG_INT_SRC_7, CG_INT_SRC_9, CG_INT_SRC_A,
CG_INT_SRC_B, CG_INT_SRC_C, CG_INT_SRC_D, CG_INT_SRC_E are
invalid for M382.

Return:

None

4.2.3.36 CG_GetNMIFlag

Get the NMI flag, which shows what triggered NMI

Prototype:

CG_NMIFactor
CG_GetNMIFlag (void)

Parameters:

None

Description:

This function will get the NMI flag, which shows what triggered NMI.

Return:

NMI value:

WDT (Bit 0) means generated from WDT.

VoltageDetection (Bit 2) means generated from voltage level detection.

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 will get the reset flag which shows the trigger of reset and clear the reset flag.

Return:

Reset flag:

PowerOn (Bit 0) means reset from power-on.

ResetPin (Bit 1) means reset from Reset pin.

WDTReset (Bit 2) means reset from WDT.
DebugReset (Bit 4) means reset from SYSRESETREQ.
OFDReset (Bit 5) means reset from OFD.

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

Reserved(Bit 1) means reserved.

uint32_t

VoltageDetection (Bit 2) means generated from voltage level detection.

4.2.4.2 CG_ResetFlag

Data Fields:

uint32_t

All specifies CG reset source.

Bit Fields:

uint32_t

PowerOn(Bit 0) means reset from power-on.

uint32_t

ResetPin(Bit 1) means reset from Reset pin.

uint32_t

WDTReset(Bit 2) means reset from WDT.

uint32_t

Reserved(Bit 3) means reserved.

uint32_t

DebugReset(Bit 4) means reset from SYSRESETREQ.

uint32_t

OFDReset(Bit 5) means reset from OFD.

5. DMAC

5.1 Overview

TOSHIBA TPM384 has a DMA controller controlled by DMA request select registers, and DMA controller has two channels. Each channel can operate in one of three transferring types (memory to memory, memory to peripheral, peripheral to memory). The priority of DMA channel 0 is higher than DMA 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\tpm384_dmac.c (*), with \Libraries\TX03_Periph_Driver\inc\tpm384_dmac.h (*) containing the API definitions for use by applications.

5.2 API Functions

5.2.1 Function List

- ◆ void DMAC_Enable(void);
- ◆ void DMAC_Disable(void);
- ◆ DMAC_INTReq DMAC_GetINTReq(void);
- ◆ DMAC_TxINTReq DMAC_GetTxINTReq(DMAC_Channel **Chx**);
- ◆ void DMAC_ClearTxINTReq(DMAC_Channel **Chx**, DMAC_INTSrc **INTSource**);
- ◆ DMAC_TxINTReq DMAC_GetRawTxINTReq(DMAC_Channel **Chx**);
- ◆ WorkState DMAC_GetChannelTxState(DMAC_Channel **Chx**);
- ◆ void DMAC_SetSWBurstReq(DMAC_ReqNum **BurstReq**);
- ◆ DMAC_BurstReqState DMAC_GetSWBurstReqState(void);
- ◆ void DMAC_SetSWSingleReq(DMAC_ReqNum **SingleReq**);
- ◆ DMAC_SingleReqState DMAC_GetSWSingleReqState(void);
- ◆ void DMAC_SetLinkedList(DMAC_Channel **Chx**, uint32_t **LinkedAddr**);
- ◆ WorkState DMAC_GetFIFOState(DMAC_Channel **Chx**);
- ◆ void DMAC_SetDMAHalt(DMAC_Channel **Chx**, FunctionalState **NewState**);
- ◆ void DMAC_SetLockedTx(DMAC_Channel **Chx**, FunctionalState **NewState**);
- ◆ void DMAC_SetTxINTConfig(DMAC_Channel **Chx**, DMAC_INTSrc **INTSource**, FunctionalState **NewState**);
- ◆ void DMAC_SetDMACChannel(DMAC_Channel **Chx**, FunctionalState **NewState**);
- ◆ void DMAC_Init(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_SetDMACChannel() 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 DMAC_SetSWBurstReq(),DMAC_GetSWBurstReqState(),DMAC_SetSWSingleReq() 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(void);
```

Parameters:

None

Description:

This function will enable DMA circuit.

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(void);
```

Parameters:

None

Description:

This function will disable DMA circuit.

Return:

None

5.2.3.3 DMAC_GetINTReq

Get DMA Channel interrupt request state.

Prototype:

```
DMAC_INTReq  
DMAC_GetINTReq(void);
```

Parameters:

None

Description:

This function will get DMA Channel interrupt request state.

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(DMAC_Channel Chx);
```

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.

Description:

This function will get channel 0 transfer interrupt state when **Chx** is **DMAC_CHANNEL_0** and get 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(DMAC_Channel Chx,  
                    DMAC_INTSrc INTSource);
```

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.

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(DMAC_Channel **Chx**);

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.

Description:
This function will get channel 0 transfer raw interrupt state when **Chx** is **DMAC_CHANNEL_0** and get 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(DMAC_Channel **Chx**);

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.

Description:
This function will get DMA channel 0 transfer state when **Chx** is **DMAC_CHANNEL_0**, and get DMA 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 DMAC_SetSWBurstReq

Set DMA burst transfer requests by software.

Prototype:

void

DMAC_SetSWBurstReq(DMAC_ReqNum **BurstReq**);

Parameters:

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

- **DMAC_SIO_0_RTX** for SIO0 Reception / Transmission,
- **DMAC_SIO_1_RTX** for SIO1 Reception / Transmission,
- **DMAC_SIO_2_RTX** for SIO2 Reception / Transmission,
- **DMAC_SIO_3_RTX** for SIO3 Reception / Transmission,
- **DMAC_SIO_4_RTX** for SIO4 Reception / Transmission,
- **DMAC_SSP0_TX** for SSP0 Transmission,
- **DMAC_SSP0_RX** for SSP0 Reception,
- **DMAC_SSP1_TX** for for SSP1 Transmission,
- **DMAC_SSP1_RX** for SSP1 Reception,

Description:

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

Return:

None

5.2.3.9 DMAC_GetSWBurstReqState

Get DMA software burst request state.

Prototype:

DMAC_BurstReqState

DMAC_GetSWBurstReqState(void);

Parameters:

None

Description:

This function will get DMA software burst request state.

Return:

The DMA burst request status.

5.2.3.10 DMAC_SetSWSingleReq

Set DMA single transfer requests by software.

Prototype:

void

DMAC_SetSWSingleReq(DMAC_ReqNum **SingleReq**);

Parameters:

SingleReq: Select burst request number, which can be:

- **DMAC_SSP0_RX** for SSP0 Reception,
- **DMAC_SSP1_RX** for SSP1 Reception,

Description:

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

Return:

None

5.2.3.11 DMAC_GetSWSingleReqState

Get DMA software single request state.

Prototype:

DMAC_SingleReqState
DMAC_GetSWSingleReqState(void);

Parameters:

None

Description:

This function will get DMA software single request state.

Return:

The DMA single request status.

5.2.3.12 DMAC_SetLinkedList

Set specified DMA Channel Linked List Item Register.

Prototype:

void
DMAC_SetLinkedList(DMAC_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 0xFFFFFFFF0.

Description:

This function will set specified DMA 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

5.2.3.13 DMAC_GetFIFOState

Indicates whether data is present in the channel FIFO

Prototype:

WorkState

DMAC_GetFIFOState(DMAC_Channel **Chx**);

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.

Description:

This function will get DMA 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**

5.2.3.14 DMAC_SetDMAHalt

Set whether ignore DMA request.

Prototype:

void

DMAC_SetDMAHalt(DMAC_Channel **Chx**,
FunctionalState **NewState**);

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.

NewState: New state of DMA halt.

This parameter can be one of the following values:

ENABLE or DISABLE

Description:

This function will set specified DMA Channel ignore DMA request.

Return:

None

5.2.3.15 DMAC_SetLockedTx

Set whether locked transfer.

Prototype:

```
void  
DMAC_SetLockedTx(DMAC_Channel Chx,  
                  FunctionalState NewState);
```

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.

NewState: New state of DMA transfer.

This parameter can be one of the following values:

ENABLE or **DISABLE**

Description:

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

Return:

None

5.2.3.16 DMAC_SetTxINTConfig

Enable or disable the specified DMA Channel transfer interrupt.

Prototype:

```
void  
DMAC_SetTxINTConfig(DMAC_Channel Chx,  
                    DMAC_INTSrc INTSource,  
                    FunctionalState NewState);
```

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.

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.

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 DMA Channel transfer end interrupt when **INTSource** is **DMAC_INT_TX_END** and enable or disable specified DMA Channel transfer error interrupt when **INTSource** is **DMAC_INT_TX_ERR**.

Return:

None

5.2.3.17 DMAC_SetDMAChannel

Enable or disable the specified DMA Channel.

Prototype:

```
void  
DMAC_SetDMAChannel(DMAC_Channel Chx,  
                   FunctionalState NewState);
```

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.

NewState: New state of DMA channel.

This parameter can be one of the following values:

ENABLE or **DISABLE**

Description:

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

Return:

None

5.2.3.18 DMAC_Init

Initialize and configure the specified DMA channel.

Prototype:

```
void  
DMAC_Init(DMAC_Channel Chx,  
          DMAC_InitTypeDef * InitStruct);
```

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.

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 DMA 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.

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.

DMAC_ReqNum

TxPeriph Set transfer destination or source peripheral (*See **Note**), which can be set as:

- **DMAC_SIO_0_RTX** for SIO0 Reception / Transmission,
- **DMAC_SIO_1_RTX** for SIO1 Reception / Transmission,
- **DMAC_SIO_2_RTX** for SIO2 Reception / Transmission,
- **DMAC_SIO_3_RTX** for SIO3 Reception / Transmission,
- **DMAC_SIO_4_RTX** for SIO4 Reception / Transmission,
- **DMAC_SSP0_TX** for SSP0 Transmission,
- **DMAC_SSP0_RX** for SSP0 Reception,
- **DMAC_SSP1_TX** for for SSP1 Transmission,
- **DMAC_SSP1_RX** for SSP1 Reception,

FunctionalState

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

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

Note:

TxPeriph is set for destination peripheral when ***TxDirection*** is **DMAC_MEMORY_TO_PERIPH**; ***TxPeriph*** is set for source peripheral when ***TxDirection*** is **DMAC_PERIPH_TO_MEMORY**. If ***TxDirection*** is **DMAC_MEMORY_TO_MEMORY**, ***TxPeriph*** is unused.

6. FC

6.1 Overview

TMPM384 device contains flash memory.
For TMPM384FD, the size of flash is 512Kbyte.

In on-board programming, the CPU is to execute software commands for rewriting or erasing the flash memory. Writing and erasing 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 please refers to the MCU data sheet.

This driver is contained in \Libraries\TX03_Periph_Driver\src\tmpm384_fc.c with \Libraries\TX03_Periph_Driver\inc\tmpm384_fc.h containing the API definitions for use by applications.

6.2 API Functions

6.2.1 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)

6.2.2 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().

6.2.3 Function Documentation

6.2.3.1 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 can not 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

6.2.3.2 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 register.

DISABLE: Protection function is not available.

ENABLE: Protection function is available.

6.2.3.3 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.

6.2.3.4 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

6.2.3.5 FC_ProgramBlockProtectState

Program the protection bits.

Prototype:

FC_Result

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.

6.2.3.6 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.

6.2.3.7 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 TPM384 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.

6.2.3.8 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.

6.2.3.9 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.

6.2.4 Data Structure Description

None.

7. GPIO

7.1 Overview

For TOSHIBA TMPM384 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/tmpm384_gpio.c, with /Libraries/TX03_Periph_Driver/inc/tmpm384_gpio.h containing the macros, data types, structures and API definitions for use by applications.

7.2 API Functions

7.2.1 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**);

7.2.2 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.

7.2.3 Function Documentation

7.2.3.1 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_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N.
- **GPIO_PP:** GPIO port P.
- **GPIO_PQ:** GPIO port Q.
- **GPIO_PR:** GPIO port R.
- **GPIO_PS:** GPIO port S.
- **GPIO_PT:** GPIO port T.
- **GPIO_PU:** GPIO port U.
- **GPIO_PV:** GPIO port V.

Description:

This function will read specified GPIO Data register.

Return:

The value read from DATA register.

7.2.3.2 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_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N.
- **GPIO_PP:** GPIO port P.
- **GPIO_PQ:** GPIO port Q.
- **GPIO_PR:** GPIO port R.
- **GPIO_PS:** GPIO port S.
- **GPIO_PT:** GPIO port T.
- **GPIO_PU:** GPIO port U.
- **GPIO_PV:** GPIO port V.

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.

7.2.3.3 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_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N.
- **GPIO_PP:** GPIO port P.
- **GPIO_PQ:** GPIO port Q.
- **GPIO_PR:** GPIO port R.
- **GPIO_PS:** GPIO port S.
- **GPIO_PT:** GPIO port T.
- **GPIO_PU:** GPIO port U.

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

Description:

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

Return:

None

7.2.3.4 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_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N.
- **GPIO_PP:** GPIO port P.
- **GPIO_PQ:** GPIO port Q.
- **GPIO_PR:** GPIO port R.
- **GPIO_PS:** GPIO port S.
- **GPIO_PT:** GPIO port T.
- **GPIO_PU:** GPIO port U.

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

7.2.3.5 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_PL**: GPIO port L.
- **GPIO_PM**: GPIO port M.
- **GPIO_PN**: GPIO port N.
- **GPIO_PP**: GPIO port P.
- **GPIO_PQ**: GPIO port Q.
- **GPIO_PR**: GPIO port R.
- **GPIO_PS**: GPIO port S.
- **GPIO_PT**: GPIO port T.
- **GPIO_PU**: GPIO port U.
- **GPIO_PV**: GPIO port V.

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

7.2.3.6 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_PL**: GPIO port L.
- **GPIO_PM**: GPIO port M.
- **GPIO_PN**: GPIO port N.
- **GPIO_PP**: GPIO port P.
- **GPIO_PQ**: GPIO port Q.
- **GPIO_PR**: GPIO port R.
- **GPIO_PS**: GPIO port S.
- **GPIO_PT**: GPIO port T.
- **GPIO_PU**: GPIO port U.

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

7.2.3.7 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_PL**: GPIO port L.
- **GPIO_PM**: GPIO port M.
- **GPIO_PN**: GPIO port N.
- **GPIO_PP**: GPIO port P.
- **GPIO_PQ**: GPIO port Q.
- **GPIO_PR**: GPIO port R.
- **GPIO_PS**: GPIO port S.
- **GPIO_PT**: GPIO port T.
- **GPIO_PU**: GPIO port U.
- **GPIO_PV**: GPIO port V.

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

7.2.3.8 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_PL**: GPIO port L.
- **GPIO_PM**: GPIO port M.
- **GPIO_PN**: GPIO port N.
- **GPIO_PP**: GPIO port P.
- **GPIO_PQ**: GPIO port Q.
- **GPIO_PR**: GPIO port R.
- **GPIO_PS**: GPIO port S.
- **GPIO_PT**: GPIO port T.
- **GPIO_PU**: GPIO port U.

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

7.2.3.9 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_PL**: GPIO port L.
- **GPIO_PM**: GPIO port M.
- **GPIO_PN**: GPIO port N.
- **GPIO_PP**: GPIO port P.
- **GPIO_PQ**: GPIO port Q.
- **GPIO_PR**: GPIO port R.
- **GPIO_PS**: GPIO port S.
- **GPIO_PT**: GPIO port T.
- **GPIO_PU**: GPIO port U.
- **GPIO_PV**: GPIO port V.

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

7.2.3.10 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_PL**: GPIO port L.
- **GPIO_PM**: GPIO port M.
- **GPIO_PN**: GPIO port N.
- **GPIO_PP**: GPIO port P.
- **GPIO_PQ**: GPIO port Q.
- **GPIO_PR**: GPIO port R.
- **GPIO_PS**: GPIO port S.
- **GPIO_PT**: GPIO port T.
- **GPIO_PU**: GPIO port U.
- **GPIO_PV**: GPIO port V.

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

7.2.3.11 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_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_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N.
- **GPIO_PP:** GPIO port P.
- **GPIO_PQ:** GPIO port Q.
- **GPIO_PR:** GPIO port R.
- **GPIO_PS:** GPIO port S.
- **GPIO_PT:** GPIO port T.
- **GPIO_PU:** GPIO port U.

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

7.2.3.12 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.
- **GPIO_PJ**: GPIO port J.
- **GPIO_PL**: GPIO port L.
- **GPIO_PM**: GPIO port M.
- **GPIO_PN**: GPIO port N.
- **GPIO_PP**: GPIO port P.
- **GPIO_PQ**: GPIO port Q.
- **GPIO_PR**: GPIO port R.
- **GPIO_PS**: GPIO port S.
- **GPIO_PT**: GPIO port T.
- **GPIO_PU**: GPIO port U.

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

7.2.3.13 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_PJ**: GPIO port J.
- **GPIO_PL**: GPIO port L.
- **GPIO_PN**: GPIO port N.
- **GPIO_PR**: GPIO port R.
- **GPIO_PT**: GPIO port T.

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.

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

7.2.3.14 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_PJ**: GPIO port J.
- **GPIO_PL**: GPIO port L.
- **GPIO_PN**: GPIO port N.
- **GPIO_PR**: GPIO port R.
- **GPIO_PT**: GPIO port T.

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.

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

7.2.4 Data Structure Description

7.2.4.1 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:** Don't have 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:** Don't have 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:** Don't have pull-down function or needn't change..

8. OFD

8.1 Overview

The oscillation frequency detector generates a reset for micro if the oscillation of high frequency for CPU clock exceeds the detection frequency range.

The OFD driver APIs provide a set of functions to enable or disable the OFD function, configure detection frequency, get the OFD status and so on.

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm384_ofd.c, with /Libraries/TX03_Periph_Driver/inc/tmpm384_ofd.h containing the macros, data types, structures and API definitions for use by applications.

8.2 API Functions

8.2.1 Function List

- ◆ void OFD_SetRegWriteMode(FunctionalState NewState);
- ◆ void OFD_Enable(void);
- ◆ void OFD_Disable(void);
- ◆ void OFD_SetDetectionFrequency(uint8_t HigherDetectionCount,
uint8_t LowerDetectionCount);
- ◆ void OFD_Reset(FunctionalState NewState);
- ◆ OFD_Status OFD_GetStatus(void);

8.2.2 Detailed Description

Functions listed above can be divided into three parts:

- 1) Initialize and configure OFD function by OFD_SetRegWriteMode(), OFD_SetDetectionFrequency (),OFD_Enable () and OFD_Disable ().
- 2) Get the OFD busy and frequency error info by OFD_GetStatus().
- 3) OFD_Reset () to Enable or disable the OFD reset.

8.2.3 Function Documentation

8.2.3.1 OFD_SetRegWriteMode

Enable or disable the writing of OFDCR2/OFDMN/OFDMX.

Prototype:

```
void  
OFD_SetRegWriteMode(FunctionalState NewState)
```

Parameters:

NewState is the new state of writing of OFDCR2/OFDMN/OFDMX registers.

This parameter can be one of the following values:

ENABLE or **DISABLE**

Description:

This function will enable writing of OFDCR2/OFDMN/OFDMX registers when **NewState** is **ENABLE**, and disable writing of OFDCR2/OFDMN/OFDMX registers when **NewState** is **DISABLE**.

Return:
None

8.2.3.2 OFD_Enable

Enable the OFD function.

Prototype:
void
OFD_Enable(void)

Parameters:
None.

Description:
This function will enable the OFD function.

Return:
None

8.2.3.3 OFD_Disable

Disable the OFD function.

Prototype:
void
OFD_Disable(void)

Parameters:
None.

Description:
This function will disable the OFD function.

Return:
None

8.2.3.4 OFD_SetDetectionFrequency

Set the count value of detection frequency.

Prototype:
void
OFD_SetDetectionFrequency(uint8_t **HigherDetectionCount**,
uint8_t **LowerDetectionCount**)

Parameters:
HigherDetectionCount is the count value of higher detection frequency.
LowerDetectionCount is the count value of lower detection frequency.

Description:

This function will set the count value of detection frequency, both higher detection frequency and lower detection frequency.

Return:

None

8.2.3.5 OFD_Reset

Enable or disable the OFD reset.

Prototype:

void
OFD_Reset(FunctionalState **NewState**)

Parameters:

NewState is the new state of enable or disable OFD reset.

This parameter can be one of the following values:

ENABLE or **DISABLE**

Description:

This function will Enable or disable the OFD reset.

Return:

None

8.2.3.6 OFD_GetStatus

Get the OFD busy and frequency error info.

Prototype:

OFD_Status
OFD_GetStatus(void)

Parameters:

None

Description:

This function will get the OFD busy and frequency error info.

Return:

OFD_Status is the structure of OFD status, which include the busy and frequency error info.

(Refer to "Data Structure Description" for details).

8.2.4 Data Structure Description

8.2.4.1 OFD_Status

Data Fields:

uint32_t

All: Data.

Bit

uint32_t

FrequencyError: 1

Frequency Error status

uint32_t

OFDBusy: 1

OFD Busy status

9. RMC

9.1 Overview

TOSHIBA TPM384 has remote control signal preprocessor: RMC0.

Reception of Remote Control Signal:

- Sampled by 32KHz clock
- Noise canceller
- Leader detection
- Batch reception up to 72bit of data

The RMC driver APIs provides a set of functions to configure each channel.

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm384_rmc.c, with /Libraries/TX03_Periph_Driver/inc/tmpm384_rmc.h containing the macros, data types, structures and API definitions for use by applications

9.2 API Functions

9.2.1 Function List

- ◆ void RMC_Enable(TSB_RMC_TypeDef * **RMCx**)
- ◆ void RMC_Disable(TSB_RMC_TypeDef * **RMCx**)
- ◆ void RMC_Init(TSB_RMC_TypeDef * **RMCx**, RMC_InitTypeDef * **RMC_InitStruct**)
- ◆ void RMC_SetRxCtrl(TSB_RMC_TypeDef * **RMCx**, FunctionalState **NewState**)
- ◆ RMC_RxDataTypeDef RMC_GetRxData(TSB_RMC_TypeDef * **RMCx**)
- ◆ void RMC_SetLeaderDetection(TSB_RMC_TypeDef * **RMCx**,
RMC_LeaderParameterTypeDef **LeaderPara**)
- ◆ void RMC_SetFallingEdgeINT(TSB_RMC_TypeDef * **RMCx**,
FunctionalState **NewState**)
- ◆ void RMC_SetSignalRxMethod(TSB_RMC_TypeDef * **RMCx**,
RMC_RxMethod **Method**)
- ◆ void RMC_SetRxTrg(TSB_RMC_TypeDef * **RMCx**, uint8_t **LowWidth**,
uint8_t **MaxDataBitCycle**)
- ◆ void RMC_SetThreshold(TSB_RMC_TypeDef * **RMCx**, uint8_t **LargerThreshold**,
uint8_t **SmallerThreshold**)
- ◆ void RMC_SetInputSignalReversed(TSB_RMC_TypeDef * **RMCx**,
FunctionalState **NewState**)
- ◆ void RMC_SetNoiseCancellation(TSB_RMC_TypeDef * **RMCx**,
uint8_t **NoiseCancellationTime**)
- ◆ RMC_INTFactor RMC_GetINTFactor(TSB_RMC_TypeDef * **RMCx**)
- ◆ RMC_LeaderDetection RMC_GetLeader(TSB_RMC_TypeDef * **RMCx**)
- ◆ void RMC_SetRxEndBitNum(TSB_RMC_TypeDef * **RMCx**,
RMC_RxEndBitsReg **Reg_x**, uint8_t **BitNum**)
- ◆ void RMC_SetSrcClk(TSB_RMC_TypeDef * **RMCx**, RMC_SrcClk **Clk**)

9.2.2 Detailed Description

Functions listed above can be divided into three parts:

- 1) Reset and set each RMC channel are handled by RMC_Enable(), RMC_Disable(), RMC_Init() and RMC_SetRxCtrl().

- 2) RMC basic function are handled by RMC_SetLeaderDetection(), SetFallingEdgeINT(), RMC_SetSignalRxMethod(), RMC_SetRxTrg(), RMC_SetThreshold(), RMC_SetInputSignalReversed(), RMC_SetNoiseCancellation(), RMC_SetRxEndBitNum() and RMC_SetSrcClk().
- 3) RMC_GetINTFactor(), RMC_GetLeader() and RMC_GetRxData() to get the receive status and save the received data from buffer.

9.2.3 Function Documentation

Note: In all of the following APIs, parameter “TSB_RMC_TypeDef * **RMCx**” should be **TSB_RMC0**

9.2.3.1 RMC_Enable

Enable the specified RMC channel.

Prototype:

void
RMC_Enable(TSB_RMC_TypeDef * **RMCx**)

Parameters:

RMCx is the specified RMC channel.

Description:

This function will enable the specified RMC channel selected by **RMCx**.
Set the RMCEN<RMCEN>bit.

Return:

None

9.2.3.2 RMC_Disable

Disable the function of specified RMC channel.

Prototype:

void
RMC_Disable(TSB_RMC_TypeDef * **RMCx**)

Parameters:

RMCx is the specified RMC channel.

Description:

This function will disable the specified RMC channel selected by **RMCx**.
Clear the RMCEN<RMCEN>bit.

Return:

None

9.2.3.3 RMC_Init

RMC registers initial.

Prototype:

void
RMC_Init(TSB_RMC_TypeDef * **RMCx**, RMC_InitTypeDef * **RMC_InitStruct**)

Parameters:

RMCx is the specified RMC channel.

RMC_InitStruct : The structure containing the basic RMC configuration.
(For details, please refer to section "Data Structure Description")

Description:

This function will initialize the specified RMC channel selected by **RMCx**.

Return:

None

9.2.3.4 RMC_SetRxCtrl

Enable or disable reception of the specified RMC channel.

Prototype:

```
void  
RMC_SetRxCtrl(TSB_RMC_TypeDef * RMCx, FunctionalState NewState)
```

Parameters:

RMCx is the specified RMC channel.

NewState is the new state for reception of RMC.
This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:

This function will enable or disable reception of the specified RMC channel selected by **RMCx**.
This function handles the RMCREN<RMCREN> bit.

Return:

None

9.2.3.5 RMC_GetRxData

Get the received data from the specified RMC channel.

Prototype:

```
RMC_RxDataTypeDef  
RMC_GetRxData(TSB_RMC_TypeDef * RMCx)
```

Parameters:

RMCx is the specified RMC channel.

Description:

Get the received data from the specified RMC channel which selected by **RMCx**.
This function reads the data from the RMCRCBUF<0-71> and RMCRCSTAT<RMCRCNUM0-6> bits.

Return:

RMC_RxDataDef: Structure to read data from the RMC receive buffer.
(Refer to "Data Structure Description" for details).

9.2.3.6 RMC_SetLeaderDetection

Configure the RMC receive control register of leader detection for the specified RMC channel.

Prototype:

```
void  
RMC_SetLeaderDetection(TSB_RMC_TypeDef * RMCx,  
                       RMC_LeaderParameterTypeDef LeaderPara)
```

Parameters:

RMCx is the specified RMC channel.

LeaderPara: The structure containing basic RMC leader detection configuration.

Data Fields:

FunctionalState **LeaderDetectionState**: ENABLE or DISABLE the leader detection. This parameter can be one of the following values:

ENABLE or **DISABLE**

uint8_t **MaxCycle**: Set <RMCLCMAX7:0> to specify a maximum cycle of leader detection. Calculating-formula of the maximum cycle:
 $RMCLCMAX \times 4 / fs[s]$.

RMC detects the first cycle as a leader if it is within the maximum cycle.

uint8_t **MinCycle**: Set <RMCLCMIN7:0> to specify a minimum cycle of leader detection. Calculating-formula of the minimum cycle: $RMCLCMIN \times 4 / fs[s]$.
RMC detects the first cycle as a leader if it exceeds the minimum cycle.

uint8_t **MaxLowWidth**: Set <RMCLLMAX7:0> to specify a maximum low width of leader detection. Calculating-formula of the maximum low width: $RMCLLMAX \times 4 / fs[s]$. RMC detects the first cycle as a leader if its low width is within the maximum low width.

uint8_t **MinLowWidth**: Set <RMCLLMIN7:0> to specify a minimum low width of leader detection. Calculating-formula of the minimum low width: $RMCLLMIN \times 4 / fs[s]$. RMC detects the first cycle as a leader if its low width exceeds the minimum low width. If $RMCR2<RMCLD> = 1$, a value less than the specified is determined as data.

FunctionalState **LeaderINTState**: ENABLE or DISABLE generation of a leader

detection interrupt by detecting a leader. This parameter can be one of the following values: **ENABLE** or **DISABLE**

Description:

This function will set the RMC leader detection configuration for specified RMC channel which selected by **RMCx**.

This function handles the $RMCR1$ register and $RMCR2<RMCLIE>$ <RMCLD> two bits.

See MCU datasheet for detail.

Return:

None

9.2.3.7 RMC_SetFallingEdgeINT

Enable or disable to generate a remote control input falling edge interrupt.

Prototype:

```
void  
RMC_SetFallingEdgeINT(TSB_RMC_TypeDef * RMCx,  
                      FunctionalState NewState)
```

Parameters:

RMCx is the specified RMC channel.

NewState: New state for generation of a remote control input falling edge interrupt.

This parameter can be one of the following values:

ENABLE or **DISABLE**

Description:

When **NewState** is **ENABLE**, this function will enable generation of a remote control input falling edge interrupt for specified RMC channel which selected by **RMCx**, and disable generation of a remote control input falling edge interrupt when **NewState** is **DISABLE**.

This function handles the RMCRCR2<RMCEIEN> bit.

Return:

None

9.2.3.8 RMC_SetSignalRxMethod

Select the method of receiving a remote control signal.

Prototype:

```
void  
RMC_SetSignalRxMethod(TSB_RMC_TypeDef * RMCx,  
                     RMC_RxMethod Method)
```

Parameters:

RMCx is the specified RMC channel.

Method: Select the RMC receive method, which can be one of:

- **RMC_RX_IN_CYCLE_METHOD**: Receive a remote control signal in cycle method.
- **RMC_RX_IN_PHASE_METHOD**: Receive a remote control signal in phase method.

Description:

This function will set receiving method of remote control signal. Two methods can be selected by this function, cycle method or phase method.

This function handles the RMCRCR2<RMCPHM> bit.

Return:

None

9.2.3.9 RMC_SetRxTrg

Set the parameters that trigger reception completion and interrupt generation for the specified RMC channel.

Prototype:

```
void  
RMC_SetRxTrg(TSB_RMC_TypeDef * RMCx,  
             uint8_t LowWidth,  
             uint8_t MaxDataBitCycle)
```

Parameters:

RMCx is the specified RMC channel.

LowWidth: Excess low width that triggers reception completion and interrupt generation.

MaxDataBitCycle: Maximum data bit cycle that triggers reception completion and interrupt generation.

Description:

This function will set the trigger for specified RMC channel.

Set **LowWidth** to RMCRCR2<RMCLL7:0> specifies an excess low width. If an excess low width is detected, reception is completed and an interrupt is generated. The low width is not detected if <RMCLL7:0> = 1111111b.

Calculating formula of an excess low width: $RMCLLx1/fs[s]$

Set **MaxDataBitCycle** to RMCRCR2<RMCDMAX7:0> specifies a threshold for detecting a maximum data bit cycle. It is detected when a data bit cycle exceeds the threshold. It is not detected when <RMCDMAX7:0> = 1111111b.

Calculating-formula of the threshold: $RMCDMAX \times 1/fs[s]$.

This function handles the RMCRCR2<RMCLL0-7> <RMCDMA0-7> bits.

Return:

None

9.2.3.10 RMC_SetThreshold

Set the parameters of threshold in a phase method for the specified RMC channel.

Prototype:

```
void  
RMC_SetThreshold(TSB_RMC_TypeDef * RMCx,  
                uint8_t LargerThreshold,  
                uint8_t SmallerThreshold)
```

Parameters:

RMCx is the specified RMC channel.

LargerThreshold: Specifies a larger threshold (within a range of 1.5T and 2T) to determine a pattern of remote control signal in a phase method. If the measured cycle exceeds the threshold, the bit is determined as "10". If not, the bit is determined as "01". Calculating formula of the threshold:

$RMCDATHx1/fs[s]$.

The LargerThreshold should less than 0x80.

SmallerThreshold: Specifies two kinds of thresholds: a threshold to determine whether a data bit is 0 or 1; a smaller threshold (within a range of 1T and 1.5T) to determine a pattern of remote control signal in a phase method.

As for the determination of data bit, if the measured cycle exceeds the threshold, the bit is determined as "1". If not, the bit is determined as "0".

Calculating-formula of the threshold: $RMCDATLx1/fs[s]$.

As for the determination of a remote control signal pattern in a phase method, if

the measured cycle exceeds the threshold, the bit is determined as "01". If not, the bit is determined as "00". Calculating formula of the threshold to determine 0 or 1: $RMCDATLx1/fs[s]$.

This function handles the $RMCR3<RMCDATH0-6> <RMCDATL0-6>$ bits.
The **SmallerThreshold** should be less than 0x80.

Description:

This function will set the parameters for thresholds in a phase method for the specified RMC channel which is selected by **RMCx**, to determine a signal pattern in phase mode and determine 0 or 1/ smaller threshold to determine a signal pattern in a phase method.

The thresholds settings are enabled only in phase method, when $<RMCPHM>$ is "1".

Return:

None

9.2.3.11 RMC_SetInputSignalReversed

Enable or disable of reversing input signal for the specified RMC channel.

Prototype:

```
void  
RMC_SetInputSignalReversed(TSB_RMC_TypeDef * RMCx,  
                           FunctionalState NewState)
```

Parameters:

RMCx is the specified RMC channel.

NewState is the new state of reversing input signal.

This parameter can be one of the following values:

ENABLE or **DISABLE**

Description:

When **NewState** is **ENABLE**, this function will enable of reversing input signal for a specified RMC channel which is selected by **RMCx**, and disable reversing input signal when **NewState** is **DISABLE**.

This function handles the $RMCR4<RMCPO>$ bit.

Return:

None

9.2.3.12 RMC_SetNoiseCancellation

Set the noise cancellation time for the specified RMC channel.

Prototype:

```
void  
RMC_SetNoiseCancellation(TSB_RMC_TypeDef * RMCx,  
                          uint8_t NoiseCancellationTime)
```

Parameters:

RMCx is the specified RMC channel.

NoiseCancellationTime: The time for Noise cancellation, which should be less than 0x10.

Description:

Specifies time that noises is cancelled by a noise canceller.

If <RMCNC3:0> = 0000b, noises are not cancelled.

Calculating formula of noise cancellation time: $RMCNC \times 1/fs[s]$.

This function handles the RMCRCR4<RMCNC0-RMCNC3> bits.

Return:

None

9.2.3.13 RMC_GetINTFactor

Get the interrupt factor for the specified RMC channel.

Prototype:

RMC_INTFactor

RMC_GetINTFactor(TSB_RMC_TypeDef * **RMCx**)

Parameters:

RMCx is the specified RMC channel.

Description:

This function will get the interrupt factor for the specified RMC channel which selected by **RMCx**. User can get the RMC receive interrupt status from this function.

This info is updated every time an interrupt is generated.

This function reads interrupt factor from RMCRCR4<RMCRLIF><RMCLOIF><RMCDMAX><RMCEDIF> bits.

Return:

RMC_INTFactor: Interrupt factor structure.

(Refer to "Data Structure Description" for details).

9.2.3.14 RMC_GetLeader

Get the leader detection result for the specified RMC channel.

Prototype:

RMC_LeaderDetection

RMC_GetLeader(TSB_RMC_TypeDef * **RMCx**)

Parameters:

RMCx is the specified RMC channel.

Description:

This function will get the leader detection result for the specified RMC channel which selected by **RMCx**.

This info is updated every time an interrupt is generated.

This function reads the leader detection status from the RMCRCR4<RMCRLDR> bits.

Return:

RMC_LeaderDetection: leader detection result, which can be one of:

RMC_LEADER_DETECTED: leader detected.

RMC_NO_LEADER: no leader detected.

9.2.3.15 RMC_SetRxEndBitNum

Specifies that the number of receive data bit.

Prototype:

```
void  
RMC_SetRxEndBitNum(TSB_RMC_TypeDef * RMCx,  
                   RMC_RxEndBitsReg Reg_x,  
                   uint8_t BitNum)
```

Parameters:

RMCx is the specified RMC channel.

Reg_x: Select the set register, which can be one of:

- **RMC_RX_END_BITS_REG_1**: RMCxEND1 register
- **RMC_RX_END_BITS_REG_2**: RMCxEND2 register
- **RMC_RX_END_BITS_REG_3**: RMCxEND3 register

BitNum: Specifies that the number of receive data bit.

Description:

This function set the number of received data bit for the specified RMC channel, which selected by **RMCx**.

This function sets the number of received data bit to the RMCxEND1, RMCxEND2, and RMCxEND3 registers.

Return:

None

9.2.3.16 RMC_SetSrcClk

Specifies that the sampling clock.

Prototype:

```
void  
RMC_SetSrcClk(TSB_RMC_TypeDef * RMCx,  
              RMC_SrcClk Clk)
```

Parameters:

RMCx is the specified RMC channel.

Clk: RMC sampling clock, which can be one of:

- **RMC_CLK_LOW_FREQUENCY**: The Low Frequency Clock(32KHz)
- **RMC_CLK_TB1OUT**: Timer output (TB1OUT).

Description:

This function specifies that the sampling clock for the specified RMC channel, which selected by **RMCx**.

This function sets the sampling clock type to the RMCxFSSEL <RMCCLK> bits.

.

Return:

None

9.2.4 Data Structure Description

9.2.4.1 RMC_RxDataDef

Data Fields:

uint8

RxDataBits: The number of received data bit.

uint32_t

RxBuf1: Received buffer 1, which reads 4 bytes data from <MCRBUF31:0>.

uint32_t

RxBuf2: Received buffer 2, which reads 4 bytes data from <MCRBUF63:32>.

uint8_t

RxBuf3: Received buffer 3, which reads 1 byte data from <MCRBUF71:64>

9.2.4.2 RMC_LeaderParameterTypeDef

Data Fields:

FunctionalState

LeaderDetectionState: ENABLE or DISABLE the leader detection.
Parameter can be one of the following values:
ENABLE or **DISABLE**

uint8_t

MaxCycle: Specifies a maximum cycle of leader detection.

uint8_t

MinCycle: Specifies a minimum cycle of leader detection.

uint8_t

MaxLowWidth: Specifies a maximum low width of leader detection.

uint8_t

MinLowWidth: Specifies a minimum low width of leader detection.

FunctionalState

LeaderINTState: ENABLE or DISABLE generation of a leader detection interrupt by detecting a leader.
Parameter can be one of the following values:
ENABLE or **DISABLE**

9.2.4.3 RMC_InitTypeDef

Data Fields:

RMC_LeaderParameterTypeDef

LeaderPara: Parameters to configure leader detection.

FunctionalState

FallingEdgeINTState: The status of enable or disable the input falling edge interrupts.
This parameter can be one of the following values:
ENABLE or **DISABLE**

RMC_RxMethod

SignalRxMethod: Which method of receiving a remote control signal.
This parameter can be one of the following values:
RMC_RX_IN_CYCLE_METHOD or
RMC_RX_IN_PHASE_METHOD

FunctionalState

InputSignalReversedState: The status of enable or disable of reversing input signal.
This parameter can be one of the following values:
ENABLE or **DISABLE**

uint8_t

NoiseCancellationTime: Noise cancellation time.
The NoiseCancellationTime should less than 0x10.

uint8_t

LowWidth: Excess low width that triggers reception completion and interrupt generation.

uint8_t

MaxDataBitCycle: Maximum data bit cycle that triggers reception completion and interrupt generation.

uint8_t

LargerThreshold: Larger threshold to determine a signal pattern in a phase method.
The LargerThreshold should less than 0x80.

uint8_t

SmallerThreshold: Smaller threshold to determine a signal pattern in a phase method.
The SmallerThreshold should less than 0x80.

9.2.4.4 RMC_INTFactor

Data Fields:

uint32_t

All: Data.

Bit

uint32_t

Reserved : 12 Reserved

uint32_t

InputFallingEdge : 1 RMC input falling edge interrupt factor

uint32_t

MaxDataBitCycle : 1 Maximum data bit cycle interrupt factor

uint32_t

LowWidthDetection : 1 Low width detection interrupt factor

uint32_t

LeaderDetection : 1 Leader detection interrupt factor

10. RTC

10.1 Overview

The Real Time Clock (RTC) in the TMPM384 has such functions as follow:

- Clock (hour, minute and second)
- Calendar (month, week, date and leap year)
- Selectable 12 (am/ pm) and 24 hour display
- Time adjustment +/- 30 seconds (by software)
- Alarm (alarm output)
- Alarm interrupt

The RTC driver APIs provide a set of functions to configure RTC clock and alarm, including such common parameters as year, leap year, month, date, day, hour, hour mode, minute and second and so on.

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm384_rtc.c, with /Libraries/TX03_Periph_Driver/inc/tmpm384_rtc.h containing the macros, data types, structures and API definitions for use by applications.

10.2 API Functions

10.2.1 Function List

- ◆ void RTC_SetSec(uint8_t **Sec**);
- ◆ uint8_t RTC_GetSec(void);
- ◆ void RTC_SetMin(RTC_FuncMode **NewMode**, uint8_t **Min**);
- ◆ uint8_t RTC_GetMin(RTC_FuncMode **NewMode**);
- ◆ uint8_t RTC_GetAMPm(RTC_FuncMode **NewMode**);
- ◆ void RTC_SetHour24(RTC_FuncMode **NewMode**, uint8_t **Hour**);
- ◆ void RTC_SetHour12(RTC_FuncMode **NewMode**, uint8_t **Hour**, uint8_t **AmPm**);
- ◆ uint8_t RTC_GetHour(RTC_FuncMode **NewMode**);
- ◆ void RTC_SetDay(RTC_FuncMode **NewMode**, uint8_t **Day**);
- ◆ uint8_t RTC_GetDay(RTC_FuncMode **NewMode**);
- ◆ void RTC_SetDate(RTC_FuncMode **NewMode**, uint8_t **Date**);
- ◆ uint8_t RTC_GetDate(RTC_FuncMode **NewMode**);
- ◆ void RTC_SetMonth(uint8_t **Month**);
- ◆ uint8_t RTC_GetMonth(void);
- ◆ void RTC_SetYear(uint8_t **Year**);
- ◆ uint8_t RTC_GetYear(void);
- ◆ void RTC_SetHourMode(uint8_t **HourMode**);
- ◆ uint8_t RTC_GetHourMode(void);
- ◆ void RTC_SetLeapYear(uint8_t **LeapYear**);
- ◆ uint8_t RTC_GetLeapYear(void);
- ◆ void RTC_SetTimeAdjustReq(void);
- ◆ RTC_ReqState RTC_GetTimeAdjustReq(void);
- ◆ void RTC_EnableClock(void);
- ◆ void RTC_DisableClock(void);
- ◆ void RTC_EnableAlarm(void);
- ◆ void RTC_DisableAlarm(void);
- ◆ void RTC_SetRTCIINT(FunctionalState **NewState**);
- ◆ void RTC_SetAlarmOutput(uint8_t **Output**);

- ◆ void RTC_ResetClockSec(void);
- ◆ RTC_ReqState RTC_GetResetClockSecReq(void);
- ◆ void RTC_ResetAlarm(void);
- ◆ void RTC_SetDateValue(RTC_DateTypeDef * **DateStruct**);
- ◆ void RTC_GetDateValue(RTC_DateTypeDef * **DateStruct**);
- ◆ void RTC_SetTimeValue(RTC_TimeTypeDef * **TimeStruct**);
- ◆ void RTC_GetTimeValue(RTC_TimeTypeDef * **TimeStruct**);
- ◆ void RTC_SetClockValue(RTC_DateTypeDef * **DateStruct**, RTC_TimeTypeDef * **TimeStruct**);
- ◆ void RTC_GetClockValue(RTC_DateTypeDef * **DateStruct**, RTC_TimeTypeDef * **TimeStruct**);
- ◆ void RTC_SetAlarmValue(RTC_AlarmTypeDef * **AlarmStruct**);
- ◆ void RTC_GetAlarmValue(RTC_AlarmTypeDef * **AlarmStruct**);

10.2.2 Detailed Description

Functions listed above can be divided into five parts:

- 1) Configure the common functions of RTC date are handled by RTC_SetDay(), RTC_GetDay(), RTC_SetDate(), RTC_GetDate(), RTC_SetMonth(), RTC_GetMonth(), RTC_SetYear(), RTC_GetYear(), RTC_SetLeapYear(), RTC_GetLeapYear(), RTC_SetDateValue(), RTC_GetDateValue(),
- 2) Configure the common functions of RTC time are handled by RTC_SetSec(), RTC_GetSec(), RTC_SetMin(), RTC_GetMin(), RTC_SetHour24(), RTC_SetHour12(), RTC_GetHour(), RTC_SetHourMode(), RTC_GetHourMode, RTC_GetAMPM(), RTC_SetTimeValue(), RTC_GetTimeValue().
- 3) RTC_EnableClock(), RTC_DisableClock(), RTC_SetTimeAdjustReq(), RTC_GetTimeAdjustReq(), RTC_ResetClockSec(), RTC_GetResetClockSec(), RTC_SetClockValue() and RTC_GetClockValue() handle for RTC clock function only.
- 4) RTC_EnableAlarm(), RTC_DisableAlarm(), RTC_ResetAlarm(), RTC_SetAlarmValue() and RTC_GetAlarmValue() handle for RTC alarm function only.
- 5) RTC_SetAlarmOutput() and RTC_SetRTCINT() handle other specified functions.

10.2.3 Function Documentation

10.2.3.1 RTC_SetSec

Set second value for RTC clock.

Prototype:

```
void  
RTC_SetSec(uint8_t Sec);
```

Parameters:

Sec: New second value, max is 59.

Description:

This function will set new second value for RTC clock. RTC register are updated synchronizing with the timing of INTRTC, so after calling this function, it should wait for RTC 1HZ interrupt occurs.

Return:

None.

10.2.3.2 RTC_GetSec

Get second value of RTC clock.

Prototype:

```
uint8_t  
RTC_GetSec(void);
```

Parameters:

None

Description:

This function will return second value of RTC clock.

Return:

Second value in the range:
0 ~ 59

10.2.3.3 RTC_SetMin

Set minute value for RTC clock or alarm.

Prototype:

```
void  
RTC_SetMin(RTC_FuncMode NewMode,  
            uint8_t Min);
```

Parameters:

NewMode: New mode of RTC, which can be set as:

- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

Min: New min value, max 59

Description:

This function will set new minute value for RTC clock when **NewMode** is **RTC_CLOCK_MODE**, and write new minute value for RTC alarm when **NewMode** is **RTC_ALARM_MODE**. RTC register are updated synchronizing with the timing of INTRTC, so after calling this function, it should wait for RTC 1HZ interrupt occurs.

Return:

None

10.2.3.4 RTC_GetMin

Get minute value of RTC clock or alarm.

Prototype:

```
uint8_t  
RTC_GetMin(RTC_FuncMode NewMode);
```

Parameters:

NewMode: New mode of RTC, which can be set as:

- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

Description:

This function will return minute value of RTC clock when **NewMode** is **RTC_CLOCK_MODE**, and return minute value of RTC alarm when **NewMode** is **RTC_ALARM_MODE**.

Return:

Minute value in the range:

0 ~ 59

10.2.3.5 RTC_GetAMPM

Get AM or PM state in the 12 Hour mode.

Prototype:

```
uint8_t  
RTC_GetAMPM(RTC_FuncMode NewMode);
```

Parameters:

NewMode: New mode of RTC, which can be set as:

- **RTC_CLOCK_MODE:** select clock function,
- **RTC_ALARM_MODE:** select alarm function.

Description:

This function will return AM or PM mode of RTC clock when **NewMode** is **RTC_CLOCK_MODE**, and return AM or PM mode of RTC alarm when **NewMode** is **RTC_ALARM_MODE**.

Return:

The mode of time:

RTC_AM_MODE: Time mode is AM.

RTC_PM_MODE: Time mode is PM.

10.2.3.6 RTC_SetHour24

Set hour value for RTC clock or alarm in the 24 Hour mode.

Prototype:

```
void  
RTC_SetHour24(RTC_FuncMode NewMode,  
              uint8_t Hour);
```

Parameters:

NewMode: New mode of RTC, which can be set as:

- **RTC_CLOCK_MODE:** select clock function,
- **RTC_ALARM_MODE:** select alarm function.

Hour: New hour value, max is 23.

Description:

This function will set new hour value for RTC clock when **NewMode** is **RTC_CLOCK_MODE**, and set new hour value for RTC alarm when **NewMode** is **RTC_ALARM_MODE**. RTC register are updated synchronizing with the timing of INTRTC, so after calling this function, it should wait for RTC 1HZ interrupt occurs.

* If hour mode is changed to 24H mode from 12H mode, **RTC_SetHour24()** should be called to rewrite the HOURR register.

Return:
None

10.2.3.7 RTC_SetHour12

Set hour value and AM/PM mode for RTC clock or alarm in the 12 Hour mode.

Prototype:
void
RTC_SetHour12(RTC_FuncMode **NewMode**,
uint8_t **Hour**,
uint8_t **AmPm**);

Parameters:

NewMode: New mode of RTC, which can be set as:

- **RTC_CLOCK_MODE:** select clock function,
- **RTC_ALARM_MODE:** select alarm function.

Hour: New hour value, max is 11.

AmPm: New time mode, which can be set as:

- **RTC_AM_MODE:** select AM mode for 12H mode,
- **RTC_PM_MODE:** select PM mode for 12H mode.

Description:

This function will set new hour value and AM/PM mode for RTC clock when **NewMode** is **RTC_CLOCK_MODE**, and set new hour value and AM/PM mode for RTC alarm when **NewMode** is **RTC_ALARM_MODE**. RTC register are updated synchronizing with the timing of INTRTC, so after calling this function, it should wait for RTC 1HZ interrupt occurs.

* If hour mode is changed to 12H mode from 24H mode, **RTC_SetHour12()** should be called to rewrite the HOURR register.

Return:
None

10.2.3.8 RTC_GetHour

Get hour value of RTC clock or alarm.

Prototype:
uint8_t
RTC_GetHour(RTC_FuncMode **NewMode**);

Parameters:

NewMode: New mode of RTC, which can be set as:

- **RTC_CLOCK_MODE:** select clock function,
- **RTC_ALARM_MODE:** select alarm function.

Description:

This function will return hour value of RTC clock when **NewMode** is **RTC_CLOCK_MODE**, and return hour value of RTC alarm when **NewMode** is **RTC_ALARM_MODE**.

Return:

In 24H mode, hour value in the range:

0 ~ 23

In 12H mode, hour value in the range:

0 ~ 11

10.2.3.9 RTC_SetDay

Set day value for RTC clock or alarm.

Prototype:

void

```
RTC_SetDay(RTC_FuncMode NewMode,  
           uint8_t Day);
```

Parameters:

NewMode: New mode of RTC, which can be set as:

- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

Day: New day value, which can be set as:

- **RTC_SUN**: Sunday.
- **RTC_MON**: Monday.
- **RTC_TUE**: Tuesday.
- **RTC_WED**: Wednesday.
- **RTC_THU**: Thursday.
- **RTC_FRI**: Friday.
- **RTC_SAT**: Saturday.

Description:

This function will set new day value for RTC clock when **NewMode** is **RTC_CLOCK_MODE**, and set new day value for RTC alarm when **NewMode** is **RTC_ALARM_MODE**. RTC register are updated synchronizing with the timing of INTRTC, so after calling this function, it should wait for RTC 1HZ interrupt occurs.

Return:

None

10.2.3.10 RTC_GetDay

Get day value of RTC clock or alarm.

Prototype:

uint8_t

```
RTC_GetDay(RTC_FuncMode NewMode);
```

Parameters:

NewMode: New mode of RTC, which can be set as:

- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

Description:

This function will return day value of RTC clock when **NewMode** is **RTC_CLOCK_MODE**, and return day value of RTC alarm when **NewMode** is **RTC_ALARM_MODE**.

Return:

Day value in the range:
0 ~ 6

10.2.3.11 RTC_SetDate

Set date value for RTC clock or alarm.

Prototype:

```
void  
RTC_SetDate(RTC_FuncMode NewMode,  
            uint8_t Date);
```

Parameters:

NewMode: New mode of RTC, which can be set as:

- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

Date: New date value, ranging from 1 to 31.

Description:

This function will set new date value for RTC clock when **NewMode** is **RTC_CLOCK_MODE**, and set new date value RTC alarm when **NewMode** is **RTC_ALARM_MODE**. RTC register are updated synchronizing with the timing of INTRTC, so after calling this function, it should wait for RTC 1HZ interrupt occurs.

Return:

None

10.2.3.12 RTC_GetDate

Get date value of RTC clock or alarm.

Prototype:

```
uint8_t  
RTC_GetDate(RTC_FuncMode NewMode);
```

Parameters:

NewMode: New mode of RTC, which can be set as:

- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

Description:

This function will return date value of RTC clock when **NewMode** is **RTC_CLOCK_MODE**, and return date value of RTC alarm when **NewMode** is **RTC_ALARM_MODE**.

Return:

Date value in the range:
1 ~ 31

10.2.3.13 RTC_SetMonth

Set month value for RTC clock.

Prototype:

```
void  
RTC_SetMonth(uint8_t Month);
```

Parameters:

Month: New month value, ranging from 1 to 12.

Description:

This function will set new month value for RTC clock. RTC register are updated synchronizing with the timing of INTRTC, so after calling this function, it should wait for RTC 1HZ interrupt occurs.

Return:

None

10.2.3.14 RTC_GetMonth

Get month value of RTC clock.

Prototype:

```
uint8_t  
RTC_GetMonth(void);
```

Parameters:

None

Description:

This function will return month value.

Return:

Month value in the range:
1 ~ 12

10.2.3.15 RTC_SetYear

Set year value for RTC clock.

Prototype:

```
void  
RTC_SetYear(uint8_t Year);
```

Parameters:

Year: New year value, max is 99.

Description:

This function will set new year value for RTC clock. RTC register are updated synchronizing with the timing of INTRTC, so after calling this function, it should wait for RTC 1HZ interrupt occurs.

Return:

None

10.2.3.16 RTC_GetYear

Get year value of RTC clock.

Prototype:

```
uint8_t  
RTC_GetYear(void);
```

Parameters:

None

Description:

This function will return year value.

Return:

Year value in the range:
0 ~ 99

10.2.3.17 RTC_SetHourMode

Select 24-hour clock or 12-hour clock.

Prototype:

```
void  
RTC_SetHourMode(uint8_t HourMode);
```

Parameters:

HourMode: New mode of hour, which can be set as:

- **RTC_12_HOUR_MODE** : Select 12H mode,
- **RTC_24_HOUR_MODE**.: Select 24H mode.

Description:

This function will select 24H mode when ***HourMode*** is **RTC_24_HOUR_MODE** and select 12H mode when ***HourMode*** is **RTC_12_HOUR_MODE**.

* Before call this function, **RTC_DisableClock()** function should be called firstly.
(See “RTC_DisableClock” for details)

Return:

None

10.2.3.18 RTC_GetHourMode

Get hour mode.

Prototype:

```
uint8_t  
RTC_GetHourMode(void);
```

Parameters:

None

Description:

This function will return hour mode.

Return:

Hour mode:

RTC_24_HOUR_MODE: Hour mode is 24H mode.

RTC_12_HOUR_MODE: Hour mode is 12H mode.

10.2.3.19 RTC_SetLeapYear

Set leap year state.

Prototype:

void

RTC_SetLeapYear(uint8_t *LeapYear*);

Parameters:

LeapYear. The state of leap year, which can be set as:

- **RTC_LEAP_YEAR_0:** Current year is a leap year.
- **RTC_LEAP_YEAR_1:** Current year is the year following a leap year.
- **RTC_LEAP_YEAR_2:** Current year is two years after a leap year.
- **RTC_LEAP_YEAR_3:** Current year is three years after a leap year.

Description:

This function will change leap year state. If *LeapYear* is **RTC_LEAP_YEAR_0**, current year is a leap year. If *LeapYear* is **RTC_LEAP_YEAR_1**, current year is the year following a leap year. If *LeapYear* is **RTC_LEAP_YEAR_2**, current year is two years after a leap year. If *LeapYear* is **RTC_LEAP_YEAR_3**, current year is three years after a leap year.

Return:

None

10.2.3.20 RTC_GetLeapYear

Get leap year state.

Prototype:

uint8_t

RTC_GetLeapYear(void);

Parameters:

None

Description:

This function will return leap year state.

Return:

The state of the leap year.

10.2.3.21 RTC_SetTimeAdjustReq

Set time adjustment + or – 30 seconds.

Prototype:

void

RTC_SetTimeAdjustReq(void);

Parameters:

None

Description:

This function will set time adjust seconds. The request is sampled when the sec counter counts up. If the time elapsed is between 0 and 29 seconds, the sec counter is cleared to "0". If the time elapsed is between 30 and 59 seconds, the min counter is carried and sec counter is cleared to "0".

Return:

None

10.2.3.22 RTC_GetTimeAdjustReq

Get time adjust request state.

Prototype:

RTC_ReqState

RTC_GetTimeAdjustReq(void);

Parameters:

None

Description:

This function will get the state of time adjust request. In order not to request repeatedly, it should be called after calling **RTC_SetTimeAdjustReq()** function.

Return:

The state of time adjustment:

RTC_NO_REQ : No adjust request.

RTC_REQ: Adjust request.

10.2.3.23 RTC_EnableClock

Enable RTC clock function.

Prototype:

void

RTC_EnableClock(void);

Parameters:

None

Description:

This function will enable clock function.

Return:

None

10.2.3.24 RTC_DisableClock

Disable RTC clock function.

Prototype:

void
RTC_DisableClock(void);

Parameters:

None

Description:

This function will disable clock function.

Return:

None

10.2.3.25 RTC_EnableAlarm

Enable RTC alarm function.

Prototype:

void
RTC_EnableAlarm(void);

Parameters:

None

Description:

This function will enable alarm function.

Return:

None

10.2.3.26 RTC_DisableAlarm

Disable RTC alarm function.

Prototype:

void
RTC_DisableAlarm(void);

Parameters:

None

Description:

This function will disable alarm function.

Return:

None

10.2.3.27 RTC_SetRTCINT

Enable or disable INTRTC.

Prototype:

void
RTC_SetRTCINT(FunctionalState **NewState**);

Parameters:

NewState: New state of INT RTC.

- **ENABLE:** Enable INTRTC.
- **DISABLE:** Disable INTRTC.

Description:

This function will enable RTCINT when **NewState** is **ENABLE**, and disable RTCINT when **NewState** is **DISABLE**.

Return:

None

10.2.3.28 RTC_SetAlarmOutput

Set output signals from ALARM pin.

Prototype:

```
void  
RTC_SetAlarmOutput(uint8_t Output);
```

Parameters:

Output: Set ALARM pin output, which can be set as:

- **RTC_LOW_LEVEL:** "0" pulse
- **RTC_PULSE_1_HZ:** 1Hz cycle "0" pulse
- **RTC_PULSE_16_HZ:** 16Hz cycle "0" pulse
- **RTC_PULSE_2_HZ:** 2Hz cycle "0" pulse
- **RTC_PULSE_4_HZ:** 4Hz cycle "0" pulse
- **RTC_PULSE_8_HZ:** 8Hz cycle "0" pulse

Description:

This function will set output signal from ALARM pin. If **Output** is **RTC_LOW_LEVEL**, Alarm pin output is "0" pulse when the alarm register corresponds with the clock. If **Output** is **RTC_PULSE_n*_HZ**, Alarm pin output is n*Hz cycle "0" pulse. (n can be one of 1,2,4,8,16)

Return:

None

10.2.3.29 RTC_ResetClockSec

Reset RTC clock second counter.

Prototype:

```
void  
RTC_ResetClockSec(void);
```

Parameters:

None

Description:

This function will reset sec counter.

Return:

None

10.2.3.30 RTC_GetResetClockSecReq

Get reset RTC clock second counter request state.

Prototype:

RTC_ReqState

RTC_GetResetClockSecReq(void);

Parameters:

None

Description:

Get request state for reset RTC clock second counter. The request is sampled using low-speed clock. In order to wait the clock stability, it should be called after calling **RTC_ResetClockSec()** function.

Return:

The state of reset clock request:

RTC_NO_REQ: No reset clock request.

RTC_REQ: Reset clock request.

10.2.3.31 RTC_ResetAlarm

Reset RTC alarm.

Prototype:

void

RTC_ResetAlarm(void);

Parameters:

None

Description:

This function will reset alarm.

Reset alarm registers, the related parameters will be set as follows.

Minute: 00, Hour: 00, Date: 01, Day of the week: Sunday

Return:

None

10.2.3.32 RTC_SetDateValue

Set the RTC clock date.

Prototype:

void

RTC_SetDateValue(RTC_DateTypeDef * **DateStruct**);

Parameters:

DateStruct: The structure containing basic date configuration including leap year state, year, month, date and day. (Refer to "Data structure Description" for details)

Description:

This function will set RTC clock date, including leap year, year, month, date and day. **RTC_SetLeapYear()**, **RTC_SetYear()**, **RTC_SetMonth()**, **RTC_SetDate()** and **RTC_Setday()** will be called by it.

Return:
None

10.2.3.33 RTC_GetDateValue

Get the RTC clock date.

Prototype:
void
RTC_GetDateValue(RTC_DateTypeDef * *DateStruct*);

Parameters:

DateStruct. The structure containing basic date configuration. (Refer to “Data structure Description” for details)

Description:
This function will get RTC clock date, including leap year, year, month, date and day. **RTC_GetLeapYear()**, **RTC_GetYear()**, **RTC_GetMonth()**, **RTC_GetDate()** and **RTC_Getday()** will be called by it.

Return:
None

10.2.3.34 RTC_SetTimeValue

Set the RTC clock time.

Prototype:
void
RTC_SetTimeValue(RTC_TimeTypeDef * *TimeStruct*);

Parameters:

TimeStruct. The structure containing basic time configuration including hour mode, hour, AM/PM mode in 12H mode, minute and second. (Refer to “Data structure Description” for details)

Description:
This function will set RTC clock time, including hour mode, hour, AM/PM mode in 12H mode, minute and second. **RTC_SetHourMode()**, **RTC_SetHour12()**, **RTC_SetHour24()**, **RTC_SetMin()** and **RTC_SetSec()** will be called by it.

Return:
None

10.2.3.35 RTC_GetTimeValue

Get the RTC time.

Prototype:

```
void  
RTC_GetTimeValue(RTC_TimeTypeDef * TimeStruct);
```

Parameters:

TimeStruct: The structure containing basic Time configuration. (Refer to “Data structure Description” for details)

Description:

This function will Get RTC clock time, including hour mode, hour, AM/PM mode in 12H mode, minute and second. **RTC_GetHourMode()**, **RTC_GetHour()**, **RTC_GetAMPM()**, **RTC_GetMin()** and **RTC_GetSec()** will be called by it.

Return:

None

10.2.3.36 RTC_SetClockValue

Set the RTC clock date and time.

Prototype:

```
void  
RTC_SetClockValue(RTC_DateTypeDef * DateStruct,  
                  RTC_TimeTypeDef * TimeStruct);
```

Parameters:

DateStruct: The structure containing basic Date configuration including leap year state, year, month, date and day.

TimeStruct: The structure containing basic Time configuration including hour mode, hour, AM/PM mode in 12H mode, minute and second. (Refer to “Data structure Description” for details)

Description:

This function will set RTC clock date and time, including leap year, year, month, date, day, hour mode, hour, AM/PM mode in 12H mode, minute and second.

RTC_SetLeapYear(), **RTC_SetYear()**, **RTC_SetMonth()**, **RTC_SetDate()**, **RTC_SetDay()**, **RTC_SetHourMode()**, **RTC_SetHour24()**, **RTC_SetHour12()**, **RTC_SetMin()** and **RTC_SetSec()** will be called by it.

Return:

None

10.2.3.37 RTC_GetClockValue

Get the RTC clock date and time.

Prototype:

```
void  
RTC_GetClockValue(RTC_DateTypeDef * DateStruct,  
                  RTC_TimeTypeDef * TimeStruct);
```

Parameters:

DateStruct: The structure containing basic Date configuration including leap year state, year, month, date and day.

TimeStruct: The structure containing basic Time configuration including hour mode, hour, AM/PM mode in 12H mode, minute and second. (Refer to “Data structure Description” for details)

Description:

This function will get RTC clock date and time, including leap year, year, month, date, day, hour mode, hour, AM/PM mode in 12H mode, minute and second.

RTC_GetLeapYear(), RTC_GetYear(), RTC_GetMonth(), RTC_GetDate(), RTC_GetDay(), RTC_GetHourMode(), RTC_GetHour(), RTC_GetAMPM(), RTC_GetMin() and RTC_GetSec() will be called by it.

Return:

None

10.2.3.38 RTC_SetAlarmValue

Set the RTC alarm date and time.

Prototype:

void
RTC_SetAlarmValue(RTC_AlarmTypeDef * **AlarmStruct**);

Parameters:

AlarmStruct: The structure containing basic alarm configuration including date, day, hour, AM/PM mode in 12H mode and minute. (Refer to “Data structure Description” for details)

Description:

This function will set RTC alarm date and time, including date, day, hour, AM/PM mode in 12H mode and minute. **RTC_SetDate(), RTC_SetDay(), RTC_SetHour12(), RTC_SetHour24() and RTC_SetMin()** will be called by it.

Return:

None

10.2.3.39 RTC_GetAlarmValue

Get the RTC alarm date and time.

Prototype:

void
RTC_GetAlarmValue(RTC_AlarmTypeDef * **AlarmStruct**);

Parameters:

AlarmStruct: The structure containing basic alarm configuration including date, day, hour, AM/PM mode in 12H mode and minute. (Refer to “Data structure Description” for details)

Description:

This function will get RTC alarm date and time, including date, day, hour, AM/PM mode in 12H mode and minute. **RTC_GetDate(), RTC_GetDay(), RTC_GetHour() , RTC_GetAMPM() and RTC_GetMin()** will be called by it.

Return:

None

10.2.4 Data Structure Description

10.2.4.1 RTC_DateTypeDef

Data Fields:

uint8_t

LeapYear set leap year state, which can be set as:

- **RTC_LEAP_YEAR_0**: Current year is a leap year.
- **RTC_LEAP_YEAR_1**: Current year is the year following a leap year.
- **RTC_LEAP_YEAR_2**: Current year is two years after a leap year.
- **RTC_LEAP_YEAR_3**: Current year is three years after a leap year

uint8_t

Year new year value, max is 99.

uint8_t

Month new month value, ranging from 1 to 12.

uint8_t

Date new date value, ranging from 1 to 31.

uint8_t

Day new day value, which can be set as:

- **RTC_SUN**: Sunday.
- **RTC_MON**: Monday.
- **RTC_TUE**: Tuesday.
- **RTC_WED**: Wednesday.
- **RTC_THU**: Thursday.
- **RTC_FRI**: Friday.
- **RTC_SAT**: Saturday.

10.2.4.2 RTC_TimeTypeDef

Data Fields:

uint8_t

HourMode select 24H mode or 12H mode, which can be set as:

- **RTC_12_HOUR_MODE**: Hour mode is 12H mode
- **RTC_24_HOUR_MODE**: Hour mode is 24H mode

uint8_t

Hour new hour value, max value is 23 in 24H mode or 11 in 12H mode.

uint8_t

AmPm select AM/PM mode for 12H mode, which can be set as:

- **RTC_AM_MODE**: select AM mode for 12H mode,
- **RTC_PM_MODE**: select PM mode for 12H mode.
- **RTC_AMPM_INVALID**: when hour mode is 24H mode.

uint8_t

Min new minute value, max is 59.

uint8_t

Sec new second value, max is 59.

10.2.4.3 RTC_AlarmTypeDef

Data Fields:

uint8_t

Date new date value of RTC alarm, ranging from 1 to 31.

uint8_t

Day new day value of RTC alarm, which can be set as:

- **RTC_SUN**: Sunday.
- **RTC_MON**: Monday.
- **RTC_TUE**: Tuesday.
- **RTC_WED**: Wednesday.
- **RTC_THU**: Thursday.
- **RTC_FRI**: Friday.
- **RTC_SAT**: Saturday.

uint8_t

Hour new hour value of RTC alarm, max value is 23 in 24H mode, max value is 11 in 12H mode.

uint8_t

AmPm select AM/PM mode for 12H mode, which can be set as:

- **RTC_AM_MODE**: select AM mode for 12H mode,
- **RTC_PM_MODE**: select PM mode for 12H mode.
- **RTC_AMPM_INVALID**: when hour mode is 24H mode.

uint8_t

Min new minute value of RTC alarm, max is 59.

11. SBI

11.1 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/tmpm384_sbi.c, with /Libraries/TX03_Periph_Driver/inc/tmpm384_sbi.h containing the macros, data types, structures and API definitions for use by applications.

11.2 API Functions

11.2.1 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_Generatel2Cstart(TSB_SBI_TypeDef* **SBIx**);
- ◆ void SBI_Generatel2Cstop(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**);

11.2.2 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 SBI channel is handled by SBI_ClearI2CINTReq(), SBI_Generatel2Cstart(), SBI_Generatel2Cstop(), 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.

11.2.3 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.

11.2.3.1 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

11.2.3.2 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

11.2.3.3 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

11.2.3.4 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

11.2.3.5 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

11.2.3.6 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

11.2.3.7 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

11.2.3.8 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

11.2.3.9 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

11.2.3.10 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 value of state of the SBI channel in I2C bus mode.

11.2.3.11 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

11.2.3.12 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

11.2.3.13 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

11.2.3.14 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 New state of free data 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

11.2.4 Data Structure Description

11.2.4.1 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.

11.2.4.2 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.

12. SSP

12.1 Overview

TOSHIBA TMPM384 contains 2 channels (SSP0, SSP1).

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/tmpm384_ssp.c, with /Libraries/TX03_Periph_Driver/inc/tmpm384_ssp.h containing the macros, data types, structures and API definitions for use by applications.

12.2 API Functions

12.2.1 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**)

12.2.2 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.

12.2.3 Function Documentation

Note: in all of the following APIs, parameter “TSB_SSP_TypeDef* **SSPx**” can be one of the following values: **SSP0** or **SSP1**

12.2.3.1 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

12.2.3.2 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

12.2.3.3 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

12.2.3.4 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

12.2.3.5 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

12.2.3.6 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

12.2.3.7 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

12.2.3.8 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

12.2.3.9 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

12.2.3.10 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

12.2.3.11 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

12.2.3.12 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

12.2.3.13 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

12.2.3.14 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.

12.2.3.15 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

12.2.3.16 SSP_SetINTConfig

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

Prototype:

```
void  
SSP_SetINTConfig(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

12.2.3.17 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.

12.2.3.18 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.

12.2.3.19 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.

12.2.3.20 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

12.2.3.21 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

12.2.4 Data Structure Description

12.2.4.1 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.

12.2.4.2 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.

13. TMRB

13.1 Overview

TOSHIBA TMPM384 contains 12 channels of multi-functional 16-bit timer/event counter (TMRB0 through TMRB11). 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:

- Pulse width measurement
- One-shot pulse generation from an external trigger pulse
- Time difference measurement

TMPM384 also has 4 channels of 16-bit multi-purpose timer (MPT), when being operated in timer mode, they are the same as common timer channels.

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/tmpm384_tmrb.c, with /Libraries/TX03_Periph_Driver/inc/tmpm384_tmrb.h containing the macros, data types, structures and API definitions for use by applications.

13.2 API Functions

13.2.1 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**, uint8_t **WriteRegMode**);
- ◆ void TMRB_SetExtStartTrg(TSB_TB_TypeDef * **TBx**, FunctionalState **NewState**, uint8_t **TrgMode**);
- ◆ void TMRB_SetClkInCoreHalt(TSB_TB_TypeDef * **TBx**, uint8_t **ClkState**);

13.2.2 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 () handle other specified functions.

13.2.3 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, TSB_TBA, TSB_TBB, TSB_TB_MPT0, TSB_TB_MPT1, TSB_TB_MPT2, TSB_TB_MPT3.

13.2.3.1 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**.
If channel is MPT, this function will also select MPT channel as timer mode.

Return:

None

13.2.3.2 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

13.2.3.3 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

13.2.3.4 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, leading timing value, trailing timing 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, trailing timing and leading timing duration for the specified TMRB channel selected by **TBx**.

Return:

None

13.2.3.5 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 TMRB2, TMRB5 and TMRB7 can be used as the capture trigger of other channels.

Note: **TMRB_CAPTURE_OUTPUT_EDGE** is only available in TMRB0 through TMRBB.

TMRB3~5: TB2OUT
TMRB6~7: TB5OUT
TMRB0~2: TB7OUT
TMRB9~B: TB8OUT

Return:

None

13.2.3.6 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.

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

13.2.3.7 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 leading timing value, Bit of **MatchTrailingTiming** Indicates if the up-counter matches with trailing timing 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 leading timing value is detected

MatchTrailingTiming(Bit1): a match with the trailing timing 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  
}
```

13.2.3.8 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 trailing timing are match.
- **TMRB_MASK_MATCH_LEADINGTIMING_INT**: Mask the interrupt the factor of which is that the value in up-counter and leading timing 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 trailing timing 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 leading timing 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

13.2.3.9 TMRB_ChangeLeadingTiming

Change the value of leading timing 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 leading timing, max. is 0xFFFF.

Description:

This function will specify the absolute value of leading timing for the specified TMRB. The actual interval of leading timing 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**.

13.2.3.10 TMRB_ChangeTrailingTiming

Change the value of trailing timing 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 trailing timing, max. is 0xFFFF.

Description:

This function will specify the absolute value of trailing timing for the specified TMRB. The actual interval of trailing timing 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.

13.2.3.11 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

13.2.3.12 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

13.2.3.13 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

13.2.3.14 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

13.2.3.15 TMRB_SetSyncMode

Enable or disable the synchronous mode of specified TMRB channel.

Prototype:

```
void  
TMRB_SetSyncMode(TSB_TB_TypeDef* TBx,  
                  FunctionalState NewState)
```

Parameters:

TBx can be

TSB_TB1, TSB_TB2, TSB_TB3, TSB_TB5, TSB_TB6, TSB_TB7, TSB_TB9, TSB_TBA, TSB_TBB.

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 TMRB1 through TMRB3, their start timing is synchronized with TMRB0. If the synchronous mode is enabled for TMRB5 through TMRB7, their start timing is synchronized with TMRB4. If the synchronous mode is enabled for TMRB9 through TMRBB, their start timing is synchronized with TMRB8.

Return:

None

Note:

TMRB1 through TMRB3, TMRB5 through TMRB7, TMRB9 through TMRBB must start counting by calling **TMRB_SetRunState()** before TMRB0, TMRB4 or TMRB8 start counting, so that start timing can be synchronized.

13.2.3.16 TMRB_SetDoubleBuf

Enable or disable double buffering for the specified TMRB channel and set the timing to write to timer register 0 and 1 when double buffer enabled.

Prototype:

```
void  
TMRB_SetDoubleBuf(TSB_TB_TypeDef* TBx,  
                  FunctionalState NewState,  
                  uint8_t WriteRegMode)
```

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.

WriteRegMode specifies timing to write to timer register 0 and 1 when double buffer enabled, which can be

- **TMRB_WRITE_REG_SEPARATE**: Timer register 0 and 1 can be written separately, even in case writing preparation is ready for only one register.
- **TMRB_WRITE_REG_SIMULTANEOUS**: In case both registers are not ready to be written, timer registers 0 and 1 can't be written.

Description:

The register TBxRG0 (**LeadingTiming**) and TBxRG1 (**TrailingTiming**) and their buffers are assigned to the same address. If double buffering is disabled, the same value is written to the registers and their buffers.

If double buffering is enabled, the value is only written to each register buffer. Therefore, to write an initial value to the registers, TBxRG0 (**LeadingTiming**) and TBxRG1 (**TrailingTiming**), the double buffering must be set to **DISABLE**. Then **ENABLE** double buffering and write the following data to the register, which can be loaded when the corresponding interrupt occurs automatically.

Return:

None

13.2.3.17 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

13.2.3.18 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

13.2.4 Data Structure Description

13.2.4.1 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.

uint32_t
TrailingTiming specifies the trailing timing 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 trailing timing, 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 leading timing value to be written into TBnRG0, max. 0xFFFF, and it can not be set larger than **TrailingTiming**.

13.2.4.2 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 trailing timing,
- **TMRB_FLIPFLOP_MATCH_LEADINGTIMING**, which means that the reversing flip-flop output will be triggered when the up-counter matches the leading timing.

13.2.4.3 TMRB_INTFactor

Data Fields:

uint32_t

All: TMRB interrupt factor.

Bit

uint32_t

MatchLeadingTiming: 1 a match with the leading timing value is detected

uint32_t

MatchTrailingTiming: 1 a match with the trailing timing value is detected

uint32_t

Overflow: 1 an up-counter is overflow

uint32_t

Reserverd: 29 -

14. SIO/UART

14.1 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/tmpm384_uart.c, with /Libraries/TX03_Periph_Driver/inc/tmpm384_uart.h containing the macros, data types, structures and API definitions for use by applications.

14.2 API Functions

14.2.1 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_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)

14.2.2 Detailed Description

Functions listed above can be divided into four parts:

- 1) Initialize and configure the common functions of each UART/SIO channel are handled by UART_Enable(), UART_Disable(), UART_Init(), UART_DefaultConfig(), SIO_Enable(), SIO_Disable() and 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_SWReset(), UART_TRxAutoDisable(), UART_SetWakeUpFunc() and UART_SetIdleMode() handle other specified functions.
- 4) Special for FIFO mode of each UART channel are handled by UART_FIFOConfig(), UART_SetFIFOTransferMode(), 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().

14.2.3 Function Documentation

Note: in all of the following APIs, parameter “TSB_SC_TypeDef* **UARTx**” can be one of the following values:

UART0, UART1, UART2, UART3, UART4.

parameter “TSB_SC_TypeDef* **SIOx**” can be one of the following values:

SIO0, SIO1, SIO2, SIO3, SIO4.

14.2.3.1 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

14.2.3.2 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

14.2.3.3 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.

14.2.3.4 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

14.2.3.5 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

14.2.3.6 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

14.2.3.7 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

14.2.3.8 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

14.2.3.9 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.

14.2.3.10 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

14.2.3.11 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

14.2.3.12 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

14.2.3.13 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

14.2.3.14 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_RTXCNT_NONE or **UART_RTXCNT_AUTODISABLE**

Description:

This function will Control automatic disabling of transmission and reception, in specified UART channel selected by **UARTx** when **TRxAutoDisable** is **UART_RTXCNT_AUTODISABLE**, and disable the channel when **TRxAutoDisable** is **UART_RTXCNT_NONE**.

Return:

None

14.2.3.15 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

14.2.3.16 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

14.2.3.17 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

14.2.3.18 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

14.2.3.19 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

14.2.3.20 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

14.2.3.21 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

14.2.3.22 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

14.2.3.23 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

14.2.3.24 UART_GetRxFIFOFillLevelStatus

Status of receive FIFO fill level.

Prototype:
uint32_t
UART_GetRxFIFOFillLevelStatus (TSB_SC_TypeDef* **UARTx**);

Parameters:
UARTx is the specified UART channel.

Description:
Status of receive 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_4B: TX FIFO fill level is 4 bytes.

14.2.3.25 UART_GetRxFIFOOverRunStatus

Receive FIFO overrun.

Prototype:
uint32_t
UART_GetRxFIFOOverRunStatus (TSB_SC_TypeDef* **UARTx**);

Parameters:
UARTx is the specified UART channel.

Description:
Receive FIFO overrun.

Return:
UART_RXFIFO_OVERRUN: Flags for RX FIFO overrun.

14.2.3.26 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_4B: TX FIFO fill level is 4 bytes.

14.2.3.27 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.

14.2.3.28 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

14.2.3.29 SIO_Disable

Disable the specified SIO channel.

Prototype:

void

SIO_Disable(TSB_SC_TypeDef* **SIOx**)

Parameters:

SIO T_x is the specified SIO channel.

Description:

This function will disable the specified SIO channel selected by **SIO x** .

Return:

None

14.2.3.30 SIO_GetRxData

Get data received from the specified SIO channel.

Prototype:

```
uint32_t  
SIO_GetRxData(TSB_SC_TypeDef* SIO $x$ )
```

Parameters:

SIO x is the specified SIO channel.

Description:

This function will get the data received from the specified SIO channel selected by **SIO x** .

Return:

Data which has been received, the data value range is 0x00 to 0xFF.

14.2.3.31 SIO_SetTxData

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

Prototype:

```
void  
SIO_SetTxData(TSB_SC_TypeDef* SIO $x$ ,  
              uint8_t Data)
```

Parameters:

SIO x 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 **SIO x** .

Return:

None

14.2.3.32 SIO_Init

Initialize and configure the specified SIO channel.

Prototype:

```
void  
SIO_Init(TSB_SC_TypeDef* SIO $x$ ,  
         uint32_t IOCIkSel,
```

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

14.2.4 Data Structure Description

14.2.4.1 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.

14.2.4.2 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

15. VLTD

15.1 Overview

The voltage detection circuit detects any decrease in the supply voltage and generates NMI.

The VLTD driver APIs provide a set of functions to enable or disable the VLTD function, configure detection voltage and get the power supply voltage status.

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm384_vltd.c, with /Libraries/TX03_Periph_Driver/inc/tmpm384_vltd.h containing the macros, data types, structures and API definitions for use by applications.

15.2 API Functions

15.2.1 Function List

- ◆ void VLTD_Enable(void);
- ◆ void VLTD_Disable(void);
- ◆ void VLTD_SetVoltage(uint32_t Voltage);
- ◆ uint32_t VLTD_GetStatus(void);

15.2.2 Detailed Description

Functions listed above can be divided into two parts:

- 1) Configure VLTD are handled by VLTD_SetVoltage(), VLTD_Enable() and VLTD_Disable().
- 2) Get the power supply voltage detection status info by VLTD_GetStatus().

15.2.3 Function Documentation

15.2.3.1 VLTD_Enable

Enable the VLTD function.

Prototype:

void
VLTD_Enable(void)

Parameters:

None.

Description:

This function will enable the VLTD function.

Return:

None.

15.2.3.2 VLTD_Disable

Disable the VLTD function.

Prototype:

void
VLTD_Disable(void)

Parameters:

None.

Description:

This function will disable the VLTD function.

Return:

None.

15.2.3.3 VLTD_SetVoltage

Select the detection voltage.

Prototype:

void
VLTD_SetVoltage(uint32_t **Voltage**)

Parameters:

Voltage is the value detection voltage.

This parameter can be one of the following values:

- **VLTD_DETECT_VOLTAGE_41**: Detection voltage = 4.1V
- **VLTD_DETECT_VOLTAGE_44**: Detection voltage = 4.4V
- **VLTD_DETECT_VOLTAGE_46**: Detection voltage = 4.6V

Description:

This function will set the value of detection voltage.

Return:

None.

15.2.3.4 VLTD_GetStatus

Get the status of current power supply voltage.

Prototype:

uint32_t
VLTD_GetStatus(void)

Parameters:

None.

Description:

This function returns the status of current power supply voltage.

Return:

The status of current power supply voltage.

The value can be one of the following:

0 means current power supply voltage is higher than the detection voltage specified by VDLVL[1:0].

1 means current power supply voltage is lower than the detection voltage specified by VDLVL[1:0].

16. WDT

16.1 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\tmpm384_wdt.c, with \Libraries\TX03_Periph_Driver\inc\tmpm384_wdt.h containing the API definitions for use by applications.

16.2 API Functions

16.2.1 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)

16.2.2 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().

16.2.3 Function Documentation

16.2.3.1 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

16.2.3.2 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

16.2.3.3 WDT_SetOverflowOutput

Set WDT to generate NMI interrupt or 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 the NMI interrupt when counter overflows.
- **WDT_WDOUT:** Set WDT to generate reset when counter overflows.

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

16.2.3.4 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 “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

16.2.3.5 WDT_Enable

Enable the WDT function.

Prototype:
void
WDT_Enable(void)

Parameters:
None

Description:
This function will enable WDT.

Return:
None

16.2.3.6 WDT_Disable

Disable the WDT function.

Prototype:
void
WDT_Disable(void)

Parameters:
None

Description:
This function will disable WDT.

Return:
None

16.2.3.7 WDT_WriteClearCode

Write the clear code.

Prototype:
void
WDT_WriteClearCode (void)

Parameters:
None

Description:
This function will clear the WDT counter.

Return:
None

16.2.4 Data Structure Description

16.2.4.1 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 the NMI interrupt when the counter overflows.

17. ENC

17.1 Overview

The TMPM384 has a two-channel incremental encoder interface (ENC0 and ENC1), which can obtain the absolute position of the motor, based on input signals from the incremental encoder.

The encoder input circuit supports four operation modes including encoder mode, sensor mode (two types) and timer mode. And the functions are as follows:

- Supports incremental encoders and Hall sensor ICs (signals of Hall sensor IC can be input directly).
- 24-bit general-purpose timer mode
- Multiply-by-4 (multiply-by-6) circuit
- Rotational direction detection circuit
- 24-bit counter
- Comparator enable/disable
- Interrupt request output: 1
- Digital noise filters for input signals

The ENC API provides a set of functions for using the TMPM384 ENC module. It includes ENC channel select, mode set, compare function set, software capture set, ENC status read, ENC counter read and so on.

This driver is contained in TX03_Periph_Driver\src\tmpm384_enc.c, with TX03_Periph_Driver\incltmpm384_enc.h containing the API definitions for use by applications.

17.2 API Functions

17.2.1 Function List

- ◆ void ENC_Enable(TSB_EN_TypeDef * **ENx**);
- ◆ void ENC_Disable(TSB_EN_TypeDef * **ENx**);
- ◆ void ENC_Init(TSB_EN_TypeDef * **ENx**, ENC_InitTypeDef * **InitStruct**);
- ◆ void ENC_SetSWCapture(TSB_EN_TypeDef * **ENx**, uint32_t **ENC_Mode**);
- ◆ void ENC_ClearCounter (TSB_EN_TypeDef * **ENx**);
- ◆ ENC_FlagStatus ENC_GetENCFlag (TSB_EN_TypeDef * **ENx**);
- ◆ void ENC_SetCounterReload (TSB_EN_TypeDef * **ENx**, uint32_t **ENC_Mode**, uint32_t **PeriodValue**);
- ◆ void ENC_SetCompareValue(TSB_EN_TypeDef * **ENx**, uint32_t **ENC_Mode**, uint32_t **CompareValue**);
- ◆ uint32_t ENC_GetCompareValue(TSB_EN_TypeDef * **ENx**);
- ◆ uint32_t ENC_GetCounterValue(TSB_EN_TypeDef * **ENx**);

17.2.2 Detailed Description

Functions listed above can be divided into three parts:

- 1) ENC setting by ENC_Init (), ENC_ClearCounter (), ENC_SetCounterReload (), ENC_SetSWCapture (), ENC_SetCompareValue ().

- 2) ENC function enable/disable by ENC_Enable (), ENC_Disable (),
- 3) ENC state or data read functions by ENC_GetENCFlag (), ENC_GetCounterValue (), ENC_GetCompareValue ().

17.2.3 Function Documentation

17.2.3.1 ENC_Enable

Enable the specified encoder operation.

Prototype:

void
ENC_Enable(TSB_EN_TypeDef * **ENx**)

Parameters:

ENx: Select ENC channel.

This parameter can be one of the following values:

- **EN0**
- **EN1**

Description:

This function will enable the specified encoder operation.

Return:

None

17.2.3.2 ENC_Disable

Disable the specified encoder operation.

Prototype:

void
ENC_Disable(TSB_EN_TypeDef * **ENx**)

Parameters:

ENx: Select ENC channel.

This parameter can be one of the following values:

- **EN0**
- **EN1**

Description:

This function will disable the specified encoder operation.

Return:

None

17.2.3.3 ENC_Init

Initialize the specified encoder operation.

Prototype:

void
ENC_Init(TSB_EN_TypeDef * **ENx**, ENC_InitTypeDef * **InitStruct**)

Parameters:

ENx: Select ENC channel.

This parameter can be one of the following values:

- **EN0**
- **EN1**

InitStruct: The structure containing basic encoder configuration (see Data Structure for details).

Description:

This function will initialize the specified encoder operation.

Return:

None

17.2.3.4 ENC_SetSWCapture

Set the specified encoder to execute software capture (timer mode/sensor mode (at timer count)).

Prototype:

void

ENC_SetSWCapture(TSB_EN_TypeDef * **ENx**, uint32_t **ENC_Mode**)

Parameters:

ENx: Select ENC channel.

This parameter can be one of the following values:

- **EN0**
- **EN1**

ENC_Mode: Select the encoder operation mode.

This parameter can be one of the following values:

- **ENC_TIMER_MODE**
- **ENC_SENSOR_TIME_MODE**

Description:

This function will set the specified encoder to execute software capture (timer mode/sensor mode (at timer count)).

Return:

None

17.2.3.5 ENC_ClearCounter

Clear pulse counter for the specified encoder.

Prototype:

void

ENC_ClearCounter (TSB_EN_TypeDef * **ENx**)

Parameters:

ENx: Select ENC channel.

This parameter can be one of the following values:

- **EN0**
- **EN1**

Description:

This function will clear pulse counter for the specified encoder.

Return:

None

17.2.3.6 ENC_GetENCFlag

Get the encoder compare flag/reverse error flag/Z-detected/rotation direction.

Prototype:

ENC_FlagStatus

ENC_GetENCFlag (TSB_EN_TypeDef * **ENx**)

Parameters:

ENx: Select ENC channel.

This parameter can be one of the following values:

- **EN0**
- **EN1**

Description:

This function will get the encoder compare flag/reverse error flag/Z-detected/rotation direction.

Return:

The union that indicates the encoder flag

ZphaseDetectFlag (bit12) means ENC Z-phase detect flag.

RotationDirection (bit13) means ENC rotation direction.

ReverseErrorFlag (bit14) means ENC sensor mode (at time count) reverse error flag.

CompareFlag (bit15) means ENC compare flag.

17.2.3.7 ENC_SetCounterReload

Set the encoder counter period.

Prototype:

void

ENC_SetCounterReload (TSB_EN_TypeDef * **ENx**, uint32_t **PeriodValue**)

Parameters:

ENx: Select ENC channel.

This parameter can be one of the following values:

- **EN0**
- **EN1**

PeriodValue: Set the encoder counter period.

This parameter can be **0x0000 - 0xFFFF**

Description:

This function will set the encoder counter period.

Return:
None

17.2.3.8 ENC_SetCompareValue

Set the encoder counter compare value.

Prototype:

```
void  
ENC_SetCompareValue(TSB_EN_TypeDef * ENx, uint32_t ENC_Mode,  
                    uint32_t CompareValue)
```

Parameters:

ENx: Select ENC channel.

This parameter can be one of the following values:

- **EN0**
- **EN1**

ENC_Mode: Select the encoder operation mode.

This parameter can be one of the following values:

- **ENC_ENCODER_MODE**
- **ENC_SENSOR_EVENT_MODE**
- **ENC_SENSOR_TIME_MODE**
- **ENC_TIMER_MODE**

CompareValue: Set the encoder counter compare value

In sensor mode (event count) and encoder mode:

This parameter can be **0x0000 - 0xFFFF**.

In sensor mode (timer count) and timer mode:

This parameter can be **0x000000 - 0xFFFFFFF**.

Description:

This function will set the encoder counter compare value.

Return:
None

17.2.3.9 ENC_GetCompareValue

Get the encoder counter compare value.

Prototype:

```
uint32_t  
ENC_GetCompareValue(TSB_EN_TypeDef * ENx)
```

Parameters:

ENx: Select ENC channel.

This parameter can be one of the following values:

- **EN0**
- **EN1**

Description:

This function will get the encoder counter compare value.

Return:

Compare value of the encoder.

17.2.3.10 ENC_GetCounterValue

Get the encoder counter/capture value.

Prototype:

```
uint32_t  
ENC_GetCounterValue(TSB_EN_TypeDef * ENx)
```

Parameters:

ENx: Select ENC channel.

This parameter can be one of the following values:

- **EN0**
- **EN1**

Description:

This function will get the encoder counter/capture value.

Return:

Value of the encoder counter

17.2.4 Data Structure Description

17.2.4.1 ENC_InitTypeDef

Data Fields:

uint32_t

ModeType Encoder input mode selection, which can be:

- **ENC_ENCODER_MODE**
- **ENC_SENSOR_EVENT_MODE**
- **ENC_SENSOR_TIME_MODE**
- **ENC_TIMER_MODE**

uint32_t

PhaseType 2-phase / 3-phase input selection (sensor mode), which can be:

- **ENC_TWO_PHASE**
- **ENC_THREE_PHASE**

uint32_t

EdgeType Edge selection of ENCZ (timer mode), which can be:

- **ENC_RISING_EDGE**
- **ENC_FALLING_EDGE**

uint32_t

CompareStatus Enable or disable the encoder compare function, which can be:

- **ENC_COMPARE_DISABLE**
- **ENC_COMPARE_ENABLE**

uint32_t

ZphaseStatus Enable or disable the Z-phase (encoder mode/timer mode), which can be:

- **ENC_ZPHASE_DISABLE**
- **ENC_ZPHASE_ENABLE**

uint32_t

FilterValue Set the noise filter effect value, which can be:

- **ENC_NO_FILTER,**
- **ENC_FILTER_VALUE31**
- **ENC_FILTER_VALUE63**
- **ENC_FILTER_VALUE127**

uint32_t

IntEn Enable or disable the encoder interrupt, which can be:

- **ENC_INTERRUPT_DISABLE**
- **ENC_INTERRUPT_ENABLE**

uint32_t

PulseDivFactor Set the encoder pulse division factor, which can be:

- **ENC_PULSE_DIV1**
- **ENC_PULSE_DIV2**
- **ENC_PULSE_DIV4**
- **ENC_PULSE_DIV8**
- **ENC_PULSE_DIV16**
- **ENC_PULSE_DIV32**
- **ENC_PULSE_DIV64**
- **ENC_PULSE_DIV128**

17.2.4.2 ENC_FlagStatus

Data Fields:

uint32_t

All specifies ENC all flag status

uint32_t

ZPhaseDetectFlag (bit12) means ENC Z-phase detect flag.

uint32_t

RotationDirection (bit13) means ENC rotation direction.

uint32_t

ReverseErrorFlag (bit14) means ENC sensor mode (at time count) reverse error flag.

uint32_t

CompareFlag (bit15) means ENC compare flag.

18. PMD

18.1 Overview

TMPM384 has two channels of motor control circuits (PMD).

The PMD enables 1-shunt sensor-less motor control by adding current-carrying output control and DC overvoltage detection input. It achieves associative motor control with AD converter.

The PMD (Programmable Motor Driver) circuit consists of the wave generation circuit and synchronous trigger generation circuit. The wave generation circuit includes the pulse width modulation circuit, current-applying control circuit, protection control circuit and dead time control circuit.

- Pulse width modulation circuit generates identical 3-phase independent PWM waveforms.
- Applying current circuit determines each upper/lower output pattern of U-, V-, W-phase.
- Protection control circuit takes place emergency stop by detecting abnormal detect input.
- Dead time control prevent a short circuit at switching upper/lower phase.
- The synchronous trigger generation circuit generates synchronous trigger signals to the AD converter.

This driver is contained in TX03_Periph_Driver\src\tmpm384_pmd.c, with TX03_Periph_Driver\inc\tmpm384_pmd.h containing the API definitions for use by applications.

18.2 API Functions

18.2.1 Function List

- ◆ void PMD_Enable(TSB_MTPD_TypeDef * **PMDx**);
- ◆ void PMD_Disable(TSB_MTPD_TypeDef * **PMDx**);
- ◆ void PMD_SetPortControl(TSB_MTPD_TypeDef * **PMDx**,
uint32_t **PortMode**);
- ◆ void PMD_Init(TSB_MTPD_TypeDef * **PMDx**,
PMD_InitTypeDef * **InitStruct**);
- ◆ void PMD_ChangePWMCycle(TSB_MTPD_TypeDef * **PMDx**,
uint32_t **CycleTiming**);
- ◆ uint32_t PMD_GetCntFlag(TSB_MTPD_TypeDef * **PMDx**);
- ◆ uint16_t PMD_GetCntValue(TSB_MTPD_TypeDef * **PMDx**);
- ◆ void PMD_SetCompareValue(TSB_MTPD_TypeDef * **PMDx**,
uint32_t **PMDPhase**,
uint32_t **Timing**);
- ◆ void PMD_SetPortOutputMode(TSB_MTPD_TypeDef * **PMDx**,
uint32_t **Mode**);
- ◆ void PMD_SetOutputPhasePolarity(TSB_MTPD_TypeDef * **PMDx**,
uint32_t **OutputPhase**,
uint32_t **Polarity**);
- ◆ void PMD_SetReflectTime(TSB_MTPD_TypeDef * **PMDx**,
uint32_t **ReflectedTime**);
- ◆ void PMD_EnableEMG(TSB_MTPD_TypeDef * **PMDx**);
- ◆ void PMD_DisableEMG(TSB_MTPD_TypeDef * **PMDx**);
- ◆ void PMD_SetEMGNoiseElimination(TSB_MTPD_TypeDef * **PMDx**,
uint32_t **NoiseElimination**);
- ◆ void PMD_SetToolBreakOutput(TSB_MTPD_TypeDef * **PMDx**,
uint32_t **Status**);
- ◆ void PMD_SetEMGMode(TSB_MTPD_TypeDef * **PMDx**,
uint32_t **Mode**);
- ◆ void PMD_EMGRelease(TSB_MTPD_TypeDef * **PMDx**);
- ◆ uint32_t PMD_GetEMGAbnormalLevel(TSB_MTPD_TypeDef * **PMDx**);
- ◆ uint32_t PMD_GetEMGCondition(TSB_MTPD_TypeDef * **PMDx**);
- ◆ void PMD_SetDeadTime(TSB_MTPD_TypeDef * **PMDx**,
uint32_t **Time**);
- ◆ void PMD_SetAllPhaseCompareValue(TSB_MTPD_TypeDef * **PMDx**,
uint32_t **UPhaseTiming**,
uint32_t **VPhaseTiming**,
uint32_t **WPhaseTiming**);
- ◆ void PMD_ChangeDutyMode(TSB_MTPD_TypeDef * **PMDx**,
uint32_t **DutyMode**);
- ◆ void PMD_SetPortOutput (TSB_MTPD_TypeDef * **PMDx**,
uint32_t **PMDPhase**,
uint8_t **Output**);
- ◆ void PMD_SetTrgCmpValue(TSB_MTPD_TypeDef * **PMDx**,
uint32_t **TRGCMP0Timing**,
uint32_t **TRGCMP1Timing**);
- ◆ void PMD_SetTrgMode(TSB_MTPD_TypeDef * **PMDx**,
uint32_t **PMDTrg**,
uint32_t **Mode**);
- ◆ void PMD_SetTrgUpdate(TSB_MTPD_TypeDef * **PMDx**,
uint32_t **PMDTrg**,

uint32_t *UpdateTiming*);
◆ Result PMD_SetEMGTrg(TSB_MTPD_TypeDef * *PMDx*,
FunctionalState *NewState*);

18.2.2 Detailed Description

Functions listed above can be divided into six parts:

- 1) Common configuration and control of each PMD channel are handled by PMD_Enable(), PMD_Disable(), PMD_SetPortControl(), PMD_Init(), PMD_ChangePWMCycle(), PMD_SetCompareValue(), PMD_SetAllPhaseCompareValue(), PMD_ChangeDutyMode().
- 2) PMD port output settings are handled by PMD_SetPortOutputMode(), PMD_SetOutputPhasePolarity(), PMD_SetReflectTime(), PMD_SetPortOutput().
- 3) PMD EMG functions are handled by PMD_EnableEMG(), PMD_DisableEMG(), PMD_SetEMGNoiseElimination(), PMD_SetToolBreakOutput(), PMD_SetEMGMode(), PMD_EMGRelease().
- 4) The status indication of each PMD channel is handled by PMD_GetCntrFlag(), PMD_GetCntValue(), PMD_GetEMGAbnormalLevel(), PMD_GetEMGCondition().
- 5) PMD dead time control is handled by PMD_SetDeadTime().
- 6) PMD ADC trigger generation circuit is handled by PMD_SetTrgCmpValue(), PMD_SetTrgMode(), PMD_SetTrgUpdate(), PMD_SetEMGTrg().

18.2.3 Function Documentation

***Note:** In all of the following APIs, parameter “TSB_MTPD_TypeDef * *PMDx*” can be *PMD0*, *PMD1*.

18.2.3.1 PMD_Enable

Enable the specified PMD channel.

Prototype:

void
PMD_Enable (TSB_MTPD_TypeDef * *PMDx*)

Parameters:

PMDx: Select the PMD channel.

Description:

This function will enable the specified PMD channel.

Return:

None

18.2.3.2 PMD_Disable

Disable the specified PMD channel.

Prototype:

void
PMD_Disable (TSB_MTPD_TypeDef * *PMDx*)

Parameters:

PMDx: Select the PMD channel.

Description:

This function will disable the specified PMD channel.

Return:

None

18.2.3.3 PMD_SetPortControl

Set PMD port control of the specified PMD channel.

Prototype:

```
void  
PMD_SetPortControl (TSB_MTPD_TypeDef * PMDx  
                    uint32_t PortMode)
```

Parameters:

PMDx: Select the PMD channel.

PortMode: The port output mode of PMD.

- **PMD_PORT_HIGH_IMPEDANCE**: High-impedance
- **PMD_PORT_PMD_OUTPUT**: PMD output

Description:

This function will set PMD port control of the specified PMD channel.

Return:

None

18.2.3.4 PMD_Init

Initialize the specified PMD channel.

Prototype:

```
void  
PMD_Init (TSB_MTPD_TypeDef * PMDx,  
          PMD_InitTypeDef * InitStruct)
```

Parameters:

PMDx: Select the PMD channel.

InitStruct: The structure containing basic PMD configuration.
(Refer to “Data Structure Description” for details).

Description:

This function will initialize the specified PMD channel.

Return:

None

18.2.3.5 PMD_ChangePWMCycle

Change the PWM cycle of the specified PMD channel.

Prototype:

void

PMD_ChangePWMCycle (TSB_MTPD_TypeDef * **PMDx**,
uint32_t **CycleTiming**)

Parameters:

PMDx: Select the PMD channel.

CycleTiming: PWM cycle, from 0x0000 to 0xFFFF.

Description:

This function will change the PWM cycle of the specified PMD channel.

Return:

None

***Note:**

If a value less than 0x10 is set, the register assumes 0x10 is set

18.2.3.6 PMD_GetCntFlag

Get the PWM counter flag of the specified PMD channel.

Prototype:

uint32_t

PMD_GetCntFlag (TSB_MTPD_TypeDef * **PMDx**)

Parameters:

PMDx: Select the PMD channel.

Description:

This function will get the PWM counter flag of the specified PMD channel.

Return:

The PWM counter flag.

The value returned can be one of the following values:

PMD_COUNTER_UP: The PWM counter is up-counting

PMD_COUNTER_DOWN : The PWM counter is down-counting

18.2.3.7 PMD_GetCntValue

Get the count value of the specified PMD channel.

Prototype:

uint16_t

PMD_GetCntValue (TSB_MTPD_TypeDef * **PMDx**)

Parameters:

PMDx: Select the PMD channel.

Description:

This function will get the count value of the specified PMD channel.

Return:

Count value of the specified PMD channel.

18.2.3.8 PMD_SetCompareValue

Set the compare value of the specified phase of the specified PMD channel.

Prototype:

```
void  
PMD_SetCompareValue (TSB_MTPD_TypeDef * PMDx,  
                     uint32_t PMDPhase,  
                     uint32_t Timing)
```

Parameters:

PMDx: Select the PMD channel.

PMDPhase: Select the phase of PMD channel.

This parameter can be one of the following values:

- **PMD_PHASE_U**: U-phase
- **PMD_PHASE_V**: V-phase
- **PMD_PHASE_W**: W-phase
- **PMD_PHASE_ALL**: All phases

Timing: Compare value, from 0x0000 to 0xFFFF.

Description:

This function will set the compare value of the specified phase of the specified PMD channel.

Return:

None

18.2.3.9 PMD_SetPortOutputMode

Set the mode of port output of the specified PMD channel.

Prototype:

```
void  
PMD_SetPortOutputMode (TSB_MTPD_TypeDef * PMDx,  
                       uint32_t Mode)
```

Parameters:

PMDx: Select the PMD channel.

Mode: The mode of port output.

This parameter can be one of the following values:

- **PMD_PORT_OUTPUT_MODE_0**: MTPDxMDCR<SYNTMD>=0
- **PMD_PORT_OUTPUT_MODE_1**: MTPDxMDCR<SYNTMD>=1

Description:

This function will set the mode of port output of the specified PMD channel.

***Note:**

MTPDxMDCR<SYNTMD>, MTPDxMDPOT<POLH><POLL>, MTPDxMDOUT<UPWN><VPWN><WPWN> <UOC> <VOC> <WOC> set the port output. (x can be 0,1)

PMD_SetPortOutputMode() set MTPDxMDCR<SYNTMD>.
PMD_SetOutputPhasePolarity() set MTPDxMDPOT<POLH><POLL>.
PMD_SetPortOutput () set MTPDxMDOUT<UPWN><VPWN> <WPWN>
<UOC> <VOC> <WOC>.

The details about the port output is in the below diagram.

MTPDxMDCR<SYNTMD>=0

Polarity: high-active(MTPDxMDPOT<POLH><POLL>="11")

MDOUT output control		MTPDxMDOUT <WPWM><VPWM><UPWM> H/L/PWM output selection			
<WOC[1]> <VOC[1]> <UOC[1]> (Upper)	<WOC[0]> <VOC[0]> ><UOC[0]> (Lower)	0 : H/L output		1 : PWM output	
		Upper output	Lower output	Upper output	Lower output
0	0	L	L	PWM	PWM
0	1	L	H	L	PWM
1	0	H	L	PWM	L
1	1	H	H	PWM	PWM

MTPDxMDCR<SYNTMD>=0

Polarity: low-active(MTPDxMDPOT<POLH><POLL>="00")

MDOUT output control		MTPDxMDOUT <WPWM><VPWM><UPWM> H/L/PWM output selection			
<WOC[1]> <VOC[1]> <UOC[1]> (Upper)	<WOC[0]> <VOC[0]> ><UOC[0]> (Lower)	0 : H/L output		1 : PWM output	
		Upper output	Lower output	Upper output	Lower output
0	0	H	H	PWM	PWM
0	1	H	L	H	PWM
1	0	L	H	PWM	H
1	1	L	L	PWM	PWM

MTPDxMDCR<SYNTMD>=1

Polarity: high-active(MTPDxMDPOT<POLH><POLL>="11")

MDOUT output control		MTPDxMDOUT <WPWM><VPWM><UPWM> H/L/PWM output selection			
<WOC[1]> <VOC[1]> <UOC[1]> (Upper)	<WOC[0]> <VOC[0]> ><UOC[0]> (Lower)	0 : H/L output		1 : PWM output	
		Upper output	Lower output	Upper output	Lower output
0	0	L	L	PWM	PWM
0	1	L	H	L	PWM
1	0	H	L	PWM	L
1	1	H	H	PWM	PWM

MTPDxMDCR<SYNTMD>=1

Polarity: low-active(MTPDxMDPOT<POLH><POLL>="00")

MDOUT output control		MTPDxMDOUT <WPWM><VPWM><UPWM> H/L/PWM output selection			
<WOC[1]> <VOC[1]> <UOC[1]> (Upper)	<WOC[0]> <VOC[0]> ><UOC[0]> (Lower)	0 : H/L output		1 : PWM output	
		Upper output	Lower output	Upper output	Lower output
0	0	H	H	PWM	PWM
0	1	H	L	H	PWM
1	0	L	H	PWM	H
1	1	L	L	PWM	PWM

Return:
None

18.2.3.10PMD_SetOutputPhasePolarity

Set polarity of the specified output port phase of the specified PMD channel.

Prototype:

void

PMD_SetOutputPhasePolarity (TSB_MTPD_TypeDef * **PMDx**,
uint32_t **OutputPhase**,
uint32_t **Polarity**)

Parameters:

PMDx: Select the PMD channel.

OutputPhase: Select the specified output port phase.

- **PMD_OUTPUT_PHASE_UPPER:** Upper phase output port
- **PMD_OUTPUT_PHASE_LOWER:** Lower phase output port

Polarity: The value of the bit.

- **PMD_POLARITY_LOW:** Low active
- **PMD_POLARITY_HIGH:** High active

Description:

This function will set polarity of the specified output port phase of the specified PMD channel.

***Note:**

- 1 Refer to function PMD_SetPortOutputMode() for more details.
- 2 When calling this function, the PMD must be disabled.

Return:

None

18.2.3.11 PMD_SetReflectTime

Choose the timing when port outputs of U-, V- and W- phase output setting is reflected of the specified PMD channel.

Prototype:

```
void  
PMD_SetReflectTime (TSB_MTPD_TypeDef * PMDx,  
                    uint32_t ReflectedTime)
```

Parameters:

PMDx: Select the PMD channel.

ReflectedTime: Select the reflected time.

- **PMD_REFLECTED_TIME_WRITE:** Reflect when write
- **PMD_REFLECTED_TIME_MIN:** Reflect when PWM counter MDCNT="1"(minimum)
- **PMD_REFLECTED_TIME_MAX:** Reflect when PWM counter MDCNT=MTPDxMDPRD<MDPRD>(maximum)
- **PMD_REFLECTED_TIME_MIN_MAX:** Reflect when PWM counter MDCNT="1"(minimum) or MTPDxMDPRD<MDPRD>(maximum)

Description:

This function will choose the timing when port outputs of U-, V- and W- phase output setting is reflected of the specified PMD channel.

***Note:**

When calling this function, the PMD must be disabled.

Return:

None

18.2.3.12 PMD_EnableEMG

Enable EMG protection of the specified PMD channel.

Prototype:

```
void  
PMD_EnableEMG (TSB_MTPD_TypeDef * PMDx)
```

Parameters:

PMDx: Select the PMD channel.

Description:

This function will enable EMG protection of the specified PMD channel.

Return:

None

18.2.3.13PMD_DisableEMG

Disable EMG protection of the specified PMD channel.

Prototype:

void

PMD_DisableEMG (TSB_MTPD_TypeDef * **PMDx**)

Parameters:

PMDx: Select the PMD channel.

Description:

This function will disable EMG protection of the specified PMD channel.

Return:

None

18.2.3.14PMD_SetEMGNoiseElimination

Set the noise elimination time for abnormal condition detection input of the specified PMD channel.

Prototype:

void

PMD_SetEMGNoiseElimination (TSB_MTPD_TypeDef * **PMDx**,
uint32_t **NoiseElimination**)

Parameters:

PMDx: Select the PMD channel.

NoiseElimination: Select the noise elimination time.

- **PMD_NOISE_ELIMINATION_NONE:** Noise filter is not used
- **PMD_NOISE_ELIMINATION_16:** Input noise elimination time 16/fsys[s]
- **PMD_NOISE_ELIMINATION_32:** Input noise elimination time 32/fsys[s]
- **PMD_NOISE_ELIMINATION_48:** Input noise elimination time 48/fsys[s]
- **PMD_NOISE_ELIMINATION_64:** Input noise elimination time 64/fsys[s]
- **PMD_NOISE_ELIMINATION_80:** Input noise elimination time 80/fsys[s]
- **PMD_NOISE_ELIMINATION_96:** Input noise elimination time 96/fsys[s]
- **PMD_NOISE_ELIMINATION_112:** Input noise elimination time 112/fsys[s]
- **PMD_NOISE_ELIMINATION_128:** Input noise elimination time 128/fsys[s]
- **PMD_NOISE_ELIMINATION_144:** Input noise elimination time 144/fsys[s]
- **PMD_NOISE_ELIMINATION_160:** Input noise elimination time 160/fsys[s]
- **PMD_NOISE_ELIMINATION_176:** Input noise elimination time 176/fsys[s]
- **PMD_NOISE_ELIMINATION_192:** Input noise elimination time 192/fsys[s]
- **PMD_NOISE_ELIMINATION_208:** Input noise elimination time 208/fsys[s]
- **PMD_NOISE_ELIMINATION_224:** Input noise elimination time 224/fsys[s]
- **PMD_NOISE_ELIMINATION_240:** Input noise elimination time 240/fsys[s]

Description:

This function will set the noise elimination time for abnormal condition detection input of the specified PMD channel.

Return:

None

18.2.3.15PMD_SetToolBreakOutput

Choose PMD output status at tool break of the specified PMD channel.

Prototype:

```
void  
PMD_SetToolBreakOutput (TSB_MTPD_TypeDef * PMDx,  
                        uint32_t Status)
```

Parameters:

PMDx: Select the PMD channel.

Status: PMD output status at tool break.

- **PMD_BREAK_STATUS_PMD**: PMD output is continued
- **PMD_BREAK_STATUS_HIGH_IMPEDANCE**: High-impedance

Description:

This function will choose PMD output status at tool break of the specified PMD channel.

Return:

None

18.2.3.16PMD_SetEMGMode

Set EMG protection mode of the specified PMD channel.

Prototype:

```
void  
PMD_SetEMGMode (TSB_MTPD_TypeDef * PMDx,  
                uint32_t Mode)
```

Parameters:

PMDx: Select the PMD channel.

Mode: EMG protection mode.

- **PMD_EMG_MODE_0**: All phase are on/PORT output and high-impedance
- **PMD_EMG_MODE_1**: All phase are off/PORT output and high-impedance
- **PMD_EMG_MODE_2**: All phase are on/PORT output is enabled
- **PMD_EMG_MODE_3**: All phase are off/PORT output is high-impedance

Description:

This function will set EMG protection mode of the specified PMD channel.

Return:

None

18.2.3.17PMD_EMGRelease

Release EMG protection status of the specified PMD channel.

Prototype:

void
PMD_EMGRelease (TSB_MTPD_TypeDef * *PMDx*)

Parameters:

PMDx: Select the PMD channel.

Description:

This function will release EMG protection status of the specified PMD channel

***Note:**

MTPDxMDOUT<UPWN><VPWN><WPWN> and MTPDxMDOUT<UOC>
<VOC> <WOC> will be cleared to 0 after the function be called. (x can be 0,1)

Return:

None

18.2.3.18PMD_GetEMGAbnormalLevel

Get the level of abnormal condition input of the specified PMD channel.

Prototype:

uint32_t
PMD_GetEMGAbnormalLevel (TSB_MTPD_TypeDef * *PMDx*)

Parameters:

PMDx: Select the PMD channel.

Description:

This function will get the level of abnormal condition input of the specified PMD channel.

Return:

The level of abnormal condition input.

The value returned can be one of the following values:

PMD_ABNORMAL_LEVEL_L: Abnormal condition input level is "L"

PMD_ABNORMAL_LEVEL_H: Abnormal condition input level is "H"

18.2.3.19PMD_GetEMGCondition

Get the EMG protection condition of the specified PMD channel.

Prototype:

uint32_t
PMD_GetEMGCondition (TSB_MTPD_TypeDef * *PMDx*)

Parameters:

PMDx: Select the PMD channel.

Description:

This function will get the EMG protection condition of the specified PMD channel.

Return:

The EMG protection condition.

The value returned can be 0 or 1.

0 means normal operation.

1 means during in EMG protection.

18.2.3.20 PMD_SetDeadTime

Set dead time of the specified PMD channel.

Prototype:

void

PMD_SetDeadTime (TSB_MTPD_TypeDef * **PMDx**,
uint32_t **Time**)

Parameters:

PMDx: Select the PMD channel.

Time: Dead time, from 0x00 to 0xFF.

Description:

This function will set dead time of the specified PMD channel.

***Note:**

When calling this function, the PMD must be disabled.

Return:

None

18.2.3.21 PMD_SetAllPhaseCompareValue

Set the compare values of the all phases of the specified PMD channel.

Prototype:

void

PMD_SetAllPhaseCompareValue (TSB_MTPD_TypeDef * **PMDx**,
uint32_t **UPhaseTiming**,
uint32_t **VPhaseTiming**,
uint32_t **WPhaseTiming**)

Parameters:

PMDx: Select the PMD channel.

UPhaseTiming: Compare value of phase U, from 0x0000 to 0xFFFF.

VPhaseTiming: Compare value of phase V, from 0x0000 to 0xFFFF.

WPhaseTiming: Compare value of phase W, from 0x0000 to 0xFFFF.

Description:

This function will set the compare values of the all phases of the specified PMD channel.

Return:
None

18.2.3.22PMD_ChangeDutyMode

Change duty mode of the specified PMD channel.

Prototype:

void
PMD_ChangeDutyMode (TSB_MTPD_TypeDef * **PMDx**,
uint32_t **DutyMode**)

Parameters:

PMDx: Select the PMD channel.

DutyMode: The duty mode of PMD.

- **PMD_DUTY_MODE_U_PHASE:** U-phase in common
- **PMD_DUTY_MODE_3_PHASE:** 3-phase independent

Description:

This function will change duty mode of the specified PMD channel.

Return:
None

18.2.3.23PMD_SetPortOutput

Set the specified output of the specified phase of the specified PMD channel.

Prototype:

Result
PMD_SetPortOutput (TSB_MTPD_TypeDef * **PMDx**,
uint32_t **PMDPhase**,
uint8_t **Output**)

Parameters:

PMDx: Select the PMD channel.

PMDPhase: Select the phase of PMD channel.

This parameter can be one of the following values:

- **PMD_PHASE_U:** U-phase
- **PMD_PHASE_V:** V-phase
- **PMD_PHASE_W:** W-phase
- **PMD_PHASE_ALL:** All phases

Output: Select the output.

This parameter can be one of the following values:

- **PMD_OUTPUT_L_L:** Upper output is L, Lower output is L.
- **PMD_OUTPUT_L_H:** Upper output is L, Lower output is H.
- **PMD_OUTPUT_H_L:** Upper output is H, Lower output is L.
- **PMD_OUTPUT_H_H:** Upper output is H, Lower output is H.
- **PMD_OUTPUT_PWM_IPWM:** Upper output is PWM, Lower output is IPWM.
- **PMD_OUTPUT_IPWM_PWM:** Upper output is IPWM, Lower output is PWM.
- **PMD_OUTPUT_H_PWM:** Upper output is H, Lower output is PWM.
- **PMD_OUTPUT_L_PWM:** Upper output is L, Lower output is PWM.

- **PMD_OUTPUT_PWM_L**:Upper output is PWM, Lower output is L.
- **PMD_OUTPUT_H_IPWM**:Upper output is H, Lower output is IPWM.
- **PMD_OUTPUT_L_IPWM**:Upper output is L, Lower output is IPWM.
- **PMD_OUTPUT_IPWM_H**:Upper output is IPWM, Lower output is H.

Description:

This function will set the specified output of the specified phase of the specified PMD channel.

Return:

Success or not

The value returned can be one of the following values:

SUCCESS: PMD output is set successfully.

ERROR: PMD output setting is failed.

***Note:**

1. IPWM is the inverting PWM.
2. Refer to function PMD_SetPortOutputMode() for details.

18.2.3.24PMD_SetTrgCmpValue

Set the ADC trigger compare registers' value of the specified PMD channel.

Prototype:

```
void  
PMD_SetTrgCmpValue(TSB_MTPD_TypeDef * PMDx,  
                   uint32_t TRGCMP0Timing,  
                   uint32_t TRGCMP1Timing)
```

Parameters:

PMDx: Select the PMD channel.

TRGCMP0Timing: Value of ADC trigger compare register 0, from 0x0001 to [MDPRD set value – 1].

TRGCMP1Timing: Value of ADC trigger compare register 1, from 0x0001 to [MDPRD set value – 1].

Description:

This function will set the ADC trigger compare registers' value of the specified PMD channel.

Return:

None

***Note:** MTPDxTRGCMPn (x can be 0 or 1) should be set in a range of 1 to [MDPRD set value – 1].

18.2.3.25PMD_SetTrgMode

Set trigger mode of the specified PMD channel.

Prototype:

```
void  
PMD_SetTrgMode (TSB_MTPD_TypeDef * PMDx,  
                uint32_t PMDTrg,
```

uint32_t *Mode*)

Parameters:

PMDx: Select the PMD channel.

PMDTrg: Select the PMD Trigger.

This parameter can be one of the following values:

- **PMD_ADC_TRG_0:** Select trigger 0
- **PMD_ADC_TRG_1:** Select trigger 1

Mode: PMD trigger mode.

This parameter can be one of the following values:

- **PMD_TRG_MODE_0:** Trigger output disabled
- **PMD_TRG_MODE_1:** Trigger output at down-count match
- **PMD_TRG_MODE_2:** Trigger output at up-count match
- **PMD_TRG_MODE_3:** Trigger output at up-/down-count match
- **PMD_TRG_MODE_4:** Trigger output at PWM carrier peak
- **PMD_TRG_MODE_5:** Trigger output at PWM carrier bottom
- **PMD_TRG_MODE_6:** Trigger output at PWM carrier peak/bottom
- **PMD_TRG_MODE_7:** Trigger output disabled

Description:

This function will set trigger mode of the specified PMD channel.

Return:

None

18.2.3.26PMD_SetTrgUpdate

Set trigger buffer update timing of the specified PMD channel.

Prototype:

```
void  
PMD_SetTrgUpdate (TSB_MTPD_TypeDef * PMDx,  
                  uint32_t PMDTrg,  
                  uint32_t UpdateTiming)
```

Parameters:

PMDx: Select the PMD channel.

PMDTrg: Select the PMD Trigger.

This parameter can be one of the following values:

- **PMD_ADC_TRG_0:** Select trigger 0
- **PMD_ADC_TRG_1:** Select trigger 1

Mode: PMDTRG0 to PMDTRG1 buffer update timing.

This parameter can be one of the following values:

- **PMD_TRG_UPDATE_SYNC:** Sync to PWM
- **PMD_TRG_UPDATE_ASYNC:** The value written to PMDTRGx is immediately reflected

Description:

This function will set trigger buffer update timing of the specified PMD channel.

Return:

None

18.2.3.27 PMD_SetEMGTrg

Enable or disable trigger output in EMG protection state of the specified PMD channel.

Prototype:

void
PMD_SetEMGTrg (TSB_MTPD_TypeDef * **PMDx**,
FunctionalState **NewState**)

Parameters:

PMDx: Select the PMD channel.

NewState: Output enable in EMG protection state.

- **ENABLE**: Enable trigger output in the protection state
- **DIABLE**: Disable trigger output in the protection state

Description:

This function will enable or disable trigger output in EMG protection state of the specified PMD channel.

Return:

None

18.2.4 Data Structure Description

18.2.4.1 PMD_InitTypeDef

Data Fields:

uint32_t

CycleMode: Specify PWM cycle extension mode, which can be:

- **PMD_PWM_NORMAL_CYCLE**: Normal cycle
- **PMD_PWM_4_FOLD_CYCLE**: 4-fold cycle

uint32_t

DutyMode: Choose DUTY mode, which can be:

- **PMD_DUTY_MODE_U_PHASE**: U-phase in common
- **PMD_DUTY_MODE_3_PHASE**: 3-phase independent

uint32_t

IntTiming: Choose PWM interrupt timing when PWM mode 1 (triangle wave) is set, which can be:

- **PMD_PWM_INT_TIMING_MINIMUM**: When PWM count MDCNT="1" is set, (minimum) interrupt request occurs
- **PMD_PWM_INT_TIMING_MAXIMUM**: When PWM count MDCNT=MTPDxMDPRD<MDPRD> is set, (maximum) interrupt request occurs

uint32_t

IntCycle: Choose PWM interrupt cycle, which can be:

- **PMD_PWM_INT_CYCLE_HALF**: Every PWM 0.5 cycle (can be set in PWM mode1 (triangle wave))
- **PMD_PWM_INT_CYCLE_1**: Every PWM 1 cycle
- **PMD_PWM_INT_CYCLE_2**: Every PWM 2 cycles

- **PMD_PWM_INT_CYCLE_4:** Every PWM 4 cycles

uint32_t

CarrierMode: Specify PWM carrier wave, which can be:

- **PMD_CARRIER_WAVE_MODE_0:** PWM mode 0 (edge PWM, sawtooth)
- **PMD_CARRIER_WAVE_MODE_1:** PWM mode 1 (center PWM, triangle wave)

uint32_t

CycleTiming: Set PWM cycle, which can be 0x0000 to 0Xffff

***Note:**

If a value less than 0x10 is set, the register assumes 0x10 is set