TOSHIBA

TOSHIBA TX03 Peripheral Driver User Guide (TMPM341)

Ver 1
Sep, 2017

TOSHIBA ELECTRONIC DEVICES & STORAGE CORPORATION

CMDR-M341UG-01E

RESTRICTIONS ON PRODUCT USE

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

1. Introduction

TOSHIBA TX03 Peripheral Driver is a set of drivers for all peripherals on the TOSHIBA TX03 series microcontrollers. TMPM341x(see **Note** below) Peripheral Driver is an important part of TOSHIBA TX03 Peripheral Driver, which are designed for TMPM341 series MCUs.

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

The design goals of TOSHIBA TMPM341x 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 TMPM341x Peripheral Drivers.

/Libraries/ TX03_CMSIS
This folder contains the TMPM341x CMSIS files: device peripheral access layer

/Libraries/TX03_Periph_Driver
This folder contains all the source code of the drivers, the core of TOSHIBA TMPM341x Peripheral Driver.

/Libraries/TX03_Periph_Driver/inc
This folder contains all the header files of TMPM341x Peripheral Drivers for each peripheral.

/Libraries/TX03_Periph_Driver/src
This folder contains all the source files of TMPM341x Peripheral Drivers for each peripheral.

/Project
This folder contains template project and examples for using TMPM341x Peripheral Driver.

/Project/Template
This folder contains template project of TOSHIBA TMPM341x Peripheral Driver.

/Project/Examples
This folder contains a set of examples for using TMPM341x Peripheral Driver

/Utilities/TMPM341-SK
This folder contains the configuration and driver files for hardware resources (e.g. led, key) on TMPM341-SK board.

***Note**: The “TMPM341x” in this document can be TMPM341FD/TMPM341FY.

3. ADC

3.1 Overview

This device contains a 12-bit, sequential-conversion analog/digital converter (ADC) with 15 analog input channels.

The 12-bit AD converter has the following features:
1. Start normal AD conversion and top-priority AD conversion by software activation, 16-bit timer(TMRB) activation or hardware activation with an external trigger input($\overline{\text{ADTRG}}$ pin).
2. AD conversion in 4 different modes:
   - Fixed-channel single conversion mode
   - Channel scan single conversion mode
   - Fixed-channel repeat conversion mode
   - Channel scan repeat conversion mode
3. Normal / Top-priority AD conversion completion interrupt
4. Normal / Top-priority AD conversion completion / busy flag
5. AD monitor function
   When the AD monitor function is enabled, an interrupt is generated if any comparison result is matched.
6. When AD conversion is completed, two types of DMA requests are supported.
7. Standby mode is supported.
8. Output switching monitor function.

The ADC API provides a set of functions for using the TMPM341x ADC modules. It includes ADC channel set, mode set, monitor function set, interrupt set, ADC status read, ADC result value read and so on.

This driver is contained in TX03_Periph_Driver\src\tmpm341_adc.c(*), with TX03_Periph_Driver\inc\tmpm341_adc.h(*) containing the API definitions for use by applications.

3.2 API Functions

3.2.1 Function List

- void ADC_SWReset(void)
- void ADC_SetClk(uint32_t ***Sample_HoldTime***, uint32_t ***Prescaler_Output***)
- void ADC_Start(void)
- void ADC_SetScanMode(FunctionalState ***NewState***)
- void ADC_SetRepeatMode(FunctionalState ***NewState***)
- void ADC_SetINTMode(uint8_t ***INTMode***)
- void ADC_SetInputChannel(uint8_t ***InputChannel***)
- void ADC_SetScanChannel(uint8_t ***StartChannel***, uint8_t ***Range***)
- void ADC_SetVrefCut(uint8_t ***VrefCtrl***)
- void ADC_SetIdleMode(FunctionalState ***NewState***)
- void ADC_SetVref(FunctionalState ***NewState***)
- void ADC_SetInputChannelTop(uint8_t ***TopInputChannel***)
- void ADC_StartTopConvert(void)
- void ADC_SetMonitor(ADC_CMPCRx ***ADCMPx***, FunctionalState ***NewState***)
- void ADC_ConfigMonitor(ADC_CMPCRx ***ADCMPx***, ADC_MonitorTypeDef* ***Monitor***)
- void ADC_SetHWTrg(uint8_t ***HwSource***, FunctionalState ***NewState***)

- void ADC_SetHWTrgTop(uint8_t ***HwSource***, FunctionalState ***NewState***)
- ADC_State ADC_GetConvertState(void)
- ADC_Result ADC_GetConvertResult (uint8_t ***ADREGx***)
- void ADC_SetClkSupply(FunctionalState ***NewState***)
- void ADC_SetDMAReq(uint8_t ***DMAReq***, FunctionalState ***NewState***)

3.2.2 Detailed Description

Functions listed above can be divided into four parts:

1) ADC setting by ADC_SetClk(), ADC_SetScanMode(), ADC_SetRepeatMode(), ADC_SetINTMode(), ADC_SetInputChannel(), ADC_SetScanChannel(), ADC_SetVref(), ADC_SetInputChannelTop(), ADC_SetMonitor(), ADC_ConfigMonitor(), ADC_SetHWTrg(), ADC_SetHWTrgTop().
2) ADC function start by ADC_Start(), ADC_StartTopConvert().
3) ADC state or data read functions by ADC_GetConvertState(), ADC_GetConvertResult().
4) ADC_SWReset(), ADC_SetVrefCut(), ADC_SetIdleMode(), ADC_SetClkSupply() and ADC_SetDMAReq() handle other specified functions.

3.2.3 Function Documentation

3.2.3.1 ADC_SWReset

Software reset ADC.

Prototype:
void
ADC_SWReset(void)

Parameters:
None

Description:
This function will software reset ADC.

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

Return:
None

3.2.3.2 ADC_SetClk

Set ADC sample hold time and prescaler output.

Prototype:
void
ADC_SetClk(uint32_t ***Sample_HoldTime***,
uint32_t ***Prescaler_Output***)

Parameters:
Sample_HoldTime: Select ADC sample hold time.
This parameter can be one of the following values:

- **ADC_CONVERSION_CLK_10:** 10x <ADCLK>
- **ADC_CONVERSION_CLK_20:** 20x <ADCLK>
- **ADC_CONVERSION_CLK_30:** 30x <ADCLK>

- **ADC_CONVERSION_CLK_40:** 40x <ADCLK>
- **ADC_CONVERSION_CLK_80:** 80x <ADCLK>

Prescaler_Output: Select ADC prescaler output(ADCLK).
This parameter can be one of the following values:

- **ADC_FC_DIVIDE_LEVEL_1:** fc
- **ADC_FC_DIVIDE_LEVEL_2:** fc / 2
- **ADC_FC_DIVIDE_LEVEL_4:** fc / 4
- **ADC_FC_DIVIDE_LEVEL_8:** fc / 8

Description:
This function will set ADC sample hold time by ***Sample_HoldTime*** and prescaler output by ***Prescaler_Output***.

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

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

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

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

Return:
None

3.2.3.3 ADC_Start

Start AD conversion.

Prototype:
void

ADC_Start(void)

Parameters:
None

Description:
This function will start normal AD conversion.

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

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

Please refer to the description of **ADC_SetScanMode(), ADC_SetRepeatMode(), ADC_SetInputChannel(), ADC_SetScanChannel()** for the details.

Before starting AD conversion, Vref should be enabled by calling **ADC_SetVref (ENABLE)**, wait for 3 μs during which time the internal reference voltage is stable, and then **ADC_Start()**.

Return:
None

3.2.3.4 ADC_SetScanMode

Enable or disable ADC scan mode.

Prototype:
void
ADC_SetScanMode(FunctionalState ***NewState***)

Parameters:
NewState: Specify ADC scan mode state
This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:
This function will enable or disable ADC scan mode.

Return:
None

3.2.3.5 ADC_SetRepeatMode

Enable or disable ADC repeat mode.

Prototype:
void
ADC_SetRepeatMode(FunctionalState ***NewState***)

Parameters:
NewState: Specify ADC repeat mode state

This parameter can be one of the following values:
ENABLE or **DISABLE**.

Description:
This function will enable or disable ADC repeat mode.

Return:
None

3.2.3.6 ADC_SetINTMode

Set ADC interrupt mode in fixed channel repeat conversion mode.

Prototype:
void
ADC_SetINTMode(uint8_t ***INTMode***)

Parameters:
INTMode: Specify AD conversion interrupt mode.
The parameter can be one of the following values:
- **ADC_INT_SINGLE:** Generate in interrupt once every single conversion.
- **ADC_INT_CONVERSION_2:** Generate interrupt once every 2 conversions.
- **ADC_INT_CONVERSION_3:** Generate interrupt once every 3 conversions.
- **ADC_INT_CONVERSION_4:** Generate interrupt once every 4 conversions.
- **ADC_INT_CONVERSION_5:** Generate interrupt once every 5 conversions.
- **ADC_INT_CONVERSION_6:** Generate interrupt once every 6 conversions.
- **ADC_INT_CONVERSION_7:** Generate interrupt once every 7 conversions.
- **ADC_INT_CONVERSION_8:** Generate interrupt once every 8 conversions.

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

Notes:
This function is valid only in fixed channel repeat conversion mode.
Examples for setting fixed channel repeat conversion mode:
1. **ADC_SetScanMode(DISABLE)**.
2. **ADC_SetRepeatMode(ENABLE)**.

Return:
None

3.2.3.7 ADC_SetInputChannel

Set ADC input channel.

Prototype:
void
ADC_SetInputChannel(uint8_t ***InputChannel***)

Parameters:
InputChannel: Analog input channel.
This parameter can be one of the following values:
ADC_AN_00, ADC_AN_01, ADC_AN_02, ADC_AN_03,
ADC_AN_04, ADC_AN_05, ADC_AN_06, ADC_AN_07,
ADC_AN_08, ADC_AN_09, ADC_AN_10, ADC_AN_11,

ADC_AN_12, ADC_AN_13, ADC_AN_14.

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

Notes:
Only one channel of **ADC_AN_00**~**ADC_AN_14** can be selected as normal conversion input each time.

Return:
None

3.2.3.8 ADC_SetScanChannel

Set ADC scan channel.

Prototype:
void
ADC_SetScanChannel (uint8_t ***StartChannel***, uint8_t ***Range***)

Parameters:
StartChannel: Specify the start channel to be scanned.
The parameter can be one of the following values:
ADC_AN_00, ADC_AN_01, ADC_AN_02, ADC_AN_03,
ADC_AN_04, ADC_AN_05, ADC_AN_06, ADC_AN_07,
ADC_AN_08, ADC_AN_09, ADC_AN_10, ADC_AN_11,
ADC_AN_12, ADC_AN_13, ADC_AN_14.

Range: Specify the range of assignable channel scan value.
The parameter can be **1** to **15**.

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

Notes:
Valid channel scan setting values are shown as follows:

StartChannel	***Range***
ADC_AN_00	1 ~ 15
ADC_AN_01	1 ~ 14
ADC_AN_02	1 ~ 13
ADC_AN_03	1 ~ 12
ADC_AN_04	1 ~ 11
ADC_AN_05	1 ~ 10
ADC_AN_06	1 ~ 9
ADC_AN_07	1 ~ 8
ADC_AN_08	1 ~ 7
ADC_AN_09	1 ~ 6
ADC_AN_10	1 ~ 5
ADC_AN_11	1 ~ 4
ADC_AN_12	1 ~ 3
ADC_AN_13	1 ~ 2
ADC_AN_14	1

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

Return:
None

3.2.3.9 ADC_SetVrefCut

Control AVREFH-AVREFL current.

Prototype:
void
ADC_SetVrefCut(uint8_t ***VrefCtrl***)

Parameters:
VrefCtrl: Specify how to apply AVREFH-AVREFL current.
This parameter can be one of the following values:

- **ADC_APPLY_VREF_IN_CONVERSION:** Apply the current only in conversion.
- **ADC_APPLY_VREF_AT_ANY_TIME:** Apply the current at any time except in RESET.

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

Return:
None

3.2.3.10 ADC_SetIdleMode

Set ADC operation in IDLE mode.

Prototype:
void
ADC_SetIdleMode(FunctionalState ***NewState***)

Parameters:
NewState: Specify ADC operation state in IDLE mode.
This parameter can be one of the following values:
ENABLE or **DISABLE**.

Description:
This function will enable or disable ADC operation state in system IDLE mode.
This function is necessary to be called before system enter IDLE mode.

Return:
None

3.2.3.11 ADC_SetVref

Set ADC Vref application control on or off.

Prototype:
void

ADC_SetVref(FunctionalState ***NewState***)

Parameters:
NewState: Specify AD conversion Vref application control.
This parameter can be one of the following values:
ENABLE or **DISABLE**.

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

Notes:
ADC_SetVref(DISABLE) should be called before system enter standby mode.

Return:
None

3.2.3.12 ADC_SetInputChannelTop

Select ADC top-priority conversion analog input channel.

Prototype:
void
ADC_SetInputChannelTop(uint8_t ***TopInputChannel***)

Parameters:
TopInputChannel: Analog input channel for top-priority conversion.
This parameter can be one of the following values:
ADC_AN_00, ADC_AN_01, ADC_AN_02, ADC_AN_03,
ADC_AN_04, ADC_AN_05, ADC_AN_06, ADC_AN_07,
ADC_AN_08, ADC_AN_09, ADC_AN_10, ADC_AN_11,
ADC_AN_12, ADC_AN_13, ADC_AN_14.

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

Notes:
Only one channel of **ADC_AN_00**~**ADC_AN_14** can be selected as Top-priority conversion input each time.

Return:
None

3.2.3.13 ADC_StartTopConvert

Start top-priority AD conversion.

Prototype:
void
ADC_StartTopConvert(void)

Parameters:
None

Description:

This function will start top-priority AD conversion.

Notes:
This function should be called after **ADC_SetInputChannelTop()**.

Return:
None

3.2.3.14 ADC_SetMonitor

Enable or disable the specified ADC monitor module.

Prototype:
void
ADC_SetMonitor(ADC_CMPCRx ***ADCMPx***,
FunctionalState ***NewState***)

Parameters:
ADCMPx: Select which compare control register will be used.
The parameter can be one of the following values:
- **ADC_CMPCR_0:** ADCMPCR0
- **ADC_CMPCR_1:** ADCMPCR1

NewState: Specify ADC monitor function state.
This parameter can be one of the following values:
ENABLE or **DISABLE**.

Description:
This device has 2 AD monitor modules which are controlled by 2 compare control registers.
This function will specify compare control register by ***ADCMPx*** setting and specify ADC monitor function enable or disable by ***NewState*** setting.

Return:
None

3.2.3.15 ADC_ConfigMonitor

Configure the specified ADC monitor module.

Prototype:
void
ADC_ConfigMonitor(ADC_CMPCRx ***ADCMPx***,
ADC_MonitorTypeDef * ***Monitor***)

Parameters:
ADCMPx: Select which compare control register will be used.
The parameter can be one of the following values:
- **ADC_CMPCR_0:** ADCMPCR0
- **ADC_CMPCR_1:** ADCMPCR1

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

Description:
This device has 2 AD monitor modules which are controlled by 2 compare control registers.
This function will specify compare control register by ***ADCMPx*** setting and specify ADC monitor configuration ***Monitor*** setting.

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

Return:
None

3.2.3.16 ADC_SetHWTrg

Set hardware trigger for normal AD conversion.

Prototype:
void
ADC_SetHWTrg(uint8_t ***HwSource***,
FunctionalState ***NewState***)

Parameters:
HwSource: Hardware source for activating normal AD conversion.
This parameter can be one of the following values:
- **ADC_EXT_TRG:** External trigger
- **ADC_MATCH_TB5RG0:** Match with timer register 0 (TB5RG0)

NewState: Specify state of hardware source for activating normal AD conversion.
This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:
This function will specify hardware trigger source for activating normal AD conversion by ***HwSource*** setting and specify hardware trigger for normal AD conversion enable or disable by ***NewState*** setting.
This function also has relation with TB5 setting.

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

Return:
None

3.2.3.17 ADC_SetHWTrgTop

Set hardware trigger for top-priority AD conversion.

Prototype:
void
ADC_SetHWTrgTop(uint8_t ***HwSource***,
FunctionalState ***NewState***)

Parameters:
HwSource: Hardware source for activating top-priority AD conversion.
This parameter can be one of the following values:

- **ADC_EXT_TRG:** External trigger
- **ADC_MATCH_TB4RG0:** Match with timer register 0 (TB4RG0)

NewState: Specify state of hardware source for activating top-priority AD conversion.
This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:
This function will specify hardware trigger source for activating top-priority AD conversion by ***HwSource*** setting and specify hardware trigger for top-priority AD conversion enable or disable by ***NewState*** setting.
This function also has relation with TB4 setting.

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

Return:
None

3.2.3.18 ADC_GetConvertState

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

Prototype:
WorkState
ADC_GetConvertState(void)

Parameters:
None

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

Return:
AD conversion state:
NormalComplete (Bit 1) means normal AD conversion is complete.
TopComplete (Bit 3) means top-priority AD conversion is complete.

3.2.3.19 ADC_GetConvertResult

Read AD conversion result.

Prototype:
ADC_Result
ADC_GetConvertResult(uint8_t ***ADREGx***)

Parameters:
ADREGx: Select ADC result register.

The parameter can be one of the following values:
ADC_REG_00, **ADC_REG_01**, **ADC_REG_02**, **ADC_REG_03**, **ADC_REG_04**, **ADC_REG_05**, **ADC_REG_06**, **ADC_REG_07**, **ADC_REG_08**, **ADC_REG_09**, **ADC_REG_10**, **ADC_REG_11**, **ADC_REG_12**, **ADC_REG_13**, **ADC_REG_14**, **ADC_REG_SP**.

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

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

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

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

Fixed-channel single mode	
Channel	Storage register
ADC_AN_00	ADC_REG_00
ADC_AN_01	ADC_REG_01
ADC_AN_02	ADC_REG_02
ADC_AN_03	ADC_REG_03
ADC_AN_04	ADC_REG_04
ADC_AN_05	ADC_REG_05
ADC_AN_06	ADC_REG_06
ADC_AN_07	ADC_REG_07
ADC_AN_08	ADC_REG_08
ADC_AN_09	ADC_REG_09
ADC_AN_10	ADC_REG_10
ADC_AN_11	ADC_REG_11
ADC_AN_12	ADC_REG_12
ADC_AN_13	ADC_REG_13
ADC_AN_14	ADC_REG_14

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

Channel scan single mode / repeat mode		
Start channel	Scan channel range	Storage register
ADC_AN_00	15 channels	ADC_REG_00 to ADC_REG_14
ADC_AN_01	14 channels	ADC_REG_01 to ADC_REG_14
ADC_AN_02	13 channels	ADC_REG_02 to ADC_REG_14
ADC_AN_03	12 channels	ADC_REG_03 to ADC_REG_14

ADC_AN_04	11 channels	ADC_REG_04 to ADC_REG_14
ADC_AN_05	10 channels	ADC_REG_05 to ADC_REG_14
ADC_AN_06	9 channels	ADC_REG_06 to ADC_REG_14
ADC_AN_07	8 channels	ADC_REG_07 to ADC_REG_14
ADC_AN_08	7 channels	ADC_REG_08 to ADC_REG_14
ADC_AN_09	6 channels	ADC_REG_09 to ADC_REG_14
ADC_AN_10	5 channels	ADC_REG_10 to ADC_REG_14
ADC_AN_11	4 channels	ADC_REG_11 to ADC_REG_14
ADC_AN_12	3 channels	ADC_REG_12 to ADC_REG_14
ADC_AN_13	2 channels	ADC_REG_13 to ADC_REG_14
ADC_AN_14	1 channel	ADC_REG_14

The ADC mode setting, please refer to relate APIs.
For top-priority AD conversion, the result is stored in ADC_REG_SP.

Return:
AD conversion result:
ADResult (Bit 0 to Bit 11) means AD result value.
Stored (Bit 12) means AD result has been stored.
OverRun (Bit 13) means new AD result is stored before the old one is read.
OutputSwitching (Bit 14) means the output switching flag of AIN port.

3.2.3.20 ADC_SetClkSupply

Enable or disable ADC clock.

Prototype:
void
ADC_SetClkSupply(FunctionalState ***NewState***)

Parameters:
NewState: Specify ADC clock supply state.
The parameter can be one of the following values:
ENABLE or **DISABLE**

Description:
This function will enable or disable ADC clock supply.

Return:
None

3.2.3.21 ADC_SetDMAReq

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

Prototype:
void
ADC_SetDMAReq(uint8_t ***DMAReq***,
FunctionalState ***NewState***)

Parameters:
DMAReq: Specify AD conversion DMA request type.
The parameter can be one of the following values:
- **ADC_DMA_REQ_NORMAL:** normal AD conversion.

- **ADC_DMA_REQ_TOP:** top-priority AD conversion.

NewState: Specify AD conversion DMA activation factor.
This parameter can be one of the following values:
ENABLE or **DISABLE**.

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

Return:
None

3.2.4 Data Structure Description

3.2.4.1 ADC_MonitorTypeDef

Data Fields for this structure:

uint8_t
CmpChannel Select which ADC Result Register to be used,
which can be:
ADC_AN_00, ADC_AN_01, ADC_AN_02, ADC_AN_03,
ADC_AN_04, ADC_AN_05, ADC_AN_06, ADC_AN_07,
ADC_AN_08, ADC_AN_09, ADC_AN_10, ADC_AN_11,
ADC_AN_12, ADC_AN_13, ADC_AN_14.

uint32_t
CmpCnt Define how many valid comparison times will be counted,
which can be **1** to **16**.

ADC_CmpCondition
Condition Conditon to compare AINx with ADCMPn (x= 0 to 14, n = 0 to 1),
which can be:
- **ADC_LARGER_THAN_CMP_REG:** If the value of the conversion result register is bigger than the comparison register 0, an interrupt is generated.
- **ADC_SMALLER_THAN_CMP_REG:** If the value of the conversion result register is smaller than the comparison register 0, an interrupt is generated.

ADC_CmpCntMode
CntMode Mode to compare AINx with ADCMPn (x = 0 to 14, n = 0 to 1),
which can be:
- **ADC_SEQUENCE_CMP_MODE:** Sequence mode.
- **ADC_CUMULATION_CMP_MODE:** Cumulation mode.

uint32_t
CmpValue Comparison value to be set in ADCMP0 or ADCMP1,
which can be **0 to 4095**

3.2.4.2 ADC_State

Data Fields for this structure:
uint32_t
All specifies AD conversion state.

Bit Fields:

uint32_t ***Reserved0*** (Bit 0)	reserved.
uint32_t ***NormalComplete*** (Bit 1)	means normal AD conversion is complete.
uint32_t ***Reserved1*** (Bit 2)	reserved.
uint32_t ***TopComplete*** (Bit 3)	means top-priority AD conversion is complete.
uint32_t ***Reserved2*** (Bit 4 to Bit 31)	reserved.

3.2.4.3 ADC_Result

Data Fields for this structure:

uint32_t
All specifies AD conversion result.

Bit Fields:

uint32_t ***ADResult*** (Bit 0 to Bit 11)	means AD result value.
uint32_t ***Stored*** (Bit 12)	means AD result has been stored.
uint32_t ***OverRun*** (Bit 13)	means new AD result is stored before the old one is read.
uint32_t ***OutputSwitching*** (Bit 14)	means the output switching flag of AIN port.
uint32_t ***Reserved*** (Bit 15 to Bit 31)	reserved.

4. CG

4.1 Overview

The CG API provides a set of functions for using the TMPM341x CG modules as the following:

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

This driver is contained in TX03_Periph_Driver\src\tmpm341_cg.c , with TX03_Periph_Driver\inc\tmpm341_cg.h containing the API definitions for use by applications.

The following symbols fosc, fpll, fc, fgear, fsys, fperiph, ΦT0 are used for kinds of clock in CG. Please refer to the clock system diagram in section “Clock System Block Diagram” of the datasheet for their meaning.

fosc : Clock generated by internal oscillator. Clock input from the X1 and X2 pins..
fpll : Clock multiplied by 8 or 16 by PLL.
fc : Clock specified by CGPLLSEL<PLLSEL> (high-speed clock).
fgear : Clock specified by CGSYSCR<GEAR[2:0]>.
fsys : Clock specified by same as fgear clock (system clock).
fperiph : Clock specified by CGSYSCR<FPSEL>.
ΦT0 : Clock specified by CGSYSCR<PRCK[2:0]> (prescaler clock).

4.2 API Functions

4.2.1 Function List

- void CG_SetFgearLevel(CG_DivideLevel ***DivideFgearFromFc***)
- CG_DivideLevel CG_GetFgearLevel(void)
- void CG_SetPhiT0Src(CG_PhiT0Src ***PhiT0Src***)
- CG_PhiT0Src CG_GetPhiT0Src(void)
- Result CG_SetPhiT0Level(CG_DivideLevel ***DividePhiT0FromFc***)
- CG_DivideLevel CG_GetPhiT0Level(void)
- void CG_SetSCOUTSrc(CG_SCOUTSrc ***Source***)
- CG_SCOUTSrc CG_GetSCOUTSrc(void)
- void CG_SetWarmUpTime(CG_WarmUpSrc ***Source***, uint16_t ***Time***)
- void CG_StartWarmUp(void)
- WorkState CG_GetWarmUpState(void)
- Result CG_SetFPLLValue(CG_FpllValue ***Newville***)
- CG_FpllValue CG_GetFPLLValue(void)
- Result CG_SetPLL(FunctionalState ***NewState***)
- FunctionalState CG_GetPLLState(void)
- Result CG_SetFosc(CG_FoscSrc Source, FunctionalState ***NewState***)
- void CG_SetFoscSrc(CG_FoscSrc ***Source***)
- CG_FoscSrc CG_GetFoscSrc(void)

- FunctionalState CG_GetFoscState(CG_FoscSrc ***Source***)
- void CG_SetSTBYMode(CG_STBYMode ***Mode***)
- CG_STBYMode CG_GetSTBYMode(void)
- void CG_SetPinStateInStop1Mode(FunctionalState ***NewState***)
- FunctionalState CG_GetPinStateInStop1Mode(void)
- void CG_SetPortKeepInStop2Mode(FunctionalState ***NewState***)
- FunctionalState CG_GetPortKeepInStop2Mode(void)
- Result CG_SetFcSrc(CG_FcSrc ***Source***)
- CG_FcSrc CG_GetFcSrc(void)
- void CG_SetFtmrdSrc(CG_FtmrdSrc ***FtmrdSrc***)
- CG_FtmrdSrc CG_GetFtmrdSrc(void)
- void CG_SetTMRDClk(FunctionalState ***NewState***)
- FunctionalState CG_GetTMRDClkState(void)
- void CG_SetProtectCtrl(FunctionalState ***NewState***)
- void CG_SetSTBYReleaseINTSrc(CG_INTSrc ***INTSource***,
  CG_INTActiveState ***ActiveState***,
  FunctionalState ***NewState***)
- CG_INTActiveState CG_GetSTBYReleaseINTState(CG_INTSrc ***INTSource***)
- void CG_ClearINTReq(CG_INTSrc ***INTSource***)
- CG_NMIFactor CG_GetNMIFlag(void)
- CG_ResetFlag CG_GetResetFlag(void)

4.2.2 Detailed Description

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

1) One group of APIs are in charge of clock selection, such as:
   CG_SetFgearLevel(), CG_GetFgearLevel(),CG_SetPhiT0Src(),CG_GetPhiT0Src(), CG_SetPhiT0Level(), CG_GetPhiT0Level(), CG_SetSCOUTSrc(), CG_GetSCOUTSrc(), CG_SetWarmUpTime(), CG_StartWarmUp(), CG_GetWarmUpState(),CG_SetFPLLValue(), CG_GetFPLLValue(),CG_SetPLL(), CG_GetPLLState(), CG_SetFosc(),CG_SetFoscSrc(), CG_GetFoscSrc(), CG_GetFoscState(),CG_SetFcSrc(),CG_GetFcSrc(),CG_SetFtmrdSrc(),CG_GetFtmrdSrc(),CG_SetTMRDClk(),CG_GetTMRDClkState(),CG_SetProtectCtrl().

2) The 2nd group of APIs handle settings of standby modes:
   CG_SetSTBYMode(), CG_GetSTBYMode( ), CG_SetPinStateInStop1Mode(), CG_GetPinStateInStop1Mode(), CG_SetPortKeepInStop2Mode(), CG_GetPortKeepInStop2Mode().

3) The other APIs handle settings of interrupts:
   CG_SetSTBYReleaseINTSrc(), CG_GetSTBYReleaseINTState(), CG_ClearINTReq(), CG_GetNMIFlag(), CG_GetResetFlag(),

4.2.3 Function Documentation

4.2.3.1 CG_SetFgearLevel

Set the dividing level between clock fgear and fc.

Prototype:
void
CG_SetFgearLevel(CG_DivideLevel ***DivideFgearFromFc***)

Parameters:
DivideFgearFromFc: the divide level between fgear and fc
The value could be the following values:

- **CG_DIVIDE_1**: fgear = fc

- **CG_DIVIDE_2**: fgear = fc/2
- **CG_DIVIDE_4**: 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

Set fperiph for PhiT0.

Prototype:
void
CG_SetPhiT0Src(CG_PhiT0Src ***PhiT0Src***)

Parameters:
PhiT0Src: Select PhiT0 source.
This parameter can be one of the following values:
- **CG_PHIT0_SRC_FGEAR** means PhiT0 source is fgear.
- **CG_PHIT0_SRC_FC** means PhiT0 source is fc.

Description:
This function selects the source for PhiT0.

Return:
None

4.2.3.4 CG_GetPhiT0Src

Get the PhiT0 source.

Prototype:
CG_PhiT0Src
CG_GetPhiT0Src (void)

Parameters:
None

Description:
This function will get the PhiT0 source.

Return:
CG_PHIT0_SRC_FGEAR means PhiT0 source is fgear.
CG_PHIT0_SRC_FC means PhiT0 source is fc.

4.2.3.5 CG_SetPhiT0Level

Set the dividing level between PhiT0 (ΦT0) and fc.

Prototype:
Result
CG_SetPhiT0Level (CG_DivideLevel ***DividePhiT0FromFc***)

Parameters:
DividePhiT0FromFc: divide level between PhiT0(ΦT0) and fc.
This parameter can be one of the following values:

- **CG_DIVIDE_1**: ΦT0 = fc
- **CG_DIVIDE_2**: ΦT0 = fc/2
- **CG_DIVIDE_4**: ΦT0 = fc/4
- **CG_DIVIDE_8**: ΦT0 = fc/8
- **CG_DIVIDE_16**: ΦT0 = fc/16
- **CG_DIVIDE_32**: ΦT0 = fc/32
- **CG_DIVIDE_64**: ΦT0 = fc/64
- **CG_DIVIDE_128**: ΦT0 = fc/128
- **CG_DIVIDE_256**: ΦT0 = fc/256
- **CG_DIVIDE_512**: ΦT0 = 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 ΦT0 and fc.

Prototype:
CG_DivideLevel
CG_GetPhiT0Level(void)

Parameters:
None

Description:
This function will get the dividing level of prescaler clock.
If the value “Reserved” is read from the register, the API will return **CG_DIVIDE_UNKNOWN**.

Return:
Dividing level between clock ΦT0 and fc, the value will be one of the following:
CG_DIVIDE_1: ΦT0 = fc
CG_DIVIDE_2: ΦT0 = fc/2
CG_DIVIDE_4: ΦT0 = fc/4
CG_DIVIDE_8: ΦT0 = fc/8
CG_DIVIDE_16: ΦT0 = fc/16
CG_DIVIDE_32: ΦT0 = fc/32
CG_DIVIDE_64: ΦT0 = fc/64
CG_DIVIDE_128 : ΦT0 = fc/128
CG_DIVIDE_256 : ΦT0 = fc/256
CG_DIVIDE_512 : ΦT0 = 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_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 ΦT0.

Description:
This function will set the clock source of SCOUT output.

Return:
None

4.2.3.8 CG_GetSCOUTSrc

Get the clock source of SCOUT output.

Prototype:
SCOUTSrc
CG_GetSCOUTSrc(void)

Parameters:
None

Description:
This function will get the clock source of SCOUT output.

Return:
The clock source of SCOUT output:

CG_SCOUT_SRC_HALF_FSYS: SCOUT source is set to fsys/2
CG_SCOUT_SRC_FSYS: SCOUT source is fsys
CG_SCOUT_SRC_PHIT0: SCOUT source is ΦT0
CG_SCOUT_SRC_ UNKNOWN: Invalid data is read.

4.2.3.9 CG_SetWarmUpTime

Set the warm up time.

Prototype:
void
CG_SetWarmUpTime (CG_WarmUpSrc ***Source***,
uint16_t ***Time***)

Parameters:
Source: select source of warm-up counter.
- **CG_WARM_UP_SRC_OSC_EXT**: external clock is selected as timer source,
- **CG_WARM_UP_SRC_OSC_INT**: internal clock is selected as timer source,

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

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

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

Example of calculating register value for warm-up time:

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

Return:
None.

4.2.3.10 CG_StartWarmUp

Start operation of warm up timer for oscillator.

Prototype:
void
CG_StartWarmUp (void)

Parameters:
None

Description:
This function will start the warm up timer.

Return:
None

4.2.3.11 CG_GetWarmUpState

Check whether warm up is completed or not.

Prototype:
WorkState
CG_GetWarmUpState (void)

Parameters:
None

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

Example of using warm-up timer:

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

Return:
Warm up state:
DONE: means warm-up operation is finished.
BUSY: means warm-up operation is in progress.

4.2.3.12 CG_SetFPLLValue

Set PLL multiplying value

Prototype:
Result
CG_SetFPLLValue(CG_FpllValue ***NewValue***)

Parameters:
NewValue:
- **CG_FPLL_MULTIPLY_8**: 8 multiplying value is selected.
- **CG_FPLL_MULTIPLY_16**: 16 multiplying value is selected.

Description:
This function sets PLL multiplying value.

Return:
SUCCESS: operation is finished successfully.
ERROR: operation is not done.

4.2.3.13 CG_GetFPLLValue

Get the value of PLL setting.

Prototype:
CG_FpllValue
CG_GetFPLLValue(void)

Parameters:
None

Description:
This function will get the PLL multiplying value.

If the value “Reserved” is read from the register, the API will return **CG_FPLL_MULTIPLY_UNKNOWN**.

Return:
The source of PLL multiplying value
CG_FPLL_MULTIPLY_8: 8 multiplying is selected.
CG_FPLL_MULTIPLY_16: 16 multiplying is selected.
CG_FPLL_MULTIPLY_UNKNOWN: invalid data is read.

4.2.3.14 CG_SetPLL

Enable or disable the PLL circuit.

Prototype:
Result
CG_SetPLL(FunctionalState ***NewState***)

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

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

Return:
SUCCESS: operation is finished successfully.
ERROR: operation is not done.

4.2.3.15 CG_GetPLLState

Get the state of PLL circuit.

Prototype:
FunctionalState
CG_GetPLLState(void)

Parameters:
None

Description:
This function will get the state of PLL circuit.

Return:
The state of PLL
ENABLE: PLL is enabled.
DISABLE: PLL is disabled.

4.2.3.16 CG_SetFosc

Enable or disable high-speed oscillator (fosc).

Prototype:
Result
CG_SetFosc(CG_FoscSrc ***Source***,
FunctionalState ***NewState***)

Parameters:
Source: select clock source of fosc.
This parameter can be one of the following values:
- **CG_FOSC_OSC_EXT**: external high-speed oscillator is selected,
- **CG_FOSC_OSC_INT**: internal high-speed oscillator is selected.

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

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

Return:
SUCCESS: operation is finished successfully.
ERROR: operation is not done.

4.2.3.17 CG_SetFoscSrc

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

Prototype:
void
CG_SetFoscSrc(CG_FoscSrc Source)

Parameters:
Source: select source for fosc.
This parameter can be one of the following values:
- **CG_FOSC_OSC_EXT**: external high-speed oscillator is selected,
- **CG_FOSC_CLKIN_EXT**: external clock input is selected.
- **CG_FOSC_OSC_INT**: internal high-speed oscillator is selected.

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

Return:
None

4.2.3.18 CG_GetFoscSrc

Get the source of the high-speed oscillator.

Prototype:
CG_FoscSrc
CG_GetFoscSrc(void)

Parameters:
None

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

Return:
The source of fosc
CG_FOSC_OSC_EXT: external high-speed oscillator is selected,
CG_FOSC_CLKIN_EXT: external clock input is selected.

CG_FOSC_OSC_INT: internal high-speed oscillator is selected.

4.2.3.19 CG_GetFoscState

Get the state of the high-speed oscillator.

Prototype:
FunctionalState
CG_GetFoscState(CG_FoscSrc Source)

Parameters:
Source: select source for fosc.
- **CG_FOSC_OSC_EXT**: external high-speed oscillator is selected,
- **CG_FOSC_OSC_INT**: internal high-speed oscillator is selected.

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

Return:
The state of fosc
ENABLE: fosc is enabled.
DISABLE: fosc is disabled.

4.2.3.20 CG_SetSTBYMode

Set the standby mode.

Prototype:
void
CG_SetSTBYMode(CG_STBYMode ***Mode***)

Parameters:
Mode: the low power consumption mode, the description of each value is as the following:
- **CG_STBY_MODE_STOP1**: STOP1 mode. All the internal circuits including the internal oscillator are brought to a stop.
- **CG_STBY_MODE_STOP2**: STOP2 mode. This mode halts main voltage supply, retaining some function operation.
- **CG_STBY_MODE_IDLE**: IDLE mode. Only CPU stop in this mode.

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

Return:
None

4.2.3.21 CG_GetSTBYMode

Get the standby mode.

Prototype:
CG_STBYMode
CG_GetSTBYMode (void)

Parameters:
None

Description:
This function will get the setting of standby mode.
If the value "Reserved" is read, "**CG_STBY_MODE_UNKNOWN**" will be returned.

Return:
The low power mode:
CG_STBY_MODE_STOP1: STOP1 mode.
CG_STBY_MODE_STOP2: STOP2 mode
CG_STBY_MODE_IDLE: IDLE mode
CG_STBY_MODE_UNKNOWN: Invalid data is read.

4.2.3.22 CG_SetPinStateInStop1Mode

Set pin status in stop1 mode

Prototype:
void
CG_SetPinStateInStop1Mode(FunctionalState ***NewState***)

Parameters:
NewState:
- **DISABLE:** <DRVE>=0
- **ENABLE:** <DRVE>=1

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

Description:
This function sets pin status in stop1 mode.

Return:
None

4.2.3.23 CG_GetPinStateInStop1Mode

Get pin status in stop1 mode

Prototype:
FunctionalState
CG_GetPinStateInStop1Mode(void)

Parameters:
None

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

Return:
The pin state in stop1 mode
DISABLE: <DRVE>=0
ENABLE: <DRVE>=1

4.2.3.24 CG_SetPortKeepInStop2Mode

Enables or disables to keep IO control signal in stop2 mode

Prototype:
void
CG_SetPortKeepInStop2Mode(FunctionalState ***NewState***)

Parameters:
NewState:
- **DISABLE:** <PTKEEP>=0
- **ENABLE:** <PTKEEP>=1

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

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

Return:
None

4.2.3.25 CG_GetPortKeepInStop2Mode

Get the pin status in stop mode

Prototype:
FunctionalState
CG_GetPinStateInStopMode (void)

Parameters:
None

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

Return:
The port keeps in stop2 mode
DISABLE: <PTKEEP>=0
ENABLE: <PTKEEP>=1

4.2.3.26 CG_SetFcSrc

Set the clock source of fc

Prototype:
Result
CG_SetFcSrc(CG_FcSrc ***Source***)

Parameters:
Source: the source for fc
This parameter can be one of the following values:
- **CG_FC_SRC_FOSC** : fc source will be set to fosc
- **CG_FC_SRC_ QUARTER_FPLL**: fc source will be set to fpll/4

Description:
This function will set the clock source of fc.

Return:
SUCCESS: set clock source for fc successfully

ERROR: clock source of fc is not changed.

4.2.3.27 CG_GetFcSrc

Get the clock source of fc.

Prototype:
CG_FcSrc
CG_GetFosc (void)

Parameters:
None

Description:
This function will get the clock source of fc.

Return:
The clock source of fc
The value returned can be one of the following values:
CG_FC_SRC_FOSC: fc source is set to fosc.
CG_FC_SRC_ QUARTER_FPLL: fc source is set to fpll/4.

4.2.3.28 CG_SetFtmrdSrc

Set the source of high-resolution PPG clock (ftmrd).

Prototype:
void
CG_SetFtmrdSrc(CG_FtmrdSrc ***FtmrdSrc***)

Parameters:
FtmrdSrc: the source for ftmrd
This parameter can be one of the following values:
- **CG_FTMRD_SRC_FPLL** : ftmrd source will be set to fpll
- **CG_FTMRD_SRC_HALF_FPLL**: ftmrd source will be set to fpll/2
- **CG_FTMRD_SRC_QUARTER_FPLL**: ftmrd source will be set to fpll/4

Description:
This function selects the source for ftmrd.

Return:
None

4.2.3.29 CG_GetFtmrdSrc

Get the source of high-resolution PPG clock (ftmrd).

Prototype:
CG_FtmrdSrc
CG_GetFtmrdSrc(void)

Parameters:
None

Description:
This function will get the source of ftmrd.

If the value “Reserved” is read, “**CG_FTMRD_SRC_UNKNOWN**” will be returned.

Return:
The source for ftmrd
The value returned can be one of the following values:
CG_FTMRD_SRC_FPLL : ftmrd source will be set to fpll.
CG_FTMRD_SRC_HALF_FPLL: ftmrd source will be set to fpll/2.
CG_FTMRD_SRC_QUARTER_FPLL: ftmrd source will be set to fpll/4.
CG_FTMRD_SRC_UNKNOWN: Invalid data is read.

4.2.3.30 CG_SetTMRDClk

Enable or disable to provide TMRD clock.

Prototype:
void
CG_SetTMRDClk(FunctionalState ***NewState***)

Parameters:
NewState:
- **DISABLE:** < TMRDCLKEN>=0 Provides TMRDCLK to TMRD will be set to Stop
- **ENABLE:** < TMRDCLKEN>=1 Provides TMRDCLK to TMRD will be set to Setup

Description:
This function enables or disables to provide TMRD clock.

Return:
None.

4.2.3.31 CG_GetTMRDClkState

Get the state of TMRD clock.

Prototype:
FunctionalState
CG_GetTMRDClkState(void)

Parameters:
None

Description:
This function will get the status of TMRD clock..

Return:
The status of TMRD clock
The value returned can be one of the following values:
DISABLE: < TMRDCLKEN>=0 Provides TMRDCLK to TMRD will be set to Stop
ENABLE: < TMRDCLKEN>=1 Provides TMRDCLK to TMRD will be set to Setup

4.2.3.32 CG_SetProtectCtrl

Enable or disable to protect CG registers.

Prototype:
void
CG_SetProtectCtrl(FunctionalState ***NewState***)

Parameters:
NewState
- **DISABLE:** < CGPROTECT>= Except 0xC1 Register write disable
- **ENABLE:** < CGPROTECT>=0xC1 Register write enable

Description:
This function enables or disables CG registers to be written.

Return:
None

4.2.3.33 CG_SetSTBYReleaseINTSrc

Set the INT source for releasing low power mode.

Prototype:
void
CG_SetSTBYReleaseINTSrc (CG_INTSrc ***INTSource***,
CG_INTActiveState ***ActiveState***,
FunctionalState ***NewState***)

Parameters:
INTSource: select the INT source for releasing standby mode
This parameter can be one of the following values:
- **CG_INT_SRC_0** : INT0
- **CG_INT_SRC_1** : INT1
- **CG_INT_SRC_2** : INT2
- **CG_INT_SRC_3** : INT3
- **CG_INT_SRC_4** : INT4
- **CG_INT_SRC_5** : INT5
- **CG_INT_SRC_6** : INT6
- **CG_INT_SRC_7** : INT7
- **CG_INT_SRC_ PHT_00**: 16-bit PHC compare interrupt 00
- **CG_INT_SRC_ PHT_01**: 16-bit PHC compare interrupt 01
- **CG_INT_SRC_ PHT_10**: 16-bit PHC compare interrupt 10
- **CG_INT_SRC_ PHT_11**: 16-bit PHC compare interrupt 11
- **CG_INT_SRC_ PHT_20**: 16-bit PHC compare interrupt 20
- **CG_INT_SRC_ PHT_21**: 16-bit PHC compare interrupt 21
- **CG_INT_SRC_ PHT_30**: 16-bit PHC compare interrupt 30
- **CG_INT_SRC_ PHT_31**: 16-bit PHC compare interrupt 31
- **CG_INT_SRC_ PHEVRY_0**: 16-bit PHC every count interrupt 0.
- **CG_INT_SRC_ PHEVRY_1**: 16-bit PHC every count interrupt 1.
- **CG_INT_SRC_ PHEVRY_2**: 16-bit PHC every count interrupt 2.
- **CG_INT_SRC_ PHEVRY_3**: 16-bit PHC every count interrupt 3
- **CG_INT_SRC_8** : INT8
- **CG_INT_SRC_9** : INT9
- **CG_INT_SRC_A** : INTA
- **CG_INT_SRC_B** : INTB

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_RISGING**: 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 INTPHT00 to INTPHT31 and INTPHEVRY0 to INTPHEVRY0, only CG_INT_ACTIVE_STATE_RISING can be set.
For INT0 to INTB, one of the following values can be set.
CG_INT_ACTIVE_STATE_L, CG_INT_ACTIVE_STATE_H,
CG_INT_ACTIVE_STATE_FALLING,
CG_INT_ACTIVE_STATE_RISING or
CG_INT_ACTIVE_STATE_BOTH_EDGES

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_PHT_00, CG_INT_SRC_PHT_01,
CG_INT_SRC_PHT_10, CG_INT_SRC_PHT_11,
CG_INT_SRC_PHT_20, CG_INT_SRC_PHT_21,
CG_INT_SRC_PHT_30, CG_INT_SRC_PHT_31,
CG_INT_SRC_PHEVRY_0, CG_INT_SRC_PHEVRY_1,
CG_INT_SRC_PHEVRY_2, CG_INT_SRC_PHEVRY_3,
CG_INT_SRC_8, CG_INT_SRC_9, CG_INT_SRC_A, CG_INT_SRC_B.

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

Return:
Active state of the input INT
The value returned can be one of the following values:
CG_INT_ACTIVE_STATE_FALLING: active on falling edge
CG_INT_ACTIVE_STATE_RISING: active on rising edge
CG_INT_ACTIVE_STATE_BOTH_EDGES: active on both edges
CG_INT_ACTIVE_STATE_INVALID: invalid

4.2.3.35 CG_ClearINTReq

Clears the input INT request.

Prototype:
void
CG_ClearINTReq(CG_INTSrc ***INTSource***)

Parameters:
INTSource: select the release INT source.
This parameter can be one of the following values:
CG_INT_SRC_0, CG_INT_SRC_1, CG_INT_SRC_2, CG_INT_SRC_3,
CG_INT_SRC_4, CG_INT_SRC_5, CG_INT_SRC_6, CG_INT_SRC_7,
CG_INT_SRC_PHT_00, CG_INT_SRC_PHT_01,
CG_INT_SRC_PHT_10, CG_INT_SRC_PHT_11,
CG_INT_SRC_PHT_20, CG_INT_SRC_PHT_21,
CG_INT_SRC_PHT_30, CG_INT_SRC_PHT_31,
CG_INT_SRC_PHEVRY_0, CG_INT_SRC_PHEVRY_1,
CG_INT_SRC_PHEVRY_2, CG_INT_SRC_PHEVRY_3,
CG_INT_SRC_8, CG_INT_SRC_9, CG_INT_SRC_A, CG_INT_SRC_B.
Description:
This function will clear the INT request for releasing standby mode.

Return:
None

4.2.3.36 CG_GetNMIFlag

Get the NMI flag that shows who triggered NMI

Prototype:
CG_NMI_Factor
CG_GetNMIFlag (void)

Parameters:
None

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

Return:
NMI value:
WDT (Bit 0) means generated from WDT.
NMIPin(Bit 1) means generated from NMI pin.

4.2.3.37 CG_GetResetFlag

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

Prototype:
CG_ResetFlag
CG_GetResetFlag(void)

Parameters:
None

Description:
This function gets the reset flag showing what triggered reset.

Return:
Reset flag:
ResetPin (Bit 0) means reset from Reset pin.
Reserved(Bit1) means reserved.
WDTReset (Bit 2) means reset from WDT.
STOP2Reset(Bit3) means reset flag by STOP2 mode release
DebugReset (Bit 4) means reset from SYSRESETREQ.
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
NMIPin(Bit 1) means generated from NMI pin..

4.2.4.2 CG_ResetFlag

Data Fields:
uint32_t
All specifies CG reset source.

Bit Fields:
uint32_t
ResetPin(Bit 0) means reset from Reset pin.
uint32_t
Reserved(Bit 1) means reserved.
uint32_t
WDTReset(Bit 2) means reset from WDT.
uint32_t
STOP2Reset(Bit 3) means reset flag by STOP2 mode release.
uint32_t
DebugReset(Bit 4) means reset from SYSRESETREQ..
uint32_t
OFDReset(Bit 5) means reset from OFD.

5. DAC

5.1 Overview

Features
- A high-resolution, 10-bit DA converter is built into each of two channels.
- Each channel is provided with a full range buffer amplifier.
- Built in AVREFH cut function.
- Built in Amplifier stops function.
- Input variation / output variation settling time : 100μs
- Circuit activated time : 10μs (Settling time not included)

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm341_dac.c, with /Libraries/TX03_Periph_Driver/inc/tmpm341_dac.h containing the macros, data types, structures and API definitions for use by applications.

5.2 API Functions

5.2.1 Function List
- void DAC_SetOutputCode(TSB_DA_TypeDef * ***DACx***,uint16_t ***OutputCode***);
- void DAC_Start(TSB_DA_TypeDef * ***DACx***);
- void DAC_Stop(TSB_DA_TypeDef * ***DACx***);

5.2.2 Detailed Description
Functions listed above can be divided into two parts:
1) Configure the DAC channel and set output value
   DAC_SetOutputCode()
2) Start and stop control
   DAC_Start(), DAC_Stop()

5.2.3 Function Documentation
Note: in all of the following APIs, unless otherwise specified, the parameter:
"TSB_DA_TypeDef * ***DACx***" can be one of the following values:
TSB_DA0, TSB_ DA1

5.2.3.1 DAC_SetOutputCode
Set output code for the specified DAC channel.

Prototype:
void
DAC_SetOutputCode(TSB_DA_TypeDef * ***DACx***,
uint16_t ***OutputCode***)

Parameters:
DACx is the specified DAC channel.
OutputCode is the value which will be converted to analog output. And its bit width is 10, so its max value is 0x3ff.

Description:

This function sets the output code for the specified DAC channel, which will be converted to analog output.

Return:
None

5.2.3.2 DAC_Start

Start the operation of the specified DAC channel

Prototype:
void
DAC_Start(TSB_DA_TypeDef * ***DACx***);

Parameters:
DACx is the specified DAC channel.

Description:
This function will start the Digital to Analog converting operation of the specified DAC channel.

Return:
None

5.2.3.3 DAC_Stop

Stop the operation of the specified DAC channel

Prototype:
void
DAC_Stop(TSB_DA_TypeDef * ***DACx***);

Parameters:
DACx is the specified DAC channel.

Description:
This function will stop the Digital to Analog converting operation of the specified DAC channel.

Return:
None

5.2.4 Data Structure Description

None

6. DMAC

6.1 Overview

TOSHIBA TMPM341x has two DMA controllers (UNITA, UNITB) controlled by DMA request select registers, and DMA controller has two channels. Each channel can operate in one of four transferring types (memory to memory, memory to peripheral, peripheral to memory, peripheral to peripheral). The priority of UNITA is higher than UNITB. And the priority of UNITA (UNITB) channel 0 is higher than UNITA (UNITB) channel 1.

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

This driver is contained in \Libraries\TX03_Periph_Driver\src\tmpm341_dmac.c, with \Libraries/TX03_Periph_Driver\inc\tmpm341_dmac.h containing the API definitions for use by applications.

6.2 API Functions

6.2.1 Function List

- void DMAC_Enable(TSB_DMAC_TypeDef * ***DMACx***);
- void DMAC_Disable(TSB_DMAC_TypeDef * ***DMACx***);
- DMAC_INTReq DMAC_GetINTReq(TSB_DMAC_TypeDef * ***DMACx***);
- DMAC_TxINTReq DMAC_GetTxINTReq(TSB_DMAC_TypeDef * ***DMACx***, DMAC_Channel ***Chx***);
- void DMAC_ClearTxINTReq(TSB_DMAC_TypeDef * ***DMACx***, DMAC_Channel ***Chx***, DMAC_INTSrc ***INTSource***);
- DMAC_TxINTReq DMAC_GetRawTxINTReq(TSB_DMAC_TypeDef * ***DMACx***, DMAC_Channel ***Chx***);
- WorkState DMAC_GetChannelTxState(TSB_DMAC_TypeDef * ***DMACx***, DMAC_Channel ***Chx***);
- void DMACA_SetSWBurstReq(DMACA_ReqNum ***BurstReq***);
- void DMACB_SetSWBurstReq(DMACB_ReqNum ***BurstReq***);
- DMAC_BurstReqState DMAC_GetSWBurstReqState(TSB_DMAC_TypeDef * ***DMACx***);
- void DMACB_SetSWSingleReq(DMACB_ReqNum ***SingleReq***);
- DMAC_SingleReqState DMAC_GetSWSingleReqState(TSB_DMAC_TypeDef * ***DMACx***);
- void DMAC_SetLinkedList(TSB_DMAC_TypeDef * ***DMACx,*** DMAC_Channel ***Chx***, uint32_t ***LinkedAddr***);
- WorkState DMAC_GetFIFOState(TSB_DMAC_TypeDef * ***DMACx,*** DMAC_Channel ***Chx***);
- void DMAC_SetDMAHalt(TSB_DMAC_TypeDef * ***DMACx,*** DMAC_Channel ***Chx***, FunctionalState ***NewState***);
- void DMAC_SetLockedTx(TSB_DMAC_TypeDef * ***DMACx,*** DMAC_Channel ***Chx***, FunctionalState ***NewState***)

- void DMAC_SetTxINTConfig(TSB_DMAC_TypeDef * ***DMACx,*** DMAC_Channel ***Chx***, DMAC_INTSrc ***INTSource***, FunctionalState ***NewState***);
- void DMAC_SetDMAChannel(TSB_DMAC_TypeDef * ***DMACx,*** DMAC_Channel ***Chx***, FunctionalState ***NewState***);
- void DMAC_Init(TSB_DMAC_TypeDef * ***DMACx*** , DMAC_Channel ***Chx***, DMAC_InitTypeDef * ***InitStruct***);

6.2.2 Detailed Description

Functions listed above can be divided into five parts:

1) The DMAC basic configure are handled by the DMAC_Enable(),DMAC_Disable(),DMAC_SetDMAChannel() and DMAC_Init() functions.
2) To get DMA transfer interrupt state, FIFO or DMA channel state are handled by DMAC_GetINTReq(),DMAC_GetTxINTReq(),DMAC_GetRawTxINTReq(), DMAC_GetChannelTxState() and DMAC_GetFIFOState().
3) To set DMA interrupt and clear DMA interrupt request are handled by DMAC_ClearTxINTReq() and DMAC_SetTxINTConfig().
4) To set DMA software request and get DMA software request are handled by DMACA_SetSWBurstReq(),DMACB_SetSWBurstReq(),DMAC_GetSWBurstReqState(),DMACB_SetSWSingleReq(),DMAC_SetLinkedList and DMAC_GetSWSingleReqState().
5) DMAC_SetDMAHalt () and DMAC_SetLockedTx() handle other specified functions.

6.2.3 Function Documentation

6.2.3.1 DMAC_Enable

Enable the DMA circuit.

Prototype:
void
DMAC_Enable(TSB_DMAC_TypeDef * ***DMACx***);

Parameters:
DMACx is the specified DMA Unit, which can be one of:
- **DMAC_UNIT_A** for DMA UNIT A,
- **DMAC_UNIT_B** for DMA UNIT B.

Description:
This function will enable UNIT A circuit when ***DMACx*** is **DMAC_UNIT_A** and Enable UNIT B circuit when ***DMACx*** is **DMAC_UNIT_B**.

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

Return:
None

6.2.3.2 DMAC_Disable

Disable the DMA circuit.

Prototype:
void

DMAC_Disable(TSB_DMAC_TypeDef * ***DMACx***);

Parameters:
DMACx is the specified DMA Unit, which can be one of:
- **DMAC_UNIT_A** for DMA UNIT A,
- **DMAC_UNIT_B** for DMA UNIT B.

Description:
This function will disable UNIT A circuit when ***DMACx*** is **DMAC_UNIT_A** and Enable UNIT B circuit when ***DMACx*** is **DMAC_UNIT_B**.

Return:
None

6.2.3.3 DMAC_GetINTReq

Get DMA Channel interrupt request state.

Prototype:
DMAC_INTReq
DMAC_GetINTReq(TSB_DMAC_TypeDef * ***DMACx***);

Parameters:
DMACx is the specified DMA Unit, which can be one of:
- **DMAC_UNIT_A** for DMA UNIT A,
- **DMAC_UNIT_B** for DMA UNIT B.

Description:
This function will get UNIT A interrupt request state when ***DMACx*** is **DMAC_UNIT_A** and get UNIT B interrupt request state when ***DMACx*** is **DMAC_UNIT_B**.

Return:
The state of interrupt request

6.2.3.4 DMAC_GetTxINTReq

Get the specified DMA Channel transfer interrupt request state.

Prototype:
DMAC_TxINTReq
DMAC_GetTxINTReq(TSB_DMAC_TypeDef * ***DMACx,***
DMAC_Channel ***Chx***);

Parameters:
DMACx is the specified DMA Unit, which can be one of:
- **DMAC_UNIT_A** for DMA UNIT A,
- **DMAC_UNIT_B** for DMA UNIT B.

Chx is the specified DMA channel, which can be one of:
- **DMAC_CHANNEL_0** for DMA channel 0,
- **DMAC_CHANNEL_1** for DMA channel 1.

Description:

This function will get specified UNIT channel 0 transfer interrupt state when ***Chx*** is **DMAC_CHANNEL_0** and get specified UNIT channel 1 transfer interrupt state when ***Chx*** is **DMAC_CHANNEL_1**.

Return:
The request states of DMA transfer interrupt.
The value returned can be one of the followings:
DMAC_TX_NO_REQ means there is no transfer interrupt request,
DMAC_TX_END_REQ means there is a transfer end interrupt request,
DMAC_TX_ERR_REQ means there is a transfer error interrupt request,
DMAC_TX_REQS means there is more than one interrupt request.

6.2.3.5 DMAC_ClearTxINTReq

Clear the transfer interrupt request.

Prototype:
void
DMAC_ClearTxINTReq(TSB_DMAC_TypeDef * ***DMACx,***
DMAC_Channel ***Chx***,
DMAC_INTSrc ***INTSource***);

Parameters:
DMACx is the specified DMA Unit, which can be one of:
- **DMAC_UNIT_A** for DMA UNIT A,
- **DMAC_UNIT_B** for DMA UNIT B.

Chx is the specified DMA channel, which can be one of:
- **DMAC_CHANNEL_0** for DMA channel 0,
- **DMAC_CHANNEL_1** for DMA channel 1.

INTSource: select the release INT source, which can be one of:
- **DMAC_INT_TX_END** for DMA transfer end interrupt,
- **DMAC_INT_TX_ERR** for DMA transfer error interrupt.

Description:
This function will clear the transfer interrupt request. When ***INTSource*** is **DMAC_INT_TX_END**, this function will clear DMA transfer end interrupt request. When ***INTSource*** is **DMAC_INT_TX_ERR**, this function will clear DMA transfer error interrupt request.

Return:
None

6.2.3.6 DMAC_GetRawTxINTReq

Get the specified DMA Channel transfer raw interrupt request state.

Prototype:
DMAC_TxINTReq
DMAC_GetRawTxINTReq(TSB_DMAC_TypeDef * ***DMACx,***
DMAC_Channel ***Chx***);

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

- **DMAC_UNIT_A** for DMA UNIT A,
- **DMAC_UNIT_B** for DMA UNIT B.

Chx is the specified DMA channel, which can be one of:
- **DMAC_CHANNEL_0** for DMA channel 0,
- **DMAC_CHANNEL_1** for DMA channel 1.

Description:
This function will get specified UNIT channel 0 transfer raw interrupt state when ***Chx*** is **DMAC_CHANNEL_0** and get specified UNIT channel 1 transfer raw interrupt state when ***Chx*** is **DMAC_CHANNEL_1**.

Return:
The request states of DMA transfer raw interrupt.
The value returned can be one of the followings:
DMAC_TX_NO_REQ means there is no transfer raw interrupt request,
DMAC_TX_END_REQ means there is a transfer end interrupt request,
DMAC_TX_ERR_REQ means there is a transfer error interrupt request,
DMAC_TX_REQS means there is more than one interrupt request.

6.2.3.7 DMAC_GetChannelTxState

Get the specified DMA Channel transfer state.

Prototype:
```
WorkState
DMAC_GetChannelTxState(TSB_DMAC_TypeDef * DMACx,
                       DMAC_Channel Chx);
```

Parameters:
DMACx is the specified DMA Unit, which can be one of:
- **DMAC_UNIT_A** for DMA UNIT A,
- **DMAC_UNIT_B** for DMA UNIT B.

Chx is the specified DMA channel, which can be one of:
- **DMAC_CHANNEL_0** for DMA channel 0,
- **DMAC_CHANNEL_1** for DMA channel 1.

Description:
This function will get specified UNIT channel 0 transfer state when ***Chx*** is **DMAC_CHANNEL_0**, and get specified UNIT channel 1 transfer state when ***Chx*** is **DMAC_CHANNEL_1**. If return value is **BUSY**, meaning the DMA channel is enabled and data transmission is in progress. If return value is **DONE**, meaning DMA channel is disabled and data transmission is complete.

Return:
The DMA transfer status.
The value returned can be one of the followings:
BUSY or **DONE**

6.2.3.8 DMACA_SetSWBurstReq

Set DMACA burst transfer requests by software.

Prototype:

void
DMACA_SetSWBurstReq(DMACA_ReqNum ***BurstReq***);

Parameters:
BurstReq: Select burst request number, which can be one of:
- **DMACA_SIO0_UART0_RX** for SIO0/UART0 Reception,
- **DMACA_SIO0_UART0_TX** for SIO0/UART0 Transmission,
- **DMACA_SIO2_UART2_RX** for SIO2/UART2 Reception,
- **DMACA_SIO2_UART2_TX** for SIO2/UART2 Transmission,
- **DMACA_SIO4_UART4_RX** for SIO4/UART4 Reception,
- **DMACA_SIO4_UART4_TX** for SIO4/UART4 Transmission,
- **DMACA_PULSE_CNT2** Two-phase pulse counter2 every count,
- **DMACA_PULSE_CNT3** Two-phase pulse counter3 every count,
- **DMACA_TMRB8_CMP_MATCH** for TMRB8 compare match,
- **DMACA_TMRB9_CMP_MATCH** for TMRB9compare match,
- **DMACA_TMRB0_CAPTURE0** for TMRB0 input capture 0,
- **DMACA_TMRB4_CAPTURE0** for TMRB4 input capture 0,
- **DMACA_TMRB4_CAPTURE1** for TMRB4 input capture 1
- **DMACA_TMRB5_CAPTURE0** for TMRB5input capture 0,
- **DMACA_TMRB5_CAPTURE1** for TMRB5input capture 1
- **DMACA_HIGH_PRIORITY_ADC X** Highest priority AD Conversion End,

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

Return:
None

6.2.3.9 DMACB_SetSWBurstReq

Set DMACB burst transfer requests by software.

Prototype:
void
DMACB_SetSWBurstReq(DMACB_ReqNum ***BurstReq***);

Parameters:
BurstReq: Select burst request number, which can be one of:
- **DMACB_TMRD00_ CMP_MATCH** for TMRD00 compare match,
- **DMACB_TMRD10_ CMP_MATCH** for TMRD10 compare match,
- **DMACB_PULSE_CNT0** Two-phase pulse counter0 every count,
- **DMACB_PULSE_CNT1** Two-phase pulse counter1 every count,
- **DMACB_TMRB6_CMP_MATCH** for TMRB6 compare match,
- **DMACB_TMRB7_CMP_MATCH** for TMRB7 compare match,
- **DMACB_TMRB0_CAPTURE1** for TMRB0 input capture 1
- **DMACB_TMRB2_CAPTURE0** for TMRB2 input capture 0,
- **DMACB_TMRB2_CAPTURE1** for TMRB2 input capture 1
- **DMACB_TMRB3_CAPTURE0** for TMRB3 input capture 0,
- **DMACB_TMRB3_CAPTURE1** for TMRB3 input capture 1,
- **DMACB_TMRB6_CAPTURE0** for TMRB6 input capture 0,
- **DMACB_TMRB6_CAPTURE1** for TMRB6 input capture 1,
- **DMACB_NORMAL_ADC** for Normal AD Conversion End,
- **DMACB_SSP_TX** for SSP Transmission,
- **DMACB_SSP_RX** for SSP Reception,

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

Return:
None

6.2.3.10 DMAC_GetSWBurstReqState

Get DMA software burst request state.

Prototype:
DMAC_BurstReqState
DMAC_GetSWBurstReqState(TSB_DMAC_TypeDef * ***DMACx***);

Parameters:
DMACx is the specified DMA Unit, which can be one of:
- **DMAC_UNIT_A** for DMA UNIT A,
- **DMAC_UNIT_B** for DMA UNIT B.

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

Return:
The DMA burst request status.

6.2.3.11 DMACB_SetSWSingleReq

Set DMA single transfer requests by software.

Prototype:
void
DMACB_SetSWSingleReq(DMACB_ReqNum ***SingleReq***);

Parameters:
SingleReq: Select burst request number, which can be:
- **DMACB_SSP_TX** for SSP Transmission,
- **DMACB_SSP_RX** for SSP Reception,

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

Return:
None

6.2.3.12 DMAC_GetSWSingleReqState

Get DMA software single request state.

Prototype:
DMAC_SingleReqState

DMAC_GetSWSingleReqState(TSB_DMAC_TypeDef * ***DMACx***);

Parameters:
DMACx is the specified DMA Unit, which can be one of:
- **DMAC_UNIT_A** for DMA UNIT A,
- **DMAC_UNIT_B** for DMA UNIT B.

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

Return:
The DMA single request status.

6.2.3.13 DMAC_SetLinkedList

Set specified DMA Channel Linked List Item Register.

Prototype:
```
void
DMAC_SetLinkedList(TSB_DMAC_TypeDef * DMACx,
                   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 0xFFFFFFF0.

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

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

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

The items that can be set with Linked List are configured with the following 4 words:
1) DMACCxSrcAddr
2) DMACCxDestAddr
3) DMACCxLLI
4) DMACCxControl

Return:
None

6.2.3.14 DMAC_GetFIFOState

Indicates whether data is present in the channel FIFO

Prototype:
WorkState
DMAC_GetFIFOState(TSB_DMAC_TypeDef * ***DMACx,***
DMAC_Channel ***Chx***);

Parameters:
DMACx is the specified DMA Unit, which can be one of:
- **DMAC_UNIT_A** for DMA UNIT A,
- **DMAC_UNIT_B** for DMA UNIT B.

Chx is the specified DMA channel, which can be one of:
- **DMAC_CHANNEL_0** for DMA channel 0,
- **DMAC_CHANNEL_1** for DMA channel 1.

Description:
This function will get specified UNIT channel 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**

6.2.3.15 DMAC_SetDMAHalt

Set whether ignore DMA request.

Prototype:
void
DMAC_SetDMAHalt(TSB_DMAC_TypeDef * ***DMACx,***
DMAC_Channel ***Chx***,
FunctionalState ***NewState***);

Parameters:
DMACx is the specified DMA Unit, which can be one of:
- **DMAC_UNIT_A** for DMA UNIT A,
- **DMAC_UNIT_B** for DMA UNIT B.

Chx is the specified DMA channel, which can be one of:
- **DMAC_CHANNEL_0** for DMA channel 0,
- **DMAC_CHANNEL_1** for DMA channel 1.

NewState: New state of DMA halt.
This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:
This function will set specified UNIT Channel ignore DMA request.

Return:
None

6.2.3.16 DMAC_SetLockedTx

Set whether locked transfer.

Prototype:
void
DMAC_SetLockedTx(TSB_DMAC_TypeDef * ***DMACx,***
DMAC_Channel ***Chx***,
FunctionalState ***NewState***);

Parameters:
DMACx is the specified DMA Unit, which can be one of:
- **DMAC_UNIT_A** for DMA UNIT A,
- **DMAC_UNIT_B** for DMA UNIT B.

Chx is the specified DMA channel, which can be one of:
- **DMAC_CHANNEL_0** for DMA channel 0,
- **DMAC_CHANNEL_1** for DMA channel 1.

NewState: New state of DMA transfer.
This parameter can be one of the following values:
ENABLE or **DISABLE**

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

Return:
None

6.2.3.17 DMAC_SetTxINTConfig

Enable or disable the specified DMA Channel transfer interrupt.

Prototype:
void
DMAC_SetTxINTConfig(TSB_DMAC_TypeDef * ***DMACx,***
DMAC_Channel ***Chx***,
DMAC_INTSrc ***INTSource***,
FunctionalState ***NewState***);

Parameters:
DMACx is the specified DMA Unit, which can be one of:
- **DMAC_UNIT_A** for DMA UNIT A,
- **DMAC_UNIT_B** for DMA UNIT B.

Chx is the specified DMA channel, which can be one of:
- **DMAC_CHANNEL_0** for DMA channel 0,
- **DMAC_CHANNEL_1** for DMA channel 1.

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

Return:
None

6.2.3.18 DMAC_SetDMAChannel

Enable or disable the specified DMA Channel.

Prototype:
void
DMAC_SetDMAChannel(TSB_DMAC_TypeDef * ***DMACx,***
DMAC_Channel ***Chx***,
FunctionalState ***NewState***);

Parameters:
DMACx is the specified DMA Unit, which can be one of:
- **DMAC_UNIT_A** for DMA UNIT A,
- **DMAC_UNIT_B** for DMA UNIT B.

Chx is the specified DMA channel, which can be one of:
- **DMAC_CHANNEL_0** for DMA channel 0,
- **DMAC_CHANNEL_1** for DMA channel 1.

NewState: New state of DMA channel.
This parameter can be one of the following values:
ENABLE or **DISABLE**

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

Return:
None

6.2.3.19 DMAC_Init

Initialize and configure the specified DMA channel.

Prototype:
void
DMAC_Init(TSB_DMAC_TypeDef * ***DMACx,***

DMAC_Channel ***Chx***,
DMAC_InitTypeDef * ***InitStruct***);

Parameters:
DMACx is the specified DMA Unit, which can be one of:
- **DMAC_UNIT_A** for DMA UNIT A,
- **DMAC_UNIT_B** for DMA UNIT B.

Chx is the specified DMA channel, which can be one of:
- **DMAC_CHANNEL_0** for DMA channel 0,
- **DMAC_CHANNEL_1** for DMA channel 1.

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

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

Note:
Please use this API to initialize DMAC transmission before calling **DMAC_SetDMAChannel()**.

Return:
None

6.2.4 Data Structure Description

6.2.4.1 DMAC_InitTypeDef

Data Fields:

uint32_t
TxDirection Set transfer direction, which can be set as:
- **DMAC_MEMORY_TO_MEMORY**: transfer method is memory to memory,
- **DMAC_MEMORY_TO_PERIPH**: transfer method is memory to peripheral,
- **DMAC_PERIPH_TO_MEMORY**: transfer method is peripheral to memory.
- **DMAC_PERIPH_TO_ PERIPH**: transfer method is peripheral to peripheral.

uint32_t
SrcAddr Set source address.

uint32_t
DstAddr Set destination address.

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

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

DMAC_BitWidth
SrcBitWidth Set transfer source bit width, which can be set as:
- **DMAC_BYTE** means transfer source bit width set as byte,
- **DMAC_HALF_WORD** means transfer source bit width set as half word,
- **DMAC_WORD** means transfer source bit width set as word.

DMAC_BitWidth
DstBitWidth Set transfer destination bit width, which can be set as:
- **DMAC_BYTE** means transfer destination bit width set as byte,
- **DMAC_HALF_WORD** means transfer destination bit width set as half word,
- **DMAC_WORD** means transfer destination bit width set as word.

DMAC_BurstSize
SrcBurstSize Set transfer source burst size, which can be set as:
- **DMAC_1_BEAT** means transfer source burst size set as 1 beat,
- **DMAC_4_BEATS** means transfer source burst size set as 4 beats,
- **DMAC_8_BEATS** means transfer source burst size set as 8 beats,
- **DMAC_16_BEATS** means transfer source burst size set as 16 beats,
- **DMAC_32_BEATS** means transfer source burst size set as 32 beats,
- **DMAC_64_BEATS** means transfer source burst size set as 64 beats,
- **DMAC_128_BEATS** means transfer source burst size set as 128 beats,
- **DMAC_256_BEATS** means transfer source burst size set as 256 beats.

DMAC_BurstSize
DstBurstSize Set transfer destination burst size, which can be set as:
- **DMAC_1_BEAT** means transfer destination burst size set as 1 beat,
- **DMAC_4_BEATS** means transfer destination burst size set as 4 beats,
- **DMAC_8_BEATS** means transfer destination burst size set as 8 beats,
- **DMAC_16_BEATS** means transfer destination burst size set as 16 beats,
- **DMAC_32_BEATS** means transfer destination burst size set as 32 beats,
- **DMAC_64_BEATS** means transfer destination burst size set as 64 beats,
- **DMAC_128_BEATS** means transfer destination burst size set as 128 beats,
- **DMAC_256_BEATS** means transfer destination burst size set as 256 beats.

uint32_t
TxSize Set the total number of transfer, MAX is 0x0FFF.

DMACA_ReqNum
A_TxDstPeriph Set transfer destination peripheral, which can be set as:
- **DMACA_SIO0_UART0_RX** for SIO0/UART0 Reception,
- **DMACA_SIO0_UART0_TX** for SIO0/UART0 Transmission,
- **DMACA_SIO2_UART2_RX** for SIO2/UART2 Reception,
- **DMACA_SIO2_UART2_TX** for SIO2/UART2 Transmission,
- **DMACA_SIO4_UART4_RX** for SIO4/UART4 Reception,
- **DMACA_SIO4_UART4_TX** for SIO4/UART4 Transmission,
- **DMACA_PULSE_CNT2** Two-phase pulse counter2 every count,
- **DMACA_PULSE_CNT3** Two-phase pulse counter3 every count,
- **DMACA_TMRB8_CMP_MATCH** for TMRB8 compare match,

- **DMACA_TMRB9_CMP_MATCH** for TMRB9compare match,
- **DMACA_TMRB0_CAPTURE0** for TMRB0 input capture 0,
- **DMACA_TMRB4_CAPTURE0** for TMRB4 input capture 0,
- **DMACA_TMRB4_CAPTURE1** for TMRB4 input capture 1
- **DMACA_TMRB5_CAPTURE0** for TMRB5 input capture 0,
- **DMACA_TMRB5_CAPTURE1** for TMRB5 input capture 1
- **DMACA_HIGH_PRIORITY_ADC X** Highest priority AD Conversion End,

DMACA_ReqNum

A_TxSrcPeriph Set transfer source peripheral, which can be set as:

- **DMACA_SIO0_UART0_RX** for SIO0/UART0 Reception,
- **DMACA_SIO0_UART0_TX** for SIO0/UART0 Transmission,
- **DMACA_SIO2_UART2_RX** for SIO2/UART2 Reception,
- **DMACA_SIO2_UART2_TX** for SIO2/UART2 Transmission,
- **DMACA_SIO4_UART4_RX** for SIO4/UART4 Reception,
- **DMACA_SIO4_UART4_TX** for SIO4/UART4 Transmission,
- **DMACA_PULSE_CNT2** Two-phase pulse counter2 every count,
- **DMACA_PULSE_CNT3** Two-phase pulse counter3 every count,
- **DMACA_TMRB8_CMP_MATCH** for TMRB8 compare match,
- **DMACA_TMRB9_CMP_MATCH** for TMRB9compare match,
- **DMACA_TMRB0_CAPTURE0** for TMRB0 input capture 0,
- **DMACA_TMRB4_CAPTURE0** for TMRB4 input capture 0,
- **DMACA_TMRB4_CAPTURE1** for TMRB4 input capture 1
- **DMACA_TMRB5_CAPTURE0** for TMRB5 input capture 0,
- **DMACA_TMRB5_CAPTURE1** for TMRB5 input capture 1
- **DMACA_HIGH_PRIORITY_ADC X** Highest priority AD Conversion End,

DMACB_ReqNum

B_TxDstPeriph Set transfer destination peripheral, which can be set as:

- **DMACB_TMRD00_ CMP_MATCH** for TMRD00 compare match,
- **DMACB_TMRD10_ CMP_MATCH** for TMRD10 compare match,
- **DMACB_PULSE_CNT0** Two-phase pulse counter0 every count,
- **DMACB_PULSE_CNT1** Two-phase pulse counter1 every count,
- **DMACB_TMRB6_CMP_MATCH** for TMRB6 compare match,
- **DMACB_TMRB7_CMP_MATCH** for TMRB7 compare match,
- **DMACB_TMRB0_CAPTURE1** for TMRB0 input capture 1
- **DMACB_TMRB2_CAPTURE0** for TMRB2 input capture 0,
- **DMACB_TMRB2_CAPTURE1** for TMRB2 input capture 1
- **DMACB_TMRB3_CAPTURE0** for TMRB3 input capture 0,
- **DMACB_TMRB3_CAPTURE1** for TMRB3 input capture 1,
- **DMACB_TMRB6_CAPTURE0** for TMRB6 input capture 0,
- **DMACB_TMRB6_CAPTURE1** for TMRB6 input capture 1,
- **DMACB_NORMAL_ADC** for Normal AD Conversion End,
- **DMACB_SSP_TX** for SSP Transmission,
- **DMACB_SSP_RX** for SSP Reception,

DMACB_ReqNum

B_TxSrcPeriph Set transfer source peripheral, which can be set as:

- **DMACB_TMRD00_ CMP_MATCH** for TMRD00 compare match,
- **DMACB_TMRD10_ CMP_MATCH** for TMRD10 compare match,
- **DMACB_PULSE_CNT0** Two-phase pulse counter0 every count,
- **DMACB_PULSE_CNT1** Two-phase pulse counter1 every count,
- **DMACB_TMRB6_CMP_MATCH** for TMRB6 compare match,
- **DMACB_TMRB7_CMP_MATCH** for TMRB7 compare match,
- **DMACB_TMRB0_CAPTURE1** for TMRB0 input capture 1
- **DMACB_TMRB2_CAPTURE0** for TMRB2 input capture 0,

- **DMACB_TMRB2_CAPTURE1** for TMRB2 input capture 1
- **DMACB_TMRB3_CAPTURE0** for TMRB3 input capture 0,
- **DMACB_TMRB3_CAPTURE1** for TMRB3 input capture 1,
- **DMACB_TMRB6_CAPTURE0** for TMRB6 input capture 0,
- **DMACB_TMRB6_CAPTURE1** for TMRB6 input capture 1,
- **DMACB_NORMAL_ADC** for Normal AD Conversion End,
- **DMACB_SSP_TX** for SSP Transmission,
- **DMACB_SSP_RX** for SSP Reception,

FunctionalState
TxINT Set transfer interrupt state, which can be set as:
- **EANBLE** means enable transfer interrupt.
- **DISABLE** means disable transfer interrupt.

7. EXB

7.1 Overview

The TMPM341x has a built-in external bus interface to connect to external memory, I/Os, etc. This interface consists of an external bus interface circuit (EBIF), a chip selector (CS) and a wait controller.

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

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

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm341_exb.c, with /Libraries/TX03_Periph_Driver/inc/tmpm341_exb.h containing the macros, data types, structures and API definitions for use by applications.

7.2 API Functions

7.2.1 Function List

- void EXB_SetBusMode(uint8_t ***BusMode***);
- void EXB_SetBusCycleExtension(uint8_t ***Cycle***);
- void EXB_Enable(uint8_t ***ChipSelect***);
- void EXB_Disable(uint8_t ***ChipSelect***) ;
- void EXB_Init(uint8_t ***ChipSelect*** , EXB_InitTypeDef* ***InitStruct***);

7.2.2 Detailed Description

Functions listed above can be divided into two parts:

1) Configure the EXB bus mode, bus cycle extension, data bus widths and the cycle of external buses based on the chip selector.
   EXB_SetBusMode(),EXB_SetBusCycleExtension() and EXB_Init().
2) Enable and disable control
   EXB_Enable(), EXB_Disable().

7.2.3 Function Documentation

7.2.3.1 EXB_SetBusMode

Set external bus mode for EXB.

Prototype:
void
EXB_SetBusMode(uint8_t ***BusMode***)

Parameters:
BusMode : select EXB bus mode.
The value could be the following values:
- **EXB_BUS_SEPARATE** for separate bus mode.
- **EXB_BUS_MULTIPLEX** for multiplex bus mode.

Description:
This function sets the bus mode for the external bus. When ***BusMode*** is **EXB_BUS_SEPARATE**, the bus mode will be separate mode. When ***BusMode*** is **EXB_BUS_MULTIPLEX**, the bus mode will be multiplex mode.

Return:
None

7.2.3.2 EXB_SetBusCycleExtension

Set the bus cycle to be double or quadruple.

Prototype:
void
EXB_SetBusCycleExtension(uint8_t ***Cycle***)

Parameters:
Cycle: Set the bus cycle to be double or quadruple.
The value could be the following values:
- **EXB_CYCLE_NONE**: EXB bus cycle will not be extended.
- **EXB_CYCLE_DOUBLE**: EXB bus cycle will be double.
- **EXB_CYCLE_QUADRUPLE**: EXB bus cycle will be quadruple.

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

Return:
None

7.2.3.3 EXB_Enable

Enable the specified chip.

Prototype:
void
EXB_Enable(uint8_t ***ChipSelect***)

Parameters:
ChipSelect is the specified chip.
The value could be the following values:
- **EXB_CS0**: for chip 0

- **EXB_CS1**: for chip 1

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

Return:
None

7.2.3.4 EXB_Disable

Disable the specified chip.

Prototype:
void
EXB_Disable(uint8_t ***ChipSelect***)

Parameters:
ChipSelect is the specified chip.
The value could be the following values:
- **EXB_CS0**: for chip 0
- **EXB_CS1**: for chip 1

Description:
This function will disable the access to the specified chip.

Return:
None

7.2.3.5 EXB_Init

Initialize the specified chip.

Prototype:
void
EXB_Init (uint8_t ***ChipSelect***,
EXB_InitTypeDef* ***InitStruct***)

Parameters:
ChipSelect is the specified chip.
The value could be the following values:
- **EXB_CS0**: for chip 0
- **EXB_CS1**: for chip 1

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

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

Return:
None

7.2.4 Data Structure Description

7.2.4.1 EXB_InitTypeDef

Data Fields:

uint8_t
AddrSpaceSize Set the address space size, which can be set as:
- **EXB_16M_BYTE**: address space is 16Mbyte,
- **EXB_8M_BYTE**: address space is 8Mbyte,
- **EXB_4M_BYTE**: address space is 4Mbyte,
- **EXB_2M_BYTE**: address space is 2Mbyte,
- **EXB_1M_BYTE**: address space is 1Mbyte,
- **EXB_512K_BYTE**: address space is 512Kbyte,
- **EXB_256K_BYTE**: address space is 256Kbyte,
- **EXB_128K_BYTE**: address space is 128Kbyte,
- **EXB_64K_BYTE**: address space is 64Kbyte.

uint8_t
StartAddr Set the start address. The max value is 0xFF.

uint8_t
BusWidth Set the data bus width, which can be set as:
- **EXB_BUS_WIDTH_BIT_8**: data bus width is 8bit,
- **EXB_BUS_WIDTH_BIT_16**: data bus width is 16bit.

uint8_t
EndType Classifies an endian type for external memory or peropheral I/O. The data value is 0 or 1.

EXB_CyclesTypeDef
Cycles Set the cycle of external buses, which consists of following members: ***InternalWait***, ***ReadSetupCycle***, ***WriteSetupCycle***, ***ALEWaitCycle*** (For multiplex bus mode only),***ReadRecoveryCycle***, ***WriteRecoveryCycle*** and ***ChipSelectRecoveryCycle***. (Refer to “EXB_CyclesTypeDef” for details)

7.2.4.2 EXB_CyclesType Def

Data Fields:

uint8_t
InternalWait Set the internal wait, which can be set as:
- **EXB_INTERNAL_WAIT_0**: 0 wait,
- **EXB_INTERNAL_WAIT_1**: 1 wait,
- **EXB_INTERNAL_WAIT_2**: 2 waits,
- **EXB_INTERNAL_WAIT_3**: 3 waits,
- **EXB_INTERNAL_WAIT_4**: 4 waits,
- **EXB_INTERNAL_WAIT_5**: 5 waits,
- **EXB_INTERNAL_WAIT_6**: 6 waits,
- **EXB_INTERNAL_WAIT_7**: 7 waits,
- **EXB_INTERNAL_WAIT_8**: 8 waits,
- **EXB_INTERNAL_WAIT_9**: 9 waits,
- **EXB_INTERNAL_WAIT_10**: 10 waits,
- **EXB_INTERNAL_WAIT_11**: 11 waits,
- **EXB_INTERNAL_WAIT_12**: 12 waits,
- **EXB_INTERNAL_WAIT_13**: 13 waits,
- **EXB_INTERNAL_WAIT_14**: 14 waits,
- **EXB_INTERNAL_WAIT_15**: 15 waits.

uint8_t

ReadSetupCycle Set the read setup cycle, which can be set as:
- **EXB_CYCLE_0**: 0 cycle,
- **EXB_CYCLE_1**: 1 cycle,
- **EXB_CYCLE_2**: 2 cycles,
- **EXB_CYCLE_4**: 4 cycles.

uint8_t
WriteSetupCycle Set the write setup cycle, which can be set as:
- **EXB_CYCLE_0**: 0 cycle,
- **EXB_CYCLE_1**: 1 cycle,
- **EXB_CYCLE_2**: 2 cycles,
- **EXB_CYCLE_4**: 4 cycles.

uint8_t
ALEWaitCycle Set the ALE waits cycle for multiplex bus, which can be set as:
- **EXB_CYCLE_0**: 0 cycle,
- **EXB_CYCLE_1**: 1 cycle,
- **EXB_CYCLE_2**: 2 cycles,
- **EXB_CYCLE_4**: 4 cycles.

uint8_t
ReadRecoveryCycle Set the read recovery cycle, which can be set as:
- **EXB_CYCLE_0**: 0 cycle,
- **EXB_CYCLE_1**: 1 cycle,
- **EXB_CYCLE_2**: 2 cycles,
- **EXB_CYCLE_3**: 3 cycles,
- **EXB_CYCLE_4**: 4 cycles,
- **EXB_CYCLE_5**: 5 cycles,
- **EXB_CYCLE_6**: 6 cycles,
- **EXB_CYCLE_8**: 8 cycles.

uint8_t
WriteRecoveryCycle Set the write recovery cycle, which can be set as:
- **EXB_CYCLE_0**: 0 cycle,
- **EXB_CYCLE_1**: 1 cycle,
- **EXB_CYCLE_2**: 2 cycles,
- **EXB_CYCLE_3**: 3 cycles,
- **EXB_CYCLE_4**: 4 cycles,
- **EXB_CYCLE_5**: 5 cycles,
- **EXB_CYCLE_6**: 6 cycles,
- **EXB_CYCLE_8**: 8 cycles.

uint8_t
ChipSelectRecoveryCycle Set the chip select recovery cycle, which can be:
- **EXB_CYCLE_0**: 0 cycle,
- **EXB_CYCLE_1**: 1 cycle,
- **EXB_CYCLE_2**: 2 cycles,
- **EXB_CYCLE_4**: 4 cycles.

8. FC

8.1 Overview

TMPM341x device contains flash memory.
For TMPM341FDXBG, the size of flash is 512Kbyte.
For TMPM341FYXBG, the size of flash is 256Kbyte.

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 refer to the MCU data sheet.

This driver is contained in \Libraries\TX03_Periph_Driver\src\tmpm341_fc.c with \Libraries\TX03_Periph_Driver \inc\tmpm341_fc.h containing the API definitions for use by applications.

8.2 API Functions

8.2.1 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)

8.2.2 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().

8.2.3 Function Documentation

8.2.3.1 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<SEBIT> to “1”.

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

Return:
None

8.2.3.2 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.

8.2.3.3 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.

8.2.3.4 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

8.2.3.5 FC_ProgramBlockProtectState

Program the protection bits

Prototype:
FunctionalState
FC_ProgramProtectState(uint8_t ***BlockNum***)

Parameters:
BlockNum:The flash block number
- **FC_BLOCK_0** for block 0.
- **FC_BLOCK_1** for block 1.
- **FC_BLOCK_2** for block 2.
- **FC_BLOCK_3** for block 3.
- **FC_BLOCK_4** for block 4.
- **FC_BLOCK_5** for block 5.

Description:

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

Return:
Result of the operation to program the protection bit
FC_SUCCESS: Set the protection bit to “1” successfully.
FC_ERROR_PROTECTED: The protection bit is “1” already, and it doesn’t need to program it again.
FC_ERROR_OVER_TIME: Program block protection bit operation over time error.

8.2.3.6 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.

8.2.3.7 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 512Byte for the TMPM341FD and 256Byte for the TMPM341FY.

Description:
This API is used to write data to specified page.
The TMPM341FD contains 128 words and the TMPM341FY contains 64 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.

8.2.3.8 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.

8.2.3.9 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.

9. GPIO

9.1 Overview

For TOSHIBA TMPM341x 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/tmpm341_gpio.c, with /Libraries/TX03_Periph_Driver/inc/tmpm341_gpio.h containing the macros, data types, structures and API definitions for use by applications.

9.2 API Functions

9.2.1 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***);

9.2.2 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.

9.2.3 Function Documentation

9.2.3.1 GPIO_ReadData

Read specified GPIO Data register.

Prototype:
uint8_t
GPIO_ReadData(GPIO_Port ***GPIO_x***);

Parameters:
GPIO_x: Select GPIO port, which can be set as:

- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PI :** GPIO port I.
- **GPIO_PJ:** GPIO port J.
- **GPIO_PK:** GPIO port K.

Description:
This function will read specified GPIO Data register.

Return:
The value read from DATA register.

9.2.3.2 GPIO_ReadDataBit

Read specified GPIO pin.

Prototype:
uint8_t
GPIO_ReadDataBit(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***);

Parameters:
GPIO_x: Select GPIO port, which can be set as:

- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D..
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PI :** GPIO port I.
- **GPIO_PJ:** GPIO port J.

- **GPIO_PK:** GPIO port K.

Bit_x: Select GPIO pin, which can be set as:
- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7.

Description:
This function will read specified GPIO pin.

Return:
The value read from GPIO pin as:
GPIO_BIT_VALUE_0: Value 0,
GPIO_BIT_VALUE_1: Value 1.

9.2.3.3 GPIO_WriteData

Write specified value to GPIO Data register.

Prototype:
```
void
GPIO_WriteData(GPIO_Port  GPIO_x,
               uint8_t Data);
```

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PI :** GPIO port I.
- **GPIO_PJ:** GPIO port J.
- **GPIO_PK:** GPIO port K.

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

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

Return:
None

9.2.3.4 GPIO_WriteDataBit

Write specified value of single bit to GPIO pin.

Prototype:
void
GPIO_WriteDataBit(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***,
uint8_t ***BitValue***) ;

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PI :** GPIO port I.
- **GPIO_PJ:** GPIO port J. .
- **GPIO_PK:** GPIO port K.

Bit_x: Select GPIO pin, which can be set as:
- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7.

BitValue: The new value of GPIO pin, which can be set as:
- **GPIO_BIT_VALUE_0**: Clear GPIO pin,
- **GPIO_BIT_VALUE_1**: Set GPIO pin.

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

Return:
None

9.2.3.5 GPIO_Init

Initialize GPIO port function.

Prototype:
void
GPIO_Init(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***,
GPIO_InitTypeDef * ***GPIO_InitStruct***);

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.

- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PI :** GPIO port I.
- **GPIO_PJ:** GPIO port J.
- **GPIO_PK:** GPIO port K.

Bit_x: Select GPIO pin, which can be set as:
- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7,
- **GPIO_BIT_ALL:** All GPIO pins can be set.
- Combination of the effective bits.

GPIO_InitStruct: The structure containing basic GPIO configuration. (Refer to Data structure Description for details)

Description:
This function will be configure GPIO pin IO mode, pull-up, pull-down function and set this pin as open drain port or CMOS port. **GPIO_SetOutput()**, **GPIO_SetInput()**,**GPIO_SetPullUp ()**,**GPIO_SetPullDown()** and **GPIO_SetOpenDrain()** will be called by it.

Return:
None

9.2.3.6 GPIO_SetOutput

Set specified GPIO pin as output port.

Prototype:
void
GPIO_SetOutput(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***);

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PI :** GPIO port I.
- **GPIO_PJ:** GPIO port J.
- **GPIO_PK:** GPIO port K.

Bit_x: Select GPIO pin, which can be set as:

- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7,
- **GPIO_BIT_ALL:** All GPIO pins can be set.
- Combination of the effective bits.

Description:
This function will set specified GPIO pin as output port.

Return:
None

9.2.3.7 GPIO_SetInput

Set specified GPIO Pin as input port.

Prototype:
void
GPIO_SetInput(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***);

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PI :** GPIO port I.
- **GPIO_PJ:** GPIO port J.
- **GPIO_PK:** GPIO port K.

Bit_x: Select GPIO pin, which can be set as:
- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7,
- **GPIO_BIT_ALL:** All GPIO pins can be set.
- Combination of the effective bits.

Description:
This function will set specified GPIO pin as input port.

Return:

None

9.2.3.8 GPIO_SetOutputEnableReg

Enable or disable specified GPIO Pin output function.

Prototype:
void
GPIO_SetOutputEnableReg(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***,
FunctionalState ***NewState***);

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PI :** GPIO port I.
- **GPIO_PJ:** GPIO port J.
- **GPIO_PK:** GPIO port K.

Bit_x: Select GPIO pin, which can be set as:
- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7,
- **GPIO_BIT_ALL:** All GPIO pins can be set.
- Combination of the effective bits.

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

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

Return:
None

9.2.3.9 GPIO_SetInputEnableReg

Enable or disable specified GPIO Pin input function.

Prototype:

void
GPIO_SetInputEnableReg(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***,
FunctionalState ***NewState***);

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PI :** GPIO port I.
- **GPIO_PJ:** GPIO port J.
- **GPIO_PK:** GPIO port K.

Bit_x: Select GPIO pin, which can be set as:
- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7,
- **GPIO_BIT_ALL:** All GPIO pins can be set.
- Combination of the effective bits.

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

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

Return:
None

9.2.3.10 GPIO_SetPullUp

Enable or disable specified GPIO Pin pull-up function.

Prototype:
void
GPIO_SetPullUp(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***,
FunctionalState ***NewState***);

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.

- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PI :** GPIO port I.
- **GPIO_PJ:** GPIO port J.
- **GPIO_PK:** GPIO port K.

Bit_x: Select GPIO pin, which can be set as:
- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7,
- **GPIO_BIT_ALL:** All GPIO pins can be set.
- Combination of the effective bits.

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

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

Return:
None

9.2.3.11 GPIO_SetPullDown

Enable or disable specified GPIO Pin pull-down function.

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

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PI :** GPIO port I.

Bit_x: Select GPIO pin, which can be set as:
- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,

- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7,
- **GPIO_BIT_ALL:** All GPIO pins can be set.
- Combination of the effective bits.

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

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

Return:
None

9.2.3.12 GPIO_SetOpenDrain

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

Prototype:
void
GPIO_SetOpenDrain(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***,
FunctionalState ***NewState***);

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PI :** GPIO port I.

Bit_x: Select GPIO pin, which can be set as:
- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7,
- **GPIO_BIT_ALL:** All GPIO pins can be set.
- Combination of the effective bits.

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

Description:

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

Return:
None

9.2.3.13 GPIO_EnableFuncReg

Enable specified GPIO function.
Prototype:
void
GPIO_EnableFuncReg(GPIO_Port ***GPIO_x***,
uint8_t ***FuncReg_x***,
uint8_t ***Bit_x***) ;

Parameters:

GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PI :** GPIO port I.
- **GPIO_PJ:** GPIO port J.
- **GPIO_PK:** GPIO port K.

FuncReg_x: The number of GPIO function register, which can be set as:
- **GPIO_FUNC_REG_1** for GPIO function register 1,
- **GPIO_FUNC_REG_2** for GPIO function register 2,
- **GPIO_FUNC_REG_3** for GPIO function register 3.
- **GPIO_FUNC_REG_4** for GPIO function register 4.

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

9.2.3.14 GPIO_DisableFuncReg

Disable specified GPIO function.

Prototype:
void
GPIO_DisableFuncReg(GPIO_Port ***GPIO_x***,
uint8_t ***FuncReg_x***,
uint8_t ***Bit_x***) ;

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PI :** GPIO port I.
- **GPIO_PJ:** GPIO port J.
- **GPIO_PK:** GPIO port K.

FuncReg_x: The number of GPIO function register, which can be set as:
- **GPIO_FUNC_REG_1** for GPIO function register 1,
- **GPIO_FUNC_REG_2** for GPIO function register 2,
- **GPIO_FUNC_REG_3** for GPIO function register 3.
- **GPIO_FUNC_REG_4** for GPIO function register 4.

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

9.2.4 Data Structure Description

9.2.4.1 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.

10. OFD

10.1 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/tmpm341_ofd.c, with /Libraries/TX03_Periph_Driver/inc/tmpm341_ofd.h containing the macros, data types, structures and API definitions for use by applications.

10.2 API Functions

10.2.1 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);

10.2.2 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.

10.2.3 Function Documentation

10.2.3.1 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

10.2.3.2 OFD_Enable

Enable the OFD function.

Prototype:
void
OFD_Enable(void)

Parameters:
None.

Description:
This function will enable the OFD function.

Return:
None

10.2.3.3 OFD_Disable

Disable the OFD function.

Prototype:
void
OFD_Disable(void)

Parameters:
None.

Description:
This function will disable the OFD function.

Return:
None

10.2.3.4 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. This value range is 0U to 0x1FFU.
LowerDetectionCount is the count value of lower detection frequency. This value range is 0U to 0x1FFU.

Description:

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

Return:
None

10.2.3.5 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

10.2.3.6 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 Structure of OFD status, which include the busy and frequency error info.
(Refer to “Data Structure Description” for details).

10.2.4 Data Structure Description

10.2.4.1 OFD_Status

Data Fields:

uint32_t
All: Data.
Bit
uint32_t

Frequency Error: 1 uint32_t	Frequency Error status
OFDBusy: 1	OFD Busy status

11. PHC

11.1 Overview

TOSHIBA TMPM341x has four channel two-phase pulse input counters. The counter is increment or decremented by one depending on the state transition of the two phase pulse that is input through PHCxIN0 and PHCxIN1 and has phase difference. PHCxIN0 and PHCxIN1 have a noise canceller, which is selected either enabled or disabled.

Two-phase pulse input counter has two compare register. An interrupt is occurred when the up-and-down counter matches them. And an interrupt is also occurred by increment or decrement the up-and-down counter.

Two-phase pulse input counter has three operation mode, they are controlled by register.
1. Normal operation mode (up/down at the fourth count)
2. Quadruple mode (up/down at each count)
3. Multiplied by two mode
- PHCxIN0 input
- PHCxIN1 input

The PHC API provides a set of functions for using the TMPM341x PHC modules. It includes PHC channel set, mode set, noise filter set, interrupt set, PHC status read, PHC result value read and so on.

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm341_phc.c, with /Libraries/TX03_Periph_Driver/inc/tmpm341_phc.h containing the macros, data types, structures and API definitions for use by applications.

11.2 API Functions

11.2.1 Function List

- void PHC_Enable(TSB_PHC_TypeDef * ***PHCx***);
- void PHC_Disable(TSB_PHC_TypeDef * ***PHCx***);
- void PHC_SetRunState(TSB_PHC_TypeDef * ***PHCx***, uint32_t ***Cmd***);
- void PHC_Init(TSB_PHC_TypeDef * ***PHCx***, PHC_InitTypeDef * ***InitStruct***);
- PHC_INTFactor PHC_GetINTFactor(TSB_PHC_TypeDef * ***PHCx***);
- void PHC_ClearINTFactor(TSB_PHC_TypeDef * ***PHCx***, uint32_t ***ClearINT***);
- void PHC_EnableInterrupt(TSB_PHC_TypeDef * ***PHCx***, uint32_t ***EnableINT***);
- void PHC_DisableInterrupt(TSB_PHC_TypeDef * ***PHCx***, uint32_t ***DisableINT***);
- uint16_t PHC_GetPulseCntValue(TSB_PHC_TypeDef * ***PHCx***);
- void PHC_ClearPulseCntValue(TSB_PHC_TypeDef * ***PHCx***);
- uint16_t PHC_GetCompareValue(TSB_PHC_TypeDef * ***PHCx***, uint8_t ***CmpReg***);
- void PHC_SetCompareValue(TSB_PHC_TypeDef * ***PHCx***, uint8_t ***CmpReg***, uint16_t ***CmpValue***);
- void PHC_SetDMAReq(TSB_PHC_TypeDef * ***PHCx***, FunctionalState ***NewState***, uint8_t ***DMAReq***);

11.2.2 Detailed Description

Functions listed above can be divided into three parts:

4) Configure and control the common functions of each PHC channel are handled by the PHC_Enable (),PHC_Disable (),PHC_Init() and void PHC_SetRunState().

5) The status indication of each PHC channel is handled by PHC_GetINTFactor(), PHC_GetPulseCntValue(), PHC_GetCompareValue()
6) PHC_ClearINTFactor(), PHC_EnableInterrupt(), PHC_DisableInterrupt(), , PHC_ClearPulseCntValue(),PHC_SetCompareValue() and PHC_SetDMAReq() handle other specified functions.

11.2.3 Function Documentation

Note: In all of the following APIs, the parameter "TSB_PHC_TypeDef * ***PHCx***" can be one of the following values:
TSB_PCH0, TSB_PCH1, TSB_PCH2, TSB_PCH3

11.2.3.1 PHC_Enable

Enable the specified PHC channel.

Prototype:
void
PHC_Enable(TSB_PHC_TypeDef* ***PHCx***)

Parameters:
PHCx is the specified PHC channel.

Description:
This function enables the specified PHC channel selected by ***PHCx***.

Return:
None

11.2.3.2 PHC_Disable

Disable the specified PHC channel.

Prototype:
void
PHC_Disable(TSB_PHC_TypeDef* ***PHCx***)

Parameters:
PHCx is the specified PHC channel.

Description:
This function disables the specified PHC channel selected by ***PHCx***.

Return:
None

11.2.3.3 PHC_ SetRunState

Start or stop of the specified PHC channel.

Prototype:
void
PHC_SetRunState(TSB_PHC_TypeDef * ***PHCx***,
uint32_t ***Cmd***);

Parameters:
PHCx is the specified PHC channel.
Cmd The command for the up-and-down counter
- **PHC_RUN** for start counter
- **PHC_STOP** for stop counter

Description:
The up-and-down counter of the specified PHC channel starts counting if ***Cmd*** is **PHC_RUN** and up-and-down counter stops counting and the value in up-and-down counter register is clear if ***Cmd*** is **PHC_STOP**.

Return:
None

11.2.3.4 PHC_Init

Initialize the specified PHC channel.

Prototype:
void
PHC_Init (TSB_PHC_TypeDef * ***PHCx***,
PHC_InitTypeDef * ***InitStruct***);

Parameters:
PHCx is the specified PHC channel.
InitStruct is the structure containing basic PHC configuration including count mode, noise filter control and counter clear control (refer to "Data Structure Description" for details).

Description:
This function initializes the specified PHC channel selected by ***PHCx***.

Return:
None

11.2.3.5 PHC_GetINTFactor

Indicate what causes the interrupt.

Prototype:
PHC_INTFactor
PHC_GetINTFactor(TSB_PHC_TypeDef* ***PHCx***)

Parameters:
PHCx is the specified PHC channel.

Description:
This function should be used in ISR to indicate the factor of interrupt. Bit of **Compare0** indicates if the counter matches with compare register0, bit of **Compare1** Indicates if the counter matches with compare register1, bit of **Overflow** indicates if overflow had occurred before the interrupt, and bit of **Underflow** indicates if underflow had occurred before the interrupt.

Return:
PHC Interrupt factor. Each bit has the following meaning:
Compare0 (Bit0): Specified PHC channel detected a match with the compare register0
Compare1 (Bit1): Specified PHC channel detected a match with the compare register1
Overflow (Bit2): Specified PHC channel detected counter is overflow
Underflow (Bit3): Specified PHC channel detected counter is underflow

Note:
It is recommended to use the following method to process different interrupt factor

```
PHC_INTFactor factor = PHC_GetINTFactor(TSB_PHC0);
if (factor.Bit.Compare0) {
    // Do A
}

if (factor.Bit.Compare1) {
    // Do B
}

if (factor.Bit.OverFlow) {
    // Do C
}

if (factor.Bit.UnderFlow) {
    // Do D
}
```

11.2.3.6 PHC_ClearINTFactor

Clear specified interrupt factor flag.

Prototype:
void
PHC_ClearINTFactor(TSB_PHC_TypeDef * ***PHCx***,
uint32_t ***ClearINT***)

Parameters:
PHCx is the specified PHC channel.
ClearINT Select the clear of PHC interrupt factor. This parameter can be:
- **PHC_FLG_CMP0**: Clears the interrupt factor which monitors compare register0 match event.
- **PHC_FLG_CMP1**: Clears the interrupt factor which monitors compare register1 match event.
- **PHC_FLG_OVERFLOW**: Clears the interrupt factor which monitors overflow event.
- **PHC_FLG_UNDERFLOW**: Clears the interrupt factor which monitors underflow event.
- **PHC_FLG_ALL**: All interrupt factors can be cleared.
- Combination of effective interrupt factor.

Description:
Because of each interrupt factor is not cleared automatically.

This function is used to clear the specified interrupt factor.

Return:
None

Note:
Each interrupt factor is not cleared automatically, initialize before using them.

11.2.3.7 PHC_ EnableInterrupt

Enable specified PHC interrupt

Prototype:
void
PHC_EnableInterrupt(TSB_PHC_TypeDef * ***PHCx***,
uint32_t ***EnableINT***);

Parameters:
PHCx is the specified PHC channel.
EnableINT Selects the clear of PHC interrupt. This parameter can be:

- **PHC_CR_INT_COMP0**: Enable INTPHTx0 interrupt, which occurred if the counter matches with compare register0.(x can be 0,1,2,3)
- **PHC_CR_INT_COMP1**: Enable INTPHTx1 interrupt, which occurred if the counter matches with compare register1. (x can be 0,1,2,3)
- **PHC_CR_INT_COMP0_AND_1**: Enable both INTPHTx0 interrupt and INTPHTx1 interrupt (x can be 0,1,2,3)
- **PHC_CR_INT_EVERY**: Enable INTPHEVRYx interrupt, which occurred if the counter value is changed. (x can be 0,1,2,3)
- **PHC_CR_INT_ALL**: All interrupt can be enabled
- Combination of effective PHC interrupt.

Description:
If **PHC_CR_INT_COMP0** is selected, the interrupt of the specified PHC channel INTPHTx0 happen when the value in counter and compare register0 are match.
If **PHC_CR_INT_COMP1** is selected, the interrupt of the specified PHC channel INTPHTx1 happen when the value in counter and compare register1 are match.
If **PHC_CR_INT_COMP0_AND_1**. the interrupt of the specified PHC channel INTPHTx0 happen when the value in counter and compare register0 are match, and the interrupt of the specified PHC channel INTPHTx1 happen when the value in counter and compare register1 are match.
If **PHC_CR_INT_EVERY**: is selected, the interrupt of the specified PHC channel INTPHEVRYx happen when the value in counter is changed.
If **PHC_CR_INT_ALL**: is selected, all above interrupt of the specified PHC channel is enabled.

Return:
None

11.2.3.8 PHC_DisableInterrupt

Disable specified PHC interrupt

Prototype:
void
PHC_DisableInterrupt(TSB_PHC_TypeDef * ***PHCx***,
uint32_t ***DisableINT***);

Parameters:
PHCx is the specified PHC channel.
DisableINT Select the clear of PHC interrupt. This parameter can be:

- **PHC_CR_INT_COMP0**: Disables INTPHTx0 interrupt, which occurred if the counter matches with compare register0.(x can be 0,1,2,3)
- **PHC_CR_INT_COMP1**: Disables INTPHTx1 interrupt, which occurred if the counter matches with compare register1. (x can be 0,1,2,3)
- **PHC_CR_INT_COMP0_AND_1**: Disables both INTPHTx0 interrupt and INTPHTx1 interrupt (x can be 0,1,2,3)
- **PHC_CR_INT_EVERY**: Disables INTPHEVRYx interrupt, which occurred if the counter value is changed. (x can be 0,1,2,3)
- **PHC_CR_INT_ALL**: All interrupt can be disabled
- Combination of effective PHC interrupt.

Description:
If **PHC_CR_INT_COMP0** is selected, the interrupt of the specified PHC channel INTPHTx0 is disabled (x can be 0,1,2,3)
If **PHC_CR_INT_COMP1** is selected, the interrupt of the specified PHC channel INTPHTx1 is disabled (x can be 0,1,2,3)
If **PHC_CR_INT_COMP0_AND_1** is selected, the interrupt of the specified PHC channel INTPHTx0 and INTPHTx1 are disabled (x can be 0,1,2,3)
If **PHC_CR_INT_EVERY**: is selected, the interrupt of the specified PHC channel INTPHEVRYx is disabled (x can be 0,1,2,3)
If **PHC_CR_INT_ALL**: is selected, all above interrupt of the specified PHC channel is disabled.

Return:
None

11.2.3.9 PHC_ GetPulseCntValue

Get the counter value of specified PHC channel

Prototype:
uint16_t
PHC_GetPulseCntValue(TSB_PHC_TypeDef * ***PHCx***);

Parameters:
PHCx is the specified PHC channel.

Description:
This function returns the value in counter of the specified PHC channel

Return:
The value of PHC counter

11.2.3.10 PHC_ ClearPulseCntValue

Clear the counter value of specified PHC channel

Prototype:
void
PHC_ ClearPulseCntValue(TSB_PHC_TypeDef * ***PHCx***);

Parameters:
PHCx is the specified PHC channel.

Description:
This function clears the value in counter of the specified PHC channel

Return:
None

Note:
Pulse is counted not synchronized with MCU operation clock. Because there is a possibility that data is read while rewriting, depending on the timing, reading out twice is recommended. In this case if the data is different read out data again.

PHC counter is an up-and-down conunter.
The counter value is **0x7FFF** after PHC_ ClearPulseCntValue function is called.

11.2.3.11 PHC_ GetCompareValue

Get the value of compare register0 or compare register1 of the specified PHC channel.

Prototype:
uint16_t
PHC_GetCompareValue(TSB_PHC_TypeDef * ***PHCx***,
uint8_t ***CmpReg***)

Parameters:
PHCx is the specified PHC channel.
CmpReg is used to choose to return the value of compare register0 or to return the value of compare register1, which can be one of the following,

- **PHC_COMP_0:** specifying compare register0.
- **PHC_COMP_1:** specifying compare register1.

Description:
This function returns the value of compare register0 of the specified PHC channel if ***CmpReg*** is **PHC_COMP_0**, and returns value of compare register1 of the specified PHC channel if ***CmpReg*** is **PHC_COMP_1**

Return:
The compare value

11.2.3.12 PHC_ SetCompareValue

Set the value of compare register0 or compare register1 of the specified PHC channel.

Prototype:
void
PHC_SetCompareValue(TSB_PHC_TypeDef * ***PHCx***,

uint8_t ***CmpReg***,
uint16_t ***CmpValue***);

Parameters:
PHCx is the specified PHC channel.
CmpReg is used to choose to set the value of compare register0 or to set the value of compare register1, which can be one of the following,
- **PHC_COMP_0:** specifying compare register0.
- **PHC_COMP_1:** specifying compare register1.

CmpValue is the value to set to compare register

Description:
This function sets ***CmpValue*** to compare register0 of the specified PHC channel if ***CmpReg*** is **PHC_COMP_0**, and sets ***CmpValue*** to compare register1 of the specified PHC channel if ***CmpReg*** is **PHC_COMP_1**

Return:
None

11.2.3.13 PHC_ SetDMAReq

Enable or disable the selected DMA request for a PHC channel.

Prototype:
void
PHC_ SetDMAReq(TSB_PHC_TypeDef * ***PHCx***,
FunctionalState ***NewState***,
uint8_t ***DMAReq***);

Parameters:
PHCx is the specified PHC channel.
NewState specifies specified DMA monitor state of specified PHC channel, which can be:
- **ENABLE:** enables specified DMA request
- **DISABLE:** disables specified DMA request

DMAReq specifies DMA request of the external inputs, which can be
- **PHC_DMA_REQ_CAPTURE_2:** Select DMA request: input capture2.

Description:
This function enables or disables the selected DMA request for the specified PHC channel.

Return:
None

Note:
When interrupt request is disabled, DMA request is not occurred even if DMA request is enabled.

11.2.4 Data Structure Description

11.2.4.1 PHC_InitTypeDef

Data Fields:

uint32_t
Mode selects PHC working mode , which can be set as:

- **PHC_CR_MODE_NORMAL:** counter is increment or decrement when the state transition of the two-phase pulse changes at fourth count
- **PHC_CR_MODE_4TIMES:** counter is increment or decrement when the state transition of the two-phase pulse changes once
- **PHC_CR_MODE_2TIMES_IN0:** counter is increment when the state transition of PHCxIN0 once
- **PHC_CR_MODE_2TIMES_IN1:** counter is increment when the state transition of PHCxIN1 changes once

uint32_t
NoiseFilterCtrl enables and disables the noise filter, which can be set as:

- **PHC_CR_NOISEFILTER_ON:** enables the noise filter
- **PHC_CR_NOISEFILTER_OFF:** disables the noise filter

uint32_t
CountClearCtrl clears or do nothing with the PHC up-and-down counter, which can be set as:

- **PHC_COUNT_CONTINUE:** Do nothing with the PHC up-and-down counter.
- **PHC_COUNT_CLR:** Clears the PHC up-and-down counter

11.2.4.2 PHC_INTFactor

Data Fields:

uint32_t
All: PHC interrupt factor.
Bit
uint32_t
Compare0: 1 a match with the compare register0 is detected
uint32_t
Compare1: 1 a match with the compare register1 is detected
uint32_t
OverFlow : 1 an up-and-down counter is overflow
uint32_t
UnderFlow : 1 an up-and-down counter is underflow
uint32_t
Reserverd : 28 Reserverd

12. SBI

12.1 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/tmpm341_sbi.c, with /Libraries/TX03_Periph_Driver/inc/tmpm341_sbi.h containing the macros, data types, structures and API definitions for use by applications.

12.2 API Functions

12.2.1 Function List

- void SBI_Enable(TSB_SBI_TypeDef* ***SBIx***);
- void SBI_Disable(TSB_SBI_TypeDef* ***SBIx***);
- void SBI_SetI2CACK(TSB_SBI_TypeDef* ***SBIx***, FunctionalState ***NewState***);
- void SBI_InitI2C(TSB_SBI_TypeDef* ***SBIx***, SBI_InitI2CTypeDef* ***InitI2CStruct***);
- void SBI_SetI2CBitNum(TSB_SBI_TypeDef* ***SBIx***, uint32_t ***I2CBitNum***);
- void SBI_SWReset(TSB_SBI_TypeDef* ***SBIx***);
- void SBI_ClearI2CINTReq(TSB_SBI_TypeDef* ***SBIx***);
- void SBI_GenerateI2CStart(TSB_SBI_TypeDef* ***SBIx***);
- void SBI_GenerateI2CStop(TSB_SBI_TypeDef* ***SBIx***);
- SBI_I2CState SBI_GetI2CState(TSB_SBI_TypeDef* ***SBIx***);
- void SBI_SetIdleMode(TSB_SBI_TypeDef* ***SBIx***, FunctionalState ***NewState***);
- void SBI_SetSendData(TSB_SBI_TypeDef* ***SBIx***, uint32_t ***Data***);
- uint32_t SBI_GetReceiveData(TSB_SBI_TypeDef* ***SBIx***);
- void SBI_SetI2CFreeDataMode(TSB_SBI_TypeDef* ***SBIx,*** FunctionalState ***NewState***);

12.2.2 Detailed Description

Functions listed above can be divided into four parts:
1) Configure and control the common functions of each SBI channel are handled by SBI_Enable(), SBI_Disable(), SBI_SetI2CACK(), SBI_SetI2CBitNum(), and SBI_InitI2C().
2) Transfer control of each TMRB channel is handled by SBI_ClearI2CINTReq(), SBI_GenerateI2Cstart(), SBI_GenerateI2Cstop(), SBI_IsI2ClastRxBitSet(), SBI_GetReceiveData().
3) The status indication of each SBI channel is handled by SBI_GetI2CState().
4) SBI_SWReset(), SBI_SetIdleMode() and SBI_EnableI2CfreeDataMode() handle other specified functions.

12.2.3 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.

12.2.3.1 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

12.2.3.2 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

12.2.3.3 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

12.2.3.4 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

12.2.3.5 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

12.2.3.6 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

12.2.3.7 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

12.2.3.8 SBI_GenerateI2CStart

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

Prototype:
void
SBI_GenerateI2CStart(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

12.2.3.9 SBI_GenerateI2CStop

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

Prototype:
void
SBI_GenerateI2CStop(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

12.2.3.10 SBI_GetI2CState

Get the SBI channel state in I2C bus mode.

Prototype:
SBI_I2CState
SBI_GetI2CState(TSB_SBI_TypeDef* ***SBIx***);

Parameters:
SBIx is the specified SBI channel.

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

Return:
The state value of the SBI channel in I2C bus.

12.2.3.11 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

12.2.3.12 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

12.2.3.13 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

12.2.3.14 SBI_SetI2CFreeDataMode

Set SBI channel working in I2C free data mode.

Prototype:
void
SBI_setI2CFreeDataMode(TSB_SBI_TypeDef* ***SBIx***,
FunctionalState ***NewState***);

Parameters:
SBIx is the specified SBI channel.
NewState specifies the state of the SBI when system is idle mode, which can be
- **ENABLE:** enables the SBI channel.
- **DISABLE:** disables the SBI channel.

Description:
The specified SBI channel can transfer data in free data format by calling this function. In free data format, master device always transmits data while slave device always receives data. If the SBI is needed to shift to transfer data in normal I2C format, call **SBI_InitI2C()**.

Return:
None

12.2.4 Data Structure Description

12.2.4.1 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.

12.2.4.2 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.

13. SSP

13.1 Overview

TOSHIBA TMPM341x contains SSP (Synchronous Serial Port) module with one channel.

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 SSPCR0 and SSPCR1.

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm341_ssp.c, with /Libraries/TX03_Periph_Driver/inc/tmpm341_ssp.h containing the macros, data types, structures and API definitions for use by applications.

13.2 API Functions

13.2.1 Function List

- void SSP_Enable(void);
- void SSP_Disable(void);
- void SSP_Init(SSP_InitTypeDef * ***InitStruct***);
- void SSP_SetClkPreScale(uint8_t ***PreScale***, uint8_t ***ClkRate***);
- void SSP_SetFrameFormat(SSP_FrameFormat ***FrameFormat***);
- void SSP_SetClkPolarity(SSP_ClkPolarity ***ClkPolarity***);
- void SSP_SetClkPhase(SSP_ClkPhase ***ClkPhase***);
- void SSP_SetDataSize(uint8_t ***DataSize***);
- void SSP_SetSlaveOutputCtrl(FunctionalState ***NewState***);
- void SSP_SetMSMode(SSP_MS_Mode ***Mode***);
- void SSP_SetLoopBackMode(FunctionalState ***NewState***);
- void SSP_SetTxData(uint16_t ***Data***);
- uint16_t SSP_GetRxData(void);
- WorkState SSP_GetWorkState(void);
- SSP_FIFOState SSP_GetFIFOState(SSP_Direction ***Direction***);
- void SSP_SetINTConfig(uint32_t ***IntSrc***);
- SSP_INTState SSP_GetINTConfig(void);
- SSP_INTState SSP_GetPreEnableINTState(void);
- SSP_INTState SSP_GetPostEnableINTState(void);
- void SSP_ClearINTFlag(uint32_t ***IntSrc***);
- void SSP_SetDMACtrl(SSP_Direction ***Direction***, FunctionalState ***NewState***);

13.2.2 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.

13.2.3 Function Documentation

13.2.3.1 SSP_Enable

Enable the SSP channel.

Prototype:
void
SSP_Enable(void)

Parameters:
None

Description:
This function is to enable SSP channel.

Return:
None

13.2.3.2 SSP_Disable

Disable the SSP channel.

Prototype:
void
SSP_ Disable(void)

Parameters:
None

Description:
This function is to disable SSP channel.

Return:
None

13.2.3.3 SSP_Init

Initialize the SSP channel through the data in structure SSP_InitTypeDef.

Prototype:
void
SSP_Init(SSP_InitTypeDef* ***InitStruct***)

Parameters:

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 SSP_InitTypeDef ***InitStruct***.
It will call the functions below:
SSP_SetFrameFormat(),
SSP_SetClkPreScale(),
SSP_SetClkPolarity(),
SSP_SetClkPhase(),
SSP_SetDataSize(),
SSP_SetMSMode().

Return:
None

13.2.3.4 SSP_SetClkPreScale

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

Prototype:

```
void
SSP_SetClkPreScale(uint8_t PreScale,
                   uint8_t ClkRate)
```

Parameters:
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 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:
BitRate = **fsys** / (**PreScale** × (1 + **ClkRate**))
Where **fsys** is the frequency of system.

Return:
None

13.2.3.5 SSP_SetFrameFormat

Specify the Frame Format of SSP channel.

Prototype:

void
SSP_SetFrameFormat(SSP_FrameFormat ***FrameFormat***)

Parameters:
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 specify the Frame Format of SSP by ***FrameFormat***, generally it is called by **SSP_Init()**.

Return:
None

13.2.3.6 SSP_SetClkPolarity

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

Prototype:
void
SSP_SetClkPolarity(SSP_ClkPolarity ***ClkPolarity***)

Parameters:
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 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

13.2.3.7 SSP_SetClkPhase

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

Prototype:
void
SSP_SetClkPhase(SSP_ClkPhase ***ClkPhase***)

Parameters:
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 specify the clock phase by ***ClkPhase*** when the Frame Format is set as SPI, generally it is called by **SSP_Init()**.

Return:
None

13.2.3.8 SSP_SetDataSize

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

Prototype:
Void
SSP_SetDataSize(uint8_t ***DataSize***)

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

Description:
This function is to set the Rx/Tx Data Size by ***DataSize***, generally it is called by **SSP_Init()**.

Return:
None

13.2.3.9 SSP_SetSlaveOutputCtrl

Enable/Disable slave mode output for the SSP channel.

Prototype:
void
SSP_SetSlaveOutputCtrl(FunctionalState ***NewState***)

Parameters:
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 Enable/Disable slave mode SPDO output by ***NewState***.

Return:
None

13.2.3.10 SSP_SetMSMode

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

Prototype:
void
SSP_SetMSMode(SSP_MS_Mode ***Mode***)

Parameters:
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 select the SSP run in Master mode or Slave mode by ***Mode***.

Return:
None

13.2.3.11 SSP_SetLoopBackMode

Set loop back mode of SSP for the SSP channel.

Prototype:
void
SSP_SetLoopBackMode(FunctionalState ***NewState***)

Parameters:
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 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

13.2.3.12 SSP_SetTxData

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

Prototype:
void
SSP_SetTxData(uint16_t ***Data***)

Parameters:
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 SSP channel.

Return:
None

13.2.3.13 SSP_GetRxData

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

Prototype:
uint16_t
SSP_GetRxData(void)

Parameters:
None

Description:
This function will read received data from Rx FIFO of the SSP channel.

Return:
Data with uint16_t type

13.2.3.14 SSP_GetWorkState

Get the Busy or Idle state of the SSP channel.

Prototype:
WorkState
SSP_GetWorkState(void)

Parameters:
None

Description:
This function will get the Busy/Idle state of the SSP channel.

Return:
WorkState type, the value means:
BUSY: SSP module is busy.
DONE: SSP module is idle.

13.2.3.15 SSP_GetFIFOState

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

Prototype:
SSP_FIFOState
SSP_GetFIFOState(SSP_Direction ***Direction***)

Parameters:
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 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_SSP, SSP_TX);
if ((fifoState == SSP_FIFO_EMPTY) || (fifoState == SSP_FIFO_NORMAL))
   { SSP_SetTxData(TSB_SSP, 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

13.2.3.16 SSP_SetINTConfig

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

Prototype:
void
SSP_SetINTConfig(uint32_t ***IntSrc***)

Parameters:
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 enable/disable interrupts by ***IntSrc***.
For example, we can enable Tx and Rx interrupt by code like below:

```
SSP_SetINTConfig( TSB_SSP, SSP_INTCFG_RX | SSP_INTCFG_TX )
```

Return:
None

13.2.3.17 SSP_GetINTConfig

Get the Enable/Disable setting for each Interrupt source in the SSP channel.

Prototype:
SSP_INTState
SSP_GetINTConfig(void)

Parameters:
None

Description:
This function will get the masked interrupt status of the SSP channel.
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.

13.2.3.18 SSP_GetPreEnableINTState

Get the raw status of each interrupt source in the SSP channel.

Prototype:

SSP_INTState
SSP_GetPreEnableINTState(void)

Parameters:
None

Description:
This function will get the pre-enable interrupt status of the SSP channel.

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.

13.2.3.19 SSP_GetPostEnableINTState

Get the SSP channel post-enable interrupt status. (after masked)

Prototype:
SSP_INTState
SSP_GetPostEnableINTState(void)

Parameters:
None.

Description:
This function will get post-enable interrupt status of the SSP channel.

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.

13.2.3.20 SSP_ClearINTFlag

Clear interrupt flag of SSP channel by writing '1' to correspond bit.

Prototype:
void
SSP_ClearINTFlag(uint32_t ***IntSrc***)

Parameters:
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 SSP channel.

Return:
None

13.2.3.21 SSP_SetDMACtrl

Enable/Disable the DMA FIFO for Rx/Tx of SSP channel.

Prototype:
void
SSP_SetDMACtrl(SSP_Direction ***Direction***,
FunctionalState ***NewState***)

Parameters:
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 SSP channel.

Return:
None

13.2.4 Data Structure Description

13.2.4.1 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.

13.2.4.2 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.

14. TMRB

14.1 Overview

TOSHIBA TMPM341x contains 10 channels of a 16-bit up-counter, two 16-bit timer registers, two 16-bit capture registers, two comparators, a capture input control, a timer flip-flop and its associated control circuit. (TMRB0 through TMRB9). Each channel can operate in the following modes:

- 16-bit interval timer mode
- 16-bit event counter mode
- 16-bit programmable square-wave output mode (PPG)
- Timer synchronous mode (capable of setting output mode for each 4ch)

The use of the capture function allows TMRBs to perform the following three measurements:

- Frequency measurement
- Pulse width measurement
- Time difference measurement

The TMRB driver APIs provide a set of functions to configure each channel, such as setting the clock division, trailingtiming and leadingtiming 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/tmpm341_tmrb.c, with /Libraries/TX03_Periph_Driver/inc/tmpm341_tmrb.h containing the macros, data types, structures and API definitions for use by applications.

14.2 API Functions

14.2.1 Function List

- void TMRB_Enable(TSB_TB_TypeDef * ***TBx***);
- void TMRB_Disable(TSB_TB_TypeDef * ***TBx***);
- void TMRB_SetRunState(TSB_TB_TypeDef * ***TBx***, uint32_t ***Cmd***);
- void TMRB_Init(TSB_TB_TypeDef * ***TBx,*** TMRB_InitTypeDef * ***InitStruct***);
- void TMRB_SetCaptureTiming(TSB_TB_TypeDef * ***TBx***, uint32_t ***CaptureTiming***);
- void TMRB_SetFlipFlop(TSB_TB_TypeDef * ***TBx***, TMRB_FFOutputTypeDef * ***FFStruct***);
- TMRB_INTFactor TMRB_GetINTFactor(TSB_TB_TypeDef * ***TBx***);
- void TMRB_SetINTMask(TSB_TB_TypeDef * ***TBx***, uint32_t ***INTMask***);
- void TMRB_ChangeLeadingTiming(TSB_TB_TypeDef * ***TBx***, uint32_t ***LeadingTiming***);
- void TMRB_ChangeTrailingTiming(TSB_TB_TypeDef * ***TBx***, uint32_t ***TrailingTiming***);
- uint16_t TMRB_GetUpCntValue(TSB_TB_TypeDef * ***TBx***);
- uint16_t TMRB_GetCaptureValue(TSB_TB_TypeDef * ***TBx***, uint8_t ***CapReg***);
- void TMRB_ExecuteSWCapture(TSB_TB_TypeDef * ***TBx***);

- void TMRB_SetIdleMode(TSB_TB_TypeDef * ***TBx***, FunctionalState ***NewState***);
- void TMRB_SetSyncMode(TSB_TB_TypeDef * ***TBx***, FunctionalState ***NewState***);
- void TMRB_SetDoubleBuf(TSB_TB_TypeDef * ***TBx***, FunctionalState ***NewState***);
- void TMRB_SetExtStartTrg(TSB_TB_TypeDef * ***TBx***, FunctionalState ***NewState***, uint8_t ***TrgMode***);
- void TMRB_SetClkInCoreHalt(TSB_TB_TypeDef * ***TBx***, uint8_t ***ClkState***);
- void TMRB_SetExtInput(TSB_TB_TypeDef * ***TBx***, uint8_t ***ExtInput***);
- void TMRB_SetDMAReq(TSB_TB_TypeDef * TBx, FunctionalState ***NewState***, uint8_t ***DMAReq***);

14.2.2 Detailed Description

Functions listed above can be divided into four parts:

3) 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().
4) Capture function of each TMRB channel is handled by TMRB_SetCaptureTiming(), and TMRB_ExecuteSWCapture().
5) The status indication of each TMRB channel is handled by TMRB_GetINTFactor(), TMRB_GetUpCntValue() and TMRB_GetCaptureValue().
6) TMRB_SetFlipFlop(), TMRB_SetINTMask(), TMRB_SetIdleMode(), TMRB_SetSyncMode(), TMRB_SetDoubleBuf(), TMRB_SetExtStartTrg() and TMRB_SetClkInCoreHalt (), TMRB_SetExtInput(), TMRB_SetDMAReq() handle other specified functions.

14.2.3 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.

14.2.3.1 TMRB_Enable

Enable the specified TMRB channel.

Prototype:
void
TMRB_Enable(TSB_TB_TypeDef* ***TBx***)

Parameters:
TBx is the specified TMRB channel.

Description:
This function will enable the specified TMRB channel selected by ***TBx***.

Return:
None

14.2.3.2 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

14.2.3.3 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

14.2.3.4 TMRB_Init

Initialize the specified TMRB channel.

Prototype:
void
TMRB_Init(TSB_TB_TypeDef* ***TBx***,
TMRB_InitTypeDef* ***InitStruct***)

Parameters:
TBx is the specified TMRB channel.
InitStruct is the structure containing basic TMRB configuration including count mode, source clock division, leadingtiming value, trailingtiming value and up-counter work mode (refer to “Data Structure Description” for details).

Description:
This function will initialize and configure the count mode, clock division, up-counter setting, trailingtiming and leadingtiming duration for the specified TMRB channel selected by ***TBx***.

Return:
None

14.2.3.5 TMRB_SetCaptureTiming

Configure the capture timing.

Prototype:
void
TMRB_SetCaptureTiming(TSB_TB_TypeDef* ***TBx***,
uint32_t ***CaptureTiming***)

Parameters:
TBx is the specified TMRB channel.

CaptureTiming specifies TMRB capture timing, which can be

- **TMRB_DISABLE_CAPTURE**: Disable the capture function of the specified TMRB channel.
- **TMRB_CAPTURE_IN_RISING**: Takes count values into capture register 0 (TBxCP0) upon rising of TBxIN pin input.
- **TMRB_CAPTURE_IN_RISING_FALLING**: Takes count values into capture register 0 (TBxCP0) upon rising of TBxIN pin input and takes count values into capture register 1 (TBxCP1) upon falling of TBxIN pin input.
- **TMRB_CAPTURE_OUTPUT_EDGE**: Takes count values into capture register 0 (TBxCP0) upon rising of TBxOUT pin input and takes count values into capture register 1 (TBxCP1) upon falling of TBxOUT pin input.

Description:
If ***CaptureTiming*** is set as **TMRB_CAPTURE_IN_RISING**, then at the time of the rising edge of input port TBxIN, the value in up-counter will be captured and saved into capture register0 (TBxCP0) of the TMRB channel.
If ***CaptureTiming*** is set as **TMRB_CAPTURE_IN_RISING_FALLING**, then at the time of the rising edge of input port TBxIN, the value in up-counter will be captured and saved into capture register0 (TBxCP0) of the TMRB channel. And the value in up-counter will be captured and saved into capture register1 (TBxCP1) upon falling of TBxIN pin input.
If ***CaptureTiming*** is set as **TMRB_CAPTURE_OUTPUT_EDGE**, then at the time of the rising edge of port TBxOUT pin, the value in up-counter will be captured and saved into capture register0 (TBxCP0) of the TMRB channel. And the value in up-counter will be captured and saved into capture register1 (TBxCP1) upon falling of TBxOUT pin input.
The flip-flop output of TMRB7, TMRB8 and TMRB9 can be used as the capture trigger of other channels.

TMRB0~1: TB7OUT
TMRB2~3: TB8OUT
TMRB4~6: TB9OUT

Return:
None

14.2.3.6 TMRB_SetFlipFlop

Configure the flip-flop function of the specified TMRB channel.

Prototype:
void
TMRB_SetFlipFlop(TSB_TB_TypeDef* ***TBx***,
TMRB_FFOutputTypeDef* ***FFStruct***)

Parameters:
TBx is the specified TMRB channel.
FFStruct is the structure containing TMRB flip-flop function configuration including flip-flop output level and flip-flop-reverse trigger (refer to "Data Structure Description" for details).

Description:
This function will set the timing of changing the flip-flop output of the specified TMRB channel. Also the level of the output can be controlled by this API.

Return:
None

14.2.3.7 TMRB_GetINTFactor

Indicate what causes the interrupt.

Prototype:
TMRB_INTFactor
TMRB_GetINTFactor(TSB_TB_TypeDef* ***TBx***)

Parameters:
TBx is the specified TMRB channel.

Description:
This function should be used in ISR to indicate the factor of interrupt. Bit of **MatchLeadingTiming** indicates if the up-counter matches with leadingtiming value, Bit of **MatchTrailingTiming** Indicates if the up-counter matches with trailingtiming value, and bit of **Overflow** indicates if overflow had occurred before the interrupt.

Return:
TMRB Interrupt factor. Each bit has the following meaning:
MatchLeadingTiming(Bit0): a match with the leadingtiming value is detected
MatchTrailingTiming(Bit1): a match with the trailingtiming value is detected
OverFlow(Bit2): an up-counter is overflow

Note:
It is recommended to use the following method to process different interrupt factor

```
TMRB_INTFactor factor = TMRB_GetINTFactor(TSB_TB0);
if (factor.Bit.MatchLeadingTiming) {
    // Do A
}

if (factor.Bit.MatchTrailingTiming) {
    // Do B
}

if (factor.Bit.OverFlow) {
```

```
    // Do C
}
```

14.2.3.8 TMRB_SetINTMask

Mask the specified TMRB interrupt.

Prototype:
void
TMRB_SetINTMask(TSB_TB_TypeDef* ***TBx***,
uint32_t ***INTMask***)

Parameters:
TBx is the specified TMRB channel.
INTMask specifies the interrupt to be masked, which can be

- **TMRB_MASK_MATCH_TRAILINGTIMING_INT**: Mask the interrupt the factor of which is that the value in up-counter and trailingtiming are match.
- **TMRB_MASK_MATCH_LEADINGTIMING_INT**: Mask the interrupt the factor of which is that the value in up-counter and leadingtiming are match.
- **TMRB_MASK_OVERFLOW_INT**: Mask the interrupt the factor of which is the occurrence of overflow.
- **TMRB_NO_INT_MASK**: Unmask the interrupt.

Description:
If **TMRB_MASK_MATCH_TRAILINGTIMING_INT** is selected, the interrupt of the specified TMRB channel will not happen when the value in up-counter and trailingtiming are match.
If **TMRB_MASK_MATCH_LEADINGTIMING_INT** is selected, the interrupt of the specified TMRB channel will not happen when the value in up-counter and leadingtiming are match.
If **TMRB_MASK_OVERFLOW_INT** is selected, the interrupt of the specified TMRB channel will not happen even if there is an occurrence of overflow.
If **TMRB_NO_INT_MASK** is selected, all interrupt masks will be cleared.

Return:
None

14.2.3.9 TMRB_ChangeLeadingTiming

Change the value of leadingtiming for the specified channel.

Prototype:
void
TMRB_ChangeLeadingTiming(TSB_TB_TypeDef* ***TBx***,
uint32_t ***LeadingTiming***)

Parameters:
TBx is the specified TMRB channel.
LeadingTiming specifies the value of leadingtiming, max. is 0xFFFF.

Description:
This function will specify the absolute value of leadingtiming for the specified TMRB. The actual interval of leadingtiming depends on the configuration of CG and the value of ***ClkDiv*** (refer to “Data Structure Description” for details).

Return:
None

Note:
LeadingTiming can not exceed ***TrailingTiming***.

14.2.3.10 TMRB_ChangeTrailingTiming

Change the value of trailingtiming for the specified channel.

Prototype:
void
TMRB_ChangeTrailingTiming(TSB_TB_TypeDef* ***TBx***,
uint32_t ***TrailingTiming***)

Parameters:
TBx is the specified TMRB channel.
TrailingTiming specifies the value of trailingtiming, max. is 0xFFFF.

Description:
This function will specify the absolute value of trailingtiming for the specified TMRB. The actual interval of trailingtiming depends on the configuration of CG and the value of ***ClkDiv*** (refer to “Data Structure Description” for details).

Return:
None

Note:
TrailingTiming must be not smaller than ***LeadingTiming***. And the value of TBxRG0/1 must be set as TBxRG0 < TBxRG1 in PPG mode.

14.2.3.11 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

14.2.3.12 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

14.2.3.13 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

14.2.3.14 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

14.2.3.15 TMRB_SetSyncMode

Enable or disable the synchronous mode of specified TMRB channel.

Prototype:
void
TMRB_SetSyncMode(TSB_TB_TypeDef* ***TBx***,
FunctionalState ***NewState***)

Parameters:
TBx is the specified TMRB channel, which can be
TSB_TB0, TSB_TB1, TSB_TB2, TSB_TB3, TSB_TB4, TSB_TB5, TSB_TB6, TSB_TB7.

NewState specifies the state of the synchronous mode of the TMRB, which can be
- **ENABLE:** enables the synchronous mode,
- **DISABLE:** disables the synchronous mode.

Description:
If the synchronous mode is enabled for TMRB0 through TMRB3, their start timing is synchronized with TMRB0. If the synchronous mode is enabled for TMRB4 through TMRB7, their start timing is synchronized with TMRB4.

Return:
None

Note:
TMRB0 through TMRB3, TMRB4through TMRB7 must start counting by calling **TMRB_SetRunState()** before TMRB0, TMRB4 start counting, so that start timing can be synchronized.

14.2.3.16 TMRB_SetDoubleBuf

Enable or disable double buffering for the specified TMRB channel.

Prototype:
void
TMRB_SetDoubleBuf(TSB_TB_TypeDef* ***TBx***,
FunctionalState ***NewState***)

Parameters:
TBx is the specified TMRB channel.

NewState specifies the state of double buffering of the TMRB, which can be
- **ENABLE:** enables double buffering,
- **DISABLE:** disables double buffering.

Description:
This function will enable or disable double buffering for the specified TMRB channel.
Return:
None

14.2.3.17 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

14.2.3.18 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

14.2.3.19 TMRB_SetExtInput

Controls the external inputs.

Prototype:
void
TMRB_SetExtInput (TSB_TB_TypeDef* ***TBx***,
uint8_t ***ExtInput***)

Parameters:
TBx is the specified TMRB channel**..**

ExtInput Controls the external inputs selection between TBxIN0/1 or PHCxIN0/1.
- **TMRB_EXT_INPUT_TBxIN:** Select external inputs TBxIN0/1.
- **TMRB_EXT_INPUT_PHCxIN:** Select external inputs PHCxIN0/1.

Description:
This function will select the external inputs.

Return:
None

14.2.3.20 TMRB_SetDMAReq

Enable or disable the selected DMA request for a TMRB channel.

Prototype:
void
TMRB_SetExtStartTrg (TSB_TB_TypeDef* ***TBx***,
FunctionalState ***NewState***,
uint8_t ***DMAReq***)

Parameters:
TBx is the specified TMRB channel.
NewState enable or disable the DMA request, which can be
- **ENABLE:** enable the DMA request,
- **DISABLE:** disable the DMA request.

DMAReq specifies DMA request of the external inputs, which can be
- **TMRB_DMA_REQ_CMP_MATCH:** Select DMA request : compare match.
- **TMRB_DMA_REQ_CAPTURE_1:** Select DMA request : input capture1.
- **TMRB_DMA_REQ_CAPTURE_0:** Select DMA request : input capture0.

Description:
This function will enable or disable the selected DMA request for the specified TMRB channel.
Note:
When mask configuration by TBxIM register is valid, DMA request does not issue even if it is enabled.

14.2.4 Data Structure Description

14.2.4.1 TMRB_InitTypeDef

Data Fields:

uint32_t
Mode selects TMRB working mode between **TMRB_INTERVAL_TIMER** (internal interval timer mode) and **TMRB_EVENT_CNT** (external event counter).

uint32_t
ClkDiv specifies the division of the source clock for the internal interval timer, which can be set as:

- **TMRB_CLK_DIV_2**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 2;
- **TMRB_CLK_DIV_8**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 8;
- **TMRB_CLK_DIV_32**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 32.
- **TMRB_CLK_DIV_64**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 64;
- **TMRB_CLK_DIV_128**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 128.
- **TMRB_CLK_DIV_256**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 256;
- **TMRB_CLK_DIV_512**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 512

uint32_t
TrailingTiming specifies the trailingtiming value to be written into TBnRG1, max. 0xFFFF.

uint32_t
UpCntCtrl selects up-counter work mode, which can be set as:

- **TMRB_FREE_RUN**, which means that the up-counter will not stop counting even when the value in it is match with trailingtiming, until it reaches 0xFFFF, then it will be cleared and starting counting from 0,
- **TMRB_AUTO_CLEAR**, which means that the up-counter will restart counting from 0 immediately when the value in up-counter matches **TrailingTiming**.

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

14.2.4.2 TMRB_FFOutputTypeDef

Data Fields:

uint32_t
FlipflopCtrl selects the level of flip-flop output which can be

- **TMRB_FLIPFLOP_INVERT:** setting output reversed by using software.
- **TMRB_FLIPFLOP_SET:** setting output to be high level.
- **TMRB_FLIPFLOP_CLEAR:** setting output to be low level.

uint32_t
FlipflopReverseTrg specifies the reverse trigger of the flip-flop output, which can be set as:

- **TMRB_DISALBE_FLIPFLOP**, which disables the flip-flop output reverse trigger,
- **TMRB_FLIPFLOP_TAKE_CATPURE_0**, which means that the reversing flip-flop output will be triggered when the up-counter value is taken into capture register 0,
- **TMRB_FLIPFLOP_TAKE_CATPURE_1**, which means that the reversing flip-flop output will be triggered when the up-counter value is taken into capture register 1,
- **TMRB_FLIPFLOP_MATCH_TRAILINGTIMING**, which means that the reversing flip-flop output will be triggered when the up-counter matches the trailingtiming,
- **TMRB_FLIPFLOP_MATCH_LEADINGTIMING**, which means that the reversing flip-flop output will be triggered when the up-counter matches the leadingtiming.

14.2.4.3 TMRB_INTFactor

Data Fields:

uint32_t
All: TMRB interrupt factor.
Bit
uint32_t
MatchLeadingTiming: 1 a match with the leadingtiming value is detected
uint32_t
MatchTrailingTiming : 1 a match with the trailingtiming value is detected
uint32_t
OverFlow : 1 an up-counter is overflow
uint32_t
Reserverd : 29 -

15. TMRD

15.1 Overview

TOSHIBA TMPM341x has TMRD module (high resolution 16-bit Timer/PPG outputs), which consists of two timer units (TMRD0 and TMRD1) and two clock setting circuits (prescalers) for supplying clocks to these timer units. The functions are as follows.

- 16-bit interval timer
- 16-bit programmable pulse generation (PPG)

16-bit interval timer has the following two modes.

- Timer mode that TMRD0 and TMRD1 operate independently
- Interlock timer mode that can start TMRD0 and TMRD1 at the same time

16-bit programmable pulse generation has the following two modes.

- PPG mode that TMRD0 and TMRD1 independently output preprogrammed pulses
- PPG mode that can change the phase relation between the pulse output by TMRD0 and that output by TMRD1 in the range from -180 degree to +180 degree

TMRD consists of the clock setting circuit and two unit timers.

The TMRD driver APIs provide a set of functions to configure TMRD module, such as setting the clock operation, timing parameters, PPG output phase, wave edge adjust, DMA request setting and set the compare 0 interrupt source, up counter's clearing way and so on.

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm341_tmrd.c, with /Libraries/TX03_Periph_Driver/inc/tmpm341_tmrd.h containing the macros, data types, structures and API definitions for use by applications.

15.2 API Functions

15.2.1 Function List

- void TMRD_Enable(TSB_TD_TypeDef * ***TDx***);
- void TMRD_Disable(TSB_TD_TypeDef * ***TDx***);
- void TMRD_SetRunStateInHalt(uint8_t ***RunState***);
- void TMRD_SetRunStateInIdle(TSB_TD_TypeDef * ***TDx***, uint8_t ***RunState***);
- void TMRD_SetMode(uint8_t ***Mode***);
- void TMRD_SetClkDivision(TSB_TD_TypeDef * ***TDx***, uint8_t ***ClkDiv***);
- void TMRD_SetUpCntCtrl(TSB_TD_TypeDef * ***TDx***, uint8_t ***UpCntCtrl***);
- void TMRD_SetPPGInitLeadingEdge(uint8_t ***PPGChannel***, uint8_t ***WaveEdge***);
- void TMRD_SetCMPRegWritePath(TSB_TD_TypeDef * ***TDx***, uint8_t ***WritePath***);
- void TMRD_SetCMP0INTSrc(TSB_TD_TypeDef * ***TDx***, uint8_t ***INTSrc***);
- void TMRD_SetRunState(TSB_TD_TypeDef * ***TDx***, uint8_t ***RunState***);
- void TMRD_SetPhaseRelation(uint8_t ***PhaseRelation***);
- void TMRD_EnableUpdateCMPReg(TSB_TD_TypeDef * ***TDx***);
- void TMRD_SetDMAReq(TSB_TD_TypeDef * ***TDx***, FunctionalState ***NewState***);

- void TMRD_SetInitTiming(TSB_TD_TypeDef * ***TDx***, TMRD_TimingTypeDef * ***TimingStruct***);
- void TMRD_ChangeTiming(uint8_t ***TimingType***, uint32_t ***Timing***);
- uint16_t TMRD_GetTiming(uint8_t ***TimingType***);

15.2.2 Detailed Description

Functions listed above can be divided into five parts:

3) Configure and control the common functions of each TMRD channel are handled by TMRD_Enable(), TMRD_Disable(), TMRD_SetUpCntCtrl (), TMRD_SetMode() and TMRD_SetRunState().
4) Clock source and division setting are handled by TMRD_SetRunStateInHalt(), TMRD_SetRunStateInIdle(), TMRD_SetClkDivision().
5) PPG output and phase control are handled by TMRD_SetPPGInitLeadingEdge(), TMRD_SetPhaseRelation().
6) Compare register and timer register control are handled by TMRD_SetCMPRegWritePath(), TMRD_EnableUpdateCMPReg(), TMRD_SetInitTiming(), TMRD_ChangeTiming() and TMRD_GetTiming().
7) Interrupt and DMA feature are handled by TMRD_SetCMP0INTSrc() and TMRD_SetDMAReq().

15.2.3 Function Documentation

Note: in all of the following APIs, unless otherwise specified, the parameter:
"TSB_TD_TypeDef * ***TDx***" can be one of the following values:
TSB_TD0, TSB_TD1.

15.2.3.1 TMRD_Enable

Enable clock signal input to TMRD channel.

Prototype:
void
TMRD_Enable(TSB_TD_TypeDef * ***TDx***)

Parameters:
TDx is the specified TMRD channel.

Description:
This function will enable the clock signal input to TMRD channel selected by ***TDx***.

Return:
None

15.2.3.2 TMRD_Disable

Disable the clock signal input to TMRD channel.

Prototype:
void
TMRD_Disable(TSB_TD_TypeDef * ***TDx***)

Parameters:
TDx is the specified TMRD channel.

Description:
This function will disable the clock signal input to TMRD channel selected by ***TDx***.

Return:
None

15.2.3.3 TMRD_SetRunStateInHalt

Set TMRD operation if a HALT instruction is executed during debugging.

Prototype:
void
TMRD_SetRunStateInHalt(uint8_t ***RunState***)

Parameters:
RunState sets the TMRD operation status, which can be:
- **TMRD_RUN:** Operation if a HALT instruction is executed during debugging.
- **TMRD_STOP:** Stop if a HALT instruction is executed during debugging.

Description:
This function will set the operation if a HALT instruction is executed during debugging.
Return:
None

15.2.3.4 TMRD_SetRunStateInIdle

Set TMRD operation during the IDLE mode.

Prototype:
void
TMRD_SetRunStateInIdle(TSB_TD_TypeDef * ***TDx***,
uint8_t ***RunState***)

Parameters:
TDx is the specified TMRD channel.

RunState sets the TMRD operation status, which can be:
- **TMRD_RUN:** Operation during the IDLE mode.
- **TMRD_STOP:** Stop during the IDLE mode.

Description:
This function will set the TMRD operation during the IDLE mode.

Return:
None

15.2.3.5 TMRD_SetMode

Set the operation mode for TMRD0 and TMRD1.

Prototype:
void
TMRD_SetMode(uint8_t ***Mode***)

Parameters:
Mode specifies TMRD operation mode, which can be
- **TMRD_MODE_BOTH_TMR**: TMRD0: Timer mode, TMRD1: Timer mode.
- **TMRD_MODE_0TMR_1PPG**: TMRD0: Timer mode, TMRD1: PPG mode.
- **TMRD_MODE_0PPG_1TMR**: TMRD0: PPG mode, TMRD1: Timer mode.
- **TMRD_MODE_BOTH_PPG**: TMRD0: PPG mode, TMRD1: PPG mode.
- **TMRD_MODE_INTERLOCK_TMR**: Interlock timer mode that allows TMRD0 and TMRD1 to start simultaneously.
- **TMRD_MODE_INTERLOCK_PPG**: Interlock PPG mode that allows TMRD0 and TMRD1 to operate in tandem.

Description:
This function will set the operation mode for TMRD0 and TMRD1. If operation mode is set to interlock PPG mode, the phase relationships between pulse generated by TMRD1 and TMRD0 can be changed.

Return:
None

Note:
If set the operation mode to **TMRD_MODE_INTERLOCK_PPG,** TMRDCLK0 and TMRDCLK1 clock cannot be configured respectively, and TMRDCLK1 and TMRDCLK0 have the same frequencies.

15.2.3.6 TMRD_SetClkDivision

Set the clock prescaler of TMRD0 and TMRD1.

Prototype:
void
TMRD_SetClkDivision(TSB_TD_TypeDef* ***TDx***,
uint8_t ***ClkDiv***)

Parameters:
TDx is the specified TMRD channel.

ClkDiv is the prescaler factor, which can be
- **TMRD_CLK_DIV_1**: TMRDCLK = ftmrd.
- **TMRD_CLK_DIV_2**: TMRDCLK = ftmrd/2.
- **TMRD_CLK_DIV_4**: TMRDCLK = ftmrd/4.
- **TMRD_CLK_DIV_8**: TMRDCLK = ftmrd/8.
- **TMRD_CLK_DIV_16**: TMRDCLK = ftmrd/16.

Description:
This function will select a clock prescaler of TMRD0 and TMRD1.

Return:
None

Note:

In the interlock PPG mode, setting the clock prescaler of TMRD1 becomes invalid, and the prescaler of TMRD1 will keep the same value with TMRD0. So setting the prescaler of TMRD0 by using this function is enough.

15.2.3.7 TMRD_SetUpCntCtrl

Select the timer up-counter operation when math the cycle.

Prototype:
void TMRD_SetUpCntCtrl(TSB_TD_TypeDef * ***TDx***, uint8_t ***UpCntCtrl***)

Parameters:
TDx is the specified TMRD channel.

UpCntCtrl is the operation mode of counter zero clearing, which can be
- **TMRD_FREE_RUN**: Operate as a free-run counter even if a match is detected.
- **TMRD_AUTO_CLEAR**: Zero cleared if a match is detected.

Description:
This function will select the up-counter operation when the match signal of CP00/CP10 is generated. If choose **TMRD_FREE_RUN**, the counter will be zero cleared when arrived at the maximum value 0xFFFF, and if choose **TMRD_AUTO_CLEAR**, the up-counter will be zero cleared when match the CP00/CP01.

Return:
None

Note:
In the PPG and interlock PPG mode, setting ***UpCntCtrl*** to **TMRD_FREE_RUN** becomes invalid.

15.2.3.8 TMRD_SetPPGInitLeadingEdge

Set the initial leading/trailing edge of PPG channels.

Prototype:
void
TMRD_SetPPGInitLeadingEdge(uint8_t ***PPGChannel***,
unt8_t ***WaveEdge***)

Parameters:
PPGChannel is the specified PPG output channel, which can be
- **TMRD_PPG_CHANNEL_A0**: PPG output signal a0.
- **TMRD_PPG_CHANNEL_A1**: PPG output signal a1.
- **TMRD_PPG_CHANNEL_B0**: PPG output signal b0.
- **TMRD_PPG_CHANNEL_B1**: PPG output signal b1.

WaveEdge specifies the initial wave edge of leading edge, which can be
- **TMRD_WAVE_EDGE_RISING**: Leading edge is rising edge and trailing edge is falling edge.

- **TMRD_WAVE_EDGE_FALLING**: Leading edge is falling edge and trailing edge is rising edge..

Description:
This function will set the initial setting of leading edge/trailing edge of a PPG output signal specified by ***WaveEdge***.

Return:
None

15.2.3.9 TMRD_SetCMPRegWritePath

Set a write path to update the compare register.

Prototype:
void
TMRD_SetCMPRegWritePath(TSB_TD_TypeDef* ***TDx***,
uint8_t ***WritePath***)

Parameters:
TDx is the specified TMRD channel.

WritePath is the path to update compare registers, which can be
- **TMRD_CMP_WRITE_DIRECT**: Directly written to the compare register by CPU instructions.
- **TMRD_CMP_WRITE_INDIRECT**: Write to compare register via the timer register. Enable flag is required.

Description:
This function will set a write path to update the compare register. When set ***WritePath*** to **TMRD_CMP_WRITE_DIRECT,** at the time when data is written into the timer register (TDxRGm), the same value is written to the corresponding compare register (TDxCPm) simultaneously. In this case, it is not necessary for an enable flag.

If set ***WritePath*** to **TMRD_CMP_WRITE_INDIRECT,** enable update flag is required by using function TMRD_EnableUpdateCMPReg(). About the update occasion, please refer to the description below in each mode:

In the timer mode:
- TD0MOD<TDCLE>="0": A value in the compare register (TDxCPm) is updated to the value of the timer register (TDxRGm) when the COUNTER overflows.
- TD0MOD<TDCLE>="1": A value in the compare register (TDxCPm) is updated to the value of the timer register (TDxRGm) when the value set in the comparator00/10 (CP00/CP10) matches the value in the counter.

In the PPG mode:
When the value set in the comparator00/10 (CP00/10) matches the value in the counter, the value in the compare register (TDxCPm) is updated to the value in the timer register (TDxRGm).

In the interlock PPG mode:
For TMRD0, the update method is the same as in PPG mode, which described above.

For TMRD1, when the value set in the comparator05 (CP05) matches the value in the compare register, the value of the compare register (TD1CPm) is updated to the value of the timer register (TD1RGm).

Return:
None

Note:
In the interlock PPG mode, writing path of TMRD1 is selected as the values in TMRD0, so only setting the writing path of TMRD0 is enough.

15.2.3.10 TMRD_SetCMP0INTSrc

Set the interrupt source of compare interrupt 0.

Prototype:
void
TMRD_SetCMP0INTSrc(TSB_TD_TypeDef* ***TDx***,
uint8_t ***INTSrc***)

Parameters:
TDx is the specified TMRD channel.

INTSrc selects the interrupt factor, which can be
- **TMRD_INT_NONE**: No interrupt factor.
- **TMRD_INT_MATCH_CYCLE**: A match signal from CP00/CP10.
- **TMRD_INT_MATCH_PHASE**: A match signal from CP05(only TMRD0 has).
- **TMRD_INT_UC_OVERFLOW**: Overflow of COUNTER.

Description:
This function will choose the interrupt source of compare interrupt 0.

Return:
None

Note:
In the PPG mode, **TMRD_INT_UC_OVERFLOW** is invalid as an interrupt factor.
In the interlock PPG mode, **TMRD_INT_MATCH_CYCLE** of TMRD1 becomes invalid as an interrupt factor.

TMRD_INT_MATCH_PHASE is invalid as an interrupt factor of TMRD1.

15.2.3.11 TMRD_SetRunState

Set the count operation of TMRD.

Prototype:
TMRD_SetRunState(TSB_TD_TypeDef* ***TDx***,
uint8_t ***RunState***)

Parameters:
TDx is the specified TMRD channel.

RunState is the counter operation of TMRD, which can be:

- **TMRD_RUN:** Starts the count operation of TMRDx.
- **TMRD_STOP:** Stop the count operation of TMRDx and zero clears COUNTER.

Description:
This function will set the count operation of TMRD.

Return:
None

Note:
In interlock timer mode and interlock PPG mode, this function becomes invalid for TMRD1, because TMRD1 will start operation in tandem with COUNTER0 of TMRD0.

15.2.3.12 TMRD_SetPhaseRelation

Set the phase relation of phase B to phase A.

Prototype:
void
TMRD_SetPhaseRelation(uint8_t ***PhaseRelation***)

Parameters:
PhaseRelation is the phase relation of phase B to phase A, which can be:

- **TMRD_PHASE_DELAY_OR_SAME:** Phase B delays or the same as phase A.
- **TMRD_PHASE_FAST_OR_SAME:** Phase B is fast or the same as phase A.

Description:
This function will set the phase relation of phase B to phase A.

Return:
None

Note:
This function is valid only in the interlock PPG mode. The output of the phase A and the phase B cannot be switched in the timer mode, the interlock mode and the PPG mode.

15.2.3.13 TMRD_EnableUpdateCMPReg

Enable to update the compare register.

Prototype:
void
TMRD_EnableUpdateCMPReg(TSB_TD_TypeDef* ***TDx***)

Parameters:

TDx is the specified TMRD channel.

Description:
This function will set a enable flag to update the compare register, for more details, please refer to the section about TMRD_SetCMPRegWritePath().

Return:
None

Note:
In the interlock PPG mode, when use this function to set the update enable flag of TMRD0, the enable flag of TMRD1 will set at the same time, please don't use this function to set TMRD1 again. And this flag will be zero cleared when a match of compare register 05(CP05) is detected.

15.2.3.14 TMRD_SetDMAReq

Enable or disable the DMA request of INTTDxCMP0 signal.

Prototype:
void
TMRD_SetDMAReq(TSB_TD_TypeDef* ***TDx***,
FunctionalState ***NewState***)

Parameters:
TDx is the specified TMRD channel.

RunState is the state of DMA request, which can be:
- **ENABLE:** Enable the DMA request.
- **DISABLE:** Disable the DMA request.

Description:
This function will enable and disable the DMA request of INTTDxCMP0, which is a factor to generate a DMA request.

Return:
None

15.2.3.15 TMRD_SetInitTiming

Initialize TMRD timing parameters.

Prototype:
void
TMRD_SetInitTiming(TSB_TD_TypeDef* ***TDx***,
TMRD_TimingTypeDef* ***TimingStruct***)

Parameters:
TDx is the specified TMRD channel.

TimingStruct is the pointer of TMRD timing parameters structure. Please refer to Data Structure for details.

Description:
This function will initialize TMRD timing parameters, which is included cycle timing, leading timing, trailing timing, phase shift timing and so on.

Return:
None

Note:
Please refer to Data Structure Description to get the setting range of each parameter in structure ***TimingStruct.***
CPRG0: ***TimingStruct***. Cycle
CPRG1: ***TimingStruct***. LeadingTiming0
CPRG2: ***TimingStruct***. TrailingTiming0
CPRG3: ***TimingStruct***. LeadingTiming1
CPRG4: ***TimingStruct***. TrailingTiming1
CPRG5: ***TimingStruct***. PhaseShiftTiming(invalid in TMRD1)

15.2.3.16 TMRD_ChangeTiming

Change the specified TMRD timing value.

Prototype:
void
TMRD_ChangeTiming(uint8_t ***TimingType***,
Uint32_t ***Timing***)

Parameters:
TimingType specifies the timing parameter type of TMRD, which can be

- **TMRD_TIMING_TD0_CYCLE:** TMRD0 cycle timing, CPRG00.
- **TMRD_TIMING_A0_LEADING:** Signal a0 leading timing (CPRG01).
- **TMRD_TIMING_A0_TRAILING:** Signal a0 trailing timing (CPRG02).
- **TMRD_TIMING_A1_LEADING:** Signal a1 leading timing (CPRG03).
- **TMRD_TIMING_A1_TRAILING:** Signal a1 trailing timing (CPRG04).
- **TMRD_TIMING_PHASE_SHIFT:** Phase shift timing (CPRG05).
- **TMRD_TIMING_TD1_CYCLE:** TMRD1 cycle timing (CPRG10).
- **TMRD_TIMING_B0_LEADING:** Signal b0 leading timing (CPRG11).
- **TMRD_TIMING_B0_TRAILING:** Signal b0 trailing timing (CPRG12).
- **TMRD_TIMING_B1_LEADING:** Signal b1 leading timing (CPRG13).
- **TMRD_TIMING_B1_TRAILING:** Signal b1 trailing timing (CPRG14).

Timing is the data range is 0x02 to 0x10000.

Description:
This function will change the specified TMRD timing value. It is very useful for users to set a small quantity or one of timing value.
Return:
None

Note:
About the value of ***Timing***, please refer to Data Structure Description to get the setting range of each parameter.

15.2.3.17 TMRD_GetTiming

Get the specified TMRD timing value.

Prototype:
uint16_t
TMRD_GetTiming(uint8_t ***TimingType***)

Parameters:
TimingType specifies the timing parameter type of TMRD, which can be

- **TMRD_TIMING_TD0_CYCLE:** TMRD0 cycle timing, CPRG00.
- **TMRD_TIMING_A0_LEADING:** Signal a0 leading timing (CPRG01).
- **TMRD_TIMING_A0_TRAILING:** Signal a0 trailing timing (CPRG02).
- **TMRD_TIMING_A1_LEADING:** Signal a1 leading timing (CPRG03).
- **TMRD_TIMING_A1_TRAILING:** Signal a1 trailing timing (CPRG04).
- **TMRD_TIMING_PHASE_SHIFT:** Phase shift timing (CPRG05).
- **TMRD_TIMING_TD1_CYCLE:** TMRD1 cycle timing (CPRG10).
- **TMRD_TIMING_B0_LEADING:** Signal b0 leading timing (CPRG11).
- **TMRD_TIMING_B0_TRAILING:** Signal b0 trailing timing (CPRG12).
- **TMRD_TIMING_B1_LEADING:** Signal b1 leading timing (CPRG13).
- **TMRD_TIMING_B1_TRAILING:** Signal b1 trailing timing (CPRG14).

Description:
This function will get the timing value from the compare register specified by ***TimingType***.

Return:
Specified timing value that is in compare register.

15.2.4 Data Structure Description

15.2.4.1 TMRD_TimingTypeDef

Data Fields:

Uint32_t
Cycle specifies the TMRD0/1 cycle value, it will set to CPRG00/10.

uint16_t
LeadingTiming0 specifies the signal a0/b0 leading timing (CPRG01/11).

uint16_t
TrailingTiming0 specifies the signal a0/b0 trailing timing (CPRG02/12).

uint16_t
LeadingTiming1 specifies the signal a1/b1 leading timing (CPRG03/13).

uint16_t
TrailingTiming1 specifies the signal a1/b1 trailing timing (CPRG04/14).

uint16_t
PhaseShiftTiming specifies the phase shift timing (CPRG05, only in TMRD0).

Note:

Please refer to the tables below to get the setting range of each parameter in different mode.

Timer Unit	Compare register	16-bit interval timer	
		<TDCLE> = "0"	<TDCLE> = "1"
TMRD0	TD0CP0	0x0000 ≤ CPRG0[15:0] ≤ 0xFFFF	0x0001 ≤ CPRG0[15:0] ≤ 0xFFFF
	TD0CP1	0x0000 ≤ CPRG1[15:0] ≤ 0xFFFF	0x0000 ≤ CPRG1[15:0] ≤ CPRG0[15:0]
	TD0CP2	0x0000 ≤ CPRG2[15:0] ≤ 0xFFFF	0x0000 ≤ CPRG2[15:0] ≤ CPRG0[15:0]
	TD0CP3	0x0000 ≤ CPRG3[15:0] ≤ 0xFFFF	0x0000 ≤ CPRG3[15:0] ≤ CPRG0[15:0]
	TD0CP4	0x0000 ≤ CPRG4[15:0] ≤ 0xFFFF	0x0000 ≤ CPRG4[15:0] ≤ CPRG0[15:0]
	TD0CP5	0x0000 ≤ CPRG5[15:0] ≤ 0xFFFF	0x0000 ≤ CPRG5[15:0] ≤ CPRG0[15:0]
TMRD1	TD1CP0	0x0000 ≤ CPRG0[15:0] ≤ 0xFFFF	0x0001 ≤ CPRG0[15:0] ≤ 0xFFFF
	TD1CP1	0x0000 ≤ CPRG1[15:0] ≤ 0xFFFF	0x0000 ≤ CPRG1[15:0] ≤ CPRG0[15:0]
	TD1CP2	0x0000 ≤ CPRG2[15:0] ≤ 0xFFFF	0x0000 ≤ CPRG2[15:0] ≤ CPRG0[15:0]
	TD1CP3	0x0000 ≤ CPRG3[15:0] ≤ 0xFFFF	0x0000 ≤ CPRG3[15:0] ≤ CPRG0[15:0]
	TD1CP4	0x0000 ≤ CPRG4[15:0] ≤ 0xFFFF	0x0000 ≤ CPRG4[15:0] ≤ CPRG0[15:0]

Setting range of the compare registers in 16-bit interval timer mode

Timer Unit	Compare register	16-bit programmable pulse generation	
		PPG	Interlock PPG
TMRD0	TD0CP0	0x0001 ≤ CPRG0[15:0] ≤ 0xFFFF	0x0001 ≤ CPRG0[15:0] ≤ 0xFFFF
	TD0CP1	0x0000 ≤ CPRG1[15:0] < CPRG2[15:0]	0x0000 ≤ CPRG1[15:0] < CPRG2[15:0]
	TD0CP2	CPRG1[15:0] < CPRG2[15:0] ≤ CPRG0[15:0]	CPRG1[15:0] < CPRG2[15:0] ≤ CPRG0[15:0]
	TD0CP3	0x0000 ≤ CPRG3[15:0] < CPRG4[15:0]	0x0000 ≤ CPRG3[15:0] < CPRG4[15:0]
	TD0CP4	CPRG3[15:0] < CPRG4[15:0] ≤ CPRG0[15:0]	CPRG3[15:0] < CPRG4[15:0] ≤ CPRG0[15:0]
	TD0CP5	don't care	0x0000 ≤ CPRG5[15:0] < (CPRG0[15:0] ÷ 2)
TMRD1	TD1CP0	0x0001 ≤ CPRG0[15:0] ≤ 0xFFFF	don't care
	TD1CP1	0x0000 ≤ CPRG1[15:0] < CPRG2[15:0]	0x0000 ≤ CPRG1[15:0] < CPRG2[15:0]
	TD1CP2	CPRG1[15:0] < CPRG2[15:0] ≤ CPRG0[15:0]	CPRG1[15:0] < CPRG2[15:0] ≤ TD0CP0<CPRG0[15:0]>
	TD1CP3	0x0000 ≤ CPRG3[15:0] < CPRG4[15:0]	0x0000 ≤ CPRG3[15:0] < CPRG4[15:0]
	TD1CP4	CPRG3[15:0] < CPRG4[15:0] ≤ CPRG0[15:0]	CPRG3[15:0] < CPRG4[15:0] ≤ TD0CP0<CPRG0[15:0]>

Setting range of the compare register when 16-bit programmable pulse is output

16. SIO/UART

16.1 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/tmpm341_uart.c, with /Libraries/TX03_Periph_Driver/inc/tmpm341_uart.h containing the macros, data types, structures and API definitions for use by applications.

16.2 API Functions

16.2.1 Function List

- void UART_Enable(TSB_SC_TypeDef* ***UARTx***)
- void UART_Disable(TSB_SC_TypeDef* ***UARTx***)
- WorkState UART_GetBufState(TSB_SC_TypeDef* ***UARTx***, uint8_t ***Direction***)
- void UART_SWReset(TSB_SC_TypeDef* ***UARTx***)
- void UART_Init(TSB_SC_TypeDef* ***UARTx***, UART_InitTypeDef* ***InitStruct***)
- uint32_t UART_GetRxData(TSB_SC_TypeDef* ***UARTx***)
- void UART_SetTxData(TSB_SC_TypeDef* ***UARTx***, uint32_t ***Data***)
- void UART_DefaultConfig(TSB_SC_TypeDef* ***UARTx***)
- UART_Err UART_GetErrState(TSB_SC_TypeDef* ***UARTx***)
- void UART_SetWakeUpFunc(TSB_SC_TypeDef* ***UARTx***,
  FunctionalState ***NewState***)
- void UART_SetIdleMode(TSB_SC_TypeDef* ***UARTx***, FunctionalState ***NewState***)
- void UART_SetRxDMAReq (TSB_SC_TypeDef* ***UARTx***, FunctionalState ***NewState***)
- void UART_SetTxDMAReq (TSB_SC_TypeDef* ***UARTx***, FunctionalState ***NewState***)
- void UART_FIFOConfig(TSB_SC_TypeDef * ***UARTx***, FunctionalState ***NewState***)
- void UART_SetFIFOTransferMode(TSB_SC_TypeDef * ***UARTx***, uint32_t ***TransferMode***)
- void UART_TRxAutoDisable(TSB_SC_TypeDef * ***UARTx***, UART_TRxDisable ***TrxAutoDisable***
- void UART_RxFIFOINTCtrl(TSB_SC_TypeDef * ***UARTx***, FunctionalState ***NewState***)
- void UART_TxFIFOINTCtrl(TSB_SC_TypeDef * ***UARTx***, FunctionalState ***NewState***)
- void UART_RxFIFOByteSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***BytesUsed***)
- void UART_RxFIFOFillLevel(TSB_SC_TypeDef * ***UARTx***,uint32_t ***RxFIFOLevel***)
- void UART_RxFIFOINTSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***RxINTCondition***)
- void UART_RxFIFOClear(TSB_SC_TypeDef * ***UARTx***)
- void UART_TxFIFOFillLevel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***TxFIFOLevel***)
- void UART_TxFIFOINTSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***TxINTCondition***)
- void UART_TxFIFOClear(TSB_SC_TypeDef * ***UARTx***)
- uint32_t UART_GetRxFIFOFillLevelStatus(TSB_SC_TypeDef * ***UARTx***)
- uint32_t UART_GetRxFIFOOverRunStatus(TSB_SC_TypeDef * ***UARTx***)

- uint32_t UART_GetTxFIFOFillLevelStatus(TSB_SC_TypeDef * ***UARTx***)
- uint32_t UART_GetTxFIFOUnderRunStatus(TSB_SC_TypeDef * ***UARTx***)
- void SIO_Enable(TSB_SC_TypeDef * ***SIOx***)
- void SIO_Disable(TSB_SC_TypeDef * ***SIOx***)
- uint8_t SIO_GetRxData(TSB_SC_TypeDef * ***SIOx***)
- void SIO_SetTxData(TSB_SC_TypeDef * ***SIOx***, uint8_t ***Data***)
- void SIO_Init(TSB_SC_TypeDef * ***SIOx***, uint32_t ***IOClkSel***, SIO_InitTypeDef * ***InitStruct***)

16.2.2 Detailed Description

Functions listed above can be divided into four parts:

1. Initialize and configure the common functions of each UART channel are handled by UART_Enable(), UART_Disable(),UART_,UART_Init() and UART_DefaultConfig(),SIO_Enable(), SIO_Disable(),SIO_,SIO_Init().
2. Transfer control and error check of each UART channel are handled by UART_GetBufState(), UART_GetRxData(), UART_SetTxData() and UART_GetErrState(),SIO_GetRxData(), SIO_SetTxData().
3. UART_SetRxDMAReq (),UART_SetTxDMAReq(), UART_SWReset(), UART_SetWakeUpFunc() and UART_SetIdleMode() handle other specified functions.
4. FIFO operation functions are UART_FIFOConfig(),UART_SetFIFOTransferMode(), UART_TrxAutoDisable(),UART_RxFIFOINTCtrl(),UART_TxFIFOINTCtrl(), UART_RxFIFOByteSel(),UART_RxFIFOFillLevel(),UART_RxFIFOINTSel(). UART_RxFIFOClear(),UART_TxFIFOFillLevel(),UART_TxFIFOINTSel(). UART_TxFIFOClear(),UART_GetRxFIFOFillLevelStatus(), UART_GetRxFIFOOverRunStatus(),UART_GetTxFIFOFillLevelStatus()and UART_GetTxFIFOUnderRunStatus().

16.2.3 Function Documentation

Note: The parameter “TSB_SC_TypeDef* ***UARTx***” of APIs UART_SetRxDMAReq and UART_SetTxDMAReq can be one of the following values:
UART0, **UART2**, **UART4**.
the other 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.

16.2.3.1 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

16.2.3.2 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

16.2.3.3 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 on going.

16.2.3.4 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

16.2.3.5 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

16.2.3.6 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

16.2.3.7 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

16.2.3.8 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

16.2.3.9 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.

16.2.3.10 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

16.2.3.11 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

16.2.3.12 UART_SetRxDMAReq

Enable or disable the specified UART channel DMA Request By receive interrupt INTRX.

Prototype:
void
UART_SetRxDMAReq (TSB_SC_TypeDef* ***UARTx***,
FunctionalState ***NewState***)

Parameters:
UARTx is the specified UART channel.
NewState is the new state of the UART channel DMA Request.

This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:
This function will enable the Rx DMA Reuqest of the specified UART channel selected by ***UARTx*** DMA Request when ***NewState*** is **ENABLE**, and disable the DMA request when ***NewState*** is **DISABLE**.

Return:
None

16.2.3.13 UART_SetTxDMAReq

Enable or disable the specified UART channel DMA Request By receive interrupt INTTX.

Prototype:
void
UART_SetTxDMAReq (TSB_SC_TypeDef* ***UARTx***,
FunctionalState ***NewState***)

Parameters:
UARTx is the specified UART channel.
NewState is the new state of the UART channel DMA Request.

This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:
This function will enable the Tx DMA Request of the specified UART channel selected by ***UARTx*** when ***NewState*** is **ENABLE**, and disable the DMAC request when ***NewState*** is **DISABLE**.

Return:
None

16.2.3.14 UART_FIFOConfig

Enable or disable the FIFO of specified UART channel.

Prototype:
void
UART_FIFOConfig(TSB_SC_TypeDef * ***UARTx,***
FunctionalState ***NewState***)

Parameters:
UARTx is the specified UART channel.
NewState is the new state of the UART FIFO.

This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:
This function will enable the FIFO of specified UART channel selected by ***UARTx*** when ***NewState*** is **ENABLE**, and disable the channel when ***NewState*** is **DISABLE**.

Return:
None

16.2.3.15 UART_SetFIFOTransferMode

Transfer mode setting.

Prototype:
void
UART_SetFIFOTransferMode(TSB_SC_TypeDef * ***UARTx,***
uint32_t ***TransferMode***)

Parameters:
UARTx is the Selected the UART channel.
TransferMode is the Transfer mode.

This parameter can be one of the following values:
UART_TRANSFER_PROHIBIT, UART_TRANSFER_HALFDPX_RX, UART_TRANSFER_HALFDPX_TX, UART_TRANSFER_FULLDPX.

Description:
This function will set the transfer mode of specified UART channel selected by ***UARTx***. The UART transfer mode has only 4 modes which above displays.

Return:
None

16.2.3.16 UART_TRxAutoDisable

Controls automatic disabling of transmission and reception.

Prototype:
void
UART_TRxAutoDisable (TSB_SC_TypeDef * ***UARTx,***
UART_TRxDisable ***TRxAutoDisable***)

Parameters:
UARTx is the specified UART channel.
TRxAutoDisable is the Disabling transmission and reception or not.

This parameter can be one of the following values:
UART_RXTXCNT_NONE or **UART_RXTXCNT_AUTODISABLE**

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

Return:
None

16.2.3.17 UART_RxFIFOINTCtrl

Enable or disable receive interrupt for receive FIFO.

Prototype:
void
UART_RxFIFOINTCtrl (TSB_SC_TypeDef * ***UARTx,***
FunctionalState ***NewState***)

Parameters:
UARTx is the specified UART channel.
NewState is the new state of receive interrupt for receive FIFO.

This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:
This function will enable receive interrupt for receive FIFO of specified UART channel selected by ***UARTx*** when ***NewState*** is **ENABLE**, and disable the channel when ***NewState*** is **DISABLE.**

Return:
None

16.2.3.18 UART_TxFIFOINTCtrl

Enable or disable transmit interrupt for transmit FIFO.

Prototype:
void
UART_TxFIFOINTCtrl (TSB_SC_TypeDef * ***UARTx,***
FunctionalState ***NewState***)

Parameters:

UARTx is the specified UART channel.
NewState is the new state of transmit interrupt for receive FIFO.

This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:
This function will enable transmit interrupt for receive FIFO of specified UART channel selected by ***UARTx*** when ***NewState*** is **ENABLE**, and disable the channel when ***NewState*** is **DISABLE.**

Return:
None

16.2.3.19 UART_RxFIFOByteSel

Bytes used in receive FIFO.

Prototype:
void
UART_RxFIFOByteSel (TSB_SC_TypeDef * ***UARTx,***
uint32_t ***BytesUsed***)

Parameters:
UARTx is the specified UART channel.
BytesUsed is the Bytes used in receive FIFO.

This parameter can be one of the following values:
UART_RXFIFO_MAX or **UART_RXFIFO_RXFLEVEL**

Description:
This function will set numbers of bytes used in receive FIFO of specified UART channel selected by ***UARTx***, But the bytes of the number should be **UART_RXFIFO_MAX** or **UART_RXFIFO_RXFLEVEL.**

Return:
None

16.2.3.20 UART_RxFIFOFillLevel

Receive FIFO fill level to generate receive interrupts.

Prototype:
void
UART_RxFIFOFillLevel (TSB_SC_TypeDef * ***UARTx,***
uint32_t ***RxFIFOLevel***)

Parameters:
UARTx is the specified UART channel.
RxFIFOLevel is the Receive FIFO fill level.

This parameter can be one of the following values:
UART_RXFIFO4B_FLEVLE_4_2B, UART_RXFIFO4B_FLEVLE_1_1B, UART_RXFIFO4B_FLEVLE_2_2B, UART_RXFIFO4B_FLEVLE_3_1B.

Description:

This function will set Receive FIFO fill level for generate receive interrupts of specified UART channel selected by ***UARTx***, But the level should be **UART_RXFIFO4B_FLEVLE_4_2B, UART_RXFIFO4B_FLEVLE_1_1B, UART_RXFIFO4B_FLEVLE_2_2B, UART_RXFIFO4B_FLEVLE_3_1B.**

Return:
None

16.2.3.21 UART_RxFIFOINTSel

Select RX interrupt generation condition.

Prototype:
void
UART_RxFIFOINTSel (TSB_SC_TypeDef * ***UARTx,***
uint32_t ***RxINTCondition***)

Parameters:
UARTx is the specified UART channel.
RxINTCondition is the RX interrupt generation condition.

This parameter can be one of the following values:
UART_RFIS_REACH_FLEVEL or **UART_RFIS_REACH_EXCEED_FLEVEL**

Description:
This function will set RX interrupt generation condition of specified UART channel selected by ***UARTx***, But the level should be **UART_RFIS_REACH_FLEVEL, UART_RFIS_REACH_EXCEED_FLEVEL.**

Return:
None

16.2.3.22 UART_RxFIFOClear

Clear Receive FIFO.

Prototype:
void
UART_RxFIFOClear (TSB_SC_TypeDef * ***UARTx***)

Parameters:
UARTx is the specified UART channel.

Description:
This function will Clear Receive FIFO of specified UART channel selected by ***UARTx***.

Return:
None

16.2.3.23 UART_TxFIFOFillLevel

Transmit FIFO fill level to generate receive interrupts.

Prototype:

void
UART_TxFIFOFillLevel (TSB_SC_TypeDef * ***UARTx,***
uint32_t ***TxFIFOLevel***)

Parameters:
UARTx is the specified UART channel.
TxFIFOLevel is the Receive FIFO fill level.

This parameter can be one of the following values:
UART_TXFIFO4B_FLEVLE_0_0B, UART_TXFIFO4B_FLEVLE_1_1B, UART_TXFIFO4B_FLEVLE_2_0B, UART_TXFIFO4B_FLEVLE_3_1B

Description:
This function will set Transmit FIFO fill level for generate receive interrupts of specified UART channel selected by ***UARTx***, But the level should be **UART_TXFIFO4B_FLEVLE_0_0B, UART_TXFIFO4B_FLEVLE_1_1B, UART_TXFIFO4B_FLEVLE_2_0B, UART_TXFIFO4B_FLEVLE_3_1B.**

Return:
None

16.2.3.24 UART_TxFIFOINTSel

Select TX interrupt generation condition.

Prototype:
void
UART_TxFIFOINTSel (TSB_SC_TypeDef * ***UARTx,***
uint32_t ***TxINTCondition***)

Parameters:
UARTx is the specified UART channel.
TxINTCondition is the TX interrupt generation condition.

This parameter can be one of the following values:
UART_TFIS_REACH_FLEVEL or **UART_TFIS_REACH_EXCEED_FLEVEL**

Description:
This function will set TX interrupt generation condition of specified UART channel selected by ***UARTx***, But the level should be **UART_TFIS_REACH_FLEVEL, UART_TFIS_REACH_EXCEED_FLEVEL.**

Return:
None

16.2.3.25 UART_TxFIFOClear

Clear Transmit FIFO.

Prototype:
void
UART_TxFIFOClear (TSB_SC_TypeDef * ***UARTx***)

Parameters:
UARTx is the specified UART channel.

Description:
This function will Clear Transmit FIFO of specified UART channel selected by ***UARTx***.

Return:
None

16.2.3.26 UART_GetRxFIFOFillLevelStatus

Indicate the status of receive FIFO fill level.

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

Parameters:
UARTx is the specified UART channel.

Description:
This function will Indicate the receive FIFO fill level status, which UART channel selected by ***UARTx*.**

Return:
UART_TRXFIFO_EMPTY: TX FIFO fill level is empty.
UART_TRXFIFO_1B: TX FIFO fill level is 1 byte.
UART_TRXFIFO_2B: TX FIFO fill level is 2 bytes.
UART_TRXFIFO_3B: TX FIFO fill level is 3 bytes.
UART_TRXFIFO_3B: TX FIFO fill level is 4 bytes.

16.2.3.27 UART_GetRxFIFOOverRunStatus

Indicate the status of Receive FIFO overrun.

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

Parameters:
UARTx is the specified UART channel.

Description:
This function will Indicate the Receive FIFO overrun status, which UART channel selected by ***UARTx*.**

Return:
UART_RXFIFO_OVERRUN: Flags for RX FIFO overrun.

16.2.3.28 UART_GetTxFIFOFillLevelStatus

Status of transmit FIFO fill level.

Prototype:

uint32_t
UART_GetTxFIFOFillLevelStatus (TSB_SC_TypeDef * ***UARTx***)

Parameters:
UARTx is the specified UART channel.

Description:
Status of transmit FIFO fill level.

Return:
UART_TRXFIFO_EMPTY: TX FIFO fill level is empty.
UART_TRXFIFO_1B: TX FIFO fill level is 1 byte.
UART_TRXFIFO_2B: TX FIFO fill level is 2 bytes.
UART_TRXFIFO_3B: TX FIFO fill level is 3 bytes.
UART_TRXFIFO_3B: TX FIFO fill level is 4 bytes.

16.2.3.29 UART_GetTxFIFOUnderRunStatus

Transmit FIFO under run.

Prototype:
uint32_t
UART_ GetTxFIFOUnderRunStatus (TSB_SC_TypeDef * ***UARTx***)

Parameters:
UARTx is the specified UART channel.

Description:
Transmit FIFO under run.

Return:
UART_TXFIFO_UNDERRUN: Flags for TX FIFO under-run.

16.2.3.30 SIO_Enable

Enable the specified SIO channel.

Prototype:
void
SIO_Enable (TSB_SC_TypeDef* ***SIOx***)

Parameters:
SIOx is the specified SIOx channel.

Description:
This function will enable the specified SIO channel selected by ***SIOx***.

Return:
None

16.2.3.31 SIO_Disable

Disable the specified SIO channel.

Prototype:
void
SIO _Disable(TSB_SC_TypeDef* ***SIOx***)

Parameters:
SIOTx is the specified SIO channel.

Description:
This function will disable the specified SIO channel selected by ***SIOx***.

Return:
None

16.2.3.32 SIO_GetRxData

Get data received from the specified SIO channel.

Prototype:
uint32_t
SIO_GetRxData(TSB_SC_TypeDef* ***SIOx***)

Parameters:
SIOx is the specified SIO channel.

Description:
This function will get the data received from the specified SIO channel selected by ***SIOx***.

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

16.2.3.33 SIO_SetTxData

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

Prototype:
void
SIO_SetTxData(TSB_SC_TypeDef* ***SIOx***,
uint8_t ***Data***)

Parameters:
SIOx is the specified SIO channel.
Data is a frame to be sent,

Description:
This function will set the data to be sent from the specified SIO channel selected by ***SIOx***.

Return:
None

16.2.3.34 SIO_Init

Initialize and configure the specified SIO channel.

Prototype:
void
SIO_Init(TSB_SC_TypeDef* ***SIOx***,
uint32_t ***IOClkSel,***
SIO_InitTypeDef* ***InitStruct***)

Parameters:
SIOx is the specified SIO channel.
IOClkSel is the work clock
InitStruct is the structure containing basic SIO configuration including baud rate, transmission direction, transfer mode.

Description:
This function will initialize and configure the baud rate, transmission direction, transfer mode for the specified SIO channel selected by ***SIOx***.

Return:
None

16.2.4 Data Structure Description

16.2.4.1 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.

16.2.4.2 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**

17. WDT

17.1 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\tmpm341_wdt.c, with \Libraries/TX03_Periph_Driver\inc\tmpm341_wdt.h containing the API definitions for use by applications.

17.2 API Functions

17.2.1 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)

17.2.2 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().

17.2.3 Function Documentation

17.2.3.1 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/fsys
- **WDT_DETECT_TIME_EXP_17:** ***DetectTime*** is 2^17/fsys
- **WDT_DETECT_TIME_EXP_19:** ***DetectTime*** is 2^19/fsys

- **WDT_DETECT_TIME_EXP_21: *DetectTime*** is 2^21/fsys
- **WDT_DETECT_TIME_EXP_23: *DetectTime*** is 2^23/fsys
- **WDT_DETECT_TIME_EXP_25: *DetectTime*** is 2^25/fsys

Description:
This function will set detection time for WDT.

Return:
None

17.2.3.2 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

17.2.3.3 WDT_SetOverflowOutput

Set WDT to generate NMI interrupt or to reset when the counter overflows.

Prototype:
void
WDT_SetOverflowOutput(uint32_t ***OverflowOutput***)

Parameters:
OverflowOutput: Select function of WDT when counter overflow.
This parameter can be one of the following values:
- **WDT_NMIINT**: Set WDT to generate NMI interrupt when counter overflow.
- **WDT_WDOUT**: Set WDT to generate reset when counter overflow.

Description:
This function will set WDT to generate NMI interrupt if the counter overflows when ***OverflowOutput*** is **WDT_NMIINT,** and set WDT to generate reset if the counter overflows when ***OverflowOutput*** is **WDT_WDOUT**.

Return:
None

17.2.3.4 WDT_Init

Initialize and configure WDT.

Prototype:
void
WDT_Init (WDT_InitTypeDef* ***InitStruct***)

Parameters:
InitStruct: The structure containing basic WDT configuration including detect time and WDT output when counter overflow. (Refer to "13.3.4 Data structure Description" for details)

Description:
This function will initialize and configure the WDT detection time and the output of WDT when the counter overflows. **WDT_SetDetectTime()** and **WDT_SetOverflowOutput()** will be called by it.

Return:
None

17.2.3.5 WDT_Enable

Enable the WDT function.

Prototype:
void
WDT_Enable(void)

Parameters:
None

Description:
This function will enable WDT.

Return:
None

17.2.3.6 WDT_Disable

Disable the WDT function.

Prototype:
void
WDT_Disable(void)

Parameters:
None

Description:

This function will disable WDT.

Return:
None

17.2.3.7 WDT_WriteClearCode

Write the clear code.

Prototype:
void
WDT_WriteClearCode (void)

Parameters:
None

Description:
This function will clear the WDT counter.

Return:
None

17.2.4 Data Structure Description

17.2.4.1 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/fsys
- **WDT_DETECT_TIME_EXP_17:** ***DetectTime*** is 2^17/fsys
- **WDT_DETECT_TIME_EXP_19:** ***DetectTime*** is 2^19/fsys
- **WDT_DETECT_TIME_EXP_21:** ***DetectTime*** is 2^21/fsys
- **WDT_DETECT_TIME_EXP_23:** ***DetectTime*** is 2^23/fsys
- **WDT_DETECT_TIME_EXP_25:** ***DetectTime*** is 2^25/fsys

uint32_t
OverflowOutput Select the action when the WDT counter overflows, which can be set as:

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