TOSHIBA

TOSHIBA TX03 Peripheral Driver User Guide (TMPM381/383)

Ver 0.102
Sep, 2017

TOSHIBA ELECTRONIC DEVICES & STORAGE CORPORATION

CMDR-M381UG-00xE

TOSHIBA

TOSHIBA

Index

1. Introduction

TOSHIBA TX03 Peripheral Driver is a set of drivers for all peripherals found on the TOSHIBA TX03 series microcontrollers. TMPM38x Peripheral Driver is an important part of TOSHIBA TX03 Peripheral Driver, which are designed for TMPM38x series MCUs.

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

The design goals of TOSHIBA TMPM38x Peripheral Driver:
- Completely written in C except the start-up routine and where not possible
- Cover all the peripherals on MCU

Note: TMPM38x stands for TMPM38x/M383

2. Organization of TOSHIBA TX03 Peripheral Driver

/Libraries
This folder contains all CMSIS files and TMPM38x Peripheral Drivers.

/Libraries/ TX03_CMSIS
This folder contains the TMPM38x CMSIS files: device peripheral access layer and core peripheral access layer.

/Libraries/TX03_Periph_Driver
This folder contains all the source code of the drivers, the core of TOSHIBA TMPM38x Peripheral Driver.

/Libraries/TX03_Periph_Driver/inc
This folder contains all the header files of TMPM38x Peripheral Drivers for each peripheral.

/Libraries/TX03_Periph_Driver/src
This folder contains all the source files of TMPM38x Peripheral Drivers for each peripheral.

/Project
This folder contains template project and examples for using TMPM38x Peripheral Driver.

/Project/Template
This folder contains template project of TOSHIBA TMPM38x Peripheral Driver.

/Project/Examples
This folder contains a set of examples for using TMPM38x Peripheral Driver

/Utilities/TMPM38x-EVAL
This folder contains the configuration and driver files for hardware resources (e.g. led, key) on TMPM38x boards.

3. ADC

3.1 Overview

TOSHIBA TMPM38x contains a 12bits/10bits (selectable) successive-approximation Analog-to-Digital Converter (ADC).

The ADC of TMPM383 has 10 external analog inputs (AIN0 to AIN9) and the ADC of TMPM381 has 18 external analog inputs (AIN0 to AIN17) which can also be used as input/output ports.

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

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm38x_adc.c, with /Libraries/TX03_Periph_Driver/inc/tmpm38x_adc.h containing the macros, data types, structures and API definitions for use by applications.

3.2 API Functions

3.2.1 Function List

- void ADC_SetClk(TSB_AD_TypeDef * ***ADx***, uint32_t ***Sample_HoldTime***,
  uint32_t ***Prescaler_Output***)
- void ADC_SetResolution(TSB_AD_TypeDef * ***ADx***, ADC_Resolution ***ADBits***)
- ADC_Resolution ADC_GetResolution(TSB_AD_TypeDef * ***ADx***)
- void ADC_Enable(TSB_AD_TypeDef * ***ADx***)
- void ADC_Disable(TSB_AD_TypeDef * ***ADx***)
- void ADC_Start(TSB_AD_TypeDef * ***ADx***, ADC_TrgType ***Trg***)
- void ADC_StopConstantTrg(TSB_AD_TypeDef * ***ADx***)
- WorkState ADC_GetConvertState(TSB_AD_TypeDef * ***ADx***, ADC_TrgType ***Trg***)
- void ADC_SetLowPowerMode(TSB_AD_TypeDef * ***ADx***, FunctionalState ***NewState***)
- void ADC_SetMonitor(TSB_AD_TypeDef * ***ADx***, ADC_MonitorTypeDef * ***Monitor***)
- void ADC_DisableMonitor(TSB_AD_TypeDef * ***ADx***, ADC_CMPCRx ***CMPCRx***)
- ADC_Result ADC_GetConvertResult(TSB_AD_TypeDef * ***ADx***,
  ADC_REGx ***ResultREGx***)
- void ADC_SetTimerTrg(TSB_AD_TypeDef * ***ADx***,
  ADC_REGx ***ResultREGx***, uint8_t ***MacroAINx***)
- void ADC_SetSWTrg(TSB_AD_TypeDef * ***ADx***,
  ADC_REGx ***ResultREGx***, uint8_t ***MacroAINx***)
- void ADC_SetConstantTrg(TSB_AD_TypeDef * ***ADx***,
  ADC_REGx ***ResultREGx***, uint8_t ***MacroAINx***)

3.2.2 Detailed Description

Functions listed above can be divided into four parts:
1) ADC setting by ADC_SetClk(), ADC_SetResolution(), ADC_SetMonitor(), ADC_DisableMonitor(), ADC_SetTimerTrg(), ADC_SetSWTrg(), ADC_SetConstantTrg().

2) ADC function enable/disable & start/stop by ADC_Enable(), ADC_Disable(), ADC_Start(), ADC_StopConstantTrg().
3) ADC state or data read functions by ADC_GetResolution(), ADC_GetConvertState(), ADC_GetConvertResult().
4) ADC_SetLowPowerMode() handle other specified functions.

3.2.3 Function Documentation

3.2.3.1 ADC_SetClk

Set ADC prescaler output(SCLK) of the specified ADC unit.

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

Parameters:
ADx: Select ADC Unit, which can be set as:
TSB_AD

Sample_HoldTime: Select ADC sample hold time, which can be set as:
- **ADC_HOLD_FIX**: write “1001b” to TSH<0:3>.

Prescaler_Output: Select ADC prescaler output, which can be set as:
- **ADC_FC_DIVIDE_LEVEL_NONE:** fc

Description:
This function will set the specified ADC unit's sample hold time by ***Sample_HoldTime*** as **ADC_HOLD_FIX** & select ADC prescaler output by ***Prescaler_Output***.

Return:
None

3.2.3.2 ADC_SetResolution

Set AD resolution.

Prototype:
void
ADC_SetResolution(TSB_AD_TypeDef * ***ADx***,
ADC_Resolution ***ADBits***)

Parameters:
ADx: Select ADC Unit, which can be set as:
TSB_AD

ADBits: Set ADC resolution as 12bits or 10bits, which can be set as:
- **ADC_10BITS:** 10bit
- **ADC_12BITS:** 12bit

Description:
This function will set the specified ADC unit's resolution by ***ADBits*** as **ADC_10BITS/ADC_12BITS**.

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

Return:
None

3.2.3.3 ADC_GetResolution

Get current AD resolution set.

Prototype:
ADC_Resolution
ADC_GetResolution(TSB_AD_TypeDef * ***ADx***)

Parameters:
ADx: Select ADC Unit, which can be set as:
TSB_AD

Description:
This function will get the specified ADC unit's resolution.

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

3.2.3.4 ADC_Enable

Enable the specified ADC unit.

Prototype:
void
ADC_Enable(TSB_AD_TypeDef * ***ADx***)

Parameters:
ADx: Select ADC Unit, which can be set as:
TSB_AD

Description:
This function will enable the specified ADC unit.

Return:
None

3.2.3.5 ADC_Disable

Disable the specified ADC unit.

Prototype:
void
ADC_Disable(TSB_AD_TypeDef * ***ADx***)

Parameters:
ADx: Select ADC Unit, which can be set as:

TSB_AD

Description:
This function will disable the specified ADC unit.

Return:
None

3.2.3.6 ADC_Start

Start the specified ADC unit with software trigger or constant trigger.

Prototype:
void
ADC_Start(TSB_AD_TypeDef * ***ADx***,
ADC_TrgType ***Trg***)

Parameters:
ADx: Select ADC Unit, which can be set as:
TSB_AD

Trg: Set trigger type, which can be set as:
- **ADC_TRG_SW**: Software triggered conversion
- **ADC_TRG_CONSTANT**: Constant AD conversion

Description:
This function will start the specified ADC unit by ***Trg*** as **ADC_TRG_SW**, **ADC_TRG_CONSTANT**.

Return:
None

3.2.3.7 ADC_StopConstantTrg

Stop the specified ADC unit when use constant trigger.

Prototype:
void
ADC_StopConstantTrg(TSB_AD_TypeDef * ***ADx***)

Parameters:
ADx: Select ADC Unit, which can be set as:
TSB_AD

Description:
This function will stop the specified ADC unit when use constant trigger.

Return:
None

3.2.3.8 ADC_GetConvertState

Get the conversion state of the specified ADC unit.

Prototype:

WorkState
ADC_GetConvertState(TSB_AD_TypeDef * ***ADx***,
ADC_TrgType ***Trg***)

Parameters:
ADx: Select ADC Unit, which can be set as:
TSB_AD

Trg**:** Set trigger type, which can be set as:
- **ADC_TRG_SW:** Software triggered conversion
- **ADC_TRG_CONSTANT:** Constant AD conversion
- **ADC_TRG_TIMER:** Timer triggered conversion

Description:
This function will get the state of the specified ADC unit's conversion as **BUSY/DONE**, when AD conversion is triggered set by ***Trg*** as **ADC_TRG_SW**, **ADC_TRG_CONSTANT** and **ADC_TRG_TIMER.**

Return:
WorkState type, the value means:
BUSY: Conversion in progress
DONE: Conversion not in process

3.2.3.9 ADC_SetLowPowerMode

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

Prototype:
void
ADC_SetLowPowerMode(TSB_AD_TypeDef * ***ADx***,
FunctionalState ***NewState***)

Parameters:
ADx: Select ADC Unit, which can be set as:
TSB_AD

NewState: Specify ADC low power mode, which can be set as:
- **DISABLE**: Exit low power mode to enter normal AD conversion
- **ENABLE**: Enter low power mode, AD conversion will not work.

Description:
This function will select ADC low power mode by ***NewState*** as **DISABLE**, or **ENABLE**.
When Power up reset, the ADC module is set in Low Power Mode so user must call **ADC_SetLowPowerMode(DISABLE)** before start AD conversion.

Return:
None

3.2.3.10 ADC_SetMonitor

Set the monitor function of the specified ADC unit and enable it.

Prototype:
void
ADC_SetMonitor(TSB_AD_TypeDef * ***ADx***,

ADC_MonitorTypeDef * ***Monitor***)

Parameters:
ADx: Select ADC Unit, which can be set as:
TSB_AD

Monitor: It is a structure with detail as below:

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

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

Description:
This function will set AD conversion result monitoring function of the specified unit by ADC_MonitorTypeDef * ***Monitor*** and enable it.

Return:
None

3.2.3.11 ADC_DisableMonitor

Disable the monitor function of the specified ADC unit.

Prototype:
void
ADC_DisableMonitor(TSB_AD_TypeDef * ***ADx***,
ADC_CMPCRx ***CMPCRx***)

Parameters:
ADx: Select ADC Unit, which can be set as:
TSB_AD

CMPCRx: Select compare control register, which can be set as:
- **ADC_CMPCR_0:** ADCMPCR0
- **ADC_CMPCR_1:** ADCMPCR1

Description:
This function will disable the monitor function of the specified ADC unit by ***CMPCRx*** as **ADC_CMPCR_0** or **ADC_CMPCR_1**.

Return:
None

3.2.3.12 ADC_GetConvertResult

Get result from the specified AD Conversion Result Register.

Prototype:
ADC_Result
ADC_GetConvertResult(TSB_AD_TypeDef * ***ADx***,
ADC_REGx ***ResultREGx***)

Parameters:
ADx: Select ADC Unit, which can be set as:
TSB_AD

ResultREGx: Set ADC result register, which can be set as:
- **ADC_REG0:** ADREG0
- **ADC_REG1:** ADREG1
- **ADC_REG2:** ADREG2
- **ADC_REG3:** ADREG3
- **ADC_REG4:** ADREG4
- **ADC_REG5:** ADREG5
- **ADC_REG6:** ADREG6
- **ADC_REG7:** ADREG7
- **ADC_REG8:** ADREG8
- **ADC_REG9:** ADREG9
- **ADC_REG10:** ADREG10
- **ADC_REG11:** ADREG11

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

Return:
AD conversion result. Each bit has the following meaning:
Stored(Bit0): AD conversion result store flag
OverRun(Bit1): Overrun flag
ADResult(Bit4 to Bit15): AD conversion result

3.2.3.13 ADC_SetTimerTrg

Set Timer Trigger Program Register of the specified ADC unit.

Prototype:
void
ADC_SetTimerTrg(TSB_AD_TypeDef * ***ADx***,
ADC_REGx ***ResultREGx***,
uint8_t ***MacroAINx***)

Parameters:
ADx: Select ADC Unit, which can be set as:
TSB_AD

ResultREGx: Set AD Conversion Result Register for programming timer triggers, which can be set as:
- **ADC_REG0:** ADREG0
- **ADC_REG1:** ADREG1
- **ADC_REG2:** ADREG2
- **ADC_REG3:** ADREG3
- **ADC_REG4:** ADREG4
- **ADC_REG5:** ADREG5
- **ADC_REG6:** ADREG6
- **ADC_REG7:** ADREG7

- **ADC_REG8:** ADREG8
- **ADC_REG9:** ADREG9
- **ADC_REG10:** ADREG10
- **ADC_REG11:** ADREG11

MacroAINx: Select AD Channel together with its enabled or disabled setting. This parameter must be inputted with macro as the format below:
- **TRG_ENABLE(y)**: Enable AD channel 'y' for ***ResultREGx***
- **TRG_DISABLE(y)**: Disable AD channel 'y' for ***ResultREGx***

'y' above can be one of the following values:
- **TMPM381**: ADC_AIN0 to ADC_AIN17
- **TMPM383**: ADC_AIN0 to ADC_AIN9

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

Return:
None

3.2.3.14 ADC_SetSWTrg

Set Software Trigger Program Register of the specified ADC unit.

Prototype:
void
ADC_SetSWTrg(TSB_AD_TypeDef * ***ADx***,
ADC_REGx ***ResultREGx***,
uint8_t ***MacroAINx***)

Parameters:
ADx: Select ADC Unit, which can be set as:
TSB_AD

ResultREGx: Set AD Conversion Result Register for programming software triggers, which can be set as:
- **ADC_REG0:** ADREG0
- **ADC_REG1:** ADREG1
- **ADC_REG2:** ADREG2
- **ADC_REG3:** ADREG3
- **ADC_REG4:** ADREG4
- **ADC_REG5:** ADREG5
- **ADC_REG6:** ADREG6
- **ADC_REG7:** ADREG7
- **ADC_REG8:** ADREG8
- **ADC_REG9:** ADREG9
- **ADC_REG10:** ADREG10
- **ADC_REG11:** ADREG11

MacroAINx: Select AD Channel together with its enabled or disabled setting. This parameter must be inputted with macro as the format below:
- **TRG_ENABLE(y)**: Enable AD channel 'y' for ***ResultREGx***
- **TRG_DISABLE(y)**: Disable AD channel 'y' for ***ResultREGx***

'y' above can be one of the following values:
- **TMPM381**: ADC_AIN0 to ADC_AIN17
- **TMPM383**: ADC_AIN0 to ADC_AIN9

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

Return:
None

3.2.3.15 ADC_SetConstantTrg

Set Constant Trigger Program Register of the specified ADC unit.

Prototype:
void
ADC_SetConstantTrg(TSB_AD_TypeDef * ***ADx***,
ADC_REGx ***ResultREGx***,
uint8_t ***MacroAINx***)

Parameters:
ADx: Select ADC Unit, which can be set as:
TSB_AD

ResultREGx: Set AD Conversion Result Register for programming constant triggers, which can be set as:
- **ADC_REG0:** ADREG0
- **ADC_REG1:** ADREG1
- **ADC_REG2:** ADREG2
- **ADC_REG3:** ADREG3
- **ADC_REG4:** ADREG4
- **ADC_REG5:** ADREG5
- **ADC_REG6:** ADREG6
- **ADC_REG7:** ADREG7
- **ADC_REG8:** ADREG8
- **ADC_REG9:** ADREG9
- **ADC_REG10:** ADREG10
- **ADC_REG11:** ADREG11

MacroAINx: Select AD Channel together with its enabled or disabled setting.
This parameter must be inputted with macro as the format below:
- **TRG_ENABLE(y)**: Enable AD channel 'y' for ***ResultREGx***
- **TRG_DISABLE(y)**: Disable AD channel 'y' for ***ResultREGx***

'y' above can be one of the following values:
- **TMPM381**: ADC_AIN0 to ADC_AIN17
- **TMPM383**: ADC_AIN0 to ADC_AIN9

Description:
This function will set AD Conversion Result Register of the specified ADC unit by ***ResultREGx*** as **ADC_REG_0**, **ADC_REG_1**, **ADC_REG_2**, **ADC_REG_3**, **ADC_REG_4**, **ADC_REG_5**, **ADC_REG_6**, **ADC_REG_7**, **ADC_REG_8**,

ADC_REG_9, **ADC_REG_10**, **ADC_REG_11**, and enable/disable REG with AIN pin by ***MacroAINx*** in Constant Trigger Program Register.

Return:
None

3.2.4 Data Structure Description

3.2.4.1 ADC_MonitorTypeDef

Data Fields for this structure:

ADC_CMPCRx
CMPCRx Select Compare Control Register, which can be:
- **ADC_CMPCR_0**: ADCMPCR0
- **ADC_CMPCR_1**: ADCMPCR1

ADC_REGx
ResultREGx Select which ADC Result Register to be used, which can be set as:
- **ADC_REG0:** ADREG0
- **ADC_REG1:** ADREG1
- **ADC_REG2:** ADREG2
- **ADC_REG3:** ADREG3
- **ADC_REG4:** ADREG4
- **ADC_REG5:** ADREG5
- **ADC_REG6:** ADREG6
- **ADC_REG7:** ADREG7
- **ADC_REG8:** ADREG8
- **ADC_REG9:** ADREG9
- **ADC_REG10:** ADREG10
- **ADC_REG11:** ADREG11

uint32_t
CmpTimes Define how many times will comparison times be counted, which can be:
1 to **16**

ADC_CmpCondition
Condition Conditon to compare ADREGm with ADCMPn(m = 0 to 11, n = 0 to 1), which can be:
- **ADC_LARGER_THAN_CMP_REG**
- **ADC_SMALLER_THAN_CMP_REG**

uint32_t
CmpValue Comparison value to be set in ADCMP0 or ADCMP1, which can be:
- **0 to 4095 for 12bits mode**
- **0 to 1023 for 10bits mode**

3.2.4.2 ADC_Result

Data Fields for this structure:

uint32_t
All: AD Conversion Result.
Bit
uint32_t
Stored**:** 1 AD result has been stored.

uint32_t ***OverRun*:**	1	Overrun flag.
uint32_t ***Reserved1*:**	2	Reserved.
uint32_t ***ADResult*:**	12	Store AD result.
uint32_t ***Reserved2*:**	16	Reserved.

4. CG

4.1 Overview

The CG API provides a set of functions for using the TMPM38x CG modules as the following:

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

This driver is contained in TX03_Periph_Driver\src\tmpm38x_cg.c, with TX03_Periph_Driver\inc\tmpm38x_cg.h containing the API definitions for use by applications.

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

fEHOSC: Clock generated by external high-speed oscillator.
fIHOSC: Clock input from internal high-speed oscillator.
fs: Clock generated by external low-speed oscillator.
fosc: fIHOSC or fEHOSC specified by CGOSCCR<OSCSEL>.
fPLL: Clock quadrupled by PLL.
fc: Clock specified by CGPLLSEL<PLLSEL> (high-speed clock).
fgear: Clock specified by CGSYSCR <GEAR[2:0]>.
fsys: Clock specified by CGSKSEL<SYSCK> (system clock).
fperiph: Clock specified by CGSYSCR <FPSEL[1:0]>.
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_SetPLL(FunctionalState ***NewState***)
- FunctionalState CG_GetPLLState(void)
- Result CG_SetFosc(CG_FoscSrc ***Source***, FunctionalState ***NewState***)
- void CG_SetFoscSrc(CG_FoscSrc ***Source***)
- CG_FoscSrc CG_GetFoscSrc(void)

- FunctionalState CG_GetFoscState(CG_FoscSrc ***Source***)
- Result CG_SetFs(FunctionalState ***NewState***)
- FunctionalState CG_GetFsState(void)
- void CG_SetPortM(CG_PortMMode ***Mode***)
- void CG_SetSTBYMode(CG_STBYMode ***Mode***)
- CG_STBYMode CG_GetSTBYMode(void)
- void CG_SetExitStopModeFosc(FunctionalState ***NewState***)
- FunctionalState CG_GetExitStopModeFoscState(void)
- void CG_SetExitStopModeFs(FunctionalState ***NewState***)
- FunctionalState CG_GetExitStopModeFsState(void)
- void CG_SetPinStateInStopMode(FunctionalState ***NewState***)
- FunctionalState CG_GetPinStateInStopMode(void)
- Result CG_SetFcSrc(CG_FcSrc ***Source***)
- CG_FcSrc CG_GetFcSrc(void)
- Result CG_SetFsysSrc(CG_FsysSrc ***Source***)
- CG_FsysSrc CG_GetFsysSrc(void)
- void CG_SetSTBYReleaseINTSrc(CG_INTSrc ***INTSource***,
  CG_INTActiveState ***ActiveState***,
  FunctionalState ***NewState***)
- CG_INTActiveState CG_GetSTBYReleaseINTState(CG_INTSrc ***INTSource***)
- void CG_ClearINTReq(CG_INTSrc ***INTSource***)
- CG_NMIFactor CG_GetNMIFlag(void)
- CG_ResetFlag CG_GetResetFlag(void)

4.2.2 Detailed Description

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

1) One group of APIs are in charge of clock selection, such as:
CG_SetFgearLevel(), CG_GetFgearLevel(), CG_SetPhiT0Src(), CG_GetPhiT0Src(), CG_SetPhiT0Level(), CG_GetPhiT0Level(), CG_SetSCOUTSrc(), CG_GetSCOUTSrc(), CG_SetWarmUpTime(), CG_StartWarmUp(), CG_GetWarmUpState(), CG_SetPLL(), CG_GetPLLState(), CG_SetFosc(),CG_SetFoscSrc(), CG_GetFoscSrc(), CG_GetFoscState(), CG_SetFs(), CG_GetFsState(), CG_SetFcSrc(), CG_GetFcSrc(), CG_SetFsysSrc(),CG_GetFsysSrc(),CG_SetPortM().
2) The 2nd group of APIs handle settings of standby modes:
CG_SetSTBYMode(), CG_GetSTBYMode( ), CG_SetExitStopModeFosc(), CG_GetExitStopModeFoscState(), CG_SetExitStopModeFs(), CG_GetExitStopModeFsState(), CG_SetPinStateInStopMode(), CG_GetPinStateInStopMode().
3) The other APIs handle settings of interrupts:
CG_SetSTBYReleaseINTSrc(), CG_GetSTBYReleaseINTState(), CG_ClearINTReq(), CG_GetNMIFlag(), CG_GetResetFlag().

4.2.3 Function Documentation

4.2.3.1 CG_SetFgearLevel

Set the dividing level between clock fgear and fc.

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

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

- **CG_DIVIDE_1**: fgear = fc
- **CG_DIVIDE_2**: fgear = fc/2
- **CG_DIVIDE_4**: fgear = fc/4
- **CG_DIVIDE_8**: fgear = fc/8
- **CG_DIVIDE_16**: fgear = fc/16

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

Return:
None

4.2.3.2 CG_GetFgearLevel

Get the dividing level between fgear and fc.

Prototype:
CG_DivideLevel
CG_GetFgearLevel(void)

Parameters:
None

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

Return:
The dividing level between clock fgear and fc.
The value returned can be one of the following values:
CG_DIVIDE_1: fgear = fc
CG_DIVIDE_2: fgear = fc/2
CG_DIVIDE_4: fgear = fc/4
CG_DIVIDE_8: fgear = fc/8
CG_DIVIDE_16: fgear = fc/16
CG_DIVIDE_UNKNOWN: invalid data is read

4.2.3.3 CG_SetPhiT0Src

Select the PhiT0(ΦT0) source between fgear, fc or fsys.

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

Parameters:
PhiT0Src: Select PhiT0 source.
This parameter can be one of the following values:
- **CG_PHIT0_SRC_FGEAR** means PhiT0 source is fgear.
- **CG_PHIT0_SRC_FC** means PhiT0 source is fc.
- **CG_PHIT0_SRC_FSYS** means PhiT0 source is fsys.

Description:
This function will select the PhiT0(ΦT0) source.

Return:
None

4.2.3.4 CG_GetPhiT0Src

Get the PhiT0 (ΦT0) source.

Prototype:
CG_PhiT0Src
CG_GetPhiT0Src(void)

Parameters:
None

Description:
This function will get the PhiT0(ΦT0) source.

Return:
CG_PHIT0_SRC_FGEAR means PhiT0 source is fgear.
CG_PHIT0_SRC_FC means PhiT0 source is fc.
CG_PHIT0_SRC_FSYS means PhiT0 source is fsys.

4.2.3.5 CG_SetPhiT0Level

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

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

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

- **CG_DIVIDE_1**: ΦT0 = fc or fsys
- **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 orΦT0 and fsys.

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 orΦT0 and fsys, the value will be one of the following:
CG_DIVIDE_1: ΦT0 = fc or fsys
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_FS**: SCOUT source is set to fs.
- **CG_SCOUT_SRC_HALF_FSYS**: SCOUT source is set to fsys/2.
- **CG_SCOUT_SRC_FSYS**: SCOUT source is set to fsys.
- **CG_SCOUT_SRC_PHIT0**: SCOUT source is set to 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_FS: SCOUT source is fs
CG_SCOUT_SRC_HALF_FSYS: SCOUT source is set to fsys/2
CG_SCOUT_SRC_FSYS: SCOUT source is fsys
CG_SCOUT_SRC_PHIT0: SCOUT source is ΦT0

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_HIGH**: External High-Speed oscillator is selected as timer source.
- **CG_WARM_UP_SRC_OSC_INT_HIGH**: Internal High-Speed oscillator is selected as timer source.
- **CG_WARM_UP_SRC_OSC_EXT_LOW**: External Low-Speed oscillator is selected as timer source.

Time: If ***Source*** is **CG_WARM_UP_SRC_OSC_EXT_HIGH** or **CG_WARM_UP_SRC_OSC_INT_HIGH,** Time value range is 0U to 0x0FFFU.
If ***Source*** is **CG_WARM_UP_SRC_OSC_EXT_LOW,** Time value range is 0U to 0x3FFFU.

Description:
This function will set the warm-up time and warm-up counter. And the formula is as the following:
Setting_value = ((warm-up time) / (input cycle time by frequency))/16

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

Return:
None.

4.2.3.10 CG_StartWarmUp

Start warm up timer.

Prototype:
void

CG_StartWarmUp(void)

Parameters:
None

Description:
This function will start the warm up timer.

Return:
None

4.2.3.11 CG_GetWarmUpState

Check that warm-up operation is in middle or completed.

Prototype:
WorkState
CG_GetWarmUpState(void)

Parameters:
None

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

Example of using warm-up timer:

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

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

4.2.3.12 CG_SetPLL

Enable or disable the PLL circuit.

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

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

Description:
This function will enable or disable the PLL circuit as the input parameter.
If the PLL is selected as fc, it can’t be disabled; in that case the API will return **ERROR**.

Return:
SUCCESS: operation is finished successfully.

ERROR: operation is not done.

4.2.3.13 CG_GetPLLState

Get the state of PLL circuit.

Prototype:
FunctionalState
CG_GetPLLState(void)

Parameters:
None

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

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

4.2.3.14 CG_SetFosc

Enable or disable the high-speed oscillator (fEHOSC or fIHOSC).

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

Parameters:
Source: select source for fosc.
- **CG_FOSC_EHOSC**: fEHOSC is selected.
- **CG_FOSC_IHOSC**: fIHOSC is selected.

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

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

4.2.3.15 CG_SetFoscSrc

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

Prototype:
void
CG_SetFoscSrc(CG_FoscSrc ***Source***)

Parameters:
Source: select source for fosc.
- **CG_FOSC_EHOSC**: fEHOSC is selected.
- **CG_FOSC_IHOSC**: fIHOSC is selected.

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

Return:
None

4.2.3.16 CG_GetFoscSrc

Get the source of the high-speed oscillator.

Prototype:
CG_FoscSrc
CG_GetFoscSrc(void)

Parameters:
None

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

Return:
The source of fosc
CG_FOSC_EHOSC: fEHOSC is selected.
CG_FOSC_IHOSC: fIHOSC is selected.

4.2.3.17 CG_GetFoscState

Get the state of the high-speed oscillator.

Prototype:
FunctionalState
CG_GetFoscState(CG_FoscSrc ***Source***)

Parameters:
Source: select source for fosc.
- **CG_FOSC_EHOSC**: fEHOSC is selected.
- **CG_FOSC_IHOSC**: fIHOSC is selected.

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

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

4.2.3.18 CG_SetFs

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

Prototype:

Result
CG_SetFs(FunctionalState ***NewState***)

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

Description:
This function will enable or disable the low-speed oscillator (fs).
When fs is selected as system clock (fsys), the low-speed oscillator (fs) can't be disabled, in that case the API will return **ERROR**.

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

4.2.3.19 CG_GetFsState

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

Prototype:
FunctionalState
CG_GetFsState (void)

Parameters:
None

Description:
This function will get the state of the low-speed oscillator (fs).

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

4.2.3.20 CG_SetPortM

Set portM as X1/X2 or general port.

Prototype:
void
CG_SetPortM(CG_PortMMode ***Mode***)

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

Description:
This function will set port M as general port when ***Mode*** is **CG_PORTM_AS_GPIO** and set port M as Hosc when ***Mode*** is **CG_PORTM_AS_HOSC**.

Return:
None

4.2.3.21 CG_SetSTBYMode

Set the standby mode.

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

Parameters:
Mode: the low power consumption mode, the description of each value is as the following:
- **CG_STBY_MODE_STOP**: STOP mode. All the internal circuits including the internal oscillator are brought to a stop.
- **CG_STBY_MODE_SLEEP**: SLEEP mode. The internal low-speed oscillator, real time clock and RMC can operate.
- **CG_STBY_MODE_IDLE**: IDLE mode. Only CPU stop in this mode.

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

Return:
None

4.2.3.22 CG_GetSTBYMode

Get the standby mode.

Prototype:
CG_STBYMode
CG_GetSTBYMode(void)

Parameters:
None

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

Return:
The low power mode:
CG_STBY_MODE_STOP: STOP mode.
CG_STBY_MODE_SLEEP: SLEEP mode.
CG_STBY_MODE_IDLE: IDLE mode.
CG_STBY_MODE_UNKNOWN: Invalid data is read.

4.2.3.23 CG_SetExitStopModeFosc

Enable or disable fosc after releasing stop mode

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

Parameters:
NewState:

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

Description:
This function will enable or disable fosc after releasing stop mode.

Return:
None

4.2.3.24 CG_GetExitStopModeFoscState

Get the state of fosc after releasing stop mode.

Prototype:
FunctionalState
CG_GetExitStopModeFoscState(void)

Parameters:
None

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

Return:
ENABLE: enable fosc after releasing stop mode.
DISABLE: do not enable fosc after releasing stop mode.

4.2.3.25 CG_SetExitStopModeFs

Enable or disable fs after releasing stop mode.

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

Parameters:
NewState:
- **ENABLE :** enable fs after releasing stop mode.
- **DISABLE:** do not enable fs after releasing stop mode.

Description:
This function will enable or disable fs after releasing stop mode.

Return:
None

4.2.3.26 CG_GetExitStopModeFsState

Get the state of fs after releasing stop mode.

Prototype:
FunctionalState
CG_GetExitStopModeFsState(void)

Parameters:
None

Description:
This function will get the state of fs after releasing stop mode.

Return:
ENABLE: enable fs after releasing stop mode.
DISABLE: do not enable fs after releasing stop mode.

4.2.3.27 CG_SetPinStateInStopMode

Specify the pin status in stop mode.

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

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

For the detailed state of port corresponding to “<DRVE>=0” or “<DRVE>=1”, please refer to the table “Pin Status in the STOP Mode” in the datasheet.

Description:
This function will specify the pin status in stop mode.

Return:
None

4.2.3.28 CG_GetPinStateInStopMode

Get the pin status in stop mode.

Prototype:
FunctionalState
CG_GetPinStateInStopMode(void)

Parameters:
None

Description:
This function will get the pin status in stop mode.

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

4.2.3.29 CG_SetFcSrc

Set the clock source of fc

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

Parameters:

Source: the source for fc
This parameter can be one of the following values:

- **CG_FC_SRC_FOSC**: fc source will be set to fosc
- **CG_FC_SRC_FPLL**: fc source will be set to fPLL

Description:
This function will set the clock source of fc.
The following conditions should be matched before calling this API

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

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

Return:
SUCCESS: set clock souce for fc successfully
ERROR: clock source of fc is not changed.

4.2.3.30 CG_GetFcSrc

Get the clock source of fc.

Prototype:
CG_FcSrc
CG_GetFosc(void)

Parameters:
None

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

Return:
The clock source of fc
The value returned can be one of the following values:
CG_FC_SRC_FOSC: fc source is set to fosc.
CG_FC_SRC_FPLL: fc source is set to fPLL.

4.2.3.31 CG_SetFsysSrc

Set the clock source of fsys.

Prototype:
Result
CG_SetFsysSrc(CG_FsysSrc ***Source***)

Parameters:
Source: select the source of system clock (fsys).
This parameter can be one of the following values:

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

Description:
This function will set the clock source of system clock (fsys).
If **CG_FSYS_SRC_FGEAR** is specified, the high-speed oscillator (fsys) should be enabled earlier; if **CG_FSYS_SRC_FS** is specified, the low-speed oscillator (fs) should be enabled earlier; otherwise, calling of this API will return **ERROR**.

Return:
SUCCESS: set clock source for fsys successfully.
ERROR: the clock source of fsys is not changed.

4.2.3.32 CG_GetFsysSrc

Get the clock source of fsys.

Prototype:
CG_FsysSrc
CG_GetFsysSrc(void)

Parameters:
None

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

Return:
Source of fsys
The value returned can be one of the following values:
CG_FSYS_SRC_FGEAR : source of fsys is set to fgear.
CG_FSYS_SRC_FS : source of fsys is set to fs.

4.2.3.33 CG_SetSTBYReleaseINTSrc

Set the INT source for releasing low power mode.

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

Parameters:
INTSource: select the INT source for releasing standby mode.
This parameter can be one of the following values:

- **CG_INT_SRC_0:** INT0.
- **CG_INT_SRC_1:** INT1.
- **CG_INT_SRC_2:** INT2.
- **CG_INT_SRC_3:** INT3.
- **CG_INT_SRC_4:** INT4.
- **CG_INT_SRC_5:** INT5.
- **CG_INT_SRC_6:** INT6. (Only for TMPM381)
- **CG_INT_SRC_7:** INT7. (Only for TMPM381)
- **CG_INT_SRC_8:** INT8.
- **CG_INT_SRC_9:** INT9. (Only for TMPM381)
- **CG_INT_SRC_A:** INTA. (Only for TMPM381)
- **CG_INT_SRC_B:** INTB. (Only for TMPM381)
- **CG_INT_SRC_C:** INTC. (Only for TMPM381)
- **CG_INT_SRC_D:** INTD. (Only for TMPM381)
- **CG_INT_SRC_E:** INTE. (Only for TMPM381)
- **CG_INT_SRC_F:** INTF.
- **CG_INT_SRC_RTC:** RTC interrupt.

- **CG_INT_SRC_RMC_RX:** receptions interrupt of RMC.

ActiveState: select the active state for release trigger.
This parameter can be one of the following values:
- **CG_INT_ACTIVE_STATE_L**: active on low level
- **CG_INT_ACTIVE_STATE_H**: active on high level
- **CG_INT_ACTIVE_STATE_FALLING**: active on falling edge
- **CG_INT_ACTIVE_STATE_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 **CG_INT_SRC_RMC_RX**, only "rising" state (**CG_INT_ACTIVE_STATE_RISING)** will be set to the register, no matter what the value of ***ActiveState*** is. For **CG_INT_SRC_RTC**, only "falling" state (**CG_INT_ACTIVE_STATE_FALLING**) will be set to the register, no matter what the value of ***ActiveState*** is.

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:** 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. (Only for TMPM381)
- **CG_INT_SRC_7:** INT7. (Only for TMPM381)
- **CG_INT_SRC_8:** INT8.
- **CG_INT_SRC_9:** INT9. (Only for TMPM381)
- **CG_INT_SRC_A:** INTA. (Only for TMPM381)
- **CG_INT_SRC_B:** INTB. (Only for TMPM381)
- **CG_INT_SRC_C:** INTC. (Only for TMPM381)
- **CG_INT_SRC_D:** INTD. (Only for TMPM381)
- **CG_INT_SRC_E:** INTE. (Only for TMPM381)
- **CG_INT_SRC_F:** INTF.
- **CG_INT_SRC_RTC:** RTC interrupt.

- **CG_INT_SRC_RMC_RX:** receptions interrupt of RMC.

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

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

4.2.3.35 CG_ClearINTReq

Clear the INT request for releasing standby mode.

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

Parameters:
INTSource: select the release INT source.
This parameter can be one of the following values:
- **CG_INT_SRC_0:** 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. (Only for TMPM381)
- **CG_INT_SRC_7:** INT7. (Only for TMPM381)
- **CG_INT_SRC_8:** INT8.
- **CG_INT_SRC_9:** INT9. (Only for TMPM381)
- **CG_INT_SRC_A:** INTA. (Only for TMPM381)
- **CG_INT_SRC_B:** INTB. (Only for TMPM381)
- **CG_INT_SRC_C:** INTC. (Only for TMPM381)
- **CG_INT_SRC_D:** INTD. (Only for TMPM381)
- **CG_INT_SRC_E:** INTE. (Only for TMPM381)
- **CG_INT_SRC_F:** INTF.
- **CG_INT_SRC_RTC:** RTC interrupt.
- **CG_INT_SRC_RMC_RX:** receptions interrupt of RMC.

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

Return:
None

4.2.3.36 CG_GetNMIFlag

Get the NMI flag, which shows what triggered NMI.

Prototype:
CG_NMIFactor
CG_ GetNMIFlag(void)

Parameters:
None

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

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

4.2.3.37 CG_GetResetFlag

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

Prototype:
CG_ResetFlag
CG_GetResetFlag(void)

Parameters:
None

Description:
This function will get the reset flag which shows the trigger of reset and clear the reset flag.

Return:
Reset flag:
PowerOn (Bit 0) means reset from power-on.
ResetPin (Bit 1) means reset from Reset pin.
WDTReset (Bit 2) means reset from WDT.
DebugReset (Bit 4) means reset from SYSRESETREQ.
OFDReset (Bit 5) means reset from OFD.

4.2.4 Data Structure Description

4.2.4.1 CG_NMIFactor

Data Fields:
uint32_t
All specifies CGNMI source generation state.

Bit Fields:

uint32_t		
WDT:	0	From WDT source.
uint32_t		
Reserved0:	31	Reserved.

4.2.4.2 CG_ResetFlag

Data Fields:
uint32_t
All specifies CG reset source.

Bit Fields:

uint32_t		
PowerOn:	1	means reset from power-on.

uint32_t **ResetPin**:	1	means reset from Reset pin.
uint32_t **WDTReset**:	1	means reset from WDT.
uint32_t **Reserved0**:	1	means reserved.
uint32_t **DebugReset**:	1	means reset from SYSRESETREQ.
uint32_t **OFDReset**:	1	means reset from OFD.
uint32_t **Reserved1**:	26	means reserved.

5. DNF

5.1 Overview

The digital noise canceller circuit can eliminate noise of input signals from external interrupt pins at the certain range.

The DNF drivers API provide a set of functions to configure DNF, including such parameters as noise filter clock, noise filter interrupt state, noise filter interrupt setting and so on.

This driver is contained in \Libraries\TX03_Periph_Driver\src\tmpm38x_dnf.c, with \Libraries/TX03_Periph_Driver\inc\tmpm38x_dnf.h (see **Note** below) containing the API definitions for use by applications.

Note: tmpm38x can be tmpm381 or tmpm383.

5.2 API Functions

5.2.1 Function List

- void DNF_SetFilterClk(uint32_t ***FilterClk***)
- uint32_t DNF_GetFilterClk(void)
- void DNF_SetInt0Filter(FunctionalState ***NewState***)
- FunctionalState DNF_GetInt0FilterState(void)
- void DNF_SetInt1Filter(FunctionalState ***NewState***)
- FunctionalState DNF_GetInt1FilterState(void)
- void DNF_SetInt2Filter(FunctionalState ***NewState***)
- FunctionalState DNF_GetInt2FilterState(void)
- void DNF_SetInt3Filter(FunctionalState ***NewState***)
- FunctionalState DNF_GetInt3FilterState(void)
- void DNF_SetInt4Filter(FunctionalState ***NewState***)
- FunctionalState DNF_GetInt4FilterState(void)
- void DNF_SetInt5Filter(FunctionalState ***NewState***)
- FunctionalState DNF_GetInt5FilterState(void)
- void DNF_SetInt6Filter(FunctionalState ***NewState***) (Only for TMPM381)
- FunctionalState DNF_GetInt6FilterState(void) (Only for TMPM381)
- void DNF_SetInt7Filter(FunctionalState ***NewState***) (Only for TMPM381)
- FunctionalState DNF_GetInt7FilterState(void) (Only for TMPM381)
- void DNF_SetInt8Filter(FunctionalState ***NewState***)
- FunctionalState DNF_GetInt8FilterState(void)
- void DNF_SetInt9Filter(FunctionalState ***NewState***) (Only for TMPM381)
- FunctionalState DNF_GetInt9FilterState(void) (Only for TMPM381)
- void DNF_SetIntAFilter(FunctionalState ***NewState***) (Only for TMPM381)
- FunctionalState DNF_GetIntAFilterState(void) (Only for TMPM381)
- void DNF_SetIntBFilter(FunctionalState ***NewState***) (Only for TMPM381)
- FunctionalState DNF_GetIntBFilterState(void) (Only for TMPM381)
- void DNF_SetIntCFilter(FunctionalState ***NewState***) (Only for TMPM381)
- FunctionalState DNF_GetIntCFilterState(void) (Only for TMPM381)
- void DNF_SetIntDFilter(FunctionalState ***NewState***) (Only for TMPM381)
- FunctionalState DNF_GetIntDFilterState(void) (Only for TMPM381)
- void DNF_SetIntEFilter(FunctionalState ***NewState***) (Only for TMPM381)

- FunctionalState DNF_GetIntEFilterState(void) (Only for TMPM381)
- void DNF_SetIntFFilter(FunctionalState ***NewState***)
- FunctionalState DNF_GetIntFFilterState(void)

5.2.2 Detailed Description

Functions listed above can be divided into two parts:

1) The noise filter clock selection are handled by the DNF_SetFilterClk(), and DNF_GetFilterClk() functions.
2) The noise filter interrupt setting are handled by the following functions: DNF_SetInt0Filter(), DNF_GetInt0FilterState(), DNF_SetInt1Filter(), DNF_GetInt1FilterState(), DNF_SetInt2Filter(), DNF_GetInt2FilterState(), DNF_SetInt3Filter(), DNF_GetInt3FilterState(), DNF_SetInt4Filter(), DNF_GetInt4FilterState(), DNF_SetInt5Filter(), DNF_GetInt5FilterState(), DNF_SetInt8Filter(), DNF_GetInt8FilterState(), DNF_SetIntFFilter(), DNF_GetIntFFilterState()

   For TMPM381:
   DNF_SetInt6Filter(), DNF_GetInt6FilterState(), DNF_SetInt7Filter(), DNF_GetInt7FilterState(),DNF_SetInt9Filter(), DNF_GetInt9FilterState(), DNF_SetIntAFilter(), DNF_GetIntAFilterState(), DNF_SetIntBFilter(), DNF_GetIntBFilterState(), DNF_SetIntCFilter(), DNF_GetIntCFilterState(), DNF_SetIntDFilter(), DNF_GetIntDFilterState(), DNF_SetIntEFilter(), DNF_GetIntEFilterState().

5.2.3 Function Documentation

5.2.3.1 DNF_SetFilterClk

Select noise filter clock.

Prototype:
void
DNF_SetFilterClk(uint32_t ***FilterClk***)

Parameters:
***FilterClk*:** Noise filter clock
This parameter can be one of the following values:
- **DNF_FILTER_CLK_STOP:** Clock control circuit stops
- **DNF_FILTER_CLK_FSYS_2:** fsys/2 clock output
- **DNF_FILTER_CLK_FSYS_4:** fsys/4 clock output
- **DNF_FILTER_CLK_FSYS_8:** fsys/8 clock output
- **DNF_FILTER_CLK_FSYS_16:** fsys/16 clock output
- **DNF_FILTER_CLK_FSYS_32:** fsys/32 clock output
- **DNF_FILTER_CLK_FSYS_64:** fsys/64 clock output
- **DNF_FILTER_CLK_FSYS_128:** fsys/128 clock output

Description:
This function will select noise filter clock.

Return:
None

5.2.3.2 DNF_GetFilterClk

Get noise filter clock.

Prototype:
uint32_t
DNF_GetFilterClk(void)

Parameters:
None

Description:
This function will get the noise filter clock.

Return:
The noise filter clock:
- **DNF_FILTER_CLK_STOP:** Clock control circuit stops
- **DNF_FILTER_CLK_FSYS_2:** fsys/2 clock output
- **DNF_FILTER_CLK_FSYS_4:** fsys/4 clock output
- **DNF_FILTER_CLK_FSYS_8:** fsys/8 clock output
- **DNF_FILTER_CLK_FSYS_16:** fsys/16 clock output
- **DNF_FILTER_CLK_FSYS_32:** fsys/32 clock output
- **DNF_FILTER_CLK_FSYS_64:** fsys/64 clock output
- **DNF_FILTER_CLK_FSYS_128**: fsys/128 clock output

5.2.3.3 DNF_SetInt0Filter

Enable or disable the INT0 noise filter.

Prototype:
void
DNF_SetInt0Filter(FunctionalState ***NewState***)

Parameters:
NewState: The INT0 noise filter state.
This parameter can be one of the following values:
- **ENABLE**: Enabled (Post-noise filtering output signal)
- **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:
This function will enable or disable the INT0 noise filter.

Return:
None

5.2.3.4 DNF_GetInt0FilterState

Get the INT0 noise filter state.

Prototype:
FunctionalState
DNF_GetInt0FilterState(void)

Parameters:
None

Description:
This function will get the INT0 noise filter state.

Return:

The INT0 noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.3.5 DNF_SetInt1Filter

Enable or disable the INT1 noise filter.

Prototype:
void
DNF_SetInt1Filter(FunctionalState ***NewState***)

Parameters:
NewState: The INT1 noise filter state.
This parameter can be one of the following values:
- **ENABLE**: Enabled (Post-noise filtering output signal)
- **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:
This function will enable or disable the INT1 noise filter.

Return:
None

5.2.3.6 DNF_GetInt1FilterState

Get the INT1 noise filter state.

Prototype:
FunctionalState
DNF_GetInt1FilterState(void)

Parameters:
None

Description:
This function will get the INT1 noise filter state.

Return:
The INT1 noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.3.7 DNF_SetInt2Filter

Enable or disable the INT2 noise filter.

Prototype:
void
DNF_SetInt2Filter(FunctionalState ***NewState***)

Parameters:
NewState: The INT2 noise filter state.
This parameter can be one of the following values:

- **ENABLE**: Enabled (Post-noise filtering output signal)
- **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:
This function will enable or disable the INT2 noise filter.

Return:
None

5.2.3.8 DNF_GetInt2FilterState

Get the INT2 noise filter state.

Prototype:
FunctionalState
DNF_GetInt2FilterState(void)

Parameters:
None

Description:
This function will get the INT2 noise filter state.

Return:
The INT2 noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.3.9 DNF_SetInt3Filter

Enable or disable the INT3 noise filter.

Prototype:
void
DNF_SetInt3Filter(FunctionalState ***NewState***)

Parameters:
NewState: The INT3 noise filter state.
This parameter can be one of the following values:

- **ENABLE**: Enabled (Post-noise filtering output signal)
- **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:
This function will enable or disable the INT3 noise filter.

Return:
None

5.2.3.10 DNF_GetInt3FilterState

Get the INT3 noise filter state.

Prototype:
FunctionalState

DNF_GetInt3FilterState(void)

Parameters:
None

Description:
This function will get the INT3 noise filter state.

Return:
The INT3 noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.3.11 DNF_SetInt4Filter

Enable or disable the INT4 noise filter.

Prototype:
void
DNF_SetInt4Filter(FunctionalState ***NewState***)

Parameters:
NewState: The INT4 noise filter state.
This parameter can be one of the following values:
- **ENABLE**: Enabled (Post-noise filtering output signal)
- **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:
This function will enable or disable the INT4 noise filter.

Return:
None

5.2.3.12 DNF_GetInt4FilterState

Get the INT4 noise filter state.

Prototype:
FunctionalState
DNF_GetInt4FilterState(void)

Parameters:
None

Description:
This function will get the INT4 noise filter state.

Return:
The INT4 noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.3.13 DNF_SetInt5Filter

Enable or disable the INT5 noise filter.

Prototype:
void
DNF_SetInt5Filter(FunctionalState ***NewState***)

Parameters:
NewState: The INT5 noise filter state.
This parameter can be one of the following values:
- **ENABLE**: Enabled (Post-noise filtering output signal)
- **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:
This function will enable or disable the INT5 noise filter.

Return:
None

5.2.3.14 DNF_GetInt5FilterState

Get the INT5 noise filter state.

Prototype:
FunctionalState
DNF_GetInt5FilterState(void)

Parameters:
None

Description:
This function will get the INT5 noise filter state.

Return:
The INT5 noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.3.15 DNF_SetInt6Filter

Enable or disable the INT6 noise filter.

Prototype:
void
DNF_SetInt6Filter(FunctionalState ***NewState***)

Parameters:
NewState: The INT6 noise filter state.
This parameter can be one of the following values:
- **ENABLE**: Enabled (Post-noise filtering output signal)
- **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:

This function will enable or disable the INT6 noise filter.

Note:
This function only supports TMPM381.

Return:
None

5.2.3.16 DNF_GetInt6FilterState

Get the INT6 noise filter state.

Prototype:
FunctionalState
DNF_GetInt6FilterState(void)

Parameters:
None

Description:
This function will get the INT6 noise filter state.

Note:
This function only supports TMPM381.

Return:
The INT6 noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.3.17 DNF_SetInt7Filter

Enable or disable the INT7 noise filter.

Prototype:
void
DNF_SetInt7Filter(FunctionalState ***NewState***)

Parameters:
NewState: The INT7 noise filter state.
This parameter can be one of the following values:
- **ENABLE**: Enabled (Post-noise filtering output signal)
- **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:
This function will enable or disable the INT7 noise filter.

Note:
This function only supports TMPM381.

Return:
None

5.2.3.18 DNF_GetInt7FilterState

Get the INT7 noise filter state.

Prototype:
FunctionalState
DNF_GetInt7FilterState(void)

Parameters:
None

Description:
This function will get the INT7 noise filter state.

Note:
This function only supports TMPM381.

Return:
The INT7 noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.3.19 DNF_SetInt8Filter

Enable or disable the INT8 noise filter.

Prototype:
void
DNF_SetInt8Filter(FunctionalState ***NewState***)

Parameters:
NewState: The INT8 noise filter state.
This parameter can be one of the following values:
- **ENABLE**: Enabled (Post-noise filtering output signal)
- **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:
This function will enable or disable the INT8 noise filter.

Return:
None

5.2.3.20 DNF_GetInt8FilterState

Get the INT8 noise filter state.

Prototype:
FunctionalState
DNF_GetInt8FilterState(void)

Parameters:
None

Description:
This function will get the INT8 noise filter state.

Return:
The INT8 noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.3.21 DNF_SetInt9Filter

Enable or disable the INT9 noise filter.

Prototype:
void
DNF_SetInt9Filter(FunctionalState ***NewState***)

Parameters:
NewState: The INT9 noise filter state.
This parameter can be one of the following values:
- **ENABLE**: Enabled (Post-noise filtering output signal)
- **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:
This function will enable or disable the INT9 noise filter.

Note:
This function only supports TMPM381.

Return:
None

5.2.3.22 DNF_GetInt9FilterState

Get the INT9 noise filter state.

Prototype:
FunctionalState
DNF_GetInt9FilterState(void)

Parameters:
None

Description:
This function will get the INT9 noise filter state.

Note:
This function only supports TMPM381.

Return:
The INT9 noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.3.23 DNF_SetIntAFilter

Enable or disable the INTA noise filter.

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

Parameters:
NewState: The INTA noise filter state.
This parameter can be one of the following values:
- **ENABLE**: Enabled (Post-noise filtering output signal)
- **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:
This function will enable or disable the INTA noise filter.

Note:
This function only supports TMPM381.

Return:
None

5.2.3.24 DNF_GetIntAFilterState

Get the INTA noise filter state.

Prototype:
FunctionalState
DNF_GetIntAFilterState(void)

Parameters:
None

Description:
This function will get the INTA noise filter state.

Note:
This function only supports TMPM381.

Return:
The INTA noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.3.25 DNF_SetIntBFilter

Enable or disable the INTB noise filter.

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

Parameters:
NewState: The INTB noise filter state.
This parameter can be one of the following values:
- **ENABLE**: Enabled (Post-noise filtering output signal)

➢ **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:
This function will enable or disable the INTB noise filter.

Note:
This function only supports TMPM381.

Return:
None

5.2.3.26 DNF_GetIntBFilterState

Get the INTB noise filter state.

Prototype:
FunctionalState
DNF_GetIntBFilterState(void)

Parameters:
None

Description:
This function will get the INTB noise filter state.

Note:
This function only supports TMPM381.

Return:
The INTB noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.3.27 DNF_SetIntCFilter

Enable or disable the INTC noise filter.

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

Parameters:
NewState: The INTC noise filter state.
This parameter can be one of the following values:
➢ **ENABLE**: Enabled (Post-noise filtering output signal)
➢ **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:
This function will enable or disable the INTC noise filter.

Note:
This function only supports TMPM381.

Return:

None

5.2.3.28 DNF_GetIntCFilterState

Get the INTC noise filter state.

Prototype:
FunctionalState
DNF_GetIntCFilterState(void)

Parameters:
None

Description:
This function will get the INTC noise filter state.

Note:
This function only supports TMPM381.

Return:
The INTC noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.3.29 DNF_SetIntDFilter

Enable or disable the INTD noise filter.

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

Parameters:
NewState: The INTD noise filter state.
This parameter can be one of the following values:
- **ENABLE**: Enabled (Post-noise filtering output signal)
- **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:
This function will enable or disable the INTD noise filter.

Note:
This function only supports TMPM381.

Return:
None

5.2.3.30 DNF_GetIntDFilterState

Get the INTD noise filter state.

Prototype:
FunctionalState
DNF_GetIntDFilterState(void)

Parameters:
None

Description:
This function will get the INTD noise filter state.

Note:
This function only supports TMPM381.

Return:
The INTD noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.3.31 DNF_SetIntEFilter

Enable or disable the INTE noise filter.

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

Parameters:
NewState: The INTE noise filter state.
This parameter can be one of the following values:
- **ENABLE**: Enabled (Post-noise filtering output signal)
- **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:
This function will enable or disable the INTE noise filter.

Note:
This function only supports TMPM381.

Return:
None

5.2.3.32 DNF_GetIntEFilterState

Get the INTE noise filter state.

Prototype:
FunctionalState
DNF_GetIntEFilterState(void)

Parameters:
None

Description:
This function will get the INTE noise filter state.

Note:
This function only supports TMPM381.

Return:

The INTE noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.3.33 DNF_SetIntFFilter

Enable or disable the INTF noise filter.

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

Parameters:
NewState: The INTF noise filter state.
This parameter can be one of the following values:
- **ENABLE**: Enabled (Post-noise filtering output signal)
- **DISABLE**: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

Description:
This function will enable or disable the INTF noise filter.

Return:
None

5.2.3.34 DNF_GetIntFFilterState

Get the INTF noise filter state.

Prototype:
FunctionalState
DNF_GetIntFFilterState(void)

Parameters:
None

Description:
This function will get the INTF noise filter state.

Return:
The INTF noise filter state:
ENABLE: Enabled (Post-noise filtering output signal)
DISABLE: Disabled (Pre-noise filtering output signal and noise filter circuit counter are cleared when releasing STOP mode.)

5.2.4 Data Structure Description

None

6. FC

6.1 Overview

TMPM38x device contains flash memory.
For TMPM38xFW, the size of flash is 128Kbyte. For TMPM38xFS, the size of flash is 64Kbyte.

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

The block configuration of flash memory please refers to the MCU data sheet.

This driver is contained in \Libraries\TX03_Periph_Driver\src\tmpm38x_fc.c with \Libraries\TX03_Periph_Driver \inc\tmpm38x_fc.h containing the API definitions for use by applications.

6.2 API Functions

6.2.1 Function List

- void FC_SetBufferState(FunctionalState ***NewState***)
- void FC_SetSecurityBit(FunctionalState ***NewState***)
- FunctionalState FC_GetSecurityBit(void)
- WorkState FC_GetBusyState(void)
- void FC_SetBlockProtectMask(uint8_t ***BlockNum***, FunctionalState ***NewState***)
- FunctionalState FC_GetBlockProtectMaskState(uint8_t ***BlockNum***)
- FunctionalState FC_GetBlockProtectState(uint8_t ***BlockNum***)
- FC_Result FC_ProgramBlockProtectState(uint8_t ***BlockNum***)
- FC_Result FC_EraseBlockProtectState(uint8_t ***BlockGroup***)
- FC_Result FC_WritePage(uint32_t ***PageAddr***, uint32_t * ***Data***)
- FC_Result FC_EraseBlock(uint32_t ***BlockAddr***)
- FC_Result FC_EraseChip(void)

6.2.2 Detailed Description

Functions listed above can be divided into seven 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().
5) The function controls flash interface with instruction Buffer
   FC_SetBufferState()
6) The function masks protect block
   FC_SetBlockProtectMask()
7) The function gets block protection mask state

FC_GetBlockProtectMaskState()

6.2.3 Function Documentation

6.2.3.1 FC_SetBufferState

Set the value of FCCR register.

Prototype:
void
FC_ SetBufferState(FunctionalState ***NewState***)

Parameters:
NewState: Set the state of FCCR register.
This parameter can be one of the following values:
- **DISABLE**: Disable Instruction Buffer (with Buffer clear)
- **ENABLE**: Enable Instruction Buffer.

Description:
After flash programing or flash erasing, it should clear instruction buffer by this function.

Return:
None

6.2.3.2 FC_SetSecurityBit

Set the value of FCSECBIT 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 FCPSRA<BLK> bits) used for the write/erase-protection function are set to “1”.
2) The FCSECBIT <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 FCSECBIT<SEBIT> to “1”.
The FCSECBIT <SECBIT> bit is set to “1” at a power-on reset right after power-on.

Return:
None

6.2.3.3 FC_GetSecurityBit

Get the value of FCSECBIT register.

Prototype:
FunctionalState
FC_GetSecurityBit(void)

Parameters:
None

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

Return:
State of FCSECBIT register.
DISABLE: Protection function is not available.
ENABLE: Protection function is available.

6.2.3.4 FC_GetBusyState

Reading FCSR register to get the status of the flash auto operation.

Prototype:
WorkState
FC_GetBusyState (void)

Parameters:
None

Description:
When the flash memory is in automatic operation, FCSR<RDY_BSY> bit is set to "0" to indicate that flash memory is busy. When the automatic operation is normally terminated, FCSR<RDY_BSY> bit is set to "1", flash memory becomes ready state to accept the next command.
Note: *Make sure that flash memory is ready before commands are issued*

Return:
Status of the flash automatic operation:
BUSY: Flash memory is in automatic operation.
DONE: Automatic operation is normally finished. The next command can be accepted and executed.

6.2.3.5 FC_SetBlockProtectMask

Set the bit protection mask

Prototype:
void
FC_SetBlockProtectMask(uint8_t ***BlockNum***,
FunctionalState ***NewState***)

Parameters:
BlockNum: The flash block number
- **FC_BLOCK_0** for block 0
- **FC_BLOCK_1** for block 1
- **FC_BLOCK_2** for block 2 (This block is not used for TMPM38xFS.)
- **FC_BLOCK_3** for block 3 (This block is not used for TMPM38xFS.)

NewState: Specifies enable/disable the bit protection mask
This parameter can be one of the following values:

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

Description:
TMPM38x can temporarily release the protect function by masking the protect bits. Using FCPMRA register to mask protect bits of block 3 through block 0. If *NewState* is ENABLE, FCPMRA<BLKM[*BlockNum*]> bit is set to "0" indicates the protection function of corresponding block is released. When *NewState* is DISABLE, FCPMRA<BLKM[*BlockNum*]> bit is set to "1", the corresponding block becomes protect state.

Return:
None

6.2.3.6 FC_GetBlockProtectMaskState

Get the specified block protection mask state

Prototype:
FunctionalState
FC_GetBlockProtectMaskState(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

Description:
With the given *BlockNum,* this function reads value of the corresponding bit in FCPMRA register: This function returns **ENABLE** if FCPMRA<BLKM[*BlockNum*]> is set to "0", and returns **DISABLE** if FCPMRA<BLKM[*BlockNum*]> is set to "1".

Return:
State of the corresponding bit in FCPMRA register.
DISABLE: Protect bit is not masked.
ENABLE: Protect bit is masked.

6.2.3.7 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

Description:

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

Return:
Block protection status.
DISABLE: Block is unprotected
ENABLE: Block is protected

6.2.3.8 FC_ProgramBlockProtectState

Program the protection bits.

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

Parameters:
BlockNum:The flash block number
- **FC_BLOCK_0** for block 0
- **FC_BLOCK_1** for block 1
- **FC_BLOCK_2** for block 2
- **FC_BLOCK_3** for block 3

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

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

6.2.3.9 FC_EraseBlockProtectState

Erase the protection bits.

Prototype:
FC_Result
FC_EraseBlockProtectState(uint8_t ***BlockGroup***)

Parameters:
BlockGroup:The flash block group
- **FC_BLOCK_GROUP_0** for the all 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 specified time error.

6.2.3.10 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 128Byte.

Description:
This API is used to write data to specified page.
The TMPM38x contains 32 words in a page. The flash can only be written page by page.
The automatic page programming is allowed only once for a page already erased. No programming can be performed twice or more time irrespective of data value whether it is “1” or “0”.
Note: An attempt to rewrite a page two or more times without erasing the content can cause damages to the device.

Return:
Result of the operation to write data to the specified page.
FC_SUCCESS: data is written to the specified page accurately.
FC_ERROR_PROTECTED: The block is protected. The write operation can’t be executed.
FC_ERROR_OVER_TIME: Write operation over time error.

6.2.3.11 FC_EraseBlock

Erase the content of specified block.

Prototype:
FC_Result
FC_EraseBlock(uint32_t ***BlockAddr***)

Parameters:
BlockAddr: The block starts address.

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

Return:
Result of the operation to erase the content of specified block.
FC_SUCCESS: the content of the specified block is erased successfully.
FC_ERROR_PROTECTED: The block is protected. The erase operation can’t be executed. The block will not be erased.
FC_ERROR_OVER_TIME: Erase operation over time error.

6.2.3.12 FC_EraseChip

Erase the content of the entire chip.

Prototype:
FC_Result
FC_EraseChip(void)

Parameters:
None

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

Return:
Result of the operation to erase the content of the entire chip.
FC_SUCCESS: If all the blocks are unprotected, the entire chip is erased. If parts of blocks are protected, only unprotected blocks are erased.
FC_ERROR_PROTECTED: All blocks are protected. The erase chip operation can't be executed.
FC_ERROR_OVER_TIME: Erase Chip operation over time error.

6.2.4 Data Structure Description

None.

7. FUART

7.1 Overview

TOSHIBA TMPM38x contains the Asynchronous serial channel (Full UART) with 50% duty cycle mode.
TOSHIBA TMPM38x contains one channel Full UART: FUART0.

The FUART driver APIs provide a set of functions to configure the Full UART channel, including such common parameters as baud rate, bit length, parity check, stop bit, flow control, and to control transfer like sending/receiving data, checking error and so on.

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm38x_fuart.c, with /Libraries/TX03_Periph_Driver/inc/tmpm38x_fuart.h containing the macros, data types, structures and API definitions for use by applications.

Note: TMPM38x stands for TMPM381 and TMPM383.
tmpm38x stands for tmpm381 and tmpm383.

7.2 API Functions

7.2.1 Function List

- void FUART_Enable(TSB_UART_TypeDef * ***FUARTx***)
- void FUART_Disable(TSB_UART_TypeDef * ***FUARTx***)
- uint32_t FUART_GetRxData(TSB_UART_TypeDef * ***FUARTx***)
- void FUART_SetTxData(TSB_UART_TypeDef * ***FUARTx***, uint32_t ***Data***)
- FUART_Err FUART_GetErrStatus(TSB_UART_TypeDef * ***FUARTx***)
- void FUART_ClearErrStatus(TSB_UART_TypeDef * ***FUARTx***)
- WorkState FUART_GetBusyState(TSB_UART_TypeDef * ***FUARTx***)
- FUART_StorageStatus FUART_GetStorageStatus(TSB_UART_TypeDef * ***FUARTx***, FUART_Direction ***Direction***)
- void FUART_Init(TSB_UART_TypeDef * ***FUARTx***, FUART_InitTypeDef * ***InitStruct***)
- void FUART_EnableFIFO(TSB_UART_TypeDef * ***FUARTx***)
- void FUART_DisableFIFO(TSB_UART_TypeDef * ***FUARTx***)
- void FUART_SetSendBreak(TSB_UART_TypeDef * ***FUARTx***, FunctionalState ***NewState***)
- void FUART_SetINTFIFOLevel(TSB_UART_TypeDef * ***FUARTx***, uint32_t ***RxLevel***, uint32_t ***TxLevel***)
- void FUART_SetINTMask(TSB_UART_TypeDef * ***FUARTx***, uint32_t ***IntMaskSrc***)
- FUART_INTStatus FUART_GetINTMask(TSB_UART_TypeDef * ***FUARTx***)
- FUART_INTStatus FUART_GetRawINTStatus(TSB_UART_TypeDef * ***FUARTx***)
- FUART_INTStatus FUART_GetMaskedINTStatus(TSB_UART_TypeDef * ***FUARTx***)
- void FUART_ClearINT(TSB_UART_TypeDef * ***FUARTx***, FUART_INTStatus ***INTStatus***)
- void FUART_EnableLoopBack(TSB_UART_TypeDef * **FUARTx**)
- void FUART_DisableLoopBack(TSB_UART_TypeDef * **FUARTx**)
- void FUART_EnableDuty(TSB_UART_TypeDef * **FUARTx**)
- void FUART_DisableDuty(TSB_UART_TypeDef * **FUARTx**)
- void FUART_SetPeriodDetection(TSB_UART_TypeDef * **FUARTx**, uint32_t **PeriodDetection**)
- void FUART_SetTxTerminalMode(TSB_UART_TypeDef * **FUARTx**, uint32_t **TxTerminalMode**)

◆ void FUART_SetStartBitTerminal(TSB_UART_TypeDef * **FUARTx**,
uint32_t **StartBitTerminal**)

7.2.2 Detailed Description

Functions listed above can be divided into five parts:

1) Full UART Configuration and Initialization, common operation
FUART_Enable(), FUART_Disable(), FUART_Init(), FUART_GetRxData(), FUART_SetTxData(), FUART_GetErrStatus(), FUART_ClearErrStatus(), FUART_GetBusyState(), FUART_GetStorageStatus(), FUART_SetSendBreak()
2) Configure FIFO and DMA.
FUART_EnableFIFO(), FUART_DisableFIFO(), FUART_SetINTFIFOLevel(),
3) Configure interrupt, get interrupt status and clear interrupt.
FUART_SetINTMask(), FUART_GetINTMask(), FUART_GetRawINTStatus(), FUART_GetMaskedINTStatus(), FUART_ClearINT().
4) Configure operation for 50% duty cycle mode.
FUART_EnableLoopBack(), FUART_DisableLoopBack(), FUART_EnableDuty(), FUART_DisableDuty(), FUART_SetPeriodDetection(), FUART_SetTxTerminalMode(), FUART_SetStartBitTerminal().

7.2.3 Function Documentation

Note**: in all of the following APIs, parameter "TSB_UART_TypeDef* ***FUARTx" can be ***FUART0***.

7.2.3.1 FUART_Enable

Enable the specified Full UART channel.

Prototype:
void
FUART_Enable(TSB_UART_TypeDef * ***FUARTx***)

Parameters:
FUARTx: The specified Full UART channel.

Description:
This API will enable the specified Full UART channel selected by ***FUARTx***.

Return:
None

7.2.3.2 FUART_Disable

Disable the specified Full UART channel.

Prototype:
void
FUART_Disable(TSB_UART_TypeDef * ***FUARTx***)

Parameters:
FUARTx: The specified Full UART channel.

Description:
This API will disable the specified Full UART channel selected by ***FUARTx***.

Return:

None

7.2.3.3 FUART_GetRxData

Get received data from the specified Full UART channel.

Prototype:
uint32_t
FUART_GetRxData(TSB_UART_TypeDef * ***FUARTx***)

Parameters:
FUARTx: The specified Full UART channel.

Description:
This API will get the data received from the specified Full UART channel selected by ***FUARTx***. It is appropriate to call the function after **FUART_GetStorageStatus(FUARTx, FUART_RX)** returns **FUART_STORAGE_NORMAL** or **FUART_STORAGE_FULL**.

Return:
The data received from the specified Full UART channel

7.2.3.4 FUART_SetTxData

Set data to be sent and start transmitting via the specified Full UART channel.

Prototype:
void
FUART_SetTxData(TSB_UART_TypeDef * ***FUARTx***,
uint32_t ***Data***)

Parameters:
FUARTx: The specified Full UART channel.
Data: A frame to be sent, which can be 5-bit, 6-bit, 7-bit or 8-bit, depending on the initialization. The Data range is 0x00 to 0xFF.

Description:
This API will set data to be sent and start transmitting via the specified Full UART channel selected by ***FUARTx***.

Return:
None

7.2.3.5 FUART_GetErrStatus

Get receive error status.

Prototype:
FUART_Err
FUART_GetErrStatus(TSB_UART_TypeDef * ***FUARTx***)

Parameters:
FUARTx: The specified Full UART channel.

Description:

This API will get the error status after a data has been transferred, so this API must be executed after **FUART_GetRxData(FUARTx)**, only in this read sequence can the right error status information be got.

Return:
- **FUART_NO_ERR** means there is no error in the last transfer.
- **FUART_OVERRUN** means that overrun occurs in the last transfer.
- **FUART_PARITY_ERR** means either even parity or odd parity fails.
- **FUART_FRAMING_ERR** means there is framing error in the last transfer.
- **FUART_BREAK_ERR** means there is break error in the last transfer.
- **FUART_ERRS** means that 2 or more errors occurred in the last transfer.

7.2.3.6 FUART_ClearErrStatus

Clear receive error status.

Prototype:
void
FUART_ClearErrStatus(TSB_UART_TypeDef * ***FUARTx***)

Parameters:
FUARTx: The specified Full UART channel.

Description:
This API will clear all the receive errors, including framing, parity, break and overrun errors.

Return:
None

7.2.3.7 FUART_GetBusyState

Get the state that whether the specified Full UART channel is transmitting data or stopped.

Prototype:
WorkState
FUART_GetBusyState(TSB_UART_TypeDef * ***FUARTx***)

Parameters:
FUARTx: The specified Full UART channel.

Description:
This API will get the work state of the specified Full UART channel to see if it is transmitting data or stopped.

Return:
Work state of the specified Full UART channel:
BUSY: The Full UART is transmitting data
DONE: The Full UART has stopped transmitting data

7.2.3.8 FUART_GetStorageStatus

Get the FIFO or hold register status.

Prototype:
FUART_StorageStatus

FUART_GetStorageStatus(TSB_UART_TypeDef * ***FUARTx***,
FUART_Direction ***Direction***)

Parameters:
FUARTx: The specified Full UART channel.
Direction: The direction of Full UART
- **FUART_RX**: for receive FIFO or receive hold register
- **FUART_TX**: for transmit FIFO or transmit hold register

Description:
When FIFO is enabled, this API will get the transmit or receive FIFO status.
When FIFO is disabled, this API will get the transmit or receive hold register status.

Return:
- **FUART_StorageStatus**: The FIFO or hold register status.
- **FUART_STORAGE_EMPTY**: The FIFO or the hold register is empty.
- **FUART_STORAGE_NORMAL**: The FIFO is normal, not empty and not full.
- **FUART_STORAGE_INVALID**: The FIFO or the hold register is in invalid status.
- **FUART_STORAGE_FULL**: The FIFO or the hold register is full.

7.2.3.9 FUART_Init

Initialize and configure the specified Full UART channel.

Prototype:
void
FUART_Init(TSB_UART_TypeDef * ***FUARTx***,
FUART_InitTypeDef * ***InitStruct***)

Parameters:
FUARTx: The specified Full UART channel.
InitStruct: The structure containing Full UART configuration including baud rate, data bits per transfer, stop bits, parity, transfer mode and flow control
(Refer to “Data Structure Description” for details).

Description:
This API will initialize and configure the baud rate, the number of bits per transfer, stop bit, parity, and transfer mode and flow control for the specified Full UART channel selected by ***FUARTx***.
This API must be executed before Full UART is enabled.

Return:
None

7.2.3.10 FUART_EnableFIFO

Enable the transmit and receive FIFO.

Prototype:
void
FUART_EnableFIFO(TSB_UART_TypeDef * ***FUARTx***)

Parameters:
FUARTx: The specified Full UART channel.

Description:
This API will enable the transmit and receive FIFO of the specified UART channel selected by ***FUARTx***.

Return:
None

7.2.3.11 FUART_DisableFIFO

Disable the transmit and receive FIFO and the mode will be changed to character mode.

Prototype:
void
FUART_DisableFIFO(TSB_UART_TypeDef * ***FUARTx***)

Parameters:
FUARTx: The specified Full UART channel.

Description:
This API will disable the transmit and receive FIFO of the specified UART channel selected by ***FUARTx***. Then the Full UART work mode will be changed from FIFO mode to character mode.

Return:
None

7.2.3.12 FUART_SetSendBreak

Generate the break condition for Full UART.

Prototype:
void
FUART_SetSendBreak(TSB_UART_TypeDef * ***FUARTx***,
FunctionalState ***NewState***)

Parameters:
FUARTx: The specified Full UART channel.
NewState: New state of the FUART send break.
- **ENABLE**: Enable the send break to generate transmit break condition
- **DISABLE**: Disable the send break

Description:
This API is used to generate the transmit break condition. For generation of the transmit break condition, the send break function must be enabled by this API while at least one frame or longer being transmitted. Even when the break condition is generated, the contents of the transmit FIFO are not affected.

Return:
None

7.2.3.13 FUART_SetINTFIFOLevel

Set the Receive and Transmit interrupt FIFO level.

Prototype:
void

FUART_SetINTFIFOLevel(TSB_UART_TypeDef * ***FUARTx***,
uint32_t ***RxLevel***,
uint32_t ***TxLevel***)

Parameters:
FUARTx: The specified Full UART channel.
RxLevel: Receive interrupt FIFO level. (Receive FIFO is 32 location deep)
- **FUART_RX_FIFO_LEVEL_4**: The data in Receive FIFO become >= 4 words
- **FUART_RX_FIFO_LEVEL_8**: The data in Receive FIFO become >= 8 words
- **FUART_RX_FIFO_LEVEL_16**: The data in Receive FIFO become >= 16 words
- **FUART_RX_FIFO_LEVEL_24**: The data in Receive FIFO become >= 24 words
- **FUART_RX_FIFO_LEVEL_28**: The data in Receive FIFO become >= 28 words

TxLevel: Transmit interrupt FIFO level. (Transmit FIFO is 32 location deep)
- **FUART_TX_FIFO_LEVEL_4**: The data in Transmit FIFO become <= 4 words
- **FUART_TX_FIFO_LEVEL_8**: The data in Transmit FIFO become <= 8 words
- **FUART_TX_FIFO_LEVEL_16**: The data in Transmit FIFO become <= 16 words
- **FUART_TX_FIFO_LEVEL_24**: The data in Transmit FIFO become <= 24 words
- **FUART_TX_FIFO_LEVEL_28**: The data in Transmit FIFO become <= 28 words

Description:
This API is used to define the FIFO level at which UARTTXINTR and UARTRXINTR are generated. The interrupts are generated based on a transition through a level rather than based on the level.

Return:
None

7.2.3.14 FUART_SetINTMask

Mask(Enable) interrupt source of the specified channel.

Prototype:
void
FUART_SetINTMask(TSB_UART_TypeDef * ***FUARTx***,
uint32_t ***IntMaskSrc***)

Parameters:
FUARTx: The specified Full UART channel.
IntMaskSrc: The interrupt source to be masked(enabled).
To enable no interrupt, use the parameter:
- **FUART_NONE_INT_MASK**

To enable the interrupt one by one, use the "OR" operation with below parameter:
- **FUART_RX_FIFO_INT_MASK**: Enable receive FIFO interrupt
- **FUART_TX_FIFO_INT_MASK**: Enable transmit FIFO interrupt
- **FUART_RX_TIMEOUT_INT_MASK**: Enable receive timeout interrupt

- **FUART_FRAMING_ERR_INT_MASK**: Enable framing error interrupt
- **FUART_PARITY_ERR_INT_MASK**: Enable parity error interrupt
- **FUART_BREAK_ERR_INT_MASK**: Enable break error interrupt
- **FUART_OVERRUN_ERR_INT_MASK**: Enable overrun error interrupt

To enable all the interrupts, use the parameter:

- **FUART_ALL_INT_MASK**

Description:
This API will enable the interrupt source of the specified channel. With using this API, interrupts specified by ***IntMaskSrc*** will be enabled, the other interrupts will be disabled.

Return:
None

7.2.3.15 FUART_GetINTMask

Get the mask(Enable) setting for each interrupt source.

Prototype:
FUART_INTStatus
FUART_GetINTMask(TSB_UART_TypeDef * ***FUARTx***)

Parameters:
FUARTx: The specified Full UART channel.

Description:
This API will get the Full UART interrupt configuration. This API can get the information that which interrupts are enabled and which interrupts are disabled.

Return:
FUART_INTStatus: The union that indicates interrupt enable configuration.
(Refer to “Data Structure Description” for details).

7.2.3.16 FUART_GetRawINTStatus

Get the raw interrupt status of the specified Full UART channel.

Prototype:
FUART_INTStatus
FUART_GetRawINTStatus(TSB_UART_TypeDef * ***FUARTx***)

Parameters:
FUARTx: The specified Full UART channel.

Description:
This API will get the raw interrupt status of the specified Full UART channel specified by ***FUARTx***.

Return:
FUART_INTStatus: The union that indicates the raw interrupt status.
(Refer to “Data Structure Description” for details).

7.2.3.17 FUART_GetMaskedINTStatus

Get the masked interrupt status of the specified Full UART channel.

Prototype:
FUART_INTStatus
FUART_GetMaskedINTStatus(TSB_UART_TypeDef * ***FUARTx***)

Parameters:
FUARTx: The specified Full UART channel.

Description:
This API will get the masked interrupt status of the specified Full UART channel specified by ***FUARTx***.

Return:
FUART_INTStatus: The union that indicates the masked interrupt status.
(Refer to “Data Structure Description” for details).

7.2.3.18 FUART_ClearINT

Clear the interrupts of the specified Full UART channel.

Prototype:
void
FUART_ClearINT(TSB_UART_TypeDef * ***FUARTx***,
FUART_INTStatus ***INTStatus***)

Parameters:
FUARTx: The specified Full UART channel.
INTStatus: The union that indicates the interrupts to be cleared. When a bit of this parameter is set to 1, the associated interrupt is cleared.
(Refer to “Data Structure Description” for details).

Description:
This API can clear the interrupts of the specified channel selected by ***FUARTx***.

Return:
None

7.2.3.19 FUART_EnableLoopBack

Enable loop back test mode of the specified Full UART channel.

Prototype:
void
FUART_ EnableLoopBack (TSB_UART_TypeDef * ***FUARTx***)

Parameters:
FUARTx: The specified Full UART channel.

Description:
This API can enable loop back test mode of the specified Full UART channel.

Return:
None

7.2.3.20 FUART_DisableLoopBack

Disable loop back test mode of the specified Full UART channel.

Prototype:
void
FUART_DisableLoopBack (TSB_UART_TypeDef * ***FUARTx***)

Parameters:
FUARTx: The specified Full UART channel.

Description:
This API can disable loop back test mode of the specified Full UART channel.

Return:
None

7.2.3.21 FUART_EnableDuty

Enable 50% duty mode of the specified Full UART channel.

Prototype:
void
FUART_ EnableDuty (TSB_UART_TypeDef * ***FUARTx***)

Parameters:
FUARTx: The specified Full UART channel.

Description:
This API can enable 50% duty mode of the specified Full UART channel.

Return:
None

7.2.3.22 FUART_DisableDuty

Disable 50% duty mode of the specified Full UART channel.

Prototype:
void
FUART_DisableDuty (TSB_UART_TypeDef * ***FUARTx***)

Parameters:
FUARTx: The specified Full UART channel.

Description:
This API can disable 50% duty mode of the specified Full UART channel.

Return:
None

7.2.3.23 FUART_SetPeriodDetection

Set “0” period detection control of the specified Full UART channel.

Prototype:
void
FUART_SetPeriodDetection(TSB_UART_TypeDef * **FUARTx**,
uint32_t **PeriodDetection**)

Parameters:

FUARTx: The specified Full UART channel.
PeriodDetection: the width of “0” period detection.
- **FUART_PERIOD_DETEC_1**: 1/16 width more than "0" detection
- **FUART_PERIOD_DETEC_2**: 2/16 width more than "0" detection
- **FUART_PERIOD_DETEC_3**: 3/16 width more than "0" detection
- **FUART_PERIOD_DETEC_4**: 4/16 width more than "0" detection
- **FUART_PERIOD_DETEC_5**: 5/16 width more than "0" detection
- **FUART_PERIOD_DETEC_6**: 6/16 width more than "0" detection
- **FUART_PERIOD_DETEC_7**: 7/16 width more than "0" detection

Description:
This API can set “0” period detection control of the specified Full UART channel.

Return:
None

7.2.3.24 FUART_SetTxTerminalMode

Select transmission terminal mode of the specified Full UART channel.

Prototype:
void
FUART_SetTxTerminalMode(TSB_UART_TypeDef * **FUARTx**,
uint32_t **TxTerminalMode**)

Parameters:
FUARTx: The specified Full UART channel.
TxTerminalMode: Transmission terminal mode.
- **FUART_1_TERMINAL**: 1 terminal mode
- **FUART_2_TERMINAL**: 2 terminal mode

Description:
This API can select transmission terminal mode of the specified Full UART channel.

Return:
None

7.2.3.25 FUART_SetStartBitTerminal

Set Terminal selection to start the start bit of the specified Full UART channel.

Prototype:
void
FUART_SetStartBitTerminal(TSB_UART_TypeDef * **FUARTx**,
uint32_t **StartBitTerminal**)

Parameters:
FUARTx: The specified Full UART channel.
StartBitTerminal: Terminal selection to start the start bit.
- FUART_TXD50A: Select UTxTXD50A
- FUART_TXD50B: Select UTxTXD50B

Description:
This API can set Terminal selection to start the start bit of the specified Full UART channel.

Return:
None

7.2.4 Data Structure Description

7.2.4.1 FUART_InitTypeDef

Data Fields:

uint32_t
BaudRate configures the Full UART communication baud rate, it can't be 0(bsp) and must be smaller than 6250000(bps).

uint32_t
DataBits specifies data bits per transfer, which can be set as:
- **FUART_DATA_BITS_5** for 5-bit mode
- **FUART_DATA_BITS_6** for 6-bit mode
- **FUART_DATA_BITS_7** for 7-bit mode
- **FUART_DATA_BITS_8** for 8-bit mode

uint32_t
StopBits specifies the length of stop bit transmission, which can be set as:
- **FUART_STOP_BITS_1** for 1 stop bit
- **FUART_STOP_BITS_2** for 2 stop bits

uint32_t
Parity specifies the parity mode, which can be set as:
- **FUART_NO_PARITY** for no parity
- **FUART_0_PARITY** for 0 parity
- **FUART_1_PARITY** for 1 parity
- **FUART_EVEN_PARITY** for even parity
- **FUART_ODD_PARITY** for odd parity

uint32_t
Mode enables or disables reception, transmission or both, which can be set as:
- **FUART_ENABLE_TX** for enabling transmission
- **FUART_ENABLE_RX** for enabling reception
- **FUART_ENABLE_TX | FUART_ENABLE_RX** for enabling both reception and transmition

uint32_t
FlowCtrl Enable or disable the hardware flow control, which can be set as:
- **FUART_NONE_FLOW_CTRL** for no flow control

7.2.4.2 FUART_INTStatus

Data Fields:

uint32_t
All: Full UART interrupt status or mask.
Bit

uint32_t
Reserved1: 1 Reserved
uint32_t
Reserved2: 1 Reserved
uint32_t
Reserved3: 1 Reserved

uint32_t ***Reserved4***:	1	Reserved
uint32_t ***RxFIFO***:	1	Receive FIFO interrupt
uint32_t ***TxFIFO***:	1	Transmit FIFO interrupt
uint32_t ***RxTimeout***:	1	Receive timeout interrupt
uint32_t ***FramingErr***:	1	Framing error interrupt
uint32_t ***ParityErr***:	1	Parity error interrupt
uint32_t ***BreakErr***:	1	Break error interrupt
uint32_t ***OverrunErr***:	1	Overrun error interrupt
uint32_t ***Reserved***:	21	Reserved

8. GPIO

8.1 Overview

For TOSHIBA TMPM38x 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/tmpm38x_gpio.c, with /Libraries/TX03_Periph_Driver/inc/tmpm38x_gpio.h containing the macros, data types, structures and API definitions for use by applications.

8.2 API Functions

8.2.1 Function List

- uint8_t GPIO_ReadData(GPIO_Port **GPIO_x**);
- uint8_t GPIO_ReadDataBit(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***);
- void GPIO_WriteData(GPIO_Port ***GPIO_x***, uint8_t ***Data***);
- void GPIO_WriteDataBit(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***, uint8_t ***BitValue***);
- void GPIO_Init(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***,
  GPIO_InitTypeDef **** GPIO_InitStruct***);
- void GPIO_SetOutput(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***);
- void GPIO_SetInput(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***);
- void GPIO_SetOutputEnableReg(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***,
  FunctionalState ***NewState***);
- void GPIO_SetInputEnableReg(GPIO_Port **GPIO_x**, uint8_t **Bit_x**,
  FunctionalState ***NewState***);
- void GPIO_SetPullUp(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***,
  FunctionalState ***NewState***);
- void GPIO_SetPullDown(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***,
  FunctionalState ***NewState***);
- void GPIO_SetOpenDrain(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***,
  FunctionalState ***NewState***);
- void GPIO_EnableFuncReg(GPIO_Port ***GPIO_x***, uint8_t ***FuncReg*_x**, uint8_t ***Bit_x***);
- void GPIO_DisableFuncReg(GPIO_Port ***GPIO_x***, uint8_t ***FuncReg_x***, uint8_t ***Bit_x***);

8.2.2 Detailed Description

Functions listed above can be divided into three parts:

1) Write/Read GPIO or GPIO pin are handled by GPIO_ReadData(), GPIO_ReadDataBit(), GPIO_WriteData() and GPIO_WriteDataBit().
2) Initialize and configure the common functions of each GPIO port are handled by GPIO_SetOutput(), GPIO_SetInput(),GPIO_SetOutputEnableReg(), GPIO_SetInputEnableReg(), GPIO_SetPullUp(),GPIO_SetPullDown(), GPIO_SetOpenDrain() and GPIO_Init().
3) GPIO_EnableFuncReg() and GPIO_DisableFuncReg() handle other specified functions.

8.2.3 Function Documentation

8.2.3.1 GPIO_ReadData

Read specified GPIO Data register.

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

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D (PD for TMPM381 only)
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G (PG for TMPM381 only)
- **GPIO_PH:** GPIO port H.
- **GPIO_PI:** GPIO port I.
- **GPIO_PJ:** GPIO port J. (PJ for TMPM381 only)
- **GPIO_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N. (PN for TMPM381 only)
- **GPIO_PP:** GPIO port P.

Description:
This function will read specified GPIO Data register.

Return:
The value read from Data register.

8.2.3.2 GPIO_ReadDataBit

Read specified GPIO pin.

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

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D. (PD for TMPM381 only)
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G. (PG for TMPM381 only)
- **GPIO_PH:** GPIO port H.
- **GPIO_PI:** GPIO port I.
- **GPIO_PJ:** GPIO port J. (PJ for TMPM381 only)
- **GPIO_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N. (PN for TMPM381 only)

- **GPIO_PP:** GPIO port P.

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

Description:
This function will read specified GPIO pin.

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

8.2.3.3 GPIO_WriteData
Write specified value to GPIO Data register.

Prototype:
void
GPIO_WriteData(GPIO_Port ***GPIO_x***,
uint8_t ***Data***)

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D. (PD for TMPM381 only)
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G. (PG for TMPM381 only)
- **GPIO_PH:** GPIO port H.
- **GPIO_PI:** GPIO port I.
- **GPIO_PJ:** GPIO port J. (PJ for TMPM381 only)
- **GPIO_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N. (PN for TMPM381 only)
- **GPIO_PP:** GPIO port P.

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

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

Return:
None

8.2.3.4 GPIO_WriteDataBit
Write specified value of single bit to GPIO pin.

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

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D. (PD for TMPM381 only)
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G. (PG for TMPM381 only)
- **GPIO_PH:** GPIO port H.
- **GPIO_PI:** GPIO port I.
- **GPIO_PJ:** GPIO port J. (PJ for TMPM381 only)
- **GPIO_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N. (PN for TMPM381 only)
- **GPIO_PP:** GPIO port P.

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

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

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

Return:
None

8.2.3.5 GPIO_Init
Initialize GPIO port function.

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

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

- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D. (PD for TMPM381 only)
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G. (PG for TMPM381 only)
- **GPIO_PH:** GPIO port H.
- **GPIO_PI:** GPIO port I.
- **GPIO_PJ:** GPIO port J. (PJ for TMPM381 only)
- **GPIO_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N. (PN for TMPM381 only)
- **GPIO_PP:** GPIO port P.

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:** GPIO pin[7:0],
- Combination of the effective bits.

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

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

Return:
None

8.2.3.6 GPIO_SetOutput
Set specified GPIO pin as output port.

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

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D. (PD for TMPM381 only)
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.

- **GPIO_PG:** GPIO port G. (PG for TMPM381 only)
- **GPIO_PH:** GPIO port H.
- **GPIO_PI:** GPIO port I.
- **GPIO_PJ:** GPIO port J.   (PJ for TMPM381 only)
- **GPIO_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N. (PN for TMPM381 only)
- **GPIO_PP:** GPIO port P.

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:** GPIO pin[7:0],
- Combination of the effective bits.

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

Return:
None

8.2.3.7 GPIO_SetInput

Set specified GPIO Pin as input port.

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

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

- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D. (PD for TMPM381 only)
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G. (PG for TMPM381 only)
- **GPIO_PH:** GPIO port H.
- **GPIO_PI:** GPIO port I.
- **GPIO_PJ:** GPIO port J.   (PJ for TMPM381 only)
- **GPIO_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N. (PN for TMPM381 only)
- **GPIO_PP:** GPIO port P.

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:** GPIO pin[7:0],
- Combination of the effective bits.

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

Note:
For TMPM381:
To use the Port H/Port I/Port J as an analog input of the AD converter, disable input on PHIE/PIIE/PJIE and disable pull-up on PHPUP/PIPUP/PJPUP.

For TMPM383:
To use the Port H/Port I as an analog input of the AD converter, disable input on PHIE/PIIE and disable pull-up on PHPUP/PIPUP.

Return:
None

8.2.3.8 GPIO_SetOutputEnableReg

Enable or disable specified GPIO Pin output function.

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

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D. (PD for TMPM381 only)
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G. (PG for TMPM381 only)
- **GPIO_PH:** GPIO port H.
- **GPIO_PI:** GPIO port I.
- **GPIO_PJ:** GPIO port J. (PJ for TMPM381 only)
- **GPIO_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N. (PN for TMPM381 only)
- **GPIO_PP:** GPIO port P.

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:** GPIO pin[7:0],
- Combination of the effective bits.

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

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

Return:
None

8.2.3.9 GPIO_SetInputEnableReg
Enable or disable specified GPIO Pin input function.

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

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D. (PD for TMPM381 only)
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G. (PG for TMPM381 only)
- **GPIO_PH:** GPIO port H.
- **GPIO_PI:** GPIO port I.
- **GPIO_PJ:** GPIO port J. (PJ for TMPM381 only)
- **GPIO_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N. (PN for TMPM381 only)
- **GPIO_PP:** GPIO port P.

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:** GPIO pin[7:0],
- Combination of the effective bits.

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

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

Return:
None

8.2.3.10 GPIO_SetPullUp
Enable or disable specified GPIO Pin pull-up function.

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

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D. (PD for TMPM381 only)
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G. (PG for TMPM381 only)
- **GPIO_PH:** GPIO port H.
- **GPIO_PI:** GPIO port I.
- **GPIO_PJ:** GPIO port J. (PJ for TMPM381 only)
- **GPIO_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N. (PN for TMPM381 only)
- **GPIO_PP:** GPIO port P.

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:** GPIO pin[7:0],
- 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 pull-up function when ***NewState*** is **DISABLE**.

Return:
None

8.2.3.11 GPIO_SetPullDown

Enable or disable specified GPIO Pin pull-down function.

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

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D. (PD for TMPM381 only)
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G. (PG for TMPM381 only)
- **GPIO_PH:** GPIO port H.
- **GPIO_PI:** GPIO port I.
- **GPIO_PJ:** GPIO port J. (PJ for TMPM381 only)
- **GPIO_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N. (PN for TMPM381 only)
- **GPIO_PP:** GPIO port P.

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:** GPIO pin[7:0],
- 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 pull-down function when ***NewState*** is **DISABLE**.

Return:
None

8.2.3.12 GPIO_SetOpenDrain

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

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

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D. (PD for TMPM381 only)
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G. (PG for TMPM381 only)
- **GPIO_PH:** GPIO port H.
- **GPIO_PI:** GPIO port I.
- **GPIO_PJ:** GPIO port J. (PJ for TMPM381 only)
- **GPIO_PL:** GPIO port L.
- **GPIO_PM:** GPIO port M.
- **GPIO_PN:** GPIO port N. (PN for TMPM381 only)
- **GPIO_PP:** GPIO port P.

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:** GPIO pin[7:0],
- Combination of the effective bits.

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

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

Return:
None

8.2.3.13 GPIO_EnableFuncReg
Enable specified GPIO function.

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

Parameters:

GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D. (PD for TMPM381 only)
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PH:** GPIO port H.
- **GPIO_PJ:** GPIO port J. (PJ for TMPM381 only)
- **GPIO_PL:** GPIO port L.
- **GPIO_PN:** GPIO port N. (PN for TMPM381 only)

FuncReg_x: The number of GPIO function register, which can be set as:
- **GPIO_FUNC_REG_1** for GPIO function register 1,
- **GPIO_FUNC_REG_2** for GPIO function register 2,
- **GPIO_FUNC_REG_3** for GPIO function register 3,
- **GPIO_FUNC_REG_4** for GPIO function register 4,
- **GPIO_FUNC_REG_5** for GPIO function register 5

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

Description:
This function will enable GPIO pin specified function.

Return:
None

8.2.3.14 GPIO_DisableFuncReg
Disable specified GPIO function.

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

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D. (PD for TMPM381 only)
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PH:** GPIO port H.
- **GPIO_PJ:** GPIO port J. (PJ for TMPM381 only)
- **GPIO_PL:** GPIO port L.
- **GPIO_PN:** GPIO port N. (PN for TMPM381 only)

FuncReg_x: The number of GPIO function register, which can be set as:
- **GPIO_FUNC_REG_1** for GPIO function register 1,
- **GPIO_FUNC_REG_2** for GPIO function register 2,
- **GPIO_FUNC_REG_3** for GPIO function register 3,
- **GPIO_FUNC_REG_4** for GPIO function register 4,
- **GPIO_FUNC_REG_5** for GPIO function register 5.

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

Description:
This function will disable GPIO pin specified function.

Return:
None

8.2.4 Data Structure Description

8.2.4.1 GPIO_InitTypeDef

Data Fields:

uint8_t
IOMode Set specified GPIO Pin as input port or output port, which can be set as:

- **GPIO_INPUT:** Set GPIO pin as input port
- **GPIO_OUTPUT:** Set GPIO pin as output port

- **GPIO_IO_MODE_NONE:** Don't change GPIO pin I/O mode.

uint8_t
PullUp Enable or disable specified GPIO Pin pull-up function, which can be set as:

- **GPIO_PULLUP_ENABLE:** Enable specified GPIO pin pull-up function**.**
- **GPIO_PULLUP_DISABLE:** Disable specified GPIO pin pull-up function**.**
- **GPIO_PULLUP_NONE:** Don't have pull-up function or needn't change.

uint8_t
OpenDrain Set specified GPIO Pin as open drain port or CMOS port, which can be set as:
- **GPIO_OPEN_DRAIN_ENABLE:** Set specified GPIO pin as open drain port**.**
- **GPIO_OPEN_DRAIN_DISABLE:** Set specified GPIO pin as CMOS port**.**
- **GPIO_OPEN_DRAIN_NONE:** Don't have open-drain function or needn't change.

uint8_t
PullDown Enable or disable specified GPIO Pin pull-down function, which can be set as:
- **GPIO_PULLDOWN_ENABLE:** Enable specified GPIO pin pull-down function**.**
- **GPIO_PULLDOWN_DISABLE:** Disable specified GPIO pin pull-down function**.**
- **GPIO_PULLDOWN_NONE:** Don't have pull-down function or needn't change.

8.2.4.2 GPIO_RegTypeDef

Data Fields:

uint8_t
PinDATA Port x data register, port data read and write by this variable.

uint8_t
PinCR Port x output control register:
- **0**: Output disable**.**
- **1**: Output enable**.**

uint8_t
PinFR[FRMAX] Function setting register. You will be able to use the functions assigned by setting "1"

uint8_t
PinOD Port x open drain control register:
- **0**: CMOS.
- **1**: Open Drain.

uint8_t
PinPUP Port x pull-up control register:
- **0**: Pull-up disable**.**
- **1**: Pull-up enable**.**

uint8_t
PinPDN Port x pull-down control register:
- **0**: Pull-down disable**.**
- **1**: Pull-down enable**.**

uint8_t
PinIE Port x input control register:
- **0**: Input disable**.**
- **1**: Input enable**.**

8.2.4.3 TSB_Port_TypeDef
Data Fields:

__IO uint32_t
DATA The “DATA” can be read and written.

__IO uint32_t
CR The “CR” can be read and written.

__IO uint32_t
FR[FRMAX] The “FR[FRMAX]” can be read and written.

uint32_t
RESERVED0[RESER] Reserved.

__IO uint32_t
OD The “OD” can be read and written.

__IO uint32_t
PUP The “PUP” can be read and written.

__IO uint32_t
PDN The “PDN” can be read and written.

uint32_t
RESERVED1 Reserved.

__IO uint32_t
IE Port x input control register.

9. OFD

9.1 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/tmpm38x_ofd.c, with /Libraries/TX03_Periph_Driver/inc/tmpm38x_ofd.h containing the macros, data types, structures and API definitions for use by applications.

9.2 API Functions

9.2.1 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);

9.2.2 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.

9.2.3 Function Documentation

9.2.3.1 OFD_SetRegWriteMode

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

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

Parameters:
NewState is the new state of writing of OFDCR2/OFDMN/OFDMX/OFDRST registers. This parameter can be one of the following values:
ENABLE or **DISABLE**

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

Return:

None

9.2.3.2 OFD_Enable

Enable the OFD function.

Prototype:
void
OFD_Enable(void)

Parameters:
None.

Description:
This function will enable the OFD function.

Return:
None

9.2.3.3 OFD_Disable

Disable the OFD function.

Prototype:
void
OFD_Disable(void)

Parameters:
None.

Description:
This function will disable the OFD function.

Return:
None

9.2.3.4 OFD_SetDetectionFrequency

Set the count value of detection frequency.

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

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

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

Return:

None

9.2.3.5 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

9.2.3.6 OFD_GetStatus

Get the OFD busy and frequency error info.

Prototype:
OFD_Status
OFD_GetStatus(void)

Parameters:
None

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

Return:
OFD_Status is the structure of OFD status, which include the busy and frequency error info.
(Refer to "Data Structure Description" for details).

9.2.4 Data Structure Description

9.2.4.1 OFD_Status

Data Fields:

uint32_t
All: Data.
Bit
uint32_t
FrequencyError: 1 Frequency Error status
uint32_t
OFDBusy: 1 OFD Busy status

10. RMC

10.1 Overview

TOSHIBA TMPM38x has remote control signal preprocessor (here after referred to as RMC) receives a remote control signal of which carrier is removed.
TMPM38x has one RMC channel: RMC.

Reception of Remote Control Signal:

- A sampling clock can be selected from either low frequency clock (32.768 kHz) or Timer output.
- Noise canceller
- Leader detection
- Batch reception up to 72bit of data

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

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm38x_rmc.c, with /Libraries/TX03_Periph_Driver/inc/tmpm38x_rmc.h containing the macros, data types, structures and API definitions for use by applications

10.2 API Functions

10.2.1 Function List

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

10.2.2 Detailed Description

Functions listed above can be divided into three parts:

1) Reset and set RMC channel are handled by RMC_Enable(), RMC_Disable(), RMC_Init() and RMC_SetRxCtrl().
2) RMC basic functions are handled by RMC_SetLeaderDetection(), SetFallingEdgeINT(), RMC_SetSignalRxMethod(), RMC_SetRxTrg(), RMC_ SetThreshold(), RMC_SetInputSignalReversed(), RMC_SetNoiseCancellation(), RMC_SetRxEndBitNum() and RMC_SetSrcClk().
3) RMC_GetINTFactor(), RMC_GetLeader() and RMC_GetRxData() to get the receive status and save the received data from buffer.

10.2.3 Function Documentation

Note: In all of the following APIs, parameter "TSB_RMC_TypeDef * ***RMCx***" should be **TSB_RMC**

10.2.3.1 RMC_Enable

Enable the specified RMC channel.

Prototype:
void
RMC_Enable(TSB_RMC_TypeDef * ***RMCx***)

Parameters:
RMCx is the specified RMC channel.

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

Return:
None

10.2.3.2 RMC_Disable

Disable the function of specified RMC channel.

Prototype:
void
RMC_Disable(TSB_RMC_TypeDef * ***RMCx***)

Parameters:
RMCx is the specified RMC channel.

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

Return:
None

10.2.3.3 RMC_Init

RMC registers initial.

Prototype:

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

Parameters:
RMCx is the specified RMC channel.

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

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

Return:
None

10.2.3.4 RMC_SetRxCtrl

Enable or disable reception of the specified RMC channel.

Prototype:
void
RMC_SetRxCtrl(TSB_RMC_TypeDef * ***RMCx***,
FunctionalState ***NewState***)

Parameters:
RMCx is the specified RMC channel.

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

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

Return:
None

10.2.3.5 RMC_GetRxData

Get the received data from the specified RMC channel.

Prototype:
RMC_RxDataTypeDef
RMC_GetRxData(TSB_RMC_TypeDef * ***RMCx***)

Parameters:
RMCx is the specified RMC channel.

Description:
Get the received data from the specified RMC channel which selected by ***RMCx***.
This function reads the data from the RMCRBUF<0-71> and RMCRSTAT<RMCRNUM0-6> bits.

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

10.2.3.6 RMC_ SetLeaderDetection

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

Prototype:
void
RMC_SetLeaderDetection(TSB_RMC_TypeDef * ***RMCx***,
RMC_LeaderParameterTypeDef ***LeaderPara***)

Parameters:
RMCx is the specified RMC channel.

LeaderPara: The structure containing basic RMC leader detection configuration.

Data Fields:

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

uint8_t **MaxCycle:** Set <RMCLCMAX7:0> to specify a maximum cycle of leader detection. Calculating-formula of the maximum cycle: RMCLCMAX×4/fs[s].
RMC detects the first cycle as a leader if it is within the maximum cycle.

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

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

uint8_t **MinLowWidth:** Set <RMCLLMIN7:0> to specify a minimum low width of leader detection. Calculating-formula of the minimum low width: RMCLLMIN×4/fs[s]. RMC detects the first cycle as a leader if its low width exceeds the minimum low width. If RMCRCR2<RMCLD> = 1, a value less than the specified is determined as data.

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

Description:
This function will set the RMC leader detection configuration for specified RMC channel which selected by ***RMCx***.
This function handles the RMCRCR1 register and RMCRCR2<RMCLIEN> <RMCLD> two bits.
See MCU datasheet for detail.

Return:

None

10.2.3.7 RMC_SetFallingEdgeINT

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

Prototype:
void
RMC_SetFallingEdgeINT(TSB_RMC_TypeDef * ***RMCx***,
FunctionalState ***NewState***)

Parameters:
RMCx is the specified RMC channel.
NewState: New state for generation of a remote control input falling edge interrupt.
This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:
When ***NewState*** is **ENABLE**, this function will enable generation of a remote control input falling edge interrupt for specified RMC channel which selected by ***RMCx***, and disable generation of a remote control input falling edge interrupt when ***NewState*** is **DISABLE**.
This function handles the RMCRCR2<RMCEDIEN> bit.

Return:
None

10.2.3.8 RMC_SetSignalRxMethod

Select the method of receiving a remote control signal.

Prototype:
void
RMC_SetSignalRxMethod(TSB_RMC_TypeDef * ***RMCx***,
RMC_RxMethod ***Method***)

Parameters:
RMCx is the specified RMC channel.
Method: Select the RMC receive method, which can be one of the following values:
- **RMC_RX_IN_CYCLE_METHOD**: Receive a remote control signal in cycle method.
- **RMC_RX_IN_PHASE_METHOD**: Receive a remote control signal in phase method.

Description:
This function will set receiving method of remote control signal. Two methods can be selected by this function, cycle method or phase method.
This function handles the RMCRCR2<RMCPHM> bit.

Return:
None

10.2.3.9 RMC_SetRxTrg

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

Prototype:
void
RMC_SetRxTrg(TSB_RMC_TypeDef * ***RMCx***,
uint8_t ***LowWidth***,
uint8_t ***MaxDataBitCycle***)

Parameters:
RMCx is the specified RMC channel.
LowWidth: Excess low width that triggers reception completion and interrupt generation.
MaxDataBitCycle: Maximum data bit cycle that triggers reception completion and interrupt generation.

Description:
This function will set the trigger for specified RMC channel.
Set ***LowWidth*** to RMCRCR2<RMCLL7:0> specifies an excess low width. If an excess low width is detected, reception is completed and an interrupt is generated. The low width is not detected if <RMCLL7:0> = 11111111b.
Calculating formula of an excess low width: RMCLLx1/fs[s]

Set ***MaxDataBitCycle*** to RMCRCR2<RMCDMAX7:0> specifies a threshold for detecting a maximum data bit cycle. It is detected when a data bit cycle exceeds the threshold. It is not detected when <RMCMAX7:0> = 11111111b.
Calculating-formula of the threshold: RMCDMAX x 1/fs[s].

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

Return:
None

10.2.3.10 RMC_SetThreshold

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

Prototype:
void
RMC_SetThreshold(TSB_RMC_TypeDef * ***RMCx***,
uint8_t ***LargerThreshold***,
uint8_t ***SmallerThreshold***)

Parameters:
RMCx is the specified RMC channel.
LargerThreshold: Specifies a larger threshold (within a range of 1.5T and 2T) to determine a pattern of remote control signal in a phase method. If the measured cycle exceeds the threshold, the bit is determined as “10”. If not, the bit is determined as “01”. Calculating formula of the threshold: RMCDATHx1/fs[s].
The LargerThreshold should less than 0x80.
SmallerThreshold: Specifies two kinds of thresholds: a threshold to determine whether a data bit is 0 or 1; a smaller threshold (within a range of 1T and 1.5T) to determine a pattern of remote control signal in a phase method.

As for the determination of data bit, if the measured cycle exceeds the threshold, the bit is determined as “1”. If not, the bit is determined as “0”.
Calculating-formula of the threshold: RMCDATL×1/fs[s].
As for the determination of a remote control signal pattern in a phase method, if the measured cycle exceeds the threshold, the bit is determined as “01”. If not, the bit is determined as “00”. Calculating formula of the threshold to determine 0 or 1: RMCDATLx1/fs[s].
This function handles the RMCRCR3<RMCDATH0-6> <RMCDATL0-6> bits.
The SmallerThreshold should less than 0x80.

Description:
This function will set the parameters for thresholds in a phase method for the specified RMC channel which selected by ***RMCx***, to determine a signal pattern in phase mode and determine 0 or 1/ smaller threshold to determine a signal pattern in a phase method.
The thresholds settings are enabled only in phase method, when <RMCPHM> is “1”.

Return:
None

10.2.3.11 RMC_SetInputSignalReversed

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

Prototype:
void
RMC_SetInputSignalReversed(TSB_RMC_TypeDef * ***RMCx***,
FunctionalState ***NewState***)

Parameters:
RMCx is the specified RMC channel.
NewState is the new state of reversing input signal.
This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:
When ***NewState*** is **ENABLE**, this function will enable of reversing input signal for a specified RMC channel which is selected by ***RMCx***, and disable reversing input signal when ***NewState*** is **DISABLE**.
This function handles the RMCRCR4<RMCPO> bit.

Return:
None

10.2.3.12 RMC_SetNoiseCancellation

Set the noise cancellation time for the specified RMC channel.

Prototype:
void
RMC_SetNoiseCancellation(TSB_RMC_TypeDef * ***RMCx***,
uint8_t ***NoiseCancellationTime***)

Parameters:

RMCx is the specified RMC channel.
NoiseCancellationTime: The time for Noise cancellation, which should be less than 0x10.

Description:
Specifies time that noises is cancelled by a noise canceller.
If <RMCNC[3:0]> = 0000b, noises are not cancelled.
Calculating formula of noise cancellation time: RMCNC x 1/fs[s].
This function handles the RMCRCR4<RMCNC[3:0]> bits.

Return:
None

10.2.3.13 RMC_GetINTFactor

Get the interrupt factor for the specified RMC channel.

Prototype:
RMC_INTFactor
RMC_GetINTFactor(TSB_RMC_TypeDef * ***RMCx***)

Parameters:
RMCx is the specified RMC channel.

Description:
This function will get the interrupt factor for the specified RMC channel which selected by ***RMCx***. User can get the RMC receive interrupt status from this function.
This info is updated every time an interrupt is generated.
This function reads interrupt factor from RMCRSTAT<RMCRLIF>< RMCLOIF >< RMCDMAX >< RMCEDIF > bits.

Return:
RMC_INTFactor: Interrupt factor structure.
(Refer to “Data Structure Description” for details).

10.2.3.14 RMC_GetLeader

Get the leader detection result for the specified RMC channel.

Prototype:
RMC_LeaderDetection
RMC_GetLeader(TSB_RMC_TypeDef * ***RMCx***)

Parameters:
RMCx is the specified RMC channel.

Description:
This function will get the leader detection result for the specified RMC channel which selected by ***RMCx***.
This info is updated every time an interrupt is generated.
This function reads the leader detection status from the RMCRSTAT <RMCRLDR> bits.

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

RMC_LEADER_DETECTED: leader detected.
RMC_NO_LEADER: no leader detected.

10.2.3.15 RMC_SetRxEndBitNum

Specifies that the number of receive data bit.

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

Parameters:
RMCx is the specified RMC channel.
Reg_x: Select the set register, which can be one of:
- **RMC_RX_END_BITS_REG_1:** RMCxEND1 register
- **RMC_RX_END_BITS_REG_2:** RMCxEND2 register
- **RMC_RX_END_BITS_REG_3:** RMCxEND3 register

BitNum: Specifies that the number of receive data bit.

Description:
This function set the number of received data bit for the specified RMC channel, which selected by ***RMCx***.
This function sets the number of received data bit to the RMCxEND1, RMCxEND2, and RMCxEND3 registers.

Return:
None

10.2.3.16 RMC_SetSrcClk

Specifies that the sampling clock.

Prototype:
void
RMC_SetSrcClk(TSB_RMC_TypeDef * ***RMCx***,
RMC_SrcClk ***Clk***)

Parameters:
RMCx is the specified RMC channel.
Clk: RMC sampling clock, which can be one of:
- **RMC_CLK_LOW_FREQUENCY:** The Low Frequency Clock(32KHz)
- **RMC_CLK_TB1OUT:** Timer output (TB1OUT).

Description:
This function specifies that the sampling clock for the specified RMC channel, which selected by ***RMCx***.
This function sets the sampling clock type to the RMCxFSSEL <RMCCLK> bit.

Return:
None

10.2.4 Data Structure Description

10.2.4.1 RMC_RxDataDef

Data Fields:

uint8_t
***RxDataBits*:** The number of received data bit.

uint32_t
***RxBuf1*:** Received buffer 1，which reads 4 bytes data from <MCRBUF[31:0]>.

uint32_t
***RxBuf2*:** Received buffer 2，which reads 4 bytes data from <MCRBUF[63:32]>.

uint8_t
***RxBuf3*:** Received buffer 3, which reads 1 byte data from <MCRBUF[71:64]>

10.2.4.2 RMC_LeaderParameterTypeDef

Data Fields:

FunctionalState

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

uint8_t
***MaxCycle*:** Specifies a maximum cycle of leader detection.

uint8_t
***MinCycle*:** Specifies a minimum cycle of leader detection.

uint8_t
***MaxLowWidth*:** Specifies a maximum low width of leader detection.

uint8_t
***MinLowWidth*:** Specifies a minimum low width of leader detection.

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

10.2.4.3 RMC_InitTypeDef

Data Fields:

RMC_LeaderParameterTypeDef
***LeaderPara*:** Parameters to configure leader detection.

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

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

FunctionalState

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

uint8_t
***NoiseCancellationTime*:** Noise cancellation time.
The NoiseCancellationTime should less than 0x10.

uint8_t
***LowWidth*:** Excess low width that triggers reception completion and interrupt generation.

uint8_t
***MaxDataBitCycle*:** Maximum data bit cycle that triggers reception completion and interrupt generation.

uint8_t
***LargerThreshold*:** Larger threshold to determine a signal pattern in a phase method.
The LargerThreshold should less than 0x80.

uint8_t
***SmallerThreshold*:** Smaller threshold to determine a signal pattern in a phase method.
The SmallerThreshold should less than 0x80.

10.2.4.4 RMC_INTFactor

Data Fields:

uint32_t
All: Data.
Bit

uint32_t ***Reserved***:	12	Reserved
uint32_t ***InputFallingEdge***:	1	RMC input falling edge interrupt factor
uint32_t ***MaxDataBitCycle***:	1	Maximum data bit cycle interrupt factor
uint32_t ***LowWidthDetection***:	1	Low width detection interrupt factor
uint32_t ***LeaderDetection***:	1	Leader detection interrupt factor

11. RTC

11.1 Overview

The Real Time Clock (RTC) in the TMPM38x has such functions as follow:

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

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

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm38x_rtc.c, with /Libraries/ TX03_Periph_Driver/inc/tmpm38x_rtc.h containing the macros, data types, structures and API definitions for use by applications.

11.2 API Functions

11.2.1 Function List

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

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

11.2.2 Detailed Description

Functions listed above can be divided into five parts:

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

11.2.3 Function Documentation

11.2.3.1 RTC_SetSec

Set second value for RTC clock.

Prototype:
void
RTC_SetSec(uint8_t ***Sec***);

Parameters:
Sec: New second value, max is 59.

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

Return:
None.

11.2.3.2 RTC_GetSec

Get second value of RTC clock.

Prototype:

uint8_t
RTC_GetSec(void);

Parameters:
None

Description:
This function will return second value of RTC clock.

Return:
Second value in the range:
0 ~ 59

11.2.3.3 RTC_SetMin

Set minute value for RTC clock or alarm.

Prototype:
void
RTC_SetMin(RTC_FuncMode ***NewMode***,
uint8_t ***Min***);

Parameters:
NewMode: New mode of RTC, which can be set as:
- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

Min: New min value, max 59

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

Return:
None

11.2.3.4 RTC_GetMin

Get minute value of RTC clock or alarm.

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

Parameters:
NewMode: New mode of RTC, which can be set as:
- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

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

Return:
Minute value in the range:
0 ~ 59

11.2.3.5 RTC_GetAMPM

Get AM or PM state in the 12 Hour mode.

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

Parameters:
NewMode: New mode of RTC, which can be set as:
- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

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

Return:
The mode of time:
RTC_AM_MODE: Time mode is AM.
RTC_PM_MODE: Time mode is PM.

11.2.3.6 RTC_SetHour24

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

Prototype:
void
RTC_SetHour24(RTC_FuncMode ***NewMode***,
uint8_t ***Hour***);

Parameters:
NewMode: New mode of RTC, which can be set as:
- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

Hour: New hour value, max is 23.

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

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

Return:
None

11.2.3.7 RTC_SetHour12

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

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

Parameters:
NewMode: New mode of RTC, which can be set as:
- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

Hour: New hour value, max is 11.
AmPm: New time mode, which can bet set as:
- **RTC_AM_MODE**: select AM mode for 12H mode,
- **RTC_PM_MODE**: select PM mode for 12H mode.

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

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

Return:
None

11.2.3.8 RTC_GetHour

Get hour value of RTC clock or alarm.

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

Parameters:
NewMode: New mode of RTC, which can be set as:
- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

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

Return:
In 24H mode, hour value in the range:
0 ~ 23
In 12H mode, hour value in the range:
0 ~ 11

11.2.3.9 RTC_SetDay

Set day value for RTC clock or alarm.

Prototype:
void
RTC_SetDay(RTC_FuncMode ***NewMode***,
uint8_t ***Day***);

Parameters:
NewMode: New mode of RTC, which can be set as:
- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

Day: New day value, which can be set as:
- **RTC_SUN:** Sunday.
- **RTC_MON:** Monday.
- **RTC_TUE:** Tuesday.
- **RTC_WED:** Wednesday.
- **RTC_THU:** Thursday.
- **RTC_FRI:** Friday.
- **RTC_SAT:** Saturday.

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

Return:
None

11.2.3.10 RTC_GetDay

Get day value of RTC clock or alarm.

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

Parameters:
NewMode: New mode of RTC, which can be set as:
- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

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

Return:
Day value in the range:
0 ~ 6

11.2.3.11 RTC_SetDate

Set date value for RTC clock or alarm.

Prototype:
void
RTC_SetDate(RTC_FuncMode ***NewMode***,
uint8_t ***Date***);

Parameters:
NewMode: New mode of RTC, which can be set as:
- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

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

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

Return:
None

11.2.3.12 RTC_GetDate

Get date value of RTC clock or alarm.

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

Parameters:
NewMode: New mode of RTC, which can be set as:
- **RTC_CLOCK_MODE**: select clock function,
- **RTC_ALARM_MODE**: select alarm function.

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

Return:
Date value in the range:
1 ~ 31

11.2.3.13 RTC_SetMonth

Set month value for RTC clock.

Prototype:
void
RTC_SetMonth(uint8_t ***Month***);

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

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

Return:
None

11.2.3.14 RTC_GetMonth

Get month value of RTC clock.

Prototype:
uint8_t
RTC_GetMonth(void);

Parameters:
None

Description:
This function will return month value.

Return:
Month value in the range:
1 ~ 12

11.2.3.15 RTC_SetYear

Set year value for RTC clock.

Prototype:
void
RTC_SetYear(uint8_t ***Year***);

Parameters:
Year: New year value, max is 99.

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

Return:
None

11.2.3.16 RTC_GetYear

Get year value of RTC clock.

Prototype:

uint8_t
RTC_GetYear(void);

Parameters:
None

Description:
This function will return year value.

Return:
Year value in the range:
0 ~ 99

11.2.3.17 RTC_SetHourMode

Select 24-hour clock or 12-hour clock.

Prototype:
void
RTC_SetHourMode(uint8_t ***HourMode***);

Parameters:
HourMode: New mode of hour, which can be set as:
- **RTC_12_HOUR_MODE** : Select 12H mode,
- **RTC_24_HOUR_MODE.**: Select 24H mode.

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

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

Return:
None

11.2.3.18 RTC_GetHourMode

Get hour mode.

Prototype:
uint8_t
RTC_GetHourMode(void);

Parameters:
None

Description:
This function will return hour mode.

Return:
Hour mode:
RTC_24_HOUR_MODE: Hour mode is 24H mode.
RTC_12_HOUR_MODE: Hour mode is 12H mode.

11.2.3.19 RTC_SetLeapYear

Set leap year state.

Prototype:
void
RTC_SetLeapYear(uint8_t ***LeapYear***);

Parameters:
LeapYear: The state of leap year, which can be set as:
- **RTC_LEAP_YEAR_0**: Current year is a leap year.
- **RTC_LEAP_YEAR_1**: Current year is the year following a leap year.
- **RTC_LEAP_YEAR_2**: Current year is two years after a leap year.
- **RTC_LEAP_YEAR_3**: Current year is three years after a leap year.

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

Return:
None

11.2.3.20 RTC_GetLeapYear

Get leap year state.

Prototype:
uint8_t
RTC_GetLeapYear(void);

Parameters:
None

Description:
This function will return leap year state.

Return:
The state of the leap year.

11.2.3.21 RTC_SetTimeAdjustReq

Set time adjustment + or – 30 seconds.

Prototype:
void
RTC_SetTimeAdjustReq(void);

Parameters:
None

Description:

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

Return:
None

11.2.3.22 RTC_GetTimeAdjustReq

Get time adjust request state.

Prototype:
RTC_ReqState
RTC_GetTimeAdjustReq(void);

Parameters:
None

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

Return:
The state of time adjustment:
RTC_NO_REQ: No adjust request.
RTC_REQ: Adjust request.

11.2.3.23 RTC_EnableClock

Enable RTC clock function.

Prototype:
void
RTC_EnableClock(void);

Parameters:
None

Description:
This function will enable clock function.

Return:
None

11.2.3.24 RTC_DisableClock

Disable RTC clock function.

Prototype:
void
RTC_DisableClock(void);

Parameters:

None

Description:
This function will disable clock function.

Return:
None

11.2.3.25 RTC_EnableAlarm

Enable RTC alarm function.

Prototype:
void
RTC_EnableAlarm(void);

Parameters:
None

Description:
This function will enable alarm function.

Return:
None

11.2.3.26 RTC_DisableAlarm

Disable RTC alarm function.

Prototype:
void
RTC_DisableAlarm(void);

Parameters:
None

Description:
This function will disable alarm function.

Return:
None

11.2.3.27 RTC_SetRTCINT

Enable or disable INTRTC.

Prototype:
void
RTC_SetRTCINT(FunctionalState ***NewState***);

Parameters:
NewState: New state of INT RTC.
- **ENABLE**: Enable INTRTC.
- **DISABLE**: Disable INTRTC.

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

Return:
None

11.2.3.28 RTC_SetAlarmOutput

Set output signals from ALARM pin.

Prototype:
void
RTC_SetAlarmOutput(uint8_t ***Output***);

Parameters:
Output: Set ALARM pin output, which can be set as:
- **RTC_LOW_LEVEL**: “0” pulse
- **RTC_PULSE_1_HZ**: 1Hz cycle “0” pulse
- **RTC_PULSE_16_HZ**: 16Hz cycle “0” pulse
- **RTC_PULSE_2_HZ**: 2Hz cycle “0” pulse
- **RTC_PULSE_4_HZ**: 4Hz cycle “0” pulse
- **RTC_PULSE_8_HZ**: 8Hz cycle “0” pulse

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

Return:
None

11.2.3.29 RTC_ResetClockSec

Reset RTC clock second counter.

Prototype:
void
RTC_ResetClockSec(void);

Parameters:
None

Description:
This function will reset sec counter.

Return:
None

11.2.3.30 RTC_GetResetClockSecReq

Get reset RTC clock second counter request state.

Prototype:
RTC_ReqState
RTC_GetResetClockSecReq(void);

Parameters:
None

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

Return:
The state of reset clock request:
RTC_NO_REQ: No reset clock request.
RTC_REQ: Reset clock request.

11.2.3.31 RTC_ResetAlarm

Reset RTC alarm.

Prototype:
void
RTC_ResetAlarm(void);

Parameters:
None

Description:
This function will reset alarm.
Reset alarm registers, the related parameters will be set as follows.
Minute: 00, Hour: 00, Date: 01, Day of the week: Sunday

Return:
None

11.2.3.32 RTC_SetDateValue

Set the RTC clock date.

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

Parameters:

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

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

Return:
None

11.2.3.33 RTC_GetDateValue

Get the RTC clock date.

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

Parameters:

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

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

Return:
None

11.2.3.34 RTC_SetTimeValue

Set the RTC clock time.

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

Parameters:

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

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

Return:
None

11.2.3.35 RTC_GetTimeValue

Get the RTC time.

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

Parameters:

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

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

Return:
None

11.2.3.36 RTC_SetClockValue

Set the RTC clock date and time.

Prototype:
void
RTC_SetClockValue(RTC_DateTypeDef * ***DateStruct***,
RTC_TimeTypeDef * ***TimeStruct***);

Parameters:

DateStruct: The structure containing basic Date configuration including leap year state, year, month, date and day.
TimeStruct: The structure containing basic Time configuration including hour mode, hour, AM/PM mode in 12H mode, minute and second. (Refer to "Data structure Description" for details)

Description:
This function will set RTC clock date and time, including leap year, year, month, date, day, hour mode, hour, AM/PM mode in 12H mode, minute and second. **RTC_SetLeapYear()**, **RTC_SetYear()**, **RTC_SetMonth()**, **RTC_SetDate()**, **RTC_SetDay()**, **RTC_SetHourMode()**, **RTC_SetHour24()**, **RTC_SetHour12()**, **RTC_SetMin()** and **RTC_SetSec()** will be called by it.

Return:
None

11.2.3.37 RTC_GetClockValue

Get the RTC clock date and time.

Prototype:
void
RTC_GetClockValue(RTC_DateTypeDef * ***DateStruct***,
RTC_TimeTypeDef * ***TimeStruct***);

Parameters:

DateStruct: The structure containing basic Date configuration including leap year state, year, month, date and day.
TimeStruct: The structure containing basic Time configuration including hour mode, hour, AM/PM mode in 12H mode, minute and second. (Refer to "Data structure Description" for details)

Description:
This function will get RTC clock date and time, including leap year, year, month, date, day, hour mode, hour, AM/PM mode in 12H mode, minute and second. **RTC_GetLeapYear()**, **RTC_GetYear()**, **RTC_GetMonth()**, **RTC_GetDate()**, **RTC_GetDay()**, **RTC_GetHourMode()**, **RTC_GetHour()**,**RTC_GetAMPM()**, **RTC_GetMin()** and **RTC_GetSec()** will be called by it.

Return:
None

11.2.3.38 RTC_SetAlarmValue

Set the RTC alarm date and time.

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

Parameters:

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

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

Return:
None

11.2.3.39 RTC_GetAlarmValue

Get the RTC alarm date and time.

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

Parameters:

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

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

Return:
None

11.2.4 Data Structure Description

11.2.4.1 RTC_DateTypeDef

Data Fields:

uint8_t
LeapYear set leap year state, which can be set as:

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

uint8_t
Year new year value, max is 99.

uint8_t
Month new month value, ranging from 1 to 12.

uint8_t
Date new date value, ranging from 1 to 31.

uint8_t
Day new day value, which can be set as:

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

11.2.4.2 RTC_TimeTypeDef

Data Fields:

uint8_t
HourMode select 24H mode or 12H mode, which can be set as:

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

uint8_t
Hour new hour value, max value is 23 in 24H mode or 11 in 12H mode.

uint8_t
AmPm select AM/PM mode for 12H mode, which can be set as:

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

uint8_t
Min new minute value, max is 59.

uint8_t
Sec new second value, max is 59.

11.2.4.3 RTC_AlarmTypeDef

Data Fields:

uint8_t
Date new date value of RTC alarm, ranging from 1 to 31.

uint8_t
Day new day value of RTC alarm, which can be set as:

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

uint8_t
Hour new hour value of RTC alarm, max value is 23 in 24H mode, max value is 11 in 12H mode.

uint8_t
AmPm select AM/PM mode for 12H mode, which can be set as:

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

uint8_t
Min new minute value of RTC alarm, max is 59.

12. SBI

12.1 Overview

This device contains one Serial Bus Interface channel. The 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/tmpm38x_sbi.c, with /Libraries/TX03_Periph_Driver/inc/tmpm38x_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 SBI channel are handled by SBI_Enable(), SBI_Disable(), SBI_SetI2CACK(), SBI_SetI2CBitNum(), and SBI_InitI2C().
2) Transfer control of SBI channel is handled by SBI_ClearI2CINTReq(), SBI_GenerateI2Cstart(), SBI_GenerateI2Cstop(), SBI_GetReceiveData().
3) The status indication of 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***” should be **TSB_SBI0**.

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 "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 mode.

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 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 TMPM38x contains SSP (Synchronous Serial Port) module with one channel SSP0.

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

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

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm38x_ssp.c, with /Libraries/TX03_Periph_Driver/inc/tmpm38x_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(TSB_SSP_TypeDef * ***SSPx***);
- void SSP_Disable(TSB_SSP_TypeDef * ***SSPx***);
- void SSP_Init(TSB_SSP_TypeDef * ***SSPx***, SSP_InitTypeDef * ***InitStruct***);
- void SSP_SetClkPreScale(TSB_SSP_TypeDef * ***SSPx***,
  uint8_t ***PreScale***, uint8_t ***ClkRate***);
- void SSP_SetFrameFormat(TSB_SSP_TypeDef * ***SSPx***,
  SSP_FrameFormat ***FrameFormat***);
- void SSP_SetClkPolarity(TSB_SSP_TypeDef * ***SSPx***,
  SSP_ClkPolarity ***ClkPolarity***);
- void SSP_SetClkPhase(TSB_SSP_TypeDef * ***SSPx***, SSP_ClkPhase ***ClkPhase***);
- void SSP_SetDataSize(TSB_SSP_TypeDef * ***SSPx***, uint8_t ***DataSize***);
- void SSP_SetSlaveOutputCtrl(TSB_SSP_TypeDef * ***SSPx***,
  FunctionalState ***NewState***);
- void SSP_SetMSMode(TSB_SSP_TypeDef * ***SSPx***, SSP_MS_Mode ***Mode***);
- void SSP_SetLoopBackMode(TSB_SSP_TypeDef * ***SSPx***,
  FunctionalState ***NewState***);
- void SSP_SetTxData(TSB_SSP_TypeDef * ***SSPx***, uint16_t ***Data***);
- uint16_t SSP_GetRxData(TSB_SSP_TypeDef * ***SSPx***);
- WorkState SSP_GetWorkState(TSB_SSP_TypeDef * ***SSPx***);
- SSP_FIFOState SSP_GetFIFOState(TSB_SSP_TypeDef * ***SSPx***,
  SSP_Direction ***Direction***);
- void SSP_SetINTConfig(TSB_SSP_TypeDef * ***SSPx***, uint32_t ***IntSrc***);
- SSP_INTState SSP_GetINTConfig(TSB_SSP_TypeDef * ***SSPx***);
- SSP_INTState SSP_GetPreEnableINTState(TSB_SSP_TypeDef * ***SSPx***);
- SSP_INTState SSP_GetPostEnableINTState(TSB_SSP_TypeDef * ***SSPx***);
- void SSP_ClearINTFlag(TSB_SSP_TypeDef * ***SSPx***, uint32_t ***IntSrc***);

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() and SSP_Disable().
6) SSP_SetSlaveOutputCtrl() and SSP_SetLoopBackMode() handle other specified functions.

13.2.3 Function Documentation

Note: in all of the following APIs, parameter "TSB_SSP_TypeDef* ***SSPx***" should be **TSB_SSP0**.

13.2.3.1 SSP_Enable

Enable the specified SSP channel.

Prototype:
void
SSP_Enable(TSB_SSP_TypeDef * ***SSPx***)

Parameters:
SSPx: Select the SSP channel.

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

Return:
None

13.2.3.2 SSP_Disable

Disable the specified SSP channel.

Prototype:
void
SSP_ Disable(TSB_SSP_TypeDef * ***SSPx***)

Parameters:
SSPx: Select the SSP channel.

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

Return:
None

13.2.3.3 SSP_Init

Initialize the specified SSP channel through the data in structure SSP_InitTypeDef.

Prototype:
void
SSP_Init(TSB_SSP_TypeDef * ***SSPx***,
    SSP_InitTypeDef* ***InitStruct***)

Parameters:
SSPx: Select the SSP channel.
InitStruct: It is a structure with detail as below:

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

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

Description:
This function will configure the SSP channel by ***SSPx*** and SSP_InitTypeDef ***InitStruct***.
It will call the functions below:
SSP_SetFrameFormat(),
SSP_SetClkPreScale(),
SSP_SetClkPolarity(),
SSP_SetClkPhase(),
SSP_SetDataSize(),
SSP_SetMSMode().

Return:
None

13.2.3.4 SSP_SetClkPreScale

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

Prototype:
void
SSP_SetClkPreScale(TSB_SSP_TypeDef * ***SSPx***,
    uint8_t ***PreScale***,
    uint8_t ***ClkRate***)

Parameters:
SSPx: Select the SSP channel.
PreScale: Clock prescale divider, must be even number from 2 to 254.
ClkRate: Serial clock rate (from 0 to 255).

Description:

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

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

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 specified SSP channel.

Prototype:
void
SSP_SetFrameFormat(TSB_SSP_TypeDef * ***SSPx***,
SSP_FrameFormat ***FrameFormat***)

Parameters:
SSPx: Select the SSP channel.
FrameFormat: Frame format of SSP which can be:

- **SSP_FORMAT_SPI:** Configure SSP module to SPI mode.
- **SSP_FORMAT_SSI:** Configure SSP module to SSI mode.
- **SSP_FORMAT_MICROWIRE:** Configure SSP module to Microwire mode.

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

Return:
None

13.2.3.6 SSP_SetClkPolarity

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

Prototype:
void
SSP_SetClkPolarity(TSB_SSP_TypeDef * ***SSPx***,
SSP_ClkPolarity ***ClkPolarity***)

Parameters:
SSPx: Select the SSP channel.
ClkPolarity: SPI clock polarity
This parameter can be one of the following values:

- **SSP_POLARITY_LOW:** SCLK pin is low level in idle state.
- **SSP_POLARITY_HIGH:** SCLK pin is high level in idle state.

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

Return:
None

13.2.3.7 SSP_SetClkPhase

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

Prototype:
void
SSP_SetClkPhase(TSB_SSP_TypeDef * ***SSPx***,
SSP_ClkPhase ***ClkPhase***)

Parameters:
SSPx: Select the SSP channel.
ClkPhase: SPI clock phase
This parameter can be one of the following values:

- **SSP_PHASE_FIRST_EDGE:** Capture data in first edge of SCLK pin.
- **SSP_PHASE_SECOND_EDGE:** Capture data in second edge of SCLK pin.

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

Return:
None

13.2.3.8 SSP_SetDataSize

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

Prototype:
Void
SSP_SetDataSize(TSB_SSP_TypeDef * ***SSPx***,
uint8_t ***DataSize***)

Parameters:
SSPx: Select the SSP channel.
DataSize: Data size select from 4 to 16.

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

Return:
None

13.2.3.9 SSP_SetSlaveOutputCtrl

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

Prototype:
void
SSP_SetSlaveOutputCtrl(TSB_SSP_TypeDef * ***SSPx***,
FunctionalState ***NewState***)

Parameters:
SSPx: Select the SSP channel.
NewState: Specifies the state of the SPDO output when SSP is set in slave mode, This parameter can be one of the following values:

- **ENABLE:** Enable the SPDO output.
- **DISABLE:** Disable the SPDO output.

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

Return:
None

13.2.3.10 SSP_SetMSMode

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

Prototype:
void
SSP_SetMSMode(TSB_SSP_TypeDef * ***SSPx***,
SSP_MS_Mode ***Mode***)

Parameters:
SSPx: Select the SSP channel.
Mode: Select the SSP mode
This parameter can be one of the following values:

- **SSP_MASTER:** SSP run in master mode.
- **SSP_SLAVE:** SSP run in slave mode.

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

Return:
None

13.2.3.11 SSP_SetLoopBackMode

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

Prototype:
void
SSP_SetLoopBackMode(TSB_SSP_TypeDef * ***SSPx***,
FunctionalState ***NewState***)

Parameters:
SSPx: Select the SSP channel.
NewState: Specifies the state for self-loop back of SSP.
This parameter can be one of the following values:

- **ENABLE**: Enable the self-loop back mode.
- **DISABLE**: Disable the self-loop back mode.

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

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

Return:
None

13.2.3.12 SSP_SetTxData

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

Prototype:
void
SSP_SetTxData(TSB_SSP_TypeDef * ***SSPx***,
uint16_t ***Data***)

Parameters:
SSPx: Select the SSP channel.
Data: 4~16bit data to be sent.

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

Return:
None

13.2.3.13 SSP_GetRxData

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

Prototype:
uint16_t
SSP_GetRxData(TSB_SSP_TypeDef * ***SSPx***)

Parameters:
SSPx: Select the SSP channel.

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

Return:
Data with uint16_t type

13.2.3.14 SSP_GetWorkState

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

Prototype:
WorkState
SSP_GetWorkState(TSB_SSP_TypeDef * ***SSPx***)

Parameters:
SSPx: Select the SSP channel.

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

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

13.2.3.15 SSP_GetFIFOState

Get the Rx/Tx FIFO state of the specified SSP channel.

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

Parameters:
SSPx: Select the SSP channel.
Direction: The direction which means transmit or receive
This parameter can be one of the following values:

- **SSP_RX**: Target is to check state of receive FIFO.
- **SSP_TX**: Target is to check state of transmit FIFO.

Description:
This function will specify SSP channel by ***SSPx,*** get the Rx/Tx FIFO state by ***Direction***.
For example, data can be sent after judging Tx FIFO is available by the code below:

```
SSP_FIFOState  fifoState;

fifoState = SSP_GetFIFOState(TSB_SSP0, SSP_TX);
if ((fifoState == SSP_FIFO_EMPTY) || (fifoState == SSP_FIFO_NORMAL))
    { SSP_SetTxData(TSB_SSP0, data_to_be_sent ); }
```

Return:
The state of SSP FIFO, which can be
SSP_FIFO_EMPTY: FIFO is empty.
SSP_FIFO_NORMAL: FIFO is not full and not empty.
SSP_FIFO_INVALID: FIFO is invalid state.
SSP_FIFO_FULL: FIFO is full

13.2.3.16 SSP_SetINTConfig

Enable/Disable interrupt source of the specified SSP channel

Prototype:
void
SSP_SetINTConfig(TSB_SSP_TypeDef * ***SSPx***,
uint32_t ***IntSrc***)

Parameters:
SSPx: Select the SSP channel.
IntSrc: The interrupt source for SSP to be enabled or disabled.
To disable all interrupt sources, use the parameter:

- **SSP_INTCFG_NONE**
- **SSP_INTFG_ALL**

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 interrupts above together, use the parameter:

- **SSP_INTCFG_ALL**

Description:
This function will specify SSP channel by ***SSPx***, enable/disable interrupts by ***IntSrc***.
For example, we can enable Tx and Rx interrupt by code like below:
SSP_SetINTConfig(TSB_SSP0, 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 specified SSP channel.

Prototype:
SSP_INTState
SSP_GetINTConfig(TSB_SSP_TypeDef * ***SSPx***)

Parameters:
SSPx: Select the SSP channel.

Description:
This function will get the masked interrupt status of the specified SSP channel by ***SSPx***.
For example, it can be used to check which interrupt source is enabled or disabled by SSP_SetINTConfig().

Return:
SSP_INTState type. It contains the state of SSP interrupt setting, for more detail refer to the description for union SSP_INTState in "Data Structure Description" part.

13.2.3.18 SSP_GetPreEnableINTState

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

Prototype:
SSP_INTState
SSP_GetPreEnableINTState(TSB_SSP_TypeDef * ***SSPx***)

Parameters:
SSPx: Select the SSP channel.

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

Return:
SSP_INTState type. It contains the pre-enable interrupt status (raw status before masked), for more detail refer to the description for union SSP_INTState in "Data Structure Description" part.

13.2.3.19 SSP_GetPostEnableINTState

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

Prototype:
SSP_INTState
SSP_GetPostEnableINTState(TSB_SSP_TypeDef * ***SSPx***)

Parameters:
SSPx: Select the SSP channel.

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

Return:
SSP_INTState type. It contains the post-enable interrupt status (after masked), for more detail refer to the description for union SSP_INTState in "Data Structure Description" part.

13.2.3.20 SSP_ClearINTFlag

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

Prototype:
void
SSP_ClearINTFlag(TSB_SSP_TypeDef * ***SSPx***,
uint32_t ***IntSrc***)

Parameters:
SSPx: Select the SSP channel.
IntSrc: The interrupt source to be cleared.
This parameter can be one of the following values:

- **SSP_INTCFG_RX_OVERRUN:** Receive overrun interrupt.
- **SSP_INTCFG_RX_TIMEOUT:** Receive timeout interrupt.
- **SSP_INTCFG_ALL:** All the 2 interrupt above together

Description:
This function will clear interrupt flag by ***IntSrc*** of the specified SSP channel by ***SSPx***.

Return:
None

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, wich 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.

uint8_t
ClkRate Serial clock rate, from 0 to 255.

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 TMPM38x contains 8 channels of multi-functional 16-bit timer/event counter (TMRB0 through TMRB7). Each channel can operate in the following modes:

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

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

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

The TMRB driver APIs provide a set of functions to configure each channel, such as setting the clock division, trailing timing and leading timing duration, capture timing and flip-flop function. And to control the running state of each channel such as controlling up-counter, the output of flip-flop and to indicate the status of each channel such as returning the factor of interrupt, value in capture registers and so on.

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm38x_tmrb.c, with /Libraries/TX03_Periph_Driver/inc/tmpm38x_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***, uint8_t ***WriteRegMode***);
- void TMRB_SetExtStartTrg(TSB_TB_TypeDef * ***TBx***, FunctionalState ***NewState***, uint8_t ***TrgMode***);

◆ void TMRB_SetClkInCoreHalt(TSB_TB_TypeDef * ***TBx***, uint8_t ***ClkState***);

14.2.2 Detailed Description

Functions listed above can be divided into four parts:

1) Configure and control the common functions of each TMRB channel are handled by TMRB_Enable(), TMRB_Disable(), TMRB_Init(), TMRB_SetRunState(), TMRB_ChangeLeadingTiming() and TMRB_ChangeTrailingTiming().
2) Capture function of each TMRB channel is handled by TMRB_SetCaptureTiming(), and TMRB_ExecuteSWCapture().
3) The status indication of each TMRB channel is handled by TMRB_GetINTFactor(), TMRB_GetUpCntValue() and TMRB_GetCaptureValue().
4) TMRB_SetFlipFlop(), TMRB_SetINTMask(), TMRB_SetIdleMode(), TMRB_SetSyncMode(), TMRB_SetDoubleBuf(), TMRB_SetExtStartTrg() and TMRB_SetClkInCoreHalt () handle other specified functions.

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.

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, leading timing value, trailing timing value and up-counter work mode (refer to “Data Structure Description” for details).

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

Return:
None

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 TMRB2 and TMRB5 can be used as the capture trigger of other channels.
Note: TMRB_CAPTURE_OUTPUT_EDGE is only available in TMRB0 through TMRB7.

TMRB3~5: TB2OUT
TMRB0~2: TB7OUT

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.

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 leading timing value, Bit of **MatchTrailingTiming** Indicates if the up-counter matches with trailing timing value, and bit of **Overflow** indicates if overflow had occurred before the interrupt.

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

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

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

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

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

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

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

Return:
None

14.2.3.9 TMRB_ChangeLeadingTiming

Change the value of leading timing for the specified channel.

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

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

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

Return:
None

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

14.2.3.10 TMRB_ChangeTrailingTiming

Change the value of trailing timing for the specified channel.

Prototype:
void

TMRB_ChangeTrailingTiming(TSB_TB_TypeDef* ***TBx***,
uint32_t ***TrailingTiming***)

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

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

Return:
None

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

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 can be
TSB_TB1, TSB_TB2, TSB_TB3, 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 TMRB1 through TMRB3, their start timing is synchronized with TMRB0. If the synchronous mode is enabled for TMRB5 through TMRB7, their start timing is synchronized with TMRB4.

Return:
None

Note:
TMRB1 through TMRB3, TMRB5 through TMRB7 must start counting by calling **TMRB_SetRunState()** before TMRB0 or 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 and set the timing to write to timer register 0 and 1 when double buffer enabled.

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

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

WriteRegMode specifies timing to write to timer register 0 and 1 when double buffer enabled, which can be
- **TMRB_WRITE_REG_SEPARATE:** Timer register 0 and 1 can be written separately, even in case writing preparation is ready for only one register.
- **TMRB_WRITE_REG_SIMULTANEOUS:** In case both registers are not ready to be written, timer registers 0 and 1 can’t be written.

Description:
The register TBxRG0 (***LeadingTiming***) and TBxRG1 (***TrailingTiming***) and their buffers are assigned to the same address. If double buffering is disabled, the same value is written to the registers and their buffers.
If double buffering is enabled, the value is only written to each register buffer. Therefore, to write an initial value to the registers, TBxRG0 (***LeadingTiming***)

and TBxRG1 (***TrailingTiming***), the double buffering must be set to **DISABLE**. Then **ENABLE** double buffering and write the following data to the register, which can be loaded when the corresponding interrupt occurs automatically.

Return:
None

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

14.2.4 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.

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

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

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

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

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

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. SIO/UART

15.1 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/tmpm38x_uart.c, with /Libraries/TX03_Periph_Driver/inc/tmpm38x_uart.h containing the macros, data types, structures and API definitions for use by applications.

15.2 API Functions

15.2.1 Function List

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

- uint8_t SIO_GetRxData(TSB_SC_TypeDef * SIOx)
- void SIO_SetTxData(TSB_SC_TypeDef * SIOx, uint8_t Data)
- void SIO_Init(TSB_SC_TypeDef * SIOx, uint32_t IOClkSel,
  SIO_InitTypeDef * InitStruct)

15.2.2 Detailed Description

Functions listed above can be divided into four parts:

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

15.2.3 Function Documentation

Note1: in all of the following APIs, parameter “TSB_SC_TypeDef* ***UARTx***” can be one of the following values:

- **For TMPM381:** UART0, UART1, UART2.
- **For TMPM383:** UART0, UART1.

Note1: in all of the following APIs, parameter “TSB_SC_TypeDef* ***SIOx***” can be one of the following values:

- **For TMPM381:** SIO0, SIO1, SIO2.
- **For TMPM383:** SIO0, SIO1.

15.2.3.1 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

15.2.3.2 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

15.2.3.3 UART_GetBufState

Indicate the state of transmission or reception buffer.

Prototype:
WorkState
UART_GetBufState(TSB_SC_TypeDef* ***UARTx***,
uint8_t ***Direction***)

Parameters:
UARTx is the specified UART channel.
Direction select the direction of transfer, which can be one of:
- **UART_RX** for reception
- **UART_TX** for transmission

Description:
When ***Direction*** is **UART_RX**, the function returns the state of the reception buffer, which can be **DONE**, meaning that the data received has been saved into the buffer, or **BUSY**, meaning that the data reception is in progress. When ***Direction*** is **UART_TX**, the function returns state of the reception buffer, which can be **DONE**, meaning that the data to be set in the buffer has been sent, or **BUSY**, the data transmission is in progress.

Return:
DONE means that the buffer can be read or written.
BUSY means that the transfer is ongoing.

15.2.3.4 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

15.2.3.5 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 and 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 and transfer mode and flow control for the specified UART channel selected by ***UARTx***.

Return:
None

15.2.3.6 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

15.2.3.7 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

15.2.3.8 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

15.2.3.9 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.

15.2.3.10 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

15.2.3.11 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

15.2.3.12 UART_FIFOConfig

Enable or disable the FIFO of specified UART channel.

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

Parameters:
UARTx is the specified UART channel.
NewState is the new state of the UART FIFO.
This parameter can be one of the following values:
ENABLE or **DISABLE**

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

Return:
None

15.2.3.13 UART_SetFIFOTransferMode

Transfer mode setting.

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

Parameters:
UARTx is the specified 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

15.2.3.14 UART_TRxAutoDisable

Controls automatic disabling of transmission and reception.

Prototype:
void

UART_TRxAutoDisable(TSB_SC_TypeDef * ***UARTx,***
UART_TRxDisable ***TRxAutoDisable***)

Parameters:
UARTx is the specified UART channel.
TRxAutoDisable is the Disabling transmission and reception or not.
This parameter can be one of the following values:
UART_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

15.2.3.15 UART_RxFIFOINTCtrl

Enable or disable receive interrupt for receive FIFO.

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

Parameters:
UARTx is the specified UART channel.
NewState is 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

15.2.3.16 UART_TxFIFOINTCtrl

Enable or disable transmit interrupt for transmit FIFO.

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

Parameters:
UARTx is the specified UART channel.
NewState is 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

15.2.3.17 UART_RxFIFOByteSel

Bytes used in receive FIFO.

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

Parameters:
UARTx is the specified UART channel.
BytesUsed is 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

15.2.3.18 UART_RxFIFOFillLevel

Receive FIFO fill level to generate receive interrupts.

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

Parameters:
UARTx is the specified UART channel.
RxFIFOLevel is 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 or UART_RXFIFO4B_FLEVLE_3_1B.**

Return:

None

15.2.3.19 UART_RxFIFOINTSel

Select RX interrupt generation condition.

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

Parameters:
UARTx is the specified UART channel.
RxINTCondition is 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

15.2.3.20 UART_RxFIFOClear

Clear Receive FIFO.

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

Parameters:
UARTx is the specified UART channel.

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

Return:
None

15.2.3.21 UART_TxFIFOFillLevel

Transmit FIFO fill level to generate transmit interrupts.

Prototype:
void
UART_TxFIFOFillLevel(TSB_SC_TypeDef * ***UARTx,***
uint32_t ***TxFIFOLevel***)

Parameters:
UARTx is the specified UART channel.

TxFIFOLevel is transmit FIFO fill level.
This parameter can be one of the following values:
UART_TXFIFO4B_FLEVEL_0_0B, UART_TXFIFO4B_FLEVEL_1_1B, UART_TXFIFO4B_FLEVEL_2_0B or UART_TXFIFO4B_FLEVEL_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_FLEVEL_0_0B, UART_TXFIFO4B_FLEVEL_1_1B, UART_TXFIFO4B_FLEVEL_2_0B or UART_TXFIFO4B_FLEVEL_3_1B.**

Return:
None

15.2.3.22 UART_TxFIFOINTSel

Select TX interrupt generation condition.

Prototype:
void
UART_TxFIFOINTSel(TSB_SC_TypeDef * ***UARTx,***
uint32_t ***TxINTCondition***)

Parameters:
UARTx is the specified UART channel.
TxINTCondition is TX interrupt generation condition.
This parameter can be one of the following values:
UART_TFIS_REACH_FLEVEL or **UART_TFIS_REACH_NOREACH_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 or UART_TFIS_REACH_NOREACH_FLEVEL.**

Return:
None

15.2.3.23 UART_TxFIFOClear

Clear Transmit FIFO.

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

Parameters:
UARTx is the specified UART channel.

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

Return:
None

15.2.3.24 UART_GetRxFIFOFillLevelStatus

Status of receive FIFO fill level.

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

Parameters:
UARTx is the specified UART channel.

Description:
Status of receive FIFO fill level.

Return:
UART_TRXFIFO_EMPTY: TX FIFO fill level is empty.
UART_TRXFIFO_1B: TX FIFO fill level is 1 byte.
UART_TRXFIFO_2B: TX FIFO fill level is 2 bytes.
UART_TRXFIFO_3B: TX FIFO fill level is 3 bytes.
UART_TRXFIFO_4B: TX FIFO fill level is 4 bytes.

15.2.3.25 UART_GetRxFIFOOverRunStatus

Receive FIFO overrun.

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

Parameters:
UARTx is the specified UART channel.

Description:
Receive FIFO overrun.

Return:
UART_RXFIFO_OVERRUN: Flags for RX FIFO overrun.

15.2.3.26 UART_GetTxFIFOFillLevelStatus

Status of transmit FIFO fill level.

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

Parameters:
UARTx is the specified UART channel.

Description:
Status of transmit FIFO fill level.

Return:
UART_TRXFIFO_EMPTY: TX FIFO fill level is empty.
UART_TRXFIFO_1B: TX FIFO fill level is 1 byte.

UART_TRXFIFO_2B: TX FIFO fill level is 2 bytes.
UART_TRXFIFO_3B: TX FIFO fill level is 3 bytes.
UART_TRXFIFO_4B: TX FIFO fill level is 4 bytes.

15.2.3.27 UART_GetTxFIFOUnderRunStatus

Transmit FIFO under run.

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

Parameters:
UARTx is the specified UART channel.

Description:
Transmit FIFO under run

Return:
UART_TXFIFO_UNDERRUN: Flags for TX FIFO under-run.

15.2.3.28 SIO_Enable

Enable the specified SIO channel.

Prototype:
void
SIO_Enable(TSB_SC_TypeDef* ***SIOx***)

Parameters:
SIOx is the specified SIO channel.

Description:
This function will enable the specified SIO channel selected by ***SIOx***.

Return:
None

15.2.3.29 SIO_Disable

Disable the specified SIO channel.

Prototype:
void
SIO_Disable(TSB_SC_TypeDef* ***SIOx***)

Parameters:
SIOx is the specified SIO channel.

Description:
This function will disable the specified SIO channel selected by ***SIOx***.

Return:
None

15.2.3.30 SIO_GetRxData

Get data received from the specified SIO channel.

Prototype:
Uint8_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.

15.2.3.31 SIO_SetTxData

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

Prototype:
void
SIO_SetTxData(TSB_SC_TypeDef* ***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

15.2.3.32 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 selected clock.
This parameter can be one of the following values:
SIO_CLK_SCLKOUTPUT or SIO_CLK_SCLKINPUT.
InitStruct is the structure containing basic SIO configuration including baud rate, transmission direction and 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

15.2.4 Data Structure Description

15.2.4.1 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.

15.2.4.2 SIO_InitTypeDef

Data Fields:
uint32_t
InputClkEdge Select the input clock edge, which can be set as:
- **SIO_SCLKS_TXDF_RXDR** Data in the transfer buffer is sent to TXDx pin one bit at a time on the falling edge of SCLKx, data from RXDx pin is received in the receive buffer one bit at a time on the rising edge of SCLKx.

- **SIO_SCLKS_TXDR_RXDF** Data in the transfer buffer is sent to TXDx pin one bit at a time on the rising edge of SCLKx, data from RXDx pin is received in the receive buffer one bit at a time on the falling edge of SCLKx.

uint32_t
- ***IntervalTime*** Setting interval time of continuous transmission, which can be set as:
- **SIO_SINT_TIME_NONE** Interval time is None.
- **SIO_SINT_TIME_SCLK_1** Interval time is 1xSCLK.
- **SIO_SINT_TIME_SCLK_2** Interval time is 2xSCLK.
- **SIO_SINT_TIME_SCLK_4** Interval time is 4xSCLK.
- **SIO_SINT_TIME_SCLK_8** Interval time is 8xSCLK.
- **SIO_SINT_TIME_SCLK_16** Interval time is 16xSCLK.
- **SIO_SINT_TIME_SCLK_32** Interval time is 32xSCLK.
- **SIO_SINT_TIME_SCLK_64** Interval time is 64xSCLK.

uint32_t
TransferMode Setting transfer mode, which can be set as:
- **SIO_TRANSFER_PROHIBIT** Transfer prohibit.
- **SIO_TRANSFER_HALFDPX_RX** Half duplex(Receive).
- **SIO_TRANSFER_HALFDPX_TX** Half duplex(Transmit).
- **SIO_TRANSFER_FULLDPX** Full duplex.

uint32_t
TransferDir sets transfer direction which could be set as:
- **SIO_LSB_FIRST** for LSB FIRST in transmission
- **SIO_MSB_FIRST** for MSB FIRST in transmission.

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
DoubleBuffer Double Buffer mode, which can be set as:
- **SIO_WBUF_DISABLE** Double buffer disable.
- **SIO_WBUF_ENABLE** Double buffer enable.

uint32_t
BaudRateClock Select the input clock for baud rate generator, which can be set as:
- **SIO_BR_CLOCK_T1** Select the input clock to baud rate generator is T1.
- **SIO_BR_CLOCK_T4** Select the input clock to baud rate generator is T4.
- **SIO_BR_CLOCK_T16** Select the input clock to baud rate generator is T16.
- **SIO_BR_CLOCK_T64** Select the input clock to baud rate generator is T64.

uint32_t
Divider Division ratio "N", which can be set as :
- **SIO_BR_DIVIDER_16** Division ratio is 16.
- **SIO_BR_DIVIDER_1** Division ratio is 1.
- **SIO_BR_DIVIDER_2** Division ratio is 2.
- **SIO_BR_DIVIDER_3** Division ratio is 3.
- **SIO_BR_DIVIDER_4** Division ratio is 4.
- **SIO_BR_DIVIDER_5** Division ratio is 5.
- **SIO_BR_DIVIDER_6** Division ratio is 6.

- **SIO_BR_DIVIDER_7** Division ratio is 7.
- **SIO_BR_DIVIDER_8** Division ratio is 8.
- **SIO_BR_DIVIDER_9** Division ratio is 9.
- **SIO_BR_DIVIDER_10** Division ratio is 10.
- **SIO_BR_DIVIDER_11** Division ratio is 11.
- **SIO_BR_DIVIDER_12** Division ratio is 12.
- **SIO_BR_DIVIDER_13** Division ratio is 13.
- **SIO_BR_DIVIDER_14** Division ratio is 14.
- **SIO_BR_DIVIDER_15** Division ratio is 15.

16. VLTD

16.1 Overview

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

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

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm38x_vltd.c, with /Libraries/TX03_Periph_Driver/inc/tmpm38x_vltd.h containing the macros, data types, structures and API definitions for use by applications.

16.2 API Functions

16.2.1 Function List

- void VLTD_Enable(void);
- void VLTD_Disable(void);

16.2.2 Detailed Description

Functions listed above have responsibilities:
Enable or disable VLTD are handled by VLTD_Enable() and VLTD_Disable().

16.2.3 Function Documentation

16.2.3.1 VLTD_Enable

Enable the VLTD function.

Prototype:
void
VLTD_Enable(void)

Parameters:
None.

Description:
This function will enable the VLTD function.

Return:
None.

16.2.3.2 VLTD_Disable

Disable the VLTD function.

Prototype:
void
VLTD_Disable(void)

Parameters:

None.

Description:
This function will disable the VLTD function.

Return:
None.

16.2.4 Data Structure Description

None

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\tmpm38x_wdt.c, with \Libraries/TX03_Periph_Driver\inc\tmpm38x_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 reset when the counter overflows.

Prototype:
void
WDT_SetOverflowOutput(uint32_t ***OverflowOutput***)

Parameters:
OverflowOutput: Select function of WDT when counter overflow.
This parameter can be one of the following values:
- **WDT_NMIINT**: Set WDT to generate the NMI interrupt when counter overflows.
- **WDT_WDOUT**: Set WDT to generate reset when counter overflows.

Description:
This function will set WDT to generate NMI interrupt if the counter overflows when ***OverflowOutput*** is **WDT_NMIINT,** and set WDT to generate reset if the counter overflows when ***OverflowOutput*** is **WDT_WDOUT**.

Return:

None

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 “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 the NMI interrupt when the counter overflows.