

TOSHIBA

TOSHIBA TX04 Peripheral Driver User Guide (TMPM461)

Ver 1
Sep, 2017

TOSHIBA ELECTRONIC DEVICES & STORAGE CORPORATION

CMDR-M461UG-01E

RESTRICTIONS ON PRODUCT USE

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

© 2017 Toshiba Electronic Devices & Storage Corporation

Index

| | |
|--|-----------|
| 1. Introduction | 1 |
| 2. Organization of TOSHIBA TX04 Peripheral Driver | 1 |
| 3. ADC | 2 |
| 3.1 Overview | 2 |
| 3.2 API Functions | 3 |
| 3.2.1 Function List | 3 |
| 3.2.2 Detailed Description..... | 3 |
| 3.2.3 Function Documentation..... | 4 |
| 3.2.4 Data Structure Description..... | 19 |
| 4. CEC | 22 |
| 4.1 Overview | 22 |
| 4.2 API Functions | 22 |
| 4.2.1 Function List | 22 |
| 4.2.2 Detailed Description..... | 23 |
| 4.2.3 Function Documentation..... | 23 |
| 4.2.4 Data Structure Description..... | 41 |
| 5. EXB | 43 |
| 5.1 Overview | 43 |
| 5.2 API Functions | 43 |
| 5.2.1 Function List | 43 |
| 5.2.2 Detailed Description..... | 43 |
| 5.2.3 Function Documentation..... | 43 |
| 5.2.4 Data Structure Description..... | 47 |
| 6. CG | 50 |
| 6.1 Overview | 50 |
| 6.2 API Functions | 50 |
| 6.2.1 Function List | 50 |
| 6.2.2 Detailed Description..... | 51 |
| 6.2.3 Function Documentation..... | 51 |
| 6.2.4 Data Structure Description..... | 68 |
| 7. FC70 | |
| 7.1 Overview | 70 |
| 7.2 API Functions | 70 |
| 7.2.1 Function List | 70 |
| 7.2.2 Detailed Description..... | 71 |
| 7.2.3 Function Documentation..... | 71 |

| | |
|--|------------|
| 7.2.4 Data Structure Description..... | 81 |
| 8. FUART..... | 82 |
| 8.1 Overview | 82 |
| 8.2 API Functions..... | 82 |
| 8.2.1 Function List | 82 |
| 8.2.2 Detailed Description..... | 83 |
| 8.2.3 Function Documentation..... | 83 |
| 8.2.4 Data Structure Description..... | 94 |
| 9. GPIO..... | 97 |
| 9.1 Overview | 97 |
| 9.2 API Functions..... | 97 |
| 9.2.1 Function List | 97 |
| 9.2.2 Detailed Description..... | 97 |
| 9.2.3 Function Documentation..... | 97 |
| 9.2.4 Data Structure Description..... | 110 |
| 10. I2C | 112 |
| 10.1 Overview | 112 |
| 10.2 API Functions..... | 112 |
| 10.2.1 Function List | 112 |
| 10.2.2 Detailed Description..... | 112 |
| 10.2.3 Function Documentation..... | 113 |
| 10.2.4 Data Structure Description..... | 119 |
| 11. IGBT | 123 |
| 11.1 Overview | 123 |
| 11.2 API Functions..... | 123 |
| 11.2.1 Function List | 123 |
| 11.2.2 Detailed Description..... | 123 |
| 11.2.3 Function Documentation..... | 124 |
| 11.2.4 Data Structure Description..... | 130 |
| 12. LVD..... | 134 |
| 12.1 Overview | 134 |
| 12.2 API Functions..... | 134 |
| 12.2.1 Function List | 134 |
| 12.2.2 Detailed Description..... | 134 |
| 12.2.3 Function Documentation..... | 134 |
| 12.2.4 Data Structure Description..... | 139 |
| 13. OFD | 140 |

| | |
|--|------------|
| 13.1 Overview | 140 |
| 13.2 API Functions | 140 |
| 13.2.1 Function List | 140 |
| 13.2.2 Detailed Description..... | 140 |
| 13.2.3 Function Documentation | 140 |
| 13.2.4 Data Structure Description..... | 143 |
| 14. RMC..... | 144 |
| 14.1 Overview | 144 |
| 14.2 API Functions | 144 |
| 14.2.1 Function List | 144 |
| 14.2.2 Detailed Description..... | 144 |
| 14.2.3 Function Documentation | 145 |
| 14.2.4 Data Structure Description..... | 153 |
| 15. RTC | 155 |
| 15.1 Overview | 155 |
| 15.2 API Functions | 155 |
| 15.2.1 Function List | 155 |
| 15.2.2 Detailed Description..... | 156 |
| 15.2.3 Function Documentation | 156 |
| 15.2.4 Data Structure Description..... | 175 |
| 16. SSP..... | 177 |
| 16.1 Overview | 177 |
| 16.2 API Functions | 177 |
| 16.2.1 Function List | 177 |
| 16.2.2 Detailed Description..... | 178 |
| 16.2.3 Function Documentation | 178 |
| 16.2.4 Data Structure Description..... | 187 |
| 17. TMRB..... | 189 |
| 17.1 Overview | 189 |
| 17.2 API Functions | 189 |
| 17.2.1 Function List | 189 |
| 17.2.2 Detailed Description..... | 190 |
| 17.2.3 Function Documentation | 190 |
| 17.2.4 Data Structure Description..... | 200 |
| 18. UART..... | 202 |
| 18.1 Overview | 202 |
| 18.2 API Functions | 202 |

| | |
|--|------------|
| 18.2.1 Function List | 202 |
| 18.2.2 Detailed Description..... | 203 |
| 18.2.3 Function Documentation | 203 |
| 18.2.4 Data Structure Description..... | 217 |
| 19. uDMAC | 220 |
| 19.1 Overview | 220 |
| 19.2 API Functions | 220 |
| 19.2.1 Function List | 220 |
| 19.2.2 Detailed Description..... | 221 |
| 19.2.3 Function Documentation | 221 |
| 19.2.4 Data Structure Description..... | 233 |
| 20. WDT | 241 |
| 20.1 Overview | 241 |
| 20.2 API Functions | 241 |
| 20.2.1 Function List | 241 |
| 20.2.2 Detailed Description..... | 241 |
| 20.2.3 Function Documentation | 241 |
| 20.2.4 Data Structure Description..... | 244 |

1. Introduction

TOSHIBA TX04 Peripheral Driver is a set of drivers for all peripherals found on the TOSHIBA TX04 series microcontrollers. TPM461x (see **Note** below) Peripheral Driver is an important part of TOSHIBA TX04 Peripheral Driver, which is designed for TPM461 series MCUs.

TOSHIBA TX04 Peripheral Driver contains a collection of macros, data types, and structures for each peripheral.

The design goals of TOSHIBA TPM461 Peripheral Driver:

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

2. Organization of TOSHIBA TX04 Peripheral Driver

/Libraries

This folder contains all CMSIS files and TPM461 Peripheral Drivers.

/Libraries/ TX04_CMSIS

This folder contains the device peripheral access layer of TPM461 CMSIS files.

/Libraries/TX04_Periph_Driver

This folder contains all the source code of the drivers, the core of TOSHIBA TPM461 Peripheral Driver.

/Libraries/TX04_Periph_Driver/inc

This folder contains all the header files of TPM461 Peripheral Drivers for each peripheral.

/Libraries/TX04_Periph_Driver/src

This folder contains all the source files of TPM461 Peripheral Drivers for each peripheral.

/Project

This folder contains template project and examples for using TPM461 Peripheral Driver.

/Project/Template

This folder contains template project of TOSHIBA TPM461 Peripheral Driver.

/Project/Examples

This folder contains a set of examples for using TPM461 Peripheral Driver

/Utilities/TPM461-EVAL

This folder contains the configuration and driver files for hardware resources (e.g. led, key) on TPM461 boards.

***Note:** The “TPM461x” in this document can be TPM461F15FG/TPM461F10FG.

3. ADC

3.1 Overview

TMPM461x contain one unit of 12-bit sequential-conversion analog/digital converters (ADC) with normal 15 analog input channels and extension 5 analog input channels. Both normal 15 analog input channels(AIN0 to AIN14) and extension 5 analog input channels are used as input/output ports.

The 12-bit AD converter has the following features:

1. Start normal AD conversion and top-priority AD conversion by software activation, internal triggers or an external trigger (ADTRG).
2. Operation 4 different modes of Normal AD conversion:
 - Fixed-channel single conversion mode
 - Channel scan single conversion mode
 - Fixed-channel repeat conversion mode
 - Channel scan repeat conversion mode
3. Operation modes of top-priority AD conversion:
 - Fixed-channel single conversion mode
4. Normal / Top-priority AD conversion completion interrupt
5. Normal / Top-priority AD conversion completion/busy flag
6. AD monitor function
 - When the AD monitor function is enabled, an interrupt is generated if any comparison result is matched.
7. AD conversion clock is controllable from f_c to $f_c/16$.
8. Current reduction function of VREF reference is supported.

The ADC API provides a set of functions for using the TMPM461x ADC modules. It includes ADC channel set, mode set, monitor function set, interrupt set, ADC status read, ADC result value read and so on.

All driver APIs are contained in /Libraries/TX04_Periph_Driver/src/tmpm461_adc.c, with /Libraries/TX04_Periph_Driver/inc/tmpm461_adc.h containing the macros, data types, structures and API definitions for use by applications.

***Note:**

- 1 Extension channel AD conversion mode are Fixed-Channel single conversion and Fixed-channel repeat conversion mode only.
- 2 To use the Port H as an analog input of the AD converter, disable input on PHIE and disable pull-up on PHPUP

3.2 API Functions

3.2.1 Function List

- ◆ void ADC_SWReset(TSB_AD_TypeDef * **ADx**)
- ◆ void ADC_SetClk(TSB_AD_TypeDef * **ADx**,
uint32_t **Sample_HoldTime**,
uint32_t **Prescaler_Output**)
- ◆ void ADC_Start(TSB_AD_TypeDef * **ADx**)
- ◆ void ADC_SetScanMode(TSB_AD_TypeDef * **ADx**,
FunctionalState **NewState**)
- ◆ void ADC_SetRepeatMode(TSB_AD_TypeDef * **ADx**,
FunctionalState **NewState**)
- ◆ void ADC_SetINTMode(TSB_AD_TypeDef * **ADx**, uint32_t **INTMode**)
- ◆ void ADC_SetInputChannel(TSB_AD_TypeDef * **ADx**, ADC_AINx **InputChannel**)
- ◆ void ADC_SetScanChannel(TSB_AD_TypeDef * **ADx**,
ADC_AINx **StartChannel**,
uint32_t **Range**)
- ◆ void ADC_SetVrefCut(TSB_AD_TypeDef * **ADx**, uint32_t **VrefCtrl**)
- ◆ void ADC_SetIdleMode(TSB_AD_TypeDef * **ADx**, FunctionalState **NewState**)
- ◆ void ADC_SetVref(TSB_AD_TypeDef * **ADx**, FunctionalState **NewState**)
- ◆ void ADC_SetInputChannelTop(TSB_AD_TypeDef * **ADx**,
ADC_AINx **TopInputChannel**)
- ◆ void ADC_StartTopConvert(TSB_AD_TypeDef * **ADx**)
- ◆ void ADC_SetMonitor(TSB_AD_TypeDef * **ADx**,
ADC_CMPCRx **ADCMPx**,
FunctionalState **NewState**)
- ◆ void ADC_ConfigMonitor(TSB_AD_TypeDef * **ADx**,
ADC_CMPCRx **ADCMPx**,
ADC_MonitorTypeDef * **Monitor**)
- ◆ void ADC_SetHWTrg(TSB_AD_TypeDef * **ADx**,
uint32_t **HWSrc**,
FunctionalState **NewState**)
- ◆ void ADC_SetHWTrgTop(TSB_AD_TypeDef * **ADx**,
uint32_t **HWSrc**,
FunctionalState **NewState**)
- ◆ ADC_State ADC_GetConvertState(TSB_AD_TypeDef * **ADx**)
- ◆ ADC_Result ADC_GetConvertResult(TSB_AD_TypeDef * **ADx**,
ADC_REGx **ADREGx**)
- ◆ void ADC_EnableTrigger(void)
- ◆ void ADC_DisableTrigger(void)
- ◆ ADC_SetTriggerStartup(ADC_TRGx **TriggerStartup**)
- ◆ ADC_SetTriggerStartupTop(ADC_TRGx **TopTriggerStartup**)
- ◆ ADC_DisableExtension(void)
- ◆ ADC_EnableExtension(void)
- ◆ ADC_SetExtChannel(ADC_EXT_AINx **ExtChannel**)
- ◆ ADC_SetExtClkSource(uint32_t **ClkSrc**)
- ◆ ADC_SetExtClk (uint32_t **Sample_HoldTime**)

3.2.2 Detailed Description

Functions listed above can be divided into six parts:

- 1) ADC setting by ADC_SetClk(), ADC_SetScanMode(), ADC_SetRepeatMode(),
ADC_SetINTMode(), ADC_SetInputChannel(), ADC_SetScanChannel(), ADC_SetVref(),
ADC_SetInputChannelTop(), ADC_SetMonitor(), ADC_ConfigMonitor(), ADC_SetHWTrg(),
ADC_SetHWTrgTop().
- 2) ADC function start by ADC_Start(), ADC_StartTopConvert().

- 3) ADC state or data read functions by ADC_GetConvertState(), ADC_GetConvertResult().
- 4) ADC_SWReset(), ADC_SetVrefCut() and ADC_SetIdleMode() handle other specified functions.
- 5) ADC_EnableTrigger(), ADC_DisableTrigger(), ADC_SetTriggerStartup(), ADC_SetTriggerStartupTop().
- 6) Extension ADC setting by ADC_DisableExtension(), ADC_EnableExtension(), ADC_SetExtChannel(), ADC_SetExtClkSource(), ADC_SetExtClk().

3.2.3 Function Documentation

3.2.3.1 ADC_SWReset

Software reset ADC.

Prototype:

```
void  
ADC_SWReset(TSB_AD_TypeDef * ADx)
```

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

Description:

This function will software reset ADC.

***Note:**

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

Return:

None

3.2.3.2 ADC_SetClk

Set ADC sample hold time and prescaler output.

Prototype:

```
void  
ADC_SetClk(TSB_AD_TypeDef * ADx,  
           uint32_t Sample_HoldTime,  
           uint32_t Prescaler_Output)
```

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

Sample_HoldTime: Select ADC sample hold time.

This parameter can be one of the following values:

- **ADC_CONVERSION_CLK_10:** 10 x <ADCLK>
- **ADC_CONVERSION_CLK_20:** 20 x <ADCLK>
- **ADC_CONVERSION_CLK_30:** 30 x <ADCLK>
- **ADC_CONVERSION_CLK_40:** 40 x <ADCLK>
- **ADC_CONVERSION_CLK_80:** 80 x <ADCLK>
- **ADC_CONVERSION_CLK_160:** 160 x <ADCLK>
- **ADC_CONVERSION_CLK_320:** 320 x <ADCLK>

Prescaler_Output: Select ADC prescaler output(ADCLK).

This parameter can be one of the following values:

- **ADC_FC_DIVIDE_LEVEL_1:** fc
- **ADC_FC_DIVIDE_LEVEL_2:** fc / 2
- **ADC_FC_DIVIDE_LEVEL_4:** fc / 4
- **ADC_FC_DIVIDE_LEVEL_8:** fc / 8
- **ADC_FC_DIVIDE_LEVEL_16:** fc / 16

Description:

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

***Note:**

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

Return:

None

3.2.3.3 ADC_Start

Start AD conversion.

Prototype:

```
void  
ADC_Start(TSB_AD_TypeDef * ADx)
```

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

Description:

This function will start normal AD conversion.

***Note:**

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

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

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

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

Return:

None

3.2.3.4 ADC_SetScanMode

Enable or disable ADC scan mode.

Prototype:

```
void  
ADC_SetScanMode(TSB_AD_TypeDef * ADx,  
                FunctionalState NewState)
```

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

NewState: Specify ADC scan mode state

This parameter can be one of the following values:

- **ENABLE** : Enable scan mode
- **DISABLE** : Disable scan mode

Description:

This function will enable or disable ADC scan mode.

Return:

None

3.2.3.5 ADC_SetRepeatMode

Enable or disable ADC repeat mode.

Prototype:

```
void  
ADC_SetRepeatMode(TSB_AD_TypeDef * ADx,  
                  FunctionalState NewState)
```

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

NewState: Specify ADC repeat mode state

This parameter can be one of the following values:

- **ENABLE** : Enable repeat mode
- **DISABLE** : Disable repeat mode

Description:

This function will enable or disable ADC repeat mode.

Return:

None

3.2.3.6 ADC_SetINTMode

Set ADC interrupt mode in fixed channel repeat conversion mode.

Prototype:

```
void  
ADC_SetINTMode(TSB_AD_TypeDef * ADx,  
               uint32_t INTMode)
```

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

INTMode: Specify AD conversion interrupt mode.

The parameter can be one of the following values:

- **ADC_INT_SINGLE:** Generate interrupt once every single conversion.
- **ADC_INT_CONVERSION_2:** Generate interrupt once every 2 conversions.
- **ADC_INT_CONVERSION_3:** Generate interrupt once every 3 conversions.
- **ADC_INT_CONVERSION_4:** Generate interrupt once every 4 conversions.
- **ADC_INT_CONVERSION_5:** Generate interrupt once every 5 conversions.
- **ADC_INT_CONVERSION_6:** Generate interrupt once every 6 conversions.
- **ADC_INT_CONVERSION_7:** Generate interrupt once every 7 conversions.
- **ADC_INT_CONVERSION_8:** Generate interrupt once every 8 conversions.

Description:

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

***Note:**

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

Examples for setting fixed channel repeat conversion mode:

1. **ADC_SetScanMode(DISABLE).**
2. **ADC_SetRepeatMode(ENABLE).**

Return:

None

3.2.3.7 ADC_SetInputChannel

Set ADC input channel.

Prototype:

```
void  
ADC_SetInputChannel(TSB_AD_TypeDef * ADx,  
                   ADC_AINx InputChannel)
```

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

InputChannel: Analog input channel.

This parameter can be one of the following values:

- **ADC_AN_00, ADC_AN_01, ADC_AN_02, ADC_AN_03,**
- **ADC_AN_04, ADC_AN_05, ADC_AN_06, ADC_AN_07**
- **ADC_AN_08, ADC_AN_09, ADC_AN_10, ADC_AN_11**
- **ADC_AN_12, ADC_AN_13, ADC_AN_14, ADC_AN_EXT**

Description:

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

***Note:**

- 1 Only one channel of **ADC_AN_00~ADC_AN_EXT** can be selected as normal conversion input each time.
- 2 When InputChannel is **ADC_AN_EXT**, it is one selected channel from **AIN15** to **AIN19**.(Please refer to the description of **ADC_SetExtChannel()**)

Return:

None

3.2.3.8 ADC_SetScanChannel

Set ADC scan channel.

Prototype:

```
void
ADC_SetScanChannel(TSB_AD_TypeDef * ADx,
                  ADC_AINx StartChannel,
                  uint32_t Range)
```

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

StartChannel: Specify the start channel to be scanned.

This parameter can be one of the following values:

- **ADC_AN_00, ADC_AN_01, ADC_AN_02, ADC_AN_03,**
- **ADC_AN_04, ADC_AN_05, ADC_AN_06, ADC_AN_07**
- **ADC_AN_08, ADC_AN_09, ADC_AN_10, ADC_AN_11**
- **ADC_AN_12, ADC_AN_13, ADC_AN_14.**

Range: Specify the range of assignable channel scan value.

This parameter can be one of the following values:

- **1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15. (note: StartChannel + Range <= 15)**

Description:

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

***Note:**

Valid channel scan setting values are shown as follows:

| StartChannel | Range (The range of assignable channel scan value) |
|---------------------|---|
| ADC_AN_00 | 1 to 15 |
| ADC_AN_01 | 1 to 14 |
| ADC_AN_02 | 1 to 13 |
| ADC_AN_03 | 1 to 12 |
| ADC_AN_04 | 1 to 11 |
| ADC_AN_05 | 1 to 10 |
| ADC_AN_06 | 1 to 9 |
| ADC_AN_07 | 1 to 8 |

| | |
|-----------|--------|
| ADC_AN_08 | 1 to 7 |
| ADC_AN_09 | 1 to 6 |
| ADC_AN_10 | 1 to 5 |
| ADC_AN_11 | 1 to 4 |
| ADC_AN_12 | 1 to 3 |
| ADC_AN_13 | 1 to 2 |
| ADC_AN_14 | 1 |

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

Return:
None

3.2.3.9 ADC_SetVrefCut

Control AVREFH-AVREFL current.

Prototype:
void
ADC_SetVrefCut(TSB_AD_TypeDef * **ADx**,
uint32_t **VrefCtrl**)

Parameters:
ADx: Select ADC unit.
This parameter can be the following value:
➤ **TSB_AD**

VrefCtrl: Specify how to apply AVREFH-AVREFL current.
This parameter can be one of the following values:
➤ **ADC_APPLY_VREF_IN_CONVERSION:** Apply the current only in conversion.
➤ **ADC_APPLY_VREF_AT_ANY_TIME:** Apply the current at any time except in RESET.

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

Return:
None

3.2.3.10 ADC_SetIdleMode

Set ADC operation in IDLE mode.

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

Parameters:
ADx: Select ADC unit.
This parameter can be the following value:
➤ **TSB_AD**

NewState: Specify ADC operation state in IDLE mode.

This parameter can be one of the following values:

- **ENABLE** : Enable ADC in IDLE mode
- **DISABLE** : Disable ADC in IDLE mode

Description:

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

Return:

None

3.2.3.11 ADC_SetVref

Set ADC Vref application control on or off.

Prototype:

```
void  
ADC_SetVref(TSB_AD_TypeDef * ADx,  
            FunctionalState NewState)
```

Parameters:

ADx: Select ADC unit.

This parameter can be one of the following values:

- **TSB_AD**

NewState: Specify AD conversion Vref application control.

This parameter can be one of the following values:

- **ENABLE** : Enable reference voltage(Vref)
- **DISABLE** : Disable reference voltage(Vref)

Description:

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

***Note:**

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

Return:

None

3.2.3.12 ADC_SetInputChannelTop

Select ADC top-priority conversion analog input channel.

Prototype:

```
void  
ADC_SetInputChannelTop(TSB_AD_TypeDef * ADx,  
                      ADC_AINx TopInputChannel)
```

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

TopInputChannel: Analog input channel for top-priority conversion.

This parameter can be one of the following values:

- **ADC_AN_00, ADC_AN_01, ADC_AN_02, ADC_AN_03,**

- **ADC_AN_04, ADC_AN_05, ADC_AN_06, ADC_AN_07**
- **ADC_AN_08, ADC_AN_09, ADC_AN_10, ADC_AN_11**
- **ADC_AN_12, ADC_AN_13, ADC_AN_14, ADC_AN_EXT**

Description:

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

***Note:**

- 1 Only one channel of **ADC_AN_00~ADC_AN_EXT** can be selected as Top-priority conversion input each time.
- 2 When **TopInputChannel** is **ADC_AN_EXT**, it is one selected channel from **AIN15** to **AIN19** (Please refer to the description of **ADC_SetExtChannel()**).

Return:

None

3.2.3.13 ADC_StartTopConvert

Start top-priority AD conversion.

Prototype:

```
void  
ADC_StartTopConvert(TSB_AD_TypeDef * ADx)
```

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

Description:

This function will start top-priority AD conversion.

***Note:**

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

Return:

None

3.2.3.14 ADC_SetMonitor

Enable or disable the specified ADC monitor module.

Prototype:

```
void  
ADC_SetMonitor(TSB_AD_TypeDef * ADx,  
               ADC_CMPCRx ADCMPx,  
               FunctionalState NewState)
```

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

ADCMPx: Select which compare control register will be used.

The parameter can be one of the following values:

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

NewState: Specify ADC monitor function state.

This parameter can be one of the following values:

- **ENABLE** : Enable ADC monitor
- **DISABLE** : Disable ADC monitor

Description:

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

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

Return:

None

3.2.3.15 ADC_ConfigMonitor

Configure the specified ADC monitor module.

Prototype:

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

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

ADCMPx: Select which compare control register will be used.

The parameter can be one of the following values:

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

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

Description:

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

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

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

Return:

None

3.2.3.16 ADC_SetHWTrg

Set hardware trigger for normal AD conversion.

Prototype:

```
void  
ADC_SetHWTrg(TSB_AD_TypeDef * ADx,  
              uint32_t HWSrc,  
              FunctionalState NewState)
```

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

HWSrc: Hardware source for activating normal AD conversion.

This parameter can be one of the following values:

- **ADC_EXTERADTRG:** ADTRG pin
- **ADC_INTERTRIGGER:** Internal trigger (selected by ADILVTRGSEL <TRGSEL>)

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

This parameter can be one of the following values:

- **ENABLE :** Enable hardware trigger source
- **DISABLE :** Disable hardware trigger source

Description:

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

***Note:**

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

Return:

None

3.2.3.17 ADC_SetHWTrgTop

Set hardware trigger for top-priority AD conversion.

Prototype:

```
void  
ADC_SetHWTrgTop(TSB_AD_TypeDef * ADx,  
                uint32_t HWSrc,  
                FunctionalState NewState)
```

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

HWSrc: Hardware source for activating top-priority AD conversion.

This parameter can be one of the following values:

- **ADC_EXTERADTRG:** ADTRG pin
- **ADC_INTERTRIGGER:** Internal trigger (selected by ADILVTRGSEL)

<HPTRGSEL>)

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

This parameter can be one of the following values:

- **ENABLE** : Enable hardware trigger source
- **DISABLE** : Disable hardware trigger source

Description:

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

***Note:**

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

Return:

None

3.2.3.18 ADC_GetConvertState

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

Prototype:

ADC_State

ADC_GetConvertState(TSB_AD_TypeDef * **ADx**)

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

Description:

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

Return:

A union with the state of AD conversion:

NormalBusy(Bit 0) : '1' means normal AD is converting

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

TopBusy(Bit 2) : '1' means top-priority AD is converting

TopComplete (Bit 3) : '1' means top-priority AD conversion is complete.

3.2.3.19 ADC_GetConvertResult

Read AD conversion result.

Prototype:

ADC_Result

ADC_GetConvertResult(TSB_AD_TypeDef * **ADx**,
ADC_REGx **ADREGx**)

Parameters:

ADx: Select ADC unit.

This parameter can be the following value:

- **TSB_AD**

ADREGx: Select ADC result register.

This parameter can be one of the following values:

- **ADC_REG_00, ADC_REG_01, ADC_REG_02, ADC_REG_03**
- **ADC_REG_04, ADC_REG_05, ADC_REG_06, ADC_REG_07**
- **ADC_REG_08, ADC_REG_09, ADC_REG_10, ADC_REG_11**
- **ADC_REG_12, ADC_REG_13, ADC_REG_14, ADC_REG_15**
- **ADC_REG_SP**

Description:

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

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

| Fixed channel single mode | | |
|----------------------------------|------------|------------------|
| Unit | Channel | Storage register |
| TSB_AD | ADC_AN_00 | ADC_REG_00 |
| | ADC_AN_01 | ADC_REG_01 |
| | ADC_AN_02 | ADC_REG_02 |
| | ADC_AN_03 | ADC_REG_03 |
| | ADC_AN_04 | ADC_REG_04 |
| | ADC_AN_05 | ADC_REG_05 |
| | ADC_AN_06 | ADC_REG_06 |
| | ADC_AN_07 | ADC_REG_07 |
| | ADC_AN_08 | ADC_REG_08 |
| | ADC_AN_09 | ADC_REG_09 |
| | ADC_AN_10 | ADC_REG_10 |
| | ADC_AN_11 | ADC_REG_11 |
| | ADC_AN_12 | ADC_REG_12 |
| | ADC_AN_13 | ADC_REG_13 |
| | ADC_AN_14 | ADC_REG_14 |
| ADC_AN_EXT | ADC_REG_15 | |

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

| Channel scan single mode / repeat mode | | | |
|---|---------------|--------------------|-----------------------------|
| Unit | Start channel | Scan channel range | Storage register |
| TSB_AD | ADC_AN_00 | 15 channels | ADC_REG_00 to ADC_REG_14 |

| | | | |
|-----------|-------------|--------------------------|----|
| ADC_AN_01 | 14 channels | ADC_REG_01 ADC_REG_14 | to |
| ADC_AN_02 | 13 channels | ADC_REG_02 ADC_REG_14 | to |
| ADC_AN_03 | 12 channels | ADC_REG_03 ADC_REG_15 | to |
| ADC_AN_04 | 11 channels | ADC_REG_04 ADC_REG_14 | to |
| ADC_AN_05 | 10 channels | ADC_REG_05 ADC_REG_14 | to |
| ADC_AN_06 | 9 channels | ADC_REG_06 ADC_REG_14 | to |
| ADC_AN_07 | 8 channels | ADC_REG_07 ADC_REG_14 | to |
| ADC_AN_08 | 7 channels | ADC_REG_08 ADC_REG_14 | to |
| ADC_AN_09 | 6 channels | ADC_REG_09 ADC_REG_14 | to |
| ADC_AN_10 | 5 channels | ADC_REG_10 ADC_REG_14 | to |
| ADC_AN_11 | 4 channels | ADC_REG_11 ADC_REG_14 | to |
| ADC_AN_12 | 3 channels | ADC_REG_12 ADC_REG_14 | to |
| ADC_AN_13 | 2 channels | ADC_REG_13 ADC_REG_14 | to |
| ADC_AN_14 | 1 channels | ADC_REG_14 | |

About the ADC mode setting, please refer to relate APIs.

***Note:**

1 For top-priority AD conversion, the result is stored in "ADC_REG_SP".

2 For 12-bit extension A/D conversion, the result is stored in "ADC_REG_15".

Return: AD conversion result:

ADResult (Bit 0 to Bit 11) : store AD result value.

Stored (Bit 12) : '1' means AD result has been stored. It will be cleared if this register is read.

OverRun (Bit 13) '1' means new AD result overwrote the old one. It will be cleared if this register is read.

3.2.3.20 ADC_EnableTrigger

Enable the trigger.

Prototype:

```
void
ADC_EnableTrigger(void)
```

Parameters:

None

Description:

This function will enable the trigger

Return:

None

3.2.3.21 ADC_DisableTrigger

Disable the trigger.

Prototype:

void
ADC_DisableTrigger(void)

Parameters:

None

Description:

This function will disable the trigger

Return:

None

3.2.3.22 ADC_SetTriggerStartup

Selects a trigger for startup of normal AD conversion

Prototype:

void
ADC_SetTriggerStartup(ADC_TRGx *TriggerStartup*)

Parameters:

TriggerStartup: trigger for startup of normal AD conversion

This parameter can be one of the following values:

- ADC_TRG_00, ADC_TRG_01, ADC_TRG_02, ADC_TRG_03
- ADC_TRG_04, ADC_TRG_05, ADC_TRG_06, ADC_TRG_07
- ADC_TRG_08, ADC_TRG_09

Description:

This function will select a trigger for startup of normal AD conversion

Return:

None

3.2.3.23 ADC_SetTriggerStartupTop

Selects a trigger for startup of top-priority AD conversion

Prototype:

void
ADC_SetTriggerStartupTop(ADC_TRGx *TopTriggerStartup*)

Parameters:

TopTriggerStartup: trigger for startup of top-priority AD conversion

This parameter can be one of the following values:

- ADC_TRG_00, ADC_TRG_01, ADC_TRG_02, ADC_TRG_03
- ADC_TRG_04, ADC_TRG_05, ADC_TRG_06, ADC_TRG_07
- ADC_TRG_08, ADC_TRG_09

Description:

This function will select a trigger for startup of top-priority AD conversion

Return:
None

3.2.3.24 ADC_DisableExtension

Disable extension ADC channel

Prototype:
void
ADC_DisableExtension(void)

Parameters:
None

Description:
This function will disable extension ADC channel

Return:
None

3.2.3.25 ADC_EnableExtension

Enable extension ADC channel

Prototype:
void
ADC_EnableExtension(void)

Parameters:
None

Description:
This function will enable extension ADC channel

Return:
None

3.2.3.26 ADC_SetExtChannel

Select the extension ADC channel

Prototype:
void
ADC_SetExtChannel(ADC_EXT_AINx *ExtChannel*)

Parameters:
ExtChannel: The extension ADC channel
This parameter can be one of the following values:
➤ ADC_EXT_AN_15, ADC_EXT_AN_16, ADC_EXT_AN_17,
➤ ADC_EXT_AN_18, ADC_EXT_AN_19

Description:
This function will select the extension ADC channel

Return:
None

3.2.3.27 ADC_SetExtClkSource

Select the sample hold time source of extension ADC channel

Prototype:
void
ADC_SetExtClkSource(uint32_t **ClkSrc**)

Parameters:
ClkSrc: The sample hold time source of extension ADC channel.
This parameter can be one of the following values:

- **ADC_EXT_CLK_SRC_EXCR:** The source is the unique source of extension ADC channel
- **ADC_EXT_CLK_SRC_ADCLK:** The source is the normal source of ADC channel

Description:
This function will select the sample hold time source of extension ADC channel

Return:
None

3.2.3.28 ADC_SetExtClk

Select the sample hold time when the source is unique source

Prototype:
void
ADC_SetExtClk (uint32_t **Sample_HoldTime**)

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

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

Description:
This function will select the sample hold time source of extension ADC channel

Return:
None

3.2.4 Data Structure Description

3.2.4.1 ADC_MonitorTypeDef

Data Fields:

ADC_AINx
CmpChannel : Select which ADC channel will be used.

It can be:

ADC_AN_00 to ADC_AN_14 and ADC_AN_EXT (16 channels)

uint32_t

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

ADC_CmpCondition

Condition Condition to compare ADC channel with Compare Register , which can be:

- **ADC_LARGER_THAN_CMP_REG**: If the value of the conversion result register is bigger than the comparison register 0, an interrupt is generated.
- **ADC_SMALLER_THAN_CMP_REG**: If the value of the conversion result register is smaller than the comparison register 0, an interrupt is generated.

ADC_CmpCntMode

CntMode Mode to compare ADC channel with Compare Register, which can be:

- **ADC_SEQUENCE_CMP_MODE**: Sequence mode.
- **ADC_CUMULATION_CMP_MODE**: Cumulation mode.

uint32_t

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

(Note: please refer to part “AD monitor function” in datasheet for more detail usage information)

3.2.4.2 ADC_State

Data Fields for this union:

uint32_t

All specifies AD conversion state.

Bit Fields:

uint32_t

NormalBusy(Bit 0) Normal A/D conversion busy flag (ADBF).
'1' means conversion is busy

uint32_t

NormalComplete (Bit 1) Normal AD conversion complete flag (EOCF).
'1' means conversion is completed

uint32_t

TopBusy(Bit 2) Top-priority A/D conversion busy flag (HPADBF). '1'
means conversion is busy

uint32_t

TopComplete (Bit 3) Top-priority AD conversion complete flag (HPEOCF). '1'
means conversion is completed

uint32_t

Reserved (Bit 4 to Bit 31) reserved.

3.2.4.3 ADC_Result

Data Fields for this union:

uint32_t

All specifies AD conversion result.

Bit Fields:

uint32_t

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

uint32_t

Stored (Bit 12) '1' means AD result has been stored.

uint32_t

OverRun (Bit 13) '1' means new AD result overwrote the old one.

uint32_t

Reserved (Bit 14 to Bit 31) reserved.

4. CEC

4.1 Overview

This IP enables to transmit or receive data that conforms to Consumer Electronics Control protocol (conforms to HDMI 1.3a specifications).

TOSHIBA TMPM461x has a CEC module with 1 channel.

Reception

- Clock sampling at fs clock or TBxOUT which is output of 16bit Timer/Event counters
 1. Adjustable noise canceling time
 2. Data reception per 1byte
- Flexible data sampling point
 1. Data reception is available even when an address discrepancy is detected.
- Error detection
 1. Cycle error(min. / max)
 2. ACK collision
 3. Waveform error

Transmission

- Data transmission per 1byte
 1. Triggered by auto-detection of bus free state
- Flexible waveform
 1. Adjustable rising edge and cycle
- Error detection
 1. Arbitration lost
 2. ACK response error

All driver APIs are contained in /Libraries/TX04_Periph_Driver/src/tmpm461_cec.c, with /Libraries/TX04_Periph_Driver/inc/tmpm461_cec.h containing the macros, data types, structures and API definitions for use by applications.

4.2 API Functions

4.2.1 Function List

- ◆ void CEC_Enable(void)
- ◆ void CEC_Disable(void)
- ◆ void CEC_SWReset(void)
- ◆ Result CEC_DefaultConfig(void)
- ◆ Result CEC_SetLogicalAddr(CEC_LogicalAddr **LogicalAddr**)
- ◆ Result CEC_AddLogicalAddr(CEC_LogicalAddr **LogicalAddr**)
- ◆ Result CEC_RemoveLogicalAddr(CEC_LogicalAddr **LogicalAddr**)
- ◆ CEC_AddrListTypeDef CEC_GetLogicalAddr(void)
- ◆ void CEC_SetRxCtrl(FunctionalState **NewState**)
- ◆ Result CEC_StartTx(void)
- ◆ void CEC_StopTx(void)
- ◆ CEC_DataTypeDef CEC_GetRxData(void)
- ◆ void CEC_SetTxData(uint8_t **Data**,CEC_EOMBit **EOM_Flag**)
- ◆ FunctionalState CEC_GetRxState(void)
- ◆ WorkState CEC_GetTxState(void)
- ◆ CEC_RxINTState CEC_GetRxINTState(void)
- ◆ CEC_TxINTState CEC_GetTxINTState(void)
- ◆ Result CEC_SetACKResponseMode(FunctionalState **NewState**)

- ◆ Result CEC_SetNoiseCancellation(CEC_LowCancellation **LowCancellation**,
CEC_HighCancellation **HighCancellation**)
- ◆ Result CEC_SetCycleConfig(CEC_CycleMin **CycleMin**, CEC_CycleMax **CycleMax**)
- ◆ Result CEC_SetDataValidTime(CEC_ValidTime **ValidTime**)
- ◆ Result CEC_SetTimeOutMode(CEC_TimeOut **TimeOut**);
- ◆ Result CEC_SetRxErrINTSuspend(FunctionalState **NewState**)
- ◆ Result CEC_SetSnoopMode(FunctionalState **NewState**)
- ◆ Result CEC_SetRxDetectWaveConfig (
CEC_Logical1RisingTimeMin **Logical1RisingTimeMin**,
CEC_Logical1RisingTimeMax **Logical1RisingTimeMax**,
CEC_Logical0RisingTimeMin **Logical0RisingTimeMin**,
CEC_Logical0RisingTimeMax **Logical0RisingTimeMax**)
- ◆ Result CEC_SetRxStartBitWaveConfig(
CEC_StartBitRisingTimeMin **RisingTimeMin**,
CEC_StartBitRisingTimeMax **RisingTimeMax**,

CEC_StartBitCycleMin **CycleMin**,
CEC_StartBitCycleMax **CycleMax**)
- ◆ Result CEC_SetRxWaveErrDetect(FunctionalState **NewState**)
- ◆ Result CEC_SetTxWaveConfig(
CEC_TxDataBitCycle **DataBitCycle**,
CEC_TxDataBitRisingTime **DataBitRisingTime**,
CEC_TxStartBitCycle **StartBitCycle**,
CEC_TxStartBitRisingTime **StartBitRisingTime**)
- ◆ Result CEC_SetTxBroadcast(FunctionalState **NewState**)
- ◆ Result CEC_SetBusFreeTime(CEC_BusFree **BusFree**)
- ◆ Result CEC_SetRxStartBitDetect(FunctionalState **NewState**)
- ◆ void CEC_SetSamplingClk(CEC_SamplingClockSrc **ClkSrc**)

4.2.2 Detailed Description

Functions listed above can be divided into three parts:

- 1) Initialize and configure the common functions of CEC are handled by
CEC_Enable(), CEC_Disable(), CEC_DefaultConfig(), CEC_SetLogicalAddr(),
CEC_AddLogicalAddr(), CEC_RemoveLogicalAddr(), CEC_GetLogicalAddr(),
CEC_SetACKResponseMode(), CEC_SetNoiseCancellation(), CEC_SetCycleConfig(),
CEC_SetDataValidTime(), CEC_SetTimeOutMode(), CEC_SetRxErrINTSuspend(),
CEC_SetSnoopMode(), CEC_SetRxDetectWaveConfig(),
CEC_SetRxStartBitWaveConfig(), CEC_SetRxWaveErrDetect(),
CEC_SetTxWaveConfig(), CEC_SetTxBroadcast(), CEC_SetBusFreeTime(),
CEC_SetRxStartBitDetect() and CEC_SetSamplingClk().
- 2) Transfer control and error check of CEC are handled by
CEC_SetRxCtrl(), CEC_StartTx(), CEC_StopTx(), CEC_GetRxData(),
CEC_SetTxData(), CEC_GetRxState(), CEC_GetTxState(), CEC_GetRxINTState() and
CEC_GetTxINTState().
- 3) CEC_SWReset() handle other specified functions.

4.2.3 Function Documentation

4.2.3.1 CEC_Enable

Enable CEC module.

Prototype:

void
CEC_Enable(void)

Parameters:
None

Description:
This function will enable the CEC module. The CEC module should be enabled before using.
Register CECEN is modified by this function.

Return:
None

4.2.3.2 CEC_Disable

Disable CEC module.

Prototype:
void
CEC_Disable(void)

Parameters:
None

Description:
Disable CEC module. When the CEC operation is disabled, no clocks are supplied to the CEC module except for the enable register. Thus power consumption can be reduced. When CEC is disabled after it was enabled, each register setting is maintained.
Register CECEN is modified by this function.

Return:
None

4.2.3.3 CEC_SWReset

Reset the CEC module.

Prototype:
void
CEC_SWReset(void)

Parameters:
None

Description:
Stop all the CEC operation and initializes the register.
The function affects as follows:
Reception: Stops immediately. The received data is discarded.
Transmission (including the CEC line): Stops immediately.
Register: All the registers other than CECEN are initialized.
Please note that software reset during transmission may cause the CEC line waveform that does not identical to the defined.

After calling this function, user can call the functions **CEC_GetRxState()** (should return **DISABLE**) and **CEC_GetTxState()** (should return **CEC_TX_STOPED**) to check reset finish or not.

CECRESET is modified by this function.

Return:

None

4.2.3.4 CEC_DefaultConfig

Initialize the CEC in the default configuration.

Prototype:

Result

CEC_DefaultConfig(void)

Parameters:

None

Description:

Initialize the CEC in the default configuration:

Idle Mode: on

Noise Cancellation Time: H: 1 cycle L: 1 cycle

Cycle Range: 2.05ms~2.75ms

Data Valid Time: 1.05ms

Time Out: 1 Bit

Rx Start Wave configure: Min of start: 3.5ms; Max of start: 3.9ms

Min of cycle: 4.3ms; Max of cycle: 4.7ms

Receive Bit Wave configure: Min of "1": 0.4ms; Max of "1": 0.8ms

Min of "0":1.3ms; Max of "0": 1.7

Send Bit Wave configure: RV

Bus free configure: 5 bit cycle

Snoop mode: On

When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.

Registers CECEN, CECADD, CECRCR1, CECRCR2, CECRCR3 and CECTCR are modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.5 CEC_SetLogicalAddr

Specify the logical address assigned to CEC.

Prototype:

Result

CEC_SetLogicalAddr(CEC_LogicalAddr **LogicalAddr**)

Parameters:

LogicalAddr: the logical address of CEC

This parameter can be one of the following values:

CEC_TV, CEC_RECORDING_DEVICE_1, CEC_RECORDING_DEVICE_2,
CEC_STB_1, CEC_DVD_1, CEC_AUDIO_SYSTEM, CEC_STB_2, CEC_STB_3,
CEC_DVD_2, CEC_RECORDING_DEVICE_3, CEC_FREE_USE,
CEC_BROADCAST

Description:

Specify the logical address assigned to CEC. After call this function, the old logical address setting will be cleared and set the new one in the register.

When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.

Register CECADD is modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.6 CEC_AddLogicalAddr

Add new logical address to CEC device.

Prototype:

Result

CEC_AddLogicalAddr(CEC_LogicalAddr *LogicalAddr*)

Parameters:

LogicalAddr: the logical address of CEC.

This parameter can be one of the following values:

CEC_TV, CEC_RECORDING_DEVICE_1, CEC_RECORDING_DEVICE_2,
CEC_STB_1, CEC_DVD_1, CEC_AUDIO_SYSTEM, CEC_STB_2, CEC_STB_3,
CEC_DVD_2, CEC_RECORDING_DEVICE_3, CEC_FREE_USE,
CEC_BROADCAST

Description:

Add one new logical address to CEC. After calling this function, the old logical address setting will be remain and just add a new address.

When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.

Register CECADD is modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.7 CEC_RemoveLogicalAddr

Remove one of logical address from CEC device.

Prototype:

Result

CEC_RemoveLogicalAddr(CEC_LogicalAddr *LogicalAddr*)

Parameters:

LogicalAddr is the logical address of CEC.

This parameter can be one of the following values:

CEC_TV, CEC_RECORDING_DEVICE_1, CEC_RECORDING_DEVICE_2,
CEC_STB_1, CEC_DVD_1, CEC_AUDIO_SYSTEM, CEC_STB_2, CEC_STB_3,
CEC_DVD_2, CEC_RECORDING_DEVICE_3, CEC_FREE_USE,
CEC_BROADCAST

Description:

Remove one of logical address from CEC device.

After calling this function, other old logical address setting will be remained.

When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.

Register CECADD is modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.8 CEC_GetLogicalAddr

Get the logical address of device.

Prototype:

CEC_AddrListTypeDef

CEC_GetLogicalAddr(void)

Parameters:

None

Description:

Get the logical address of device include the number of logical address and the address list.

Return:

The logical address list of device:

CEC_AddrListTypeDef: The structure of the logical address list (refer to "4.3.4 Data Structure Description" for details).

4.2.3.9 CEC_SetRxCtrl

Enable/Disable data reception of CEC.

Prototype:

void

CEC_SetRxCtrl(FunctionalState **NewState**)

Parameters:

NewState: New state of the data reception function.

- **ENABLE** : enable reception function
- **DISABLE** : disable reception function

Description:

Control the reception operation of CEC. It takes a little time to reflect the setting of the <CECREN> bit to the circuit. After calling this function, user can call the function **CEC_GetRxState()** to check enable/disable finish or not.

Register CECREN is modified by this function.

Return:

None

4.2.3.10 CEC_StartTx

Start data transmission of CEC.

Prototype:

Result
CEC_StartTx(void)

Parameters:

None

Description:

Start a frame data transmission of CEC.
When the transmission is in progress, the function will return **ERROR** and do nothing.
Register CECTEN is modified by this function.

Return:

SUCCESS means set successful.
ERROR means set failed and do nothing.

4.2.3.11 CEC_StopTx

Stop data transmission of CEC.

Prototype:

void
CEC_StopTx(void)

Parameters:

None

Description:

To stop transmission including the EOM bit that indicates "1". This generates a transmission completion interrupt.
Register CECTEN is modified by this function.

Return:

None

4.2.3.12 CEC_GetRxData

Read the data received.

Prototype:

CEC_DataTypeDef
CEC_GetRxData (void)

Parameters:

None

Description:

Read the received data. The data include the one byte of received data. The bit 7 is the MSB, the received ACK state and the received EOM state. The function should be called as soon as a reception interrupt is generated. The reception continues from the

first data block until the final data block which has the EOM bit indicating “1”. After detecting the final data block, CEC waits for the next start bit.

Return:

The received data from receive Buffer.

CEC_DataTypeDef: The structure of the data received (see Data Structure “4.3.4 Data Structure Description” for details).

4.2.3.13 CEC_SetTxData

Set the data to be sent.

Prototype:

```
void  
CEC_SetTxData(uint8_t Data,  
               CEC_EOMBit EOM_Flag)
```

Parameters:

Data: The 1 byte data to be sent.

EOM_Flag: Specify the EOM bit to transmit, which can be:

- **CEC_EOM** means last data of the frame.
- **CEC_NO_EOM** means not last data of the frame.

Description:

Set the data to be sent. The first byte of the frame should be set by this function before calling the function **CEC_StartTx()**. When CEC starts transmitting the first bit of a byte of data, a transmit interrupt is generated. Subsequent to the transmit interrupt, a byte of next data can be set to the transmit buffer by this function. Data transfer continues in the above sequence until **EOM_Bit** is set to CEC_EOM. Register CECTBUF is modified by this function.

Return:

None

4.2.3.14 CEC_GetRxState

Get the Receive state of CEC.

Prototype:

```
FunctionalState  
CEC_GetRxState (void)
```

Parameters:

None

Description:

Get the Receive state of CEC.

Return:

ENABLE means CEC reception is enabled.

DISABLE means CEC reception is disabled.

4.2.3.15 CEC_GetTxState

Get the transmission state of CEC

Prototype:

WorkState
CEC_GetTxState(void)

Parameters:

None

Description:

Get the transmission state of CEC.

Return:

BUSY means transmission is in progress.

DONE means transmission is completed or an interrupt is generated.

4.2.3.16 CEC_GetRxINTState

Get the receive interrupt state of CEC

Prototype:

CEC_RxINTState
CEC_GetRxINTState(void)

Parameters:

None

Description:

Get the receive interrupt state of CEC and return the result. Each bit is described as follow: **RxEnd**(Bit 0) means 1 byte of data reception is completed, **RxStartBit**(Bit 1) means a start bit is detected, **MAXCycleErr**(Bit 2) means The maximum cycle error detected, **MINCycleErr**(Bit 3) means the minimum cycle error detected, **ACKCollision**(Bit 4) means ACK collision detected, **BufOverrun**(Bit 5) means receive buffer overrun detected and **WaveformErr**(Bit 6) means Waveform error detected

Return:

State of Receive Interrupt, Each bit is described as below:

RxEnd(Bit 0) means 1 byte of data reception is completed,

RxStartBit(Bit 1) means a start bit is detected,

MAXCycleErr(Bit 2) means the maximum cycle error detected,

MINCycleErr(Bit 3) means the minimum cycle error detected,

ACKCollision(Bit 4) means ACK collision detected,

BufOverrun(Bit 5) means receive buffer overrun detected,

WaveformErr(Bit 6) means wave form error detected,

4.2.3.17 CEC_GetTxINTState

Get the transmission interrupt state of CEC.

Prototype:

CEC_TxINTState
CEC_GetTxINTState(void)

Parameters:

None

Description:

Get the transmission interrupt state of CEC and return the result. Each bit is described as follow:

TxStart(Bit 0) means 1 byte of data transmission is started, **TxEnd**(Bit 1) means data transmission including the EOM bit is completed, **ArbitrationLost**(Bit 2) means arbitration lost occurs, **ACKErr**(Bit 3) means ACK error detected and **BufUnderrun**(Bit 4) means transmit buffer underrun detected.

Return:

State of Transmit Interrupt, Each bit is described as follow:

TxStart(Bit 0) means 1 byte of data transmission is started,
TxEnd(Bit 1) means data transmission including the EOM bit is completed,
ArbitrationLost(Bit 2) means arbitration lost occurs,
ACKErr(Bit 3) means ACK error detected,
BufUnderrun(Bit 4) means transmit buffer underrun detected,

4.2.3.18 CEC_SetACKResponseMode

Set the ACK response mode of CEC.

Prototype:

Result

CEC_SetACKResponseMode(FunctionalState **NewState**)

Parameters:

NewState: New state of the ACK Response Mode.

- **ENABLE** : enable ACK Response Mode
- **DISABLE** : disable ACK Response Mode

Description:

Call this function enables you to specify if logical "0" is sent or not as an ACK response to the data block when destination address corresponds with the address set in the logical address register. The header block sends logical "0" as an ACK response regardless of the bit setting when detecting the addresses corresponding. Please refer to the "Preconfiguration (5) ACK Response of CEC" in the datasheet. When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.

Register CECRCR1<CEACKDIS> is modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.19 CEC_SetNoiseCancellation

Set the Noise Cancellation Mode of CEC.

Prototype:

Result

CEC_SetNoiseCancellation(CEC_LowCancellation **LowCancellation**,
CEC_HighCancellation **HighCancellation**)

Parameters:

LowCancellation: The number of “0” samplings for noise. This parameter can be one of the following values:

- **CEC_LOW_CANCELLATION_1:** 1 cycle,
- **CEC_LOW_CANCELLATION_2:** 2 cycles,
- **CEC_LOW_CANCELLATION_3:** 3 cycles,
- **CEC_LOW_CANCELLATION_4:** 4 cycles,

HighCancellation: The number of “1” samplings for noise Cancellation. This parameter can be one of the following values:

- **CEC_HIGH_CANCELLATION_1:** 1 cycle,
- **CEC_HIGH_CANCELLATION_2:** 2 cycles,
- **CEC_HIGH_CANCELLATION_3:** 3 cycles,
- **CEC_HIGH_CANCELLATION_4:** 4 cycles.

Description:

The noise cancellation time is configurable with the **LowCancellation** and **HighCancellation** by this function. You can configure the time to detect “1” and “0” respectively. It is considered as noise if “1”s or “0”s of the same number as the specified value are not sampled. Please refer to the “Preconfiguration (2) Noise Cancellation Time of CEC” in the datasheet.

When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.

Registers CECRCR1< CECHNC>, CECRCR1< CECLNC> are modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.20 CEC_SetCycleConfig

Set the cycle error detected configuration of CEC

Prototype:

Result

CEC_SetCycleConfig(CEC_CycleMin **CycleMin**,
CEC_CycleMax **CycleMax**)

Parameters:

CycleMin: Time to identify as min. cycle error. This parameter can be one of the following values:

- **CEC_CYCLE_MIN_0:** 2.05ms,
- **CEC_CYCLE_MIN_1:** 2.05ms+1cycle,
- **CEC_CYCLE_MIN_2:** 2.05ms+2cycles,
- **CEC_CYCLE_MIN_3:** 2.05ms+3cycles,
- **CEC_CYCLE_MIN_4:** 2.05ms-1cycles,
- **CEC_CYCLE_MIN_5:** 2.05ms-2cycles,
- **CEC_CYCLE_MIN_6:** 2.05ms-3cycles,
- **CEC_CYCLE_MIN_7:** 2.05ms-4cycles.

CycleMax: Time to identify as max. cycle error. This parameter can be one of the following

- **CEC_CYCLE_MAX_0:** 2.75ms,
- **CEC_CYCLE_MAX_1:** 2.75ms+1cycles,
- **CEC_CYCLE_MAX_2:** 2.75ms+2cycles,
- **CEC_CYCLE_MAX_3:** 2.75ms+3cycles,
- **CEC_CYCLE_MAX_4:** 2.75ms-1cycles,
- **CEC_CYCLE_MAX_5:** 2.75ms-2cycles,

- **CEC_CYCLE_MAX_6**: 2.75ms-3cycles,
- **CEC_CYCLE_MAX_7**: 2.75ms-4cycles.

Description:

Call this function to detect a cycle error.

You can specify the time to detect a cycle error for each sampling clock cycle between the ranges of -4 to +3 cycles from the maximum or minimum time set in the CEC standard. Detecting an error during data reception causes an error interrupt, and CEC waits for the next start bit. The received data is discarded.

When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.

Registers **CECR1< CECMIN>**, **CECR1< CECMAX>** are modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.21 CEC_SetDataValidTime

Specify the point of determining the data as 0 or 1.

Prototype:

Result

CEC_SetDataValidTime(CEC_ValidTime *ValidTime*)

Parameters:

ValidTime: Point of determining the data as 0 or 1. This parameter can be one of the following values:

- **CEC_VALID_TIME_0**: 1.05ms,
- **CEC_VALID_TIME_1**: 1.05ms+2cycles,
- **CEC_VALID_TIME_2**: 1.05ms+4cycles,
- **CEC_VALID_TIME_3**: 1.05ms+6cycles,
- **CEC_VALID_TIME_4**: 1.05ms-2cycles,
- **CEC_VALID_TIME_5**: 1.05ms-4cycles,
- **CEC_VALID_TIME_6**: 1.05ms-6cycles.

Description:

Call this function for configuring the point of determining the data as “0” or “1”.

You can specify it per two sampling clock cycles between the ranges of + or - 6 cycles with approx. 1.05 ms from the bit start point. Please refer to the “Preconfiguration (4) Point of Determining Data Time of CEC” in the datasheet.

When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.

Register **CECR1< CECDAT>** is modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.22 CEC_SetTimeOutMode

Specify the time to determine a timeout.

Prototype:

Result

CEC_SetTimeOutMode(CEC_TimeOut *TimeOut*)

Parameters:

TimeOut: The Number of Cycle to identify time out. This parameter can be one of the following values:

- **CEC_TIME_OUT_1_BIT:** 1 bit cycle,
- **CEC_TIME_OUT_2_BIT:** 2 bit cycle,
- **CEC_TIME_OUT_3_BIT:** 3 bit cycle.

Description:

Call this function to specify the time to determine a time out.

This is used when the setting of a receive error interrupt suspension, which is specified in set error interrupt is suspended.

When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.

Register CECRCR1< CECTOUT> is modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.23 CEC_SetRxErrINTSuspend

Specify if a receive error interrupt is suspended or not.

Prototype:

Result

CEC_SetRxErrINTSuspend(FunctionalState *NewState*)

Parameters:

NewState: The state of a receive error interrupt is suspended or not.

- **ENABLE:** enable the error interrupt
- **DISABLE:** disable the error interrupt

Description:

Call this function to specify if a reception error interrupt (maximum cycle error, buffer overrun and waveform error) is suspended or not. Setting **ENABLE** generates no interrupt at the error detection.

When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.

Register CECRCR1< CECRIHLD> is modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.24 CEC_SetSnoopMode

Specify if data is received or not when destination address does not correspond with the address set in the logical address register.

Prototype:

Result

CEC_SetSnoopMode(FunctionalState *NewState*)

Parameters:

NewState: The state of snoop mode enable or not.

- **ENABLE:** enable snoop mode
- **DISABLE:** disable snoop mode

Description:

By calling this function, you can specify whether the data should be received or not when destination address does not correspond with the address set in the logical address register. In this case, the data is received as usual, and an interrupt is generated by detecting an error. However, an ACK response of neither the header block nor the data block is sent. A broadcast message is received regardless of the Snoop Mode.

When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.

Register CECRCR1< CECOTH> is modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.25 CEC_SetRxDetectWaveConfig

Specify waveform error detection range

Prototype:

Result

```
CEC_SetRxDetectWaveConfig(  
    CEC_Logical1RisingTimeMin Logical1RisingTimeMin,  
    CEC_Logical1RisingTimeMax Logical1RisingTimeMax,  
    CEC_Logical0RisingTimeMin Logical0RisingTimeMin,  
    CEC_Logical0RisingTimeMax Logical0RisingTimeMax)
```

Parameters:

Logical1RisingTimeMin: The fastest rising timing of logical “1” determined as proper.

This parameter can be one of the following values:

- **CEC_LOGICAL_1_RISING_TIME_MIN_0:** 0.4ms,
- **CEC_LOGICAL_1_RISING_TIME_MIN_1:** 0.4ms-1cycle,
- **CEC_LOGICAL_1_RISING_TIME_MIN_2:** 0.4ms-2cycles,
- **CEC_LOGICAL_1_RISING_TIME_MIN_3:** 0.4ms-3cycles,
- **CEC_LOGICAL_1_RISING_TIME_MIN_4:** 0.4ms-4cycles,
- **CEC_LOGICAL_1_RISING_TIME_MIN_5:** 0.4ms-5cycles,
- **CEC_LOGICAL_1_RISING_TIME_MIN_6:** 0.4ms-6cycles,
- **CEC_LOGICAL_1_RISING_TIME_MIN_7:** 0.4ms-7cycles.

Logical1RisingTimeMax: The latest rising timing of logical “1” determined as proper waveform. This parameter can be one of the following values:

- **CEC_LOGICAL_1_RISING_TIME_MAX_0:** 0.8ms,
- **CEC_LOGICAL_1_RISING_TIME_MAX_1:** 0.8ms+1cycle,
- **CEC_LOGICAL_1_RISING_TIME_MAX_2:** 0.8ms+2cycles,
- **CEC_LOGICAL_1_RISING_TIME_MAX_3:** 0.8ms+3cycles,
- **CEC_LOGICAL_1_RISING_TIME_MAX_4:** 0.8ms+4cycles,
- **CEC_LOGICAL_1_RISING_TIME_MAX_5:** 0.8ms+5cycles,
- **CEC_LOGICAL_1_RISING_TIME_MAX_6:** 0.8ms+6cycles,
- **CEC_LOGICAL_1_RISING_TIME_MAX_7:** 0.8ms+7cycles.

Logical0RisingTimeMin: The fastest rising timing of logical “0” determined as proper.

This parameter can be one of the following values:

- **CEC_LOGICAL_0_RISING_TIME_MIN_0:** 1.3ms,
- **CEC_LOGICAL_0_RISING_TIME_MIN_1:** 1.3ms -1cycle,

- **CEC_LOGICAL_0_RISING_TIME_MIN_2**: 1.3ms -2cycles,
- **CEC_LOGICAL_0_RISING_TIME_MIN_3**: 1.3ms -3cycles,
- **CEC_LOGICAL_0_RISING_TIME_MIN_4**: 1.3ms -4cycles,
- **CEC_LOGICAL_0_RISING_TIME_MIN_5**: 1.3ms -5cycles,
- **CEC_LOGICAL_0_RISING_TIME_MIN_6**: 1.3ms -6cycles,
- **CEC_LOGICAL_0_RISING_TIME_MIN_7**: 1.3ms -7cycles.

Logical0RisingTimeMax: The latest rising timing of logical “0” determined as proper waveform. This parameter can be one of the following values:

- **CEC_LOGICAL_0_RISING_TIME_MAX_0**: 1.7ms,
- **CEC_LOGICAL_0_RISING_TIME_MAX_1**: 1.7ms +1cycle,
- **CEC_LOGICAL_0_RISING_TIME_MAX_2**: 1.7ms +2cycles,
- **CEC_LOGICAL_0_RISING_TIME_MAX_3**: 1.7ms +3cycles,
- **CEC_LOGICAL_0_RISING_TIME_MAX_4**: 1.7ms +4cycles,
- **CEC_LOGICAL_0_RISING_TIME_MAX_5**: 1.7ms +5cycles,
- **CEC_LOGICAL_0_RISING_TIME_MAX_6**: 1.7ms +6cycles,
- **CEC_LOGICAL_0_RISING_TIME_MAX_7**: 1.7ms +7cycles.

Description:

Call this function to detect an error when a received waveform is out of the defined tolerance range,

You can specify the detection time *with Logical1RisingTimeMin,*

Logical1RisingTimeMax, Logical0RisingTimeMin and *Logical0RisingTimeMax.*

Please refer to the “Preconfiguration (10) Waveform Error Detection of CEC” in the datasheet.

When the CEC reception is enabled or the transmission is in progress, the function will return ERROR and do nothing.

Register CECRCR3 is modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.26 CEC_SetRxStartBitWaveConfig

Specify the cycles to detect a start bit and the rising timing of a start bit in its detection.

Prototype:

Result

```
CEC_SetRxStartBitWaveConfig(  
    CEC_StartBitRisingTimeMin RisingTimeMin,  
    CEC_StartBitRisingTimeMax RisingTimeMax,  
    CEC_StartBitCycleMin CycleMin,  
    CEC_StartBitCycleMax CycleMax)
```

Parameters:

RisingTimeMin: Min. time of start bit rising timing. This parameter can be one of the following values:

- **CEC_START_BIT_RISING_TIME_MIN_0**: 3.5ms,
- **CEC_START_BIT_RISING_TIME_MIN_1**: 3.5ms-1cycle,
- **CEC_START_BIT_RISING_TIME_MIN_2**: 3.5ms-2cycles,
- **CEC_START_BIT_RISING_TIME_MIN_3**: 3.5ms-3cycles,
- **CEC_START_BIT_RISING_TIME_MIN_4**: 3.5ms-4cycles,
- **CEC_START_BIT_RISING_TIME_MIN_5**: 3.5ms-5cycles,
- **CEC_START_BIT_RISING_TIME_MIN_6**: 3.5ms-6cycles,
- **CEC_START_BIT_RISING_TIME_MIN_7**: 3.5ms-7cycles.

RisingTimeMax: Max. time of start bit rising timing. This parameter can be one of the following values:

- **CEC_START_BIT_RISING_TIME_MAX_0:** 3.9ms,
- **CEC_START_BIT_RISING_TIME_MAX_1:** 3.9ms+1cycle,
- **CEC_START_BIT_RISING_TIME_MAX_2:** 3.9ms+2cycles,
- **CEC_START_BIT_RISING_TIME_MAX_3:** 3.9ms+3cycles,
- **CEC_START_BIT_RISING_TIME_MAX_4:** 3.9ms+4cycles,
- **CEC_START_BIT_RISING_TIME_MAX_5:** 3.9ms+5cycles,
- **CEC_START_BIT_RISING_TIME_MAX_6:** 3.9ms+6cycles,
- **CEC_START_BIT_RISING_TIME_MAX_7:** 3.9ms+7cycles.

CycleMin: Min. cycle to detect start bit. This parameter can be one of the following values:

- **CEC_START_BIT_CYCLE_MIN_0:** 4.3ms,
- **CEC_START_BIT_CYCLE_MIN_1:** 4.3ms-1cycle,
- **CEC_START_BIT_CYCLE_MIN_2:** 4.3ms -2cycles,
- **CEC_START_BIT_CYCLE_MIN_3:** 4.3ms -3cycles,
- **CEC_START_BIT_CYCLE_MIN_4:** 4.3ms -4cycles,
- **CEC_START_BIT_CYCLE_MIN_5:** 4.3ms -5cycles,
- **CEC_START_BIT_CYCLE_MIN_6:** 4.3ms -6cycles,
- **CEC_START_BIT_CYCLE_MIN_7:** 4.3ms -7cycles.

CycleMax: Max. cycle to detect start bit. This parameter can be one of the following values:

- **CEC_START_BIT_CYCLE_MAX_0:** 4.7ms,
- **CEC_START_BIT_CYCLE_MAX_1:** 4.7ms+1cycle,
- **CEC_START_BIT_CYCLE_MAX_2:** 4.7ms +2cycles,
- **CEC_START_BIT_CYCLE_MAX_3:** 4.7ms +3cycles,
- **CEC_START_BIT_CYCLE_MAX_4:** 4.7ms +4cycles,
- **CEC_START_BIT_CYCLE_MAX_5:** 4.7ms +5cycles,
- **CEC_START_BIT_CYCLE_MAX_6:** 4.7ms +6cycles,
- **CEC_START_BIT_CYCLE_MAX_7:** 4.7ms +7cycles.

Description:

Calling this function allows you to specify the rising timing and a cycle of the start bit detection respectively.

RisingTimeMin is to specify the fastest start bit rising timing. **RisingTimeMax** is to specify the latest start bit rising. **CycleMin** is to specify the minimum cycle of a start bit. **CycleMax** is to specify the maximum cycle of a start bit.

Please refer to the “Preconfiguration (9) Start Bit Detection of CEC” in the datasheet. When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.

Register CECRCR2 is modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.27 CEC_SetRxWaveErrDetect

Enable or disable the function that detects a received waveform does not identical to the one defined and generates waveform error interrupt.

Prototype:

Result

CEC_SetRxWaveErrDetect(FunctionalState **NewState**)

Parameters:

NewState: The state of Waveform error detection Mode.

- **ENABLE:** enable the waveform error detection mode.
- **DISABLE:** disable the waveform error detection mode.

Description:

Enable or disable the function that detects a received waveform does not identical to the one defined and generates waveform error interrupt.

When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.

Register CECRCR3< CECWAVEN> is modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.28 CEC_SetTxWaveConfig

Specify the waveform parameter for transmit

Prototype:

Result

```
CEC_SetTxWaveConfig(CEC_TxDataBitCycle DataBitCycle ,  
                    CEC_TxDataBitRisingTime DataBitRisingTime,  
                    CEC_TxStartBitCycle StartBitCycle,  
                    CEC_TxStartBitRisingTime StartBitRisingTime)
```

Parameters:

DataBitCycle: the cycle of a data bit. parameter can be one of the following values:

- **CEC_TX_DATA_BIT_CYCLE_0:** RV (Reference value: 2.4ms approx.),
- **CEC_TX_DATA_BIT_CYCLE_1:** RV-1cycle,
- **CEC_TX_DATA_BIT_CYCLE_2:** RV-2cycles,
- **CEC_TX_DATA_BIT_CYCLE_3:** RV-3cycles,
- **CEC_TX_DATA_BIT_CYCLE_4:** RV-4cycles,
- **CEC_TX_DATA_BIT_CYCLE_5:** RV-5cycles,
- **CEC_TX_DATA_BIT_CYCLE_6:** RV-6cycles,
- **CEC_TX_DATA_BIT_CYCLE_7:** RV-7cycles,
- **CEC_TX_DATA_BIT_CYCLE_8:** RV-8cycles,
- **CEC_TX_DATA_BIT_CYCLE_9:** RV-9cycles,
- **CEC_TX_DATA_BIT_CYCLE_10:** RV-10cycles,
- **CEC_TX_DATA_BIT_CYCLE_11:** RV-11cycles,
- **CEC_TX_DATA_BIT_CYCLE_12:** RV-12cycles,
- **CEC_TX_DATA_BIT_CYCLE_13:** RV-13cycles,
- **CEC_TX_DATA_BIT_CYCLE_14:** RV-14cycles,
- **CEC_TX_DATA_BIT_CYCLE_15:** RV-15cycles.

DataBitRisingTime: The rising timing of a data bit. parameter can be one of the following values:

- **CEC_TX_DATA_BIT_RISING_TIME_0:** RV (logical "1": 0.6 ms approx., logical "0": 1.5 ms approx),
- **CEC_TX_DATA_BIT_RISING_TIME_1:** RV-1cycle,
- **CEC_TX_DATA_BIT_RISING_TIME_2:** RV-2cycles,
- **CEC_TX_DATA_BIT_RISING_TIME_3:** RV-3cycles,

StartBitCycle: the cycle of a start bit. This parameter can be one of the following values:

- **CEC_TX_START_BIT_CYCLE_0:** RV (4.5ms approx.),
- **CEC_TX_START_BIT_CYCLE_1:** RV-1cycle,
- **CEC_TX_START_BIT_CYCLE_2:** RV-2cycles,
- **CEC_TX_START_BIT_CYCLE_3:** RV-3cycles,

- **CEC_TX_START_BIT_CYCLE_4**: RV-4cycles,
- **CEC_TX_START_BIT_CYCLE_5**: RV-5cycles,
- **CEC_TX_START_BIT_CYCLE_6**: RV-6cycles,
- **CEC_TX_START_BIT_CYCLE_7**: RV-7cycles.

StartBitRisingTime: the rising timing of a start bit. This parameter can be one of the following values:

- **CEC_TX_START_BIT_RISING_TIME_0**: RV (3.7ms approx.),
- **CEC_TX_START_BIT_RISING_TIME_1**: RV-1cycle,
- **CEC_TX_START_BIT_RISING_TIME_2**: RV-2cycles,
- **CEC_TX_START_BIT_RISING_TIME_3**: RV-3cycles,
- **CEC_TX_START_BIT_RISING_TIME_4**: RV-4cycles,
- **CEC_TX_START_BIT_RISING_TIME_5**: RV-5cycles,
- **CEC_TX_START_BIT_RISING_TIME_6**: RV-6cycles,
- **CEC_TX_START_BIT_RISING_TIME_7**: RV-7cycles.

Description:

Both start bit and data bit are capable of adjusting the rising timing and cycle. With the **DataBitCycle**, **DataBitRisingTime**, **StartBitCycle** and **StartBitRisingTime**, the timing can be specified between the defined fastest rising/cycle timing and the reference value.

Please refer to the “Preconfiguration (3) Adjusting Transmission Waveform of CEC” in the datasheet.

When the transmission is in progress, the function will return **ERROR** and do nothing. Register CECTCR is modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.29 CEC_SetTxBroadcast

Set message as a broadcast message or not.

Prototype:

Result

CEC_SetTxBroadcast(FunctionalState **NewState**)

Parameters:

NewState: The state of Broadcast mode.

- **ENABLE**: enable the Broadcast mode.
- **DISABLE**: disable the Broadcast mode.

Description:

Call this function and set **NewState** to **ENABLE** when transmitting a broadcast message. If **NewState** is **ENABLE**, “0” response during an ACK cycle results in an error. If **NewState** is **DISABLE**, “1” response during an ACK cycle results in an error. When the transmission is in progress, the function will return **ERROR** and do nothing. Register CECTCR< CECBRD> is modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.2.3.30 CEC_SetBusFreeTime

Specify time of a bus to be free that checked before transmission.

Prototype:

Result

CEC_SetBusFreeTime (CEC_BusFree *BusFree*)**Parameters:****BusFree:** the cycle of a data bit. This parameter can be one of the following values:

- CEC_BUS_FREE_1_BIT: 1 bit cycle,
- CEC_BUS_FREE_2_BIT: 2 bit cycles,
- CEC_BUS_FREE_3_BIT: 3 bit cycles,
- CEC_BUS_FREE_4_BIT: 4 bit cycles,
- CEC_BUS_FREE_5_BIT: 5 bit cycles,
- CEC_BUS_FREE_6_BIT: 6 bit cycles,
- CEC_BUS_FREE_7_BIT: 7 bit cycles,
- CEC_BUS_FREE_8_BIT: 8 bit cycles,
- CEC_BUS_FREE_9_BIT: 9 bit cycles,
- CEC_BUS_FREE_10_BIT: 10 bit cycles,
- CEC_BUS_FREE_11_BIT: 11 bit cycles,
- CEC_BUS_FREE_12_BIT: 12 bit cycles,
- CEC_BUS_FREE_13_BIT: 13 bit cycles,
- CEC_BUS_FREE_14_BIT: 14 bit cycles,
- CEC_BUS_FREE_15_BIT: 15 bit cycles,
- CEC_BUS_FREE_16_BIT: 16 bit cycles.

Description:

Configure the wait time for a bus to be free with calling this function. It can be specified cycle from 1 cycle to 16 cycles.

Start point to check if a bus is free is the end of final bit. If a bus is free for specified **CEC_BUS_FREE_1_BIT**, transmission starts. Please refer to the “Preconfiguration (1) Bus Free Wait Time of CEC” in the datasheet.

When the transmission is in progress, the function will return **ERROR** and do nothing. Register CECTCR< CECFREE> is modified by this function.

Return:**SUCCESS** means set successful.**ERROR** means set failed and do nothing.

4.2.3.31 CEC_SetRxStartBitDetect

Enable or disable the function that detects a reception of start bit and generates interrupt.

Prototype:

Result

CEC_SetRxStartBitDetect(FunctionalState *NewState*)**Parameters:****NewState:** The state of Start bit detection Mode.

- **ENABLE:** enable Rx start bit detection mode.
- **DISABLE:** disable Rx start bit detection mode.

Description:

Enable or disable the function that detects a reception of start bit and generates interrupt.

When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.
Register CECRCR3< CEC RSTAEN> is modified by this function.

Return:
SUCCESS means set successful.
ERROR means set failed and do nothing.

4.2.3.32 CEC_SetSamplingClk

Specify the sampling clock to CEC.

Prototype:
void
void CEC_SetSamplingClk(CEC_SamplingClockSrc **ClkSrc**)

Parameters:
ClkSrc: The sampling clock source of CEC. This parameter can be one of the following values:
➤ **CEC_SAMPLING_CLK_SRC_LOW_SPEED:** Low-speed clock (fs)
➤ **CEC_SAMPLING_CLK_SRC_TBxOUT:** TBxOUT

Description:
Specify the sampling clock to CEC.
Call this function first to select **ClkSrc** as the sampling clock after starting (permitting) the CEC operation.
Register FSSEL< CECCLK> is modified by this function.

Return:
None

4.2.4 Data Structure Description

4.2.4.1 CEC_DataTypeDef

Data Fields:
uint8_t
Data Reads one byte of data received. The bit 7 is the MSB.
CEC_ACKState
ACKBit The received ACK bit which can be:
➤ **CEC_ACK** means the ACK bit is "1"
➤ **CEC_NO_ACK** means the ACK bit is "0"

CEC_EOMBit
EOMBit The received EOM bit which can be:
➤ **CEC_EOM** means the EOM bit is "1"
➤ **CEC_NO_EOM** means the EOM bit is "0"

4.2.4.2 CEC_AddrListTypeDef

Data Fields:
uint8_t
AddrNumber is the number of logical address.
CEC_LogicalAddr

AddrList[16] is the logical address list of the CEC device.

Every item of the list can be one of the following values: CEC_TV,
CEC_RECORDING_DEVICE_1, CEC_RECORDING_DEVICE_2,CEC_STB_1,
CEC_DVD_1, CEC_AUDIO_SYSTEM, CEC_STB_2,CEC_STB_3,CEC_DVD_2,
CEC_RECORDING_DEVICE_3, CEC_FREE_USE, CEC_BROADCAST

5. EXB

5.1 Overview

The TMPM461x has a built-in external bus interface to connect to external memory, I/Os, etc. This interface consists of an external bus interface circuit (EBIF), a chip selector (CS) and a wait controller.

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

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

All driver APIs are contained in /Libraries/TX04_Periph_Driver/src/tmpm461_exb.c, with /Libraries/TX04_Periph_Driver/inc/tmpm461_exb.h containing the macros, data types, structures and API definitions for use by applications.

5.2 API Functions

5.2.1 Function List

- ◆ void EXB_SetBusMode(uint8_t **BusMode**);
- ◆ void EXB_SetBusCycleExtension(uint8_t **Cycle**);
- ◆ void EXB_Enable(uint8_t **ChipSelect**);
- ◆ void EXB_Disable(uint8_t **ChipSelect**);
- ◆ void EXB_Init(uint8_t **ChipSelect**, EXB_InitTypeDef* **InitStruct**);
- ◆ Result EXB_SetClkOutputDivision(uint8_t **ClkDiv**);
- ◆ void EXB_EnableClkOutput(void);
- ◆ void EXB_DisableClkOutput(void);
- ◆ FunctionalState EXB_GetClkOutputState(void);

5.2.2 Detailed Description

Functions listed above can be divided into three parts:

- 1) Configure the EXB bus mode, bus cycle extension, data bus widths and the cycle of external buses based on the chip selector.
EXB_SetBusMode(), EXB_SetBusCycleExtension() and EXB_Init().
- 2) Enable and disable control of EXB.
EXB_Enable(), EXB_Disable().
- 3) Configure the clock output of EXB or get the state of the clock output.
EXB_SetClkOutputDivision(), EXB_EnableClkOutput(), EXB_DisableClkOutput(), EXB_GetClkOutputState().

5.2.3 Function Documentation

5.2.3.1 EXB_SetBusMode

Set external bus mode for EXB.

Prototype:

void
EXB_SetBusMode(uint8_t **BusMode**)

Parameters:

BusMode : select EXB bus mode.

The value could be the following values:

- **EXB_BUS_MULTIPLEX** for multiplex bus mode.
- **EXB_BUS_SEPARATE** for separate bus mode

Description:

This function sets the bus mode for the external bus.

When **BusMode** is **EXB_BUS_MULTIPLEX**, the bus mode will be multiplex mode.

When **BusMode** is **EXB_BUS_SEPARATE**, the bus mode will be separate mode.

Return:

None

5.2.3.2 EXB_SetBusCycleExtension

Set the bus cycle to be double or quadruple.

Prototype:

void

EXB_SetBusCycleExtension(uint8_t **Cycle**)

Parameters:

Cycle: Set the bus cycle to be double or quadruple.

The value could be the following values:

- **EXB_CYCLE_NONE**: EXB bus cycle will not be extended.
- **EXB_CYCLE_DOUBLE**: EXB bus cycle will be double.
- **EXB_CYCLE_QUADRUPLE**: EXB bus cycle will be quadruple.

Description:

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

Return:

None

5.2.3.3 EXB_Enable

Enable the specified chip.

Prototype:

void

EXB_Enable(uint8_t **ChipSelect**)

Parameters:

ChipSelect is the specified chip.

The value could be the following values:

- **EXB_CS0**: for chip 0
- **EXB_CS1**: for chip 1
- **EXB_CS2**: for chip 2
- **EXB_CS3**: for chip 3

Description:

This function will enable the access to the specified chip.

Return:

None

5.2.3.4 EXB_Disable

Disable the specified chip.

Prototype:

void
EXB_Disable(uint8_t *ChipSelect*)

Parameters:

ChipSelect is the specified chip.
The value could be the following values:

- **EXB_CS0**: for chip 0
- **EXB_CS1**: for chip 1
- **EXB_CS2**: for chip 2
- **EXB_CS3**: for chip 3

Description:

This function will disable the access to the specified chip.

Return:

None

5.2.3.5 EXB_Init

Initialize the specified chip.

Prototype:

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

Parameters:

ChipSelect is the specified chip.
The value could be the following values:

- **EXB_CS0**: for chip 0
- **EXB_CS1**: for chip 1
- **EXB_CS2**: for chip 2
- **EXB_CS3**: for chip 3

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

Description:

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

Return:

None

5.2.3.6 EXB_SetClkOutputDivision

Set the clock division setting of EXB clock output.

Prototype:

Result
EXB_SetClkOutputDivision (uint8_t *ClkDiv*)

Parameters:

ClkDiv is the clock division setting.

The value could be the following values:

- **EXB_CLK_DIVISION_FSYS_2**: the clock division is fsys/2
- **EXB_CLK_DIVISION_FSYS_4**: the clock division is fsys/4
- **EXB_CLK_DIVISION_FSYS_8**: the clock division is fsys/8

Description:

This function will set the clock division setting of EXB clock output.

***Note:**

The clock output must be disable when call function **EXB_SetClkOutputDivision**

The return value will be ERROR when beyond above situation

Return:

SUCCESS: operation is finished successfully.

ERROR: operation is not done.

5.2.3.7 EXB_EnableClkOutput

Enable the EXB clock output..

Prototype:

void
EXB_EnableClkOutput(void)

Parameters:

None

Description:

This function will enable the EXB clock output.

Return:

None

5.2.3.8 EXB_DisableClkOutput

Disable the EXB clock output..

Prototype:

void
EXB_DisableClkOutput(void)

Parameters:

None

Description:

This function will disable the EXB clock output.

Return:

None

5.2.3.9 EXB_GetClkOutputState

Get the state of EXB clock output.

Prototype:

void
EXB_GetClkOutputState(void)

Parameters:

None

Description:

This function will get the state of EXB clock output.

Return:

The state of EXB clock output
ENABLE: EXB clock output is enabled.
DISABLE: EXB clock output is disabled.

5.2.4 Data Structure Description

5.2.4.1 EXB_InitTypeDef

Data Fields:

uint8_t

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

- **EXB_16M_BYTE:** address space is 16Mbyte,
- **EXB_8M_BYTE:** address space is 8Mbyte,
- **EXB_4M_BYTE:** address space is 4Mbyte,
- **EXB_2M_BYTE:** address space is 2Mbyte,
- **EXB_1M_BYTE:** address space is 1Mbyte,
- **EXB_512K_BYTE:** address space is 512Kbyte,
- **EXB_256K_BYTE:** address space is 256Kbyte,
- **EXB_128K_BYTE:** address space is 128Kbyte,
- **EXB_64K_BYTE:** address space is 64Kbyte.

uint8_t

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

uint8_t

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

- **EXB_BUS_WIDTH_BIT_8:** data bus width is 8bit,
- **EXB_BUS_WIDTH_BIT_16:** data bus width is 16bit.

EXB_CyclesTypeDef

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

InternalWait, **ReadSetupCycle**, **WriteSetupCycle**, **ALEWaitCycle** (For multiplex bus mode only), **ReadRecoveryCycle**, **WriteRecoveryCycle** and **ChipSelectRecoveryCycle**. (Refer to "EXB_CyclesTypeDef" for details)

uint8_t

WaitSignal Wait signal selection, which can be set as:

- **EXB_WAIT_SIGNAL_LOW:** Active of wait signal is "Low",
- **EXB_WAIT_SIGNAL_HIGH:** Active of wait signal is "High".

uint8_t

WaitFunction Wait function selection, which can be set as:

- **EXB_WAIT_FUNCTION_INT**: Wait function is internal wait,
- **EXB_WAIT_FUNCTION_EXT**: Wait function is external wait.

5.2.4.2 EXB_CyclesType Def

Data Fields:

uint8_t

Wait Selection of number of waits, which can be set as:

- **EXB_WAIT_0**: 0 wait,
- **EXB_WAIT_1**: 1 wait,
- **EXB_WAIT_2**: 2 waits,
- **EXB_WAIT_3**: 3 waits,
- **EXB_WAIT_4**: 4 waits,
- **EXB_WAIT_5**: 5 waits,
- **EXB_WAIT_6**: 6 waits,
- **EXB_WAIT_7**: 7 waits,
- **EXB_WAIT_8**: 8 waits,
- **EXB_WAIT_9**: 9 waits,
- **EXB_WAIT_10**: 10 waits,
- **EXB_WAIT_11**: 11 waits,
- **EXB_WAIT_12**: 12 waits,
- **EXB_WAIT_13**: 13 waits,
- **EXB_WAIT_14**: 14 waits,
- **EXB_WAIT_15**: 15 waits.

uint8_t

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

- **EXB_CYCLE_0**: 0 cycle,
- **EXB_CYCLE_1**: 1 cycle,
- **EXB_CYCLE_2**: 2 cycles,
- **EXB_CYCLE_4**: 4 cycles.

uint8_t

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

- **EXB_CYCLE_0**: 0 cycle,
- **EXB_CYCLE_1**: 1 cycle,
- **EXB_CYCLE_2**: 2 cycles,
- **EXB_CYCLE_4**: 4 cycles.

uint8_t

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

- **EXB_CYCLE_0**: 0 cycle,
- **EXB_CYCLE_1**: 1 cycle,
- **EXB_CYCLE_2**: 2 cycles,
- **EXB_CYCLE_4**: 4 cycles.

uint8_t

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

- **EXB_CYCLE_0**: 0 cycle,

- **EXB_CYCLE_1**: 1 cycle,
- **EXB_CYCLE_2**: 2 cycles,
- **EXB_CYCLE_3**: 3 cycles,
- **EXB_CYCLE_4**: 4 cycles,
- **EXB_CYCLE_5**: 5 cycles,
- **EXB_CYCLE_6**: 6 cycles,
- **EXB_CYCLE_8**: 8 cycles.

uint8_t

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

- **EXB_CYCLE_0**: 0 cycle,
- **EXB_CYCLE_1**: 1 cycle,
- **EXB_CYCLE_2**: 2 cycles,
- **EXB_CYCLE_3**: 3 cycles,
- **EXB_CYCLE_4**: 4 cycles,
- **EXB_CYCLE_5**: 5 cycles,
- **EXB_CYCLE_6**: 6 cycles,
- **EXB_CYCLE_8**: 8 cycles.

uint8_t

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

- **EXB_CYCLE_0**: 0 cycle,
- **EXB_CYCLE_1**: 1 cycle,
- **EXB_CYCLE_2**: 2 cycles,
- **EXB_CYCLE_4**: 4 cycles.

6. CG

6.1 Overview

The CG API provides a set of functions for using the TMPM461x CG modules as the following:

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

This driver is contained in /TX04_Periph_Driver/src/tmpm461_cg.c, with /TX04_Periph_Driver/inc/tmpm461_cg.h containing the API definitions for use by applications.

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

EHCLKIN : Clock input from the X1 pins

EHOSC : Output clock from the external high-speed oscillator

ELOSC : Output clock from the external Low-speed oscillator

IHOSC : Output clock from the internal high-speed oscillator.(for SYS)

IHOSC2 : Output clock from the external high-speed oscillator.(for OFD)

FOSCHI : Clock specified by CGOSCCR<HOSCON>

fosc : Clock specified by CGOSCCR<OSCSEL>

fppll : Clock multiplied by PLL.

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

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

fsys : Clock specified by CGSYSCR<GEAR[2:0]>.(system clock)

fperiph : Clock specified by CGSYSCR<FPSEL[2:0]>.

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

6.2 API Functions

6.2.1 Function List

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

- ◆ FunctionalState CG_GetPLLState(void)
- ◆ Result CG_SetFosc(CG_FoscSrc Source, FunctionalState **NewState**)
- ◆ void CG_SetFoscSrc(CG_FoscSrc **Source**)
- ◆ CG_FoscSrc CG_GetFoscSrc(void)
- ◆ FunctionalState CG_GetFoscState(CG_FoscSrc **Source**)
- ◆ void CG_SetSTBYMode(CG_STBYMode **Mode**)
- ◆ CG_STBYMode CG_GetSTBYMode(void)
- ◆ void CG_SetPortKeepInStop2Mode(FunctionalState **NewState**)
- ◆ FunctionalState CG_GetPortKeepInStop2Mode(void)
- ◆ Result CG_SetFcSrc(CG_FcSrc **Source**)
- ◆ CG_FcSrc CG_GetFcSrc(void)
- ◆ void CG_SetProtectCtrl(FunctionalState **NewState**)
- ◆ void CG_SetSTBYReleaseINTSrc(CG_INTSrc **INTSource**,
CG_INTActiveState **ActiveState**,
FunctionalState **NewState**)
- ◆ CG_INTActiveState CG_GetSTBYReleaseINTState(CG_INTSrc **INTSource**)
- ◆ void CG_ClearINTReq(CG_INTSrc **INTSource**)
- ◆ CG_NMIFactor CG_GetNMIFlag(void)
- ◆ FunctionalState CG_GetIOSCFlashFlag(void)
- ◆ CG_ResetFlag CG_GetResetFlag(void)
- ◆ void CG_SetADCClkSupply(FunctionalState **NewState**)
- ◆ void CG_SetInternalOscForOFD(FunctionalState **NewState**)
- ◆ void CG_SetFcPeriphA(uint32_t **Periph**, FunctionalState **NewState**)
- ◆ void CG_SetFcPeriphB(uint32_t **Periph**, FunctionalState **NewState**)
- ◆ void CG_SetFs(FunctionalState **NewState**)

6.2.2 Detailed Description

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

- 1) One group of APIs are in charge of clock selection, such as:
CG_SetFgearLevel(), CG_GetFgearLevel(), CG_SetPhiT0Src(), CG_GetPhiT0Src(),
CG_SetPhiT0Level(), CG_GetPhiT0Level(), CG_SetSCOUTSrc(), CG_GetSCOUTSrc(),
CG_SetWarmUpTime(), CG_StartWarmUp(),
CG_GetWarmUpState(), CG_SetFPLLValue(), CG_GetFPLLValue(), CG_SetPLL(),
CG_GetPLLState(), CG_SetFosc(), CG_SetFoscSrc(), CG_GetFoscSrc(),
CG_GetFoscState(), CG_SetFcSrc(), CG_GetFcSrc(), CG_SetProtectCtrl().
- 2) The 2nd group of APIs handle settings of standby modes:
CG_SetSTBYMode(), CG_GetSTBYMode(),
CG_SetPortKeepInStop2Mode(), CG_GetPortKeepInStop2Mode().
- 3) The 3rd group of APIs handle settings of interrupts:
CG_SetSTBYReleaseINTSrc(), CG_GetSTBYReleaseINTState(), CG_ClearINTReq(),
CG_GetNMIFlag(), CG_GetResetFlag().
- 4) The other APIs control clock supply for peripherals:
CG_SetADCClkSupply(), CG_SetInternalOscForOFD(), CG_SetFcPeriphA(),
CG_SetFcPeriphB(), CG_GetIOSCFlashFlag(), CG_SetFs().

6.2.3 Function Documentation

6.2.3.1 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

6.2.3.2 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

6.2.3.3 CG_SetPhiT0Src

Set fperiph for PhiT0.

Prototype:

void
CG_SetPhiT0Src(CG_PhiT0Src *PhiT0Src*)

Parameters:

PhiT0Src: Select PhiT0 source.

This parameter can be one of the following values:

- **CG_PHIT0_SRC_FGEAR** means PhiT0 source is fgear.
- **CG_PHIT0_SRC_FC** means PhiT0 source is fc.

Description:

This function selects the source for PhiT0.

Return:

None

6.2.3.4 CG_GetPhiT0Src

Get the PhiT0 source.

Prototype:

CG_PhiT0Src
CG_GetPhiT0Src(void)

Parameters:

None

Description:

This function will get the PhiT0 source.

Return:

CG_PHIT0_SRC_FGEAR means PhiT0 source is fgear.

CG_PHIT0_SRC_FC means PhiT0 source is fc.

6.2.3.5 CG_SetPhiT0Level

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

Prototype:

Result
CG_SetPhiT0Level(CG_DivideLevel *DividePhiT0FromFc*)

Parameters:

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

This parameter can be one of the following values:

- **CG_DIVIDE_1**: $\Phi T0 = fc$
- **CG_DIVIDE_2**: $\Phi T0 = fc/2$
- **CG_DIVIDE_4**: $\Phi T0 = fc/4$
- **CG_DIVIDE_8**: $\Phi T0 = fc/8$
- **CG_DIVIDE_16**: $\Phi T0 = fc/16$
- **CG_DIVIDE_32**: $\Phi T0 = fc/32$
- **CG_DIVIDE_64**: $\Phi T0 = fc/64$
- **CG_DIVIDE_128**: $\Phi T0 = fc/128$
- **CG_DIVIDE_256**: $\Phi T0 = fc/256$
- **CG_DIVIDE_512**: $\Phi T0 = fc/512$

Description:

This function will set the dividing level of prescaler clock.

Return:

SUCCESS means the setting has been written to registers successfully.

ERROR means the setting has not been written to registers.

6.2.3.6 CG_GetPhiT0Level

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

Prototype:

CG_DivideLevel
CG_GetPhiT0Level(void)

Parameters:

None

Description:

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

Return:

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

CG_DIVIDE_1: $\Phi T0 = f_c$
CG_DIVIDE_2: $\Phi T0 = f_c/2$
CG_DIVIDE_4: $\Phi T0 = f_c/4$
CG_DIVIDE_8: $\Phi T0 = f_c/8$
CG_DIVIDE_16: $\Phi T0 = f_c/16$
CG_DIVIDE_32: $\Phi T0 = f_c/32$
CG_DIVIDE_64: $\Phi T0 = f_c/64$
CG_DIVIDE_128: $\Phi T0 = f_c/128$
CG_DIVIDE_256: $\Phi T0 = f_c/256$
CG_DIVIDE_512: $\Phi T0 = f_c/512$
CG_DIVIDE_UNKNOWN: invalid data is read.

6.2.3.7 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_FSYS**: SCOUT source is set to fsys.
➤ **CG_SCOUT_SRC_FS**: SCOUT source is set to fs.

Description:

This function will set the clock source of SCOUT output.

Return:

➤ None
➤

6.2.3.8 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_FSYS: SCOUT source is set to fsys

CG_SCOUT_SRC_FS: SCOUT source is fs

6.2.3.9 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_INT_HIGH:** internal high-speed oscillator is selected as timer source.
- **CG_WARM_UP_SRC_OSC_EXT_HIGH:** external 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:

Number of warm-up cycle. It is between 0x0000 and 0xFFFFU.

Description:

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

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

Example of calculating register value for warm-up time:

/* When using high-speed oscillator 10MHz, and set warm-up time 5ms. */

So value = (warm-up time to set) / (input frequency cycle(s)) = 5ms / (1/10MHz)
= 5000cycle = 0xC350.

Round lower 4 bit off, set 0xC35 to CGOSCCR<WUPT[11:0]>

Return:

None.

6.2.3.10 CG_StartWarmUp

Start operation of warm up timer for oscillator.

Prototype:

void

CG_StartWarmUp(void)

Parameters:

None

Description:

This function will start the warm up timer.

Return:
None

6.2.3.11 CG_GetWarmUpState

Check whether warm up is completed or not.

Prototype:
WorkState
CG_GetWarmUpState(void)

Parameters:
None

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

```
Example of using warm-up timer:  
CG_SetWarmUpTime(CG_WARM_UP_SRC_OSC_EXT_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.

6.2.3.12 CG_SetFPLLValue

Set PLL multiplying value

Prototype:
Result
CG_SetFPLLValue(uint32_t **NewValue**)

Parameters:
NewValue:

- **CG_8M_MUL_4_FPLL:**
Input clock 8MHz, output clock 32MHz (4 multiplying)
- **CG_8M_MUL_6_FPLL:**
Input clock 8MHz, output clock 48MHz (6 multiplying)
- **CG_8M_MUL_8_FPLL:**
Input clock 8MHz, output clock 64MHz (8 multiplying)
- **CG_8M_MUL_10_FPLL:**
Input clock 8MHz, output clock 80MHz (10 multiplying)
- **CG_8M_MUL_12_FPLL:**
Input clock 8MHz, output clock 96MHz (12 multiplying)
- **CG_10M_MUL_4_FPLL:**
Input clock 10MHz, output clock 40MHz (4 multiplying)
- **CG_10M_MUL_5_FPLL:**
Input clock 10MHz, output clock 50MHz (5 multiplying)
- **CG_10M_MUL_6_FPLL:**
Input clock 10MHz, output clock 60MHz (6 multiplying)
- **CG_10M_MUL_10_FPLL:**

- Input clock 10MHz, output clock 100MHz (10 multiplying)
- **CG_10M_MUL_12_FPLL:**
Input clock 10MHz, output clock 120MHz (12 multiplying)
- **CG_12M_MUL_4_FPLL:**
Input clock 16MHz, output clock 48MHz (4 multiplying)
- **CG_12M_MUL_10_FPLL:**
Input clock 12MHz, output clock 120MHz (10 multiplying)
- **CG_16M_MUL_4_FPLL:**
Input clock 16MHz, output clock 64MHz (4 multiplying)
- **CG_16M_MUL_6_FPLL:**
Input clock 16MHz, output clock 96MHz (6 multiplying)

Description:

This function sets PLL multiplying value.

Return:

SUCCESS: operation is finished successfully.

ERROR: operation is not done.

6.2.3.13 CG_GetFPLLValue

Get the value of PLL setting.

Prototype:

uint32_t

CG_GetFPLLValue(void)

Parameters:

None

Description:

This function will get the PLL multiplying value.

If the other value is read from the register, it means the value is reserved.

Return:

The source of PLL multiplying value

- **CG_8M_MUL_4_FPLL:**
Input clock 8MHz, output clock 32MHz (4 multiplying)
- **CG_8M_MUL_6_FPLL:**
Input clock 8MHz, output clock 48MHz (6 multiplying)
- **CG_8M_MUL_8_FPLL:**
Input clock 8MHz, output clock 64MHz (8 multiplying)
- **CG_8M_MUL_10_FPLL:**
Input clock 8MHz, output clock 80MHz (10 multiplying)
- **CG_8M_MUL_12_FPLL:**
Input clock 8MHz, output clock 96MHz (12 multiplying)
- **CG_10M_MUL_4_FPLL:**
Input clock 10MHz, output clock 40MHz (4 multiplying)
- **CG_10M_MUL_5_FPLL:**
Input clock 10MHz, output clock 50MHz (5 multiplying)
- **CG_10M_MUL_6_FPLL:**
Input clock 10MHz, output clock 60MHz (6 multiplying)
- **CG_10M_MUL_10_FPLL:**
Input clock 10MHz, output clock 100MHz (10 multiplying)
- **CG_10M_MUL_12_FPLL:**
Input clock 10MHz, output clock 120MHz (12 multiplying)

- **CG_12M_MUL_4_FPLL:**
Input clock 16MHz, output clock 48MHz (4 multiplying)
- **CG_12M_MUL_10_FPLL:**
Input clock 12MHz, output clock 120MHz (10 multiplying)
- **CG_16M_MUL_4_FPLL:**
Input clock 16MHz, output clock 64MHz (4 multiplying)
- **CG_16M_MUL_6_FPLL:**
Input clock 16MHz, output clock 96MHz (6 multiplying)

6.2.3.14 CG_SetPLL

Enable or disable the PLL circuit.

Prototype:

Result

CG_SetPLL(FunctionalState **NewState**)

Parameters:

NewState:

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

Description:

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

Return:

SUCCESS: operation is finished successfully.

ERROR: operation is not done.

6.2.3.15 CG_GetPLLState

Get the state of PLL circuit.

Prototype:

FunctionalState

CG_GetPLLState(void)

Parameters:

None

Description:

This function will get the state of PLL circuit.

Return:

The state of PLL

ENABLE: PLL is enabled.

DISABLE: PLL is disabled.

6.2.3.16 CG_SetFosc

Enable or disable high-speed oscillator (fosc).

Prototype:

Result

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

Parameters:

Source: select clock source of fosc.

This parameter can be one of the following values:

- **CG_FOSC_OSC_EXT:** external high-speed oscillator is selected,
- **CG_FOSC_OSC_INT:** internal high-speed oscillator is selected.

NewState

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

Description:

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

Return:

SUCCESS: operation is finished successfully.

ERROR: operation is not done.

6.2.3.17 CG_SetFoscSrc

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

Prototype:

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

Parameters:

Source: select source for fosc.

This parameter can be one of the following values:

- **CG_FOSC_OSC_EXT:** external high-speed oscillator is selected,
- **CG_FOSC_CLKIN_EXT:** external clock input is selected.
- **CG_FOSC_OSC_INT:** internal high-speed oscillator is selected.

Description:

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

Return:

None

6.2.3.18 CG_GetFoscSrc

Get the source of the high-speed oscillator.

Prototype:

```
CG_FoscSrc  
CG_GetFoscSrc(void)
```

Parameters:

None

Description:

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

Return:

The source of fosc

- CG_FOSC_OSC_EXT:** external high-speed oscillator is selected,
- CG_FOSC_CLKIN_EXT:** external clock input is selected.
- CG_FOSC_OSC_INT:** internal high-speed oscillator is selected.

6.2.3.19 CG_GetFoscState

Get the state of the high-speed oscillator.

Prototype:

FunctionalState
CG_GetFoscState(CG_FoscSrc Source)

Parameters:

Source: select source for fosc.

- **CG_FOSC_OSC_EXT:** external high-speed oscillator is selected,
- **CG_FOSC_OSC_INT:** internal high-speed oscillator is selected.

Description:

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

Return:

The state of fosc

ENABLE: fosc is enabled.

DISABLE: fosc is disabled.

6.2.3.20 CG_SetSTBYMode

Set the standby mode.

Prototype:

void
CG_SetSTBYMode(CG_STBYMode **Mode**)

Parameters:

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

- **CG_STBY_MODE_STOP1:** STOP1 mode. All the internal circuits including the internal oscillator are brought to a stop.
- **CG_STBY_MODE_STOP2:** STOP2 mode. This mode halts main voltage supply, retaining some function operation.
- **CG_STBY_MODE_IDLE:** IDLE mode. Only CPU stop in this mode.

Description:

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

Return:

None

6.2.3.21 CG_GetSTBYMode

Get the standby mode.

Prototype:

CG_STBYMode
CG_GetSTBYMode(void)

Parameters:

None

Description:

This function will get the setting of standby mode.

If the value "Reserved" is read, "CG_STBY_MODE_UNKNOWN" will be returned.

Return:

The low power mode:

CG_STBY_MODE_STOP1: STOP1 mode.

CG_STBY_MODE_STOP2: STOP2 mode

CG_STBY_MODE_IDLE: IDLE mode

CG_STBY_MODE_UNKNOWN: Invalid data is read.

6.2.3.22 CG_SetPortKeepInStop2Mode

Enables or disables to keep IO control signal in stop2 mode

Prototype:

void

CG_SetPortKeepInStop2Mode(FunctionalState *NewState*)

Parameters:***NewState:***

➤ **DISABLE:** <PTKEEP>=0

➤ **ENABLE:** <PTKEEP>=1

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

Description:

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

Return:

None

6.2.3.23 CG_GetPortKeepInStop2Mode

Get the pin status in stop2 mode

Prototype:

FunctionalState

CG_GetPinStateInStopMode(void)

Parameters:

None

Description:

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

Return:

The port keeps in stop2 mode

DISABLE: <PTKEEP>=0

ENABLE: <PTKEEP>=1

6.2.3.24 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.

Return:

SUCCESS: set clock source for fc successfully

ERROR: clock source of fc is not changed.

6.2.3.25 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.

6.2.3.26 CG_SetProtectCtrl

Enable or disable to protect CG registers.

Prototype:

void

CG_SetProtectCtrl(FunctionalState **NewState**)

Parameters:

NewState

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

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

Description:

This function enables or disables CG registers to be written.

Return:

None

6.2.3.27 CG_SetSTBYReleaseINTSrc

Set the INT source for releasing low power mode.

Prototype:

```
void  
CG_SetSTBYReleaseINTSrc(CG_INTSrc INTSource,  
                        CG_INTActiveState ActiveState,  
                        FunctionalState NewState)
```

Parameters:

INTSource: select the INT source for releasing standby mode

This parameter can be one of the following values:

- **CG_INT_SRC_0** : INT0
- **CG_INT_SRC_1** : INT1
- **CG_INT_SRC_2** : INT2
- **CG_INT_SRC_3** : INT3
- **CG_INT_SRC_4** : INT4
- **CG_INT_SRC_5** : INT5
- **CG_INT_SRC_6** : INT6
- **CG_INT_SRC_7** : INT7
- **CG_INT_SRC_8** : INT8
- **CG_INT_SRC_9** : INT9
- **CG_INT_SRC_A** : INTA
- **CG_INT_SRC_B** : INTB
- **CG_INT_SRC_C** : INTC
- **CG_INT_SRC_D** : INTD
- **CG_INT_SRC_E** : INTE
- **CG_INT_SRC_F** : INTF
- **CG_INT_SRC_CECRX**: CEC reception interrupt
- **CG_INT_SRC_CECTX**: CEC transmission interrupt
- **CG_INT_SRC_RMCRX0**: RMC reception interrupt(channel 0)
- **CG_INT_SRC_RTC** : INTRTC

ActiveState: select the active state for release trigger.

For **CG_INT_SRC_RTC**, this parameter can only be

- **CG_INT_ACTIVE_STATE_FALLING**: active on falling edge

For **CG_INT_SRC_CECRX**, **CG_INT_SRC_CECTX** and **CG_INT_SRC_RMCRX0**, this parameter can only be

- **CG_INT_ACTIVE_STATE_RISING**: active on rising edge

For the other interrupt source, this parameter can be one of the following values:

- **CG_INT_ACTIVE_STATE_L**: active on low level
- **CG_INT_ACTIVE_STATE_H**: active on high level
- **CG_INT_ACTIVE_STATE_FALLING**: active on falling edge
- **CG_INT_ACTIVE_STATE_RISING**: active on rising edge
- **CG_INT_ACTIVE_STATE_BOTH_EDGES**: active on both edges

NewState: enable or disable this release trigger

This parameter can be one of the following values:

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

Description:

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

Return:

None

6.2.3.28 CG_GetSTBYReleaseINTState

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

Prototype:

CG_INT_ActiveState
CG_GetSTBYReleaseINTSrc(CG_INTSrc *INTSource*)

Parameters:

INTSource: select the release INT source

This parameter can be one of the following values:

CG_INT_SRC_0, CG_INT_SRC_1, CG_INT_SRC_2, CG_INT_SRC_3,
CG_INT_SRC_4, CG_INT_SRC_5, CG_INT_SRC_6, CG_INT_SRC_7,
CG_INT_SRC_8, CG_INT_SRC_9, CG_INT_SRC_A, CG_INT_SRC_B,
CG_INT_SRC_C, CG_INT_SRC_D, CG_INT_SRC_E , CG_INT_SRC_F
CG_INT_SRC_CECRX, CG_INT_SRC_CECTX,
CG_INT_SRC_RMCRX0, CG_INT_SRC_RTC.

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

6.2.3.29 CG_ClearINTReq

Clears the input INT request.

Prototype:

void
CG_ClearINTReq(CG_INTSrc *INTSource*)

Parameters:

INTSource: select the release INT source.

This parameter can be one of the following values:

CG_INT_SRC_0, CG_INT_SRC_1, CG_INT_SRC_2, CG_INT_SRC_3,
CG_INT_SRC_4, CG_INT_SRC_5, CG_INT_SRC_6, CG_INT_SRC_7,
CG_INT_SRC_8, CG_INT_SRC_9, CG_INT_SRC_A, CG_INT_SRC_B,
CG_INT_SRC_C, CG_INT_SRC_D, CG_INT_SRC_E , CG_INT_SRC_F
CG_INT_SRC_CECRX, CG_INT_SRC_CECTX,
CG_INT_SRC_RMCRX0, CG_INT_SRC_RTC.

Description:

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

Return:

None

6.2.3.30 CG_GetNMIFlag

Get the NMI flag that shows who triggered NMI

Prototype:

CG_NMI_Factor
CG_GetNMIFlag (void)

Parameters:

None

Description:

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

Return:

NMI value:

WDT (Bit 0) means generated from WDT.

NMIPin (Bit 1) means generated from NMI pin.

DetectLowVoltage (Bit 2) means generated when detect low voltage by LVD.

ReturnLowVoltage (Bit 4) means generated from Flash ECC

6.2.3.31 CG_GetIOSCFlashFlag

Get the flag for stopping of the internal high-speed oscillator or writing to the flash memory.

Prototype:

FunctionalState
CG_GetIOSCFlashFlag(void)

Parameters:

None

Description:

This function gets the flag for stopping of the internal high-speed oscillator or writing to the flash memory.

***Note:**

For programming into the Flash memory after entering into Normal mode from Stop2 mode, it is required to confirm that CGRSTFLG<OSCFLF> is read as "1".

Return:

Flag for stopping of the internal high-speed oscillator or writing to the flash memory:

ENABLE: Can stop the internal high-speed oscillator and write to the flash memory.

DISABLE: Can't stop the internal high-speed oscillator and write to the flash memory.

6.2.3.32 CG_GetResetFlag

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

Prototype:

CG_ResetFlag
CG_GetResetFlag(void)

Parameters:

None

Description:

This function gets the reset flag showing what triggered reset.

Return:

Reset flag:

ResetPin (Bit0) Reset by power on reset
WDTReset (Bit 2) means reset from WDT.
STOP2Reset(Bit3) means reset flag by STOP2 mode release
DebugReset (Bit 4) means reset from SYSRESETREQ.
OFDReset (Bit5) Rest by OFD
LVDReset (Bit6) Rest by LVD

6.2.3.33 CG_SetADCClkSupply

Enable or disable supplying clock fsys for ADC.

Prototype:

void
CG_SetADCClkSupply(FunctionalState **NewState**)

Parameters:

NewState: New state of clock fsys supply setting for ADC.

This parameter can be one of the following values:

- **ENABLE** : Enable ADC clock supply
- **DISABLE** : Disable ADC clock supply

Description:

This function will enable or disable supplying clock fsys for ADC.

Return:

None

6.2.3.34 CG_SetInternalOscForOFD

Enables or stops internal high-speed oscillator oscillation for OFD

Prototype:

void
CG_SetInternalOscForOFD(FunctionalState **NewState**)

Parameters:

NewState

- **ENABLE:** to enable internal high-speed oscillator 2 oscillation for OFD.
- **DISABLE:** to disable internal high-speed oscillator 2 oscillation for OFD.

Description:

This function enables or stops internal high-speed oscillator oscillation for OFD.

Return:

None

6.2.3.35 CG_SetFcPeriphA

Enable or disable supplying clock fsys to peripherals.

Prototype:

void
CG_SetFcPeriphA(uint32_t **Periph**,
FunctionalState **NewState**)

Parameters:

Periph: The target peripheral that CG supplies clock fc for

This parameter can be one of the following values or their combination:

- **CG_FC_PERIPH_PORTA:** Clock control for PORT A
- **CG_FC_PERIPH_PORTB:** Clock control for PORT B
- **CG_FC_PERIPH_PORTC:** Clock control for PORT C
- **CG_FC_PERIPH_PORTD:** Clock control for PORT D
- **CG_FC_PERIPH_PORTE:** Clock control for PORT E
- **CG_FC_PERIPH_PORTF:** Clock control for PORT F
- **CG_FC_PERIPH_PORTG:** Clock control for PORT G
- **CG_FC_PERIPH_PORTH:** Clock control for PORT H
- **CG_FC_PERIPH_PORTJ:** Clock control for PORT J
- **CG_FC_PERIPH_PORTK:** Clock control for PORT K
- **CG_FC_PERIPH_TMRB0:** Clock control for TMRB0
- **CG_FC_PERIPH_TMRB1:** Clock control for TMRB1
- **CG_FC_PERIPH_TMRB2:** Clock control for TMRB2
- **CG_FC_PERIPH_TMRB3:** Clock control for TMRB3
- **CG_FC_PERIPH_TMRB4:** Clock control for TMRB4
- **CG_FC_PERIPH_TMRB5:** Clock control for TMRB5
- **CG_FC_PERIPH_TMRB6:** Clock control for TMRB6
- **CG_FC_PERIPH_TMRB7:** Clock control for TMRB7
- **CG_FC_PERIPH_TMRB8:** Clock control for TMRB8
- **CG_FC_PERIPH_TMRB9:** Clock control for TMRB9
- **CG_FC_PERIPH_TMRB10:** Clock control for TMRB10
- **CG_FC_PERIPH_TMRB11:** Clock control for TMRB11
- **CG_FC_PERIPH_TMRB12:** Clock control for TMRB12
- **CG_FC_PERIPH_TMRB13:** Clock control for TMRB13
- **CG_FC_PERIPH_TMRB14:** Clock control for TMRB14
- **CG_FC_PERIPH_TMRB15:** Clock control for TMRB15
- **CG_FC_PERIPH_MPT0:** Clock control for MPT0
- **CG_FC_PERIPH_MPT1:** Clock control for MPT1
- **CG_FC_PERIPH_TRACE:** Clock control for TRACE
- **CG_FC_PERIPHA_ALL:** ALL clock control

NewState

- **ENABLE:** Enable supplying clock fsys to peripherals.
- **DISABLE:** Disable supplying clock fsys to peripherals.

Description:

This function enables or disables supplying clock fsys to peripherals

Return:

None

6.2.3.36 CG_SetFcPeriphB

Enable or disable supplying clock fsys to peripherals.

Prototype:

void

CG_SetFcPeriphB(uint32_t ***Periph***,
FunctionalState ***NewState***)

Parameters:

Periph: The target peripheral that CG supplies clock fc for

This parameter can be one of the following values or their combination:

- **CG_FC_PERIPH_SIO_UART0:** Clock control for SIO/UART0
- **CG_FC_PERIPH_SIO_UART1:** Clock control for SIO/UART1
- **CG_FC_PERIPH_SIO_UART2:** Clock control for SIO/UART2
- **CG_FC_PERIPH_SIO_UART3:** Clock control for SIO/UART3

- **CG_FC_PERIPH_SIO_UART4**: Clock control for SIO/UART4
- **CG_FC_PERIPH_SIO_UART5**: Clock control for SIO/UART5
- **CG_FC_PERIPH_UART0**: Clock control for UART0
- **CG_FC_PERIPH_UART1**: Clock control for UART1
- **CG_FC_PERIPH_I2C0**: Clock control for I2C0
- **CG_FC_PERIPH_I2C1**: Clock control for I2C1
- **CG_FC_PERIPH_I2C2**: Clock control for I2C2
- **CG_FC_PERIPH_I2C3**: Clock control for I2C3
- **CG_FC_PERIPH_I2C4**: Clock control for I2C4
- **CG_FC_PERIPH_SSP0**: Clock control for SSP0
- **CG_FC_PERIPH_SSP1**: Clock control for SSP1
- **CG_FC_PERIPH_SSP2**: Clock control for SSP2
- **CG_FC_PERIPH_EBIF**: Clock control for EBIF
- **CG_FC_PERIPH_DMACA**: Clock control for DMAC A
- **CG_FC_PERIPH_DMACB**: Clock control for DMAC B
- **CG_FC_PERIPH_DMACC**: Clock control for DMAC C
- **CG_FC_PERIPH_DMAIF**: Clock control for DMACIF
- **CG_FC_PERIPH_ADC**: Clock control for ADC
- **CG_FC_PERIPH_WDT**: Clock control for WDT
- **CG_FC_PERIPH_OFD**: Clock control for OFD
- **CG_FC_PERIPHB_ALL**: ALL clock control

NewState

- **ENABLE**: Enable supplying clock fsys to peripherals.
- **DISABLE**: Disable supplying clock fsys to peripherals.

Description:

This function enables or disables supplying clock fsys to peripherals

Return:

None

6.2.3.37 CG_SetFs

Enable or disable external low-speed oscillator (fs) for RMC & RTC & CEC

Prototype:

void

CG_SetFs(FunctionalState **NewState**)

Parameters:

NewState

- **ENABLE**: to enable external low-speed oscillator for RMC & RTC & CEC.
- **DISABLE**: to disable external low-speed oscillator for RMC & RTC & CEC.

Description:

This enables or disables external low-speed oscillator (fs) for RMC & RTC & CEC.

Return:

None

6.2.4 Data Structure Description

6.2.4.1 CG_NMIFactor

Data Fields:

uint32_t

All specifies CGNMI source generation state.

Bit Fields:

uint32_t

WDT(Bit 0) means generated from WDT.

uint32_t

NMIPin(Bit 1) means generated from NMI pin.

uint32_t

DetectLowVoltage(Bit 2) means generated when detect low voltage by LVD.

uint32_t

Reserved1 (Bit3) Reserved

uint32_t

ReturnLowVoltage(Bit 4) Means generated from Flash ECC

uint32_t

Reserved2 (Bit5~bit31) Reserved

6.2.4.2 CG_ResetFlag

Data Fields:

uint32_t

All specifies CG reset source.**Bit Fields:**

uint32_t

ResetPin(Bit0) Reset from RESET pin

uint32_t

Reserved1 (Bit1) Reserved

uint32_t

WDTReset(Bit2) Reset from WDT

uint32_t

STOP2Reset(Bit3) Reset flag by STOP2 mode release

uint32_t

DebugReset(Bit4) Reset from SYSRESETREQ

uint32_t

OFDReset(Bit5) Rest by OFD

uint32_t

LVDReset(Bit6) Rest by LVD

uint32_t

Reserved2 (Bit7~bit31) Reserved

7. FC

7.1 Overview

TMPM461x device contains flash memory.

For TMPM461F10FG, the size of flash is 1024Kbytes.

For TMPM461F15FG, the size of flash is 1536Kbytes.

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\TX04_Periph_Driver\src\tmpr461_fc.c with \Libraries\TX04_Periph_Driver\inc\tmpr461_fc.h containing the API definitions for use by applications.

7.2 API Functions

7.2.1 Function List

- ◆ void FC_SetSecurityBit(FunctionalState **NewState**)
- ◆ FunctionalState FC_GetSecurityBit(void)
- ◆ WorkState FC_GetBusyState(void)
- ◆ FunctionalState FC_GetBlockProtectState(uint8_t **BlockNum**)
- ◆ FunctionalState FC_GetPageProtectState(uint8_t **PageNum**)
- ◆ FunctionalState FC_GetAbortState(void)
- ◆ uint32_t FC_GetSwapSize(void);
- ◆ uint32_t FC_GetSwapState(void);
- ◆ void FC_SelectArea(uint8_t **AreaNum**, FunctionalState **NewState**);
- ◆ void FC_SetAbortion(void);
- ◆ void FC_ClearAbortion(void);
- ◆ void FC_SetClkDiv(uint8_t **ClkDiv**);
- ◆ void FC_SetProgramCount(uint8_t **ProgramCount**);
- ◆ void FC_SetEraseCounter(uint8_t **EraseCounter**);
- ◆ FC_Result FC_ProgramBlockProtectState(uint8_t **BlockNum**);
- ◆ FC_Result FC_ProgramPageProtectState(uint8_t **PageNum**);
- ◆ FC_Result FC_EraseProtectState(void);
- ◆ FC_Result FC_WritePage(uint32_t **PageAddr**, uint32_t * **Data**);
- ◆ FC_Result FC_EraseBlock(uint32_t **BlockAddr**);
- ◆ FC_Result FC_EraseArea(uint32_t **AreaAddr**);
- ◆ FC_Result FC_ErasePage(uint32_t **PageAddr**);
- ◆ FC_Result FC_EraseChip(void);
- ◆ FC_Result FC_SetSwpsrBit(uint8_t **BitNum**);
- ◆ uint32_t FC_GetSwpsrBitValue(uint8_t **BitNum**);

7.2.2 Detailed Description

Functions listed above can be divided into five 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(), FC_GetPageProtectState().
- 3) The functions change the protection status of each block:
FC_ProgramBlockProtectState(), FC_ProgramPageProtectState(),
FC_EraseProtectState().
- 4) Use automatic operation command of flash.
FC_WritePage(), FC_EraseBlock(), FC_EraseChip(), FC_EraseArea(),
FC_ErasePage(), FC_SetSwpsrBit().
- 5) Others:
FC_GetAbortState(), FC_GetSwapSize(), FC_GetSwapState(), FC_SelectArea(),
FC_SetAbortion(), FC_ClearAbortion(), FC_SetClkDiv(), FC_SetProgramCount(),
FC_SetEraseCounter(), FC_GetSwpsrBitValue().

7.2.3 Function Documentation

7.2.3.1 FC_SetSecurityBit

Set the value of SECBIT register.

Prototype:

void
FC_SetSecurityBit (FunctionalState **NewState**)

Parameters:

NewState: Select the state of SECBIT register.

This parameter can be one of the following values:

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

Description:

- 1) All the protection bits (the FCPSRx register) 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<SECBIT> to "1".

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

Return:

None

7.2.3.2 FC_GetSecurityBit

Get the value of SECBIT register.

Prototype:

FunctionalState
FC_GetSecurityBit(void)

Parameters:

None

Description:

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

Return:

State of SECBIT register.

DISABLE: Protection function is not available.

ENABLE: Protection function is available.

7.2.3.3 FC_GetBusyState

Get the status of the flash auto operation.

Prototype:

WorkState

FC_GetBusyState (void)

Parameters:

None

Description:

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

Return:

Status of the flash automatic operation:

BUSY: Flash memory is in automatic operation.

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

7.2.3.4 FC_GetBlockProtectState

Get the specified block protection state

Prototype:

FunctionalState

FC_GetBlockProtectState(uint8_t *BlockNum*).

Parameters:

BlockNum: The flash block number

For TMPM461F10FG:

FC_BLOCK_1 to FC_BLOCK_31

For TMPM461F15FG:

FC_BLOCK_1 to FC_BLOCK_47

Description:

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

Return:

Block protection status.

DISABLE: Block is unprotected

ENABLE: Block is protected

7.2.3.5 FC_GetPageProtectState

Get the page protection status.

Prototype:

FunctionalState

FC_GetPageProtectState(uint8_t *PageNum*)

Parameters:

PageNum:The flash page number

➤ **FC_PAGE_0** to **FC_PAGE_7**

Description:

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

Return:

Page protection status.

DISABLE: Page is unprotected

ENABLE: Page is protected

7.2.3.6 FC_GetAbortState

Get the status of the auto operation is aborted or not

Prototype:

FunctionalState

FC_GetAbortState(void)

Parameters:

None

Description:

Once the auto operation is aborted by FCCR<WEABORT>, "1" is set to this bit.

Return:

the status of the auto operation is aborted or not.:

DISABLE: The aborted is disabled

DONE: The aborted is enabled

7.2.3.7 FC_GetSwapSize

Get the swap size.

Prototype:

uint32_t

FC_GetSwapSize(void)

Parameters:

None

Description:

Get the swap size.

Return:

the swap size.

FC_SWAP_SIZE_4K: The swap size is 4K bytes.

FC_SWAP_SIZE_8K: The swap size is 8K bytes.

FC_SWAP_SIZE_16K: The swap size is 16K bytes.

FC_SWAP_SIZE_32K: The swap size is 32K bytes.

FC_SWAP_SIZE_AREA1: Area 0 swaps with area 1.

FC_SWAP_SIZE_AREA2: Area 0 swaps with area 2.

7.2.3.8 FC_GetSwapState

Get the swap state.

Prototype:

uint32_t

FC_GetSwapState(void)

Parameters:

None

Description:

Get the swap state.

Return:

the swap state.

FC_SWAP_RELEASE: Release the swap.

FC_SWAP_PROHIBIT: Setting is prohibited..

FC_SWAPPING: In swapping.

FC_SWAP_INITIAL: Release the swap (Initial state).

7.2.3.9 FC_SelectArea

Specifies an "area" in the Flash memory.

Prototype:

void

FC_SelectArea(uint8_t **AreaNum** ,
FunctionalState **NewState**)

Parameters:

AreaNum: The flash area number

For TMPM461F15FG:

➤ **FC_AREA_0, FC_AREA_1, FC_AREA_2, FC_AREA_ALL**

For TMPM461F10FG:

➤ **FC_AREA_0, FC_AREA_1, FC_AREA_ALL**

NewState: Specify area state.

This parameter can be one of the following values:

➤ **ENABLE:** Select the area.

➤ **DISABLE:** Unselect the area.

Description:

Specifies an "area" in the Flash memory that is targeted by Flash memory operation command

Return:
None

7.2.3.10 FC_SetAbortion

Set abortion of auto operation command.

Prototype:
void
FC_SetAbortion(void)

Parameters:
None

Description:
Set abortion of auto operation command.

Return:
None

7.2.3.11 FC_ClearAbortion

Clear FCSR<WEABORT> to "0" command.

Prototype:
void
FC_ClearAbortion(void)

Parameters:
None

Description:
Clear FCSR<WEABORT> to "0" command.

Return:
None

7.2.3.12 FC_SetClkDiv

Set Frequency division ratio to change the clock.

Prototype:
void
FC_SetClkDiv(uint8_t *ClkDiv*)

Parameters:
ClkDiv: The divisor of the system clock
➤ FC_Clk_Div_1 to FC_Clk_Div_32

Description:
Set Frequency division ratio to change the clock (WCLK: $f_{sys}/(DIV+1)$)n automatic operation to 8 to 12MHz.

Return:
None

7.2.3.13 FC_SetProgramCount

Set the number of counts that makes a programming time (CNT/WCLK) by automatic program execution command

Prototype:

void
FC_SetProgramCount(uint8_t *ProgramCount*)

Parameters:

ProgramCount: the counter of the divided system clock for flash program

- FC_PROG_CNT_250, FC_PROG_CNT_300, FC_PROG_CNT_350.

Description:

Set the number of counts that makes a programming time (CNT/WCLK) by automatic program execution command be within the range of 20 to 40 μsec.

Return:

None

7.2.3.14 FC_SetEraseCounter

Set the number of counts until erase time (CNT/WCLK) will be 100 ~ 130msec using each auto erase command.

Prototype:

void
FC_SetEraseCounter (uint8_t *EraseCounter*)

Parameters:

EraseCounter: the number of counts until erase time (CNT/WCLK) will be 100~130msec using each auto erase command

- FC_ERAS_CNT_85, FC_ERAS_CNT_90,
- FC_ERAS_CNT_95, FC_ERAS_CNT_100,
- FC_ERAS_CNT_105, FC_ERAS_CNT_110
- FC_ERAS_CNT_115, FC_ERAS_CNT_120,
- FC_ERAS_CNT_125, FC_ERAS_CNT_130,
- FC_ERAS_CNT_135, FC_ERAS_CNT_140.

Description:

Set the number of counts until erase time (CNT/WCLK) will be 100 ~ 130msec using each auto erase command.

Return:

None

7.2.3.15 FC_ProgramBlockProtectState

Program the protection bits.

Prototype:

FC_Result
FC_ProgramProtectState(uint8_t *BlockNum*)

Parameters:

For TMPM461F10FG:

- FC_BLOCK_1 to FC_BLOCK_31

For **TMPM461F15FG**:

➤ **FC_BLOCK_1** to **FC_BLOCK_47**

Description:

This API is used to set the protection bit to “1” so that the corresponding block can be protected. When the block is protected, it can’t be written or erased.

One protection bit will be programmed when this API is executed each time.

Return:

Result of the operation to program the protection bit.

FC_SUCCESS: Set the protection bit to “1” successfully.

FC_ERROR_PROTECTED: The protection bit is “1” already, and it doesn’t need to program it again.

FC_ERROR_OVER_TIME: Program block protection bit operation over time error.

7.2.3.16 FC_ProgramPageProtectState

Program the protection bits.

Prototype:

FC_Result

FC_ProgramProtectState(uint8_t *PageNum*)

Parameters:

➤ **FC_PAGE_0** to **FC_PAGE_7**

Description:

This API is used to set the protection bit to “1” so that the corresponding page 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 page protection bit operation over time error.

7.2.3.17 FC_EraseProtectState

Erase the protection bits.

Prototype:

FC_Result

FC_EraseBlockProtectState(void)

Parameters:

None

Description:

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

The whole flash protection bit 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 page protection bits operation over time error.

7.2.3.18 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 FC_PAGE_SIZE.

Description:

This API is used to write data to specified page.

It contains 1024 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:**

1 An attempt to rewrite a page two or more times without erasing the content can cause damages to the device.

2 For programming into the Flash memory after entering into Normal mode from Stop2 mode, it is required to confirm that CGRSTFLG<OSCFLF> is read as "1".

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 or page is protected. The write operation can't be executed.

FC_ERROR_OVER_TIME: Write operation over time error.

7.2.3.19 FC_EraseBlock

Erase the content of specified block.

Prototype:

FC_Result

FC_EraseBlock(uint32_t **BlockAddr**)

Parameters:

BlockAddr: The block starts address.

Description:

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

Return:

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

FC_SUCCESS: the content of the specified block is erased successfully.

FC_ERROR_PROTECTED: The block is protected. The erase operation can't be executed. The block will not be erased.

FC_ERROR_OVER_TIME: Erase operation over time error.

7.2.3.20 FC_EraseArea

Erase the content of specified area.

Prototype:

FC_Result

FC_EraseArea(uint32_t **AreaAddr**)

Parameters:

AreaAddr: The block starts address.

Description:

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

Return:

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

FC_SUCCESS: the content of the specified area is erased successfully.

FC_ERROR_PROTECTED: The area is protected. The erase operation can't be executed. The area will not be erased.

FC_ERROR_OVER_TIME: Erase operation over time error.

7.2.3.21 FC_ErasePage

Erase the content of specified page.

Prototype:

FC_Result

FC_ErasePage(uint32_t **PageAddr**)

Parameters:

PageAddr: The page starts address.

Description:

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

Return:

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

FC_SUCCESS: the content of the specified page is erased successfully.

FC_ERROR_PROTECTED: The page is protected. The erase operation can't be executed. The page will not be erased.

FC_ERROR_OVER_TIME: Erase operation over time error.

7.2.3.22 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.

7.2.3.23 FC_SetSwapsrBit

Setting values of FCSWPSR[10:0] by memory swap command.

Prototype:

FC_Result

FC_SetSwapsrBit(uint8_t *BitNum*)

Parameters:

BitNum: The FCSWPSR bit number to be set

This parameter can be one of the following values:

FC_SWPSR_BIT_0 to FC_SWPSR_BIT_10

Description:

Setting values of FCSWPSR[10:0] by memory swap command. Automatic memory swap is a command to write "1" to each bit of FCSWPSR[10:0] in the units of 1-bit. A bit cannot be set to "0" independently. All bits should be cleared to "0" using automatic protect bit erase command.

Return:

Result of the operation to set the FCSWPSR bit.

FC_SUCCESS: Set the FCSWPSR bit successfully.

FC_ERROR_OVER_TIME: Set the FCSWPSR bit operation over time error.

7.2.3.24 FC_GetSwpsrBitValue

Get the value of the special bit of FCSWPSR[10:0].

Prototype:

uint32_t

FC_GetSwpsrBitValue(uint8_t *BitNum*)

Parameters:

BitNum: The special bit of SWPSR.

This parameter can be one of the following values:

FC_SWPSR_BIT_0 to FC_SWPSR_BIT_10

Description:

Get the value of the special bit of FCSWPSR[10:0].

Return:

The value returned can be one of the following values:

FC_BIT_VALUE_0, FC_BIT_VALUE_1.

7.2.4 Data Structure Description

None

8. FUART

8.1 Overview

TOSHIBA TMPM461x contains the Asynchronous serial channel (Full UART) with Modem control.

TOSHIBA TMPM461x contains two channels Full UART: FUART0, FUART1.

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/TX04_Periph_Driver/src/tmpm461_fuart.c, with /Libraries/TX04_Periph_Driver/inc/tmpm461_fuart.h containing the macros, data types, structures and API definitions for use by applications.

8.2 API Functions

8.2.1 Function List

- ◆ void FUART_Enable(TSB_FUART_TypeDef * **FUARTx**)
- ◆ void FUART_Disable(TSB_FUART_TypeDef * **FUARTx**)
- ◆ uint32_t FUART_GetRxData(TSB_FUART_TypeDef * **FUARTx**)
- ◆ void FUART_SetTxData(TSB_FUART_TypeDef * **FUARTx**, uint32_t **Data**)
- ◆ FUART_Err FUART_GetErrStatus(TSB_FUART_TypeDef * **FUARTx**)
- ◆ void FUART_ClearErrStatus(TSB_FUART_TypeDef * **FUARTx**)
- ◆ WorkState FUART_GetBusyState(TSB_FUART_TypeDef * **FUARTx**)
- ◆ FUART_StorageStatus FUART_GetStorageStatus(
TSB_FUART_TypeDef * **FUARTx**, FUART_Direction **Direction**)
- ◆ void FUART_SetIrDADivisor(TSB_FUART_TypeDef * **FUARTx**, uint32_t **Divisor**)
- ◆ void FUART_Init(
TSB_FUART_TypeDef * **FUARTx**, FUART_InitTypeDef * **InitStruct**)
- ◆ void FUART_EnableFIFO(TSB_FUART_TypeDef * **FUARTx**)
- ◆ void FUART_DisableFIFO(TSB_FUART_TypeDef * **FUARTx**)
- ◆ void FUART_SetSendBreak(
TSB_FUART_TypeDef * **FUARTx**, FunctionalState **NewState**)
- ◆ void FUART_SetIrDAEncodeMode(
TSB_FUART_TypeDef * **FUARTx**, uint32_t **Mode**)
- ◆ Result FUART_EnableIrDA(TSB_FUART_TypeDef * **FUARTx**)
- ◆ void FUART_DisableIrDA(TSB_FUART_TypeDef * **FUARTx**)
- ◆ void FUART_SetINTFIFOLevel(
TSB_FUART_TypeDef * **FUARTx**, uint32_t **RxLevel**, uint32_t **TxLevel**)
- ◆ void FUART_SetINTMask(TSB_FUART_TypeDef * **FUARTx**, uint32_t **IntMaskSrc**)
- ◆ FUART_INTStatus FUART_GetINTMask(TSB_FUART_TypeDef * **FUARTx**)
- ◆ FUART_INTStatus FUART_GetRawINTStatus(TSB_FUART_TypeDef * **FUARTx**)
- ◆ FUART_INTStatus FUART_GetMaskedINTStatus(TSB_FUART_TypeDef * **FUARTx**)
- ◆ void FUART_ClearINT(
TSB_FUART_TypeDef * **FUARTx**, FUART_INTStatus **INTStatus**)
- ◆ void FUART_SetDMAOnErr(
TSB_FUART_TypeDef * **FUARTx**, FunctionalState **NewState**)
- ◆ void FUART_SetFIFODMA(TSB_FUART_TypeDef * **FUARTx**,
FUART_Direction **Direction**, FunctionalState **NewState**)
- ◆ FUART_AllModemStatus FUART_GetModemStatus(

- ◆ void FUART_SetRTSSStatus(
TSB_FUART_TypeDef * **FUARTx**)
- ◆ void FUART_SetDTRStatus(
TSB_FUART_TypeDef * **FUARTx**, FUART_ModemStatus **Status**)

8.2.2 Detailed Description

Functions listed above can be divided into five parts:

- 1) Full UART Configuration and Initialization, common operation
FUART_Enable(), FUART_Disable(), FUART_Init(), FUART_GetRxData(),
FUART_SetTxData(), FUART_GetErrStatus(), FUART_ClearErrStatus(),
FUART_GetBusyState(), FUART_GetStorageStatus(), FUART_SetSendBreak()
- 2) Configure FIFO and DMA.
FUART_EnableFIFO(), FUART_DisableFIFO(), FUART_SetINTFIFOLevel(),
FUART_SetFIFODMA(), FUART_SetDMAOnErr().
- 3) Configure interrupt, get interrupt status and clear interrupt.
FUART_SetINTMask(), FUART_GetINTMask(), FUART_GetRawINTStatus(),
FUART_GetMaskedINTStatus(), FUART_ClearINT().
- 4) Modem control.
FUART_GetModemStatus(), FUART_SetRTSSStatus(), FUART_SetDTRStatus().
- 5) Configure IrDA.
FUART_EnableIrDA(), FUART_DisableIrDA(), FUART_SetIrDAEncodeMode(),
FUART_SetIrDADivisor().

8.2.3 Function Documentation

***Note:** in all of the following APIs, parameter “TSB_FUART_TypeDef* **FUARTx**” can be **FUART0** or **FUART1**.

8.2.3.1 FUART_Enable

Enable the specified Full UART channel.

Prototype:

void
FUART_Enable(TSB_FUART_TypeDef * **FUARTx**)

Parameters:

FUARTx: The specified Full UART channel.

Description:

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

Return:

None

8.2.3.2 FUART_Disable

Disable the specified Full UART channel.

Prototype:

void
FUART_Disable(TSB_FUART_TypeDef * **FUARTx**)

Parameters:

FUARTx: The specified Full UART channel.

Description:

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

Return:

None

8.2.3.3 FUART_GetRxData

Get received data from the specified Full UART channel.

Prototype:

uint32_t

FUART_GetRxData(TSB_FUART_TypeDef * **FUARTx**)

Parameters:

FUARTx: The specified Full UART channel.

Description:

This API will get the data received from the specified Full UART channel selected by **FUARTx**. It is appropriate to call the function after

FUART_GetStorageStatus(FUARTx, FUART_RX) returns **FUART_STORAGE_NORMAL** or **FUART_STORAGE_FULL**.

Return:

The data received from the specified Full UART channel

8.2.3.4 FUART_SetTxData

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

Prototype:

void

FUART_SetTxData(TSB_FUART_TypeDef * **FUARTx**,
uint32_t **Data**)

Parameters:

FUARTx: The specified Full UART channel.

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

Description:

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

Return:

None

8.2.3.5 FUART_GetErrStatus

Get receive error status.

Prototype:

FUART_Err

FUART_GetErrStatus(TSB_FUART_TypeDef * **FUARTx**)

Parameters:

FUARTx: The specified Full UART channel.

Description:

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

Return:

FUART_NO_ERR means there is no error in the last transfer.

FUART_OVERRUN means that overrun occurs in the last transfer.

FUART_PARITY_ERR means either even parity or odd parity fails.

FUART_FRAMING_ERR means there is framing error in the last transfer.

FUART_BREAK_ERR means there is break error in the last transfer.

FUART_ERRS means that 2 or more errors occurred in the last transfer.

8.2.3.6 FUART_ClearErrStatus

Clear receive error status.

Prototype:

void

FUART_ClearErrStatus(TSB_FUART_TypeDef * **FUARTx**)

Parameters:

FUARTx: The specified Full UART channel.

Description:

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

Return:

None

8.2.3.7 FUART_GetBusyState

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

Prototype:

WorkState

FUART_GetBusyState(TSB_FUART_TypeDef * **FUARTx**)

Parameters:

FUARTx: The specified Full UART channel.

Description:

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

Return:

Work state of the specified Full UART channel:

BUSY: The Full UART is transmitting data

DONE: The Full UART has stopped transmitting data

8.2.3.8 FUART_GetStorageStatus

Get the FIFO or hold register status.

Prototype:

FUART_StorageStatus

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

Parameters:

FUARTx: The specified Full UART channel.

Direction: The direction of Full UART

- **FUART_RX**: for receive FIFO or receive hold register
- **FUART_TX**: for transmit FIFO or transmit hold register

Description:

When FIFO is enabled, this API will get the transmit or receive FIFO status.

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

Return:

FUART_StorageStatus: The FIFO or hold register status.

FUART_STORAGE_EMPTY: The FIFO or the hold register is empty.

FUART_STORAGE_NORMAL: The FIFO is normal, not empty and not full.

FUART_STORAGE_INVALID: The FIFO or the hold register is in invalid status.

FUART_STORAGE_FULL: The FIFO or the hold register is full.

8.2.3.9 FUART_SetIrDADivisor

Get error flag of the transfer from the specified UART channel.

Prototype:

void

FUART_SetIrDADivisor(TSB_FUART_TypeDef * *FUARTx*,
uint32_t *Divisor*)

Parameters:

FUARTx: The specified Full UART channel.

Divisor: The IrDA Low-power divisor (from 0x01 to 0xFF)

Description:

This API will set IrDA Low-power divisor to generate the IrLPBaud16 signal by dividing down of UARTCLK.

This API must be executed before the IrDA circuit is enabled.

Return:

None

8.2.3.10 FUART_Init

Initialize and configure the specified Full UART channel.

Prototype:

```
void  
FUART_Init(TSB_FUART_TypeDef * FUARTx,  
           FUART_InitTypeDef * InitStruct)
```

Parameters:

FUARTx: The specified Full UART channel.

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

Description:

This API will initialize and configure the baud rate, the number of bits per transfer, stop bit, parity, transfer mode and flow control for the specified Full UART channel selected by **FUARTx**.

This API must be executed before Full UART is enabled.

Return:

None

8.2.3.11 FUART_EnableFIFO

Enable the transmit and receive FIFO.

Prototype:

```
void  
FUART_EnableFIFO(TSB_FUART_TypeDef * FUARTx)
```

Parameters:

FUARTx: The specified Full UART channel.

Description:

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

Return:

None

8.2.3.12 FUART_DisableFIFO

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

Prototype:

```
FUART_DisableFIFO(TSB_FUART_TypeDef * FUARTx)
```

Parameters:

FUARTx: The specified Full UART channel.

Description:

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

Return:
None

8.2.3.13 FUART_SetSendBreak

Generate the break condition for Full UART.

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

Parameters:

FUARTx: The specified Full UART channel.

NewState: New state of the FUART send break.

- **ENABLE:** Enable the send break to generate transmit break condition
- **DISABLE:** Disable the send break

Description:

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

Return:
None

8.2.3.14 FUART_SetIrDAEncodeMode

Select IrDA encoding mode for transmitting 0 bits.

Prototype:
void
FUART_SetIrDAEncodeMode(TSB_FUART_TypeDef * **FUARTx**,
uint32_t **Mode**)

Parameters:

FUARTx: The specified Full UART channel.

Mode: IrDA encoding mode select for transmitting 0 bits.

- **FUART_IRDA_3_16_BIT_PERIOD_MODE:** 0 bits are transmitted as an active high pulse of 3/16th of the bit period.
- **FUART_IRDA_3_TIMES_IRLPBAUD16_MODE:** 0 bits are transmitted with a pulse width that is 3 times the period of the IrLPBaud16 input signal.

Description:

This API selects IrDA encoding mode. Change IrDA encoding mode to **FUART_IRDA_3_TIMES_IRLPBAUD16_MODE** can reduce power consumption but might decrease transmission distance.

Return:
None

8.2.3.15 FUART_EnableIrDA

Enable the IrDA circuit.

Prototype:

Result

FUART_EnableIrDA(TSB_FUART_TypeDef * **FUARTx**)

Parameters:

FUARTx: The specified Full UART channel.

Description:

If Full UART is enabled, this API will enable IrDA circuit. If Full UART is disabled, this API will do nothing and return an error.

Return:

SUCCESS: Enable IrDA circuit successfully.

ERROR: The UART channel is disabled, can not enable IrDA circuit.

8.2.3.16 FUART_DisableIrDA

Disable the IrDA circuit.

Prototype:

void

FUART_DisableIrDA(TSB_FUART_TypeDef * **FUARTx**)

Parameters:

FUARTx: The specified Full UART channel.

Description:

If Full UART is enabled, this API will disable IrDA circuit. If Full UART is disabled, this API will do nothing, IrDA circuit doesn't work originally.

Return:

None

8.2.3.17 FUART_SetINTFIFOLevel

Set the Receive and Transmit interrupt FIFO level.

Prototype:

void

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

Parameters:

FUARTx: The specified Full UART channel.

RxLevel: Receive interrupt FIFO level. (Receive FIFO is 32 location deep)

- **FUART_RX_FIFO_LEVEL_4**: The data in Receive FIFO become >= 4 words
- **FUART_RX_FIFO_LEVEL_8**: The data in Receive FIFO become >= 8 words
- **FUART_RX_FIFO_LEVEL_16**: The data in Receive FIFO become >= 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

8.2.3.18 FUART_SetINTMask

Mask(Enable) interrupt source of the specified channel.

Prototype:

```
void  
FUART_SetINTMask(TSB_FUART_TypeDef * FUARTx,  
uint32_t IntMaskSrc)
```

Parameters:

FUARTx: The specified Full UART channel.

IntMaskSrc: The interrupt source to be masked(enabled).

To enable no interrupt, use the parameter:

- **FUART_NONE_INT_MASK**

To enable the interrupt one by one, use the “OR” operation with below parameter:

- **FUART_RIN_MODEM_INT_MASK:** Enable RIN interrupt
- **FUART_CTS_MODEM_INT_MASK:** Enable CTS modem interrupt
- **FUART_DCD_MODEM_INT_MASK:** Enable DCD modem interrupt
- **FUART_DSR_MODEM_INT_MASK:** Enable DSR modem interrupt
- **FUART_RX_FIFO_INT_MASK:** Enable receive FIFO interrupt
- **FUART_TX_FIFO_INT_MASK:** Enable transmit FIFO interrupt
- **FUART_RX_TIMEOUT_INT_MASK:** Enable receive timeout interrupt
- **FUART_FRAMING_ERR_INT_MASK:** Enable framing error interrupt
- **FUART_PARITY_ERR_INT_MASK:** Enable parity error interrupt
- **FUART_BREAK_ERR_INT_MASK:** Enable break error interrupt
- **FUART_OVERRUN_ERR_INT_MASK:** Enable overrun error interrupt

To enable all the interrupts, use the parameter:

- **FUART_ALL_INT_MASK**

Description:

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

Return:

None

8.2.3.19 FUART_GetINTMask

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

Prototype:

FUART_INTStatus

FUART_GetINTMask(TSB_FUART_TypeDef * *FUARTx*)**Parameters:***FUARTx*: The specified Full UART channel.**Description:**

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

Return:

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

8.2.3.20 FUART_GetRawINTStatus

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

Prototype:

FUART_INTStatus

FUART_GetRawINTStatus(TSB_FUART_TypeDef * *FUARTx*)**Parameters:***FUARTx*: The specified Full UART channel.**Description:**

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

Return:

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

8.2.3.21 FUART_GetMaskedINTStatus

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

Prototype:

FUART_INTStatus

FUART_GetMaskedINTStatus(TSB_FUART_TypeDef * *FUARTx*)**Parameters:***FUARTx*: The specified Full UART channel.**Description:**

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

Return:

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

8.2.3.22 FUART_ClearINT

Clear the interrupts of the specified Full UART channel.

Prototype:

```
void  
FUART_ClearINT(TSB_FUART_TypeDef * FUARTx,  
FUART_INTStatus INTStatus)
```

Parameters:

FUARTx: The specified Full UART channel.

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

Description:

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

Return:

None

8.2.3.23 FUART_SetDMAOnErr

Enable or disable the DMA receive request output on assertion of a UART error interrupt.

Prototype:

```
void  
FUART_SetDMAOnErr(TSB_FUART_TypeDef * FUARTx,  
FunctionalState NewState)
```

Parameters:

FUARTx: The specified Full UART channel.

NewState: New state of the DMA receive request output on assertion of a UART error interrupt.

- **ENABLE:** The DMA on error is available, the DMA receive request output, UARTRXDMSREQ or UARTRXDMABREQ, is disabled on assertion of a UART error interrupt.
- **DISABLE:** The DMA on error is not available, the DMA receive request output, UARTRXDMSREQ or UARTRXDMABREQ, is enabled on assertion of a UART error interrupt.

Description:

This API is used to enable or disable the DMA receive request output on assertion of a UART error interrupt.

Return:

None

8.2.3.24 FUART_SetFIFODMA

Enable or Disable the Transmit FIFO DMA or Receive FIFO DMA.

Prototype:

```
void
```

FUART_SetFIFODMA(TSB_FUART_TypeDef * **FUARTx**,
FUART_Direction **Direction**,
FunctionalState **NewState**)

Parameters:

FUARTx: The specified Full UART channel.

Direction: The direction of Full UART.

➤ **FUART_RX**: Receive FIFO

➤ **FUART_TX**: Transmit FIFO

NewState: New state of the FIFO DMA.

➤ **ENABLE**: Enable FIFO DMA

➤ **DISABLE**: Disable FIFO DMA

Description:

This API will enable or disable the Transmit FIFO DMA or Receive FIFO DMA.

The bus width must be set to 8-bits, if you transfer the data of transmit/ receive FIFO by using DMAC.

Return:

None

8.2.3.25 FUART_GetModemStatus

Get all the Modem Status, include: CTS, DSR, DCD, RIN, DTR, and RTS.

Prototype:

FUART_AllModemStatus

FUART_GetModemStatus(TSB_FUART_TypeDef * **FUARTx**)

Parameters:

FUARTx: The specified Full UART channel.

Description:

This API will get all the Modem Status, include: CTS, DSR, DCD, RIN, DTR, and RTS.

Return:

FUART_AllModemStatus: The union that indicates all the modem status.

(Refer to “Data Structure Description” for details).

8.2.3.26 FUART_SetRTSStatus

Set the Full UART RTS(Request To Send) modem status output.

Prototype:

void

FUART_SetRTSStatus(TSB_FUART_TypeDef * **FUARTx**,
FUART_ModemStatus **Status**)

Parameters:

FUARTx: The specified Full UART channel.

Status: RTS modem status output.

➤ **FUART_MODEM_STATUS_0**: The modem status output is 0.

➤ **FUART_MODEM_STATUS_1**: The modem status output is 1.

Description:

This API will set the Full UART RTS(Request To Send) modem status output.

Return:

None

8.2.3.27 FUART_SetDTRStatus

Set the Full UART DTR(Data Transmit Ready) modem status output.

Prototype:

```
void  
FUART_SetDTRStatus(TSB_FUART_TypeDef * FUARTx,  
FUART_ModemStatus Status)
```

Parameters:

FUARTx: The specified Full UART channel.

Status: DTR modem status output.

- **FUART_MODEM_STATUS_0**: The modem status output is 0.
- **FUART_MODEM_STATUS_1**: The modem status output is 1.

Description:

This API will set the Full UART DTR(Data Transmit Ready) modem status output.

Return:

None

8.2.4 Data Structure Description

8.2.4.1 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 2950000(bsp).

uint32_t

DataBits specifies data bits per transfer, which can be set as:

- **UART_DATA_BITS_5** for 5-bit mode
- **UART_DATA_BITS_6** for 6-bit mode
- **UART_DATA_BITS_7** for 7-bit mode
- **UART_DATA_BITS_8** for 8-bit mode

uint32_t

StopBits specifies the length of stop bit transmission, which can be set as:

- **UART_STOP_BITS_1** for 1 stop bit
- **UART_STOP_BITS_2** for 2 stop bits

uint32_t

Parity specifies the parity mode, which can be set as:

- **UART_NO_PARITY** for no parity
- **UART_0_PARITY** for 0 parity
- **UART_1_PARITY** for 1 parity
- **UART_EVEN_PARITY** for even parity
- **UART_ODD_PARITY** for odd parity

uint32_t

Mode enables or disables reception, transmission or both, which can be set as:

- **UART_ENABLE_TX** for enabling transmission
- **UART_ENABLE_RX** for enabling reception
- **UART_ENABLE_TX | UART_ENABLE_RX** for enabling both reception and transmission

uint32_t

FlowCtrl Enable or disable the hardware flow control, which can be set as:

- **UART_NONE_FLOW_CTRL** for no flow control
- **UART_CTS_FLOW_CTRL** for enabling CTS flow control
- **UART_RTS_FLOW_CTRL** for enabling RTS flow control
- **UART_CTS_FLOW_CTRL | UART_RTS_FLOW_CTRL** for enabling both CTS and RTS flow control

8.2.4.2 FUART_INTStatus

Data Fields:

uint32_t

All: Full UART interrupt status or mask.

Bit

uint32_t

RIN: 1 RIN modem interrupt

uint32_t

CTS: 1 CTS modem interrupt

uint32_t

DCD: 1 DCD modem interrupt

uint32_t

DSR: 1 DSR modem interrupt

uint32_t

RxFIFO: 1 Receive FIFO interrupt

uint32_t

TxFIFO: 1 Transmit FIFO interrupt

uint32_t

RxTimeout: 1 Receive timeout interrupt

uint32_t

FramingErr: 1 Framing error interrupt

uint32_t

ParityErr: 1 Parity error interrupt

uint32_t

BreakErr: 1 Break error interrupt

uint32_t

OverrunErr: 1 Overrun error interrupt

uint32_t

Reserved: 21 Reserved

8.2.4.3 FUART_AllModemStatus

Data Fields:

uint32_t

All: Full UART All Modem Status

Bit

uint32_t

| | |
|---------------------------------|--------------------|
| CTS: 1 uint32_t | CTS modem status d |
| DSR: 1 uint32_t | DSR modem status |
| DCD: 1 uint32_t | DCD modem status |
| Reserved1: 5 uint32_t | Reserved |
| RI: 1 uint32_t | RIN modem status |
| Reserved2: 1 uint32_t | Reserved |
| DTR: 1 uint32_t | DTR modem status |
| RTS: 1 uint32_t | RTS modem status |
| Reserved3: 20 | Reserved |

9. GPIO

9.1 Overview

For TOSHIBA TMPM461x 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/TX04_Periph_Driver/src/ tmpm461_gpio.c, with /Libraries/TX04_Periph_Driver/inc/tmpm461_gpio.h containing the macros, data types, structures and API definitions for use by applications.

9.2 API Functions

9.2.1 Function List

- ◆ uint16_t GPIO_ReadData(GPIO_Port **GPIO_x**)
- ◆ uint16_t GPIO_ReadDataBit(GPIO_Port **GPIO_x**, uint16_t **Bit_x**)
- ◆ void GPIO_WriteData(GPIO_Port **GPIO_x**, uint16_t **Data**)
- ◆ void GPIO_WriteDataBit(GPIO_Port **GPIO_x**, uint16_t **Bit_x**, uint16_t **BitValue**)
- ◆ void GPIO_Init(GPIO_Port **GPIO_x**, uint16_t **Bit_x**,
GPIO_InitTypeDef * **GPIO_InitStruct**)
- ◆ void GPIO_SetOutput(GPIO_Port **GPIO_x**, uint16_t **Bit_x**)
- ◆ void GPIO_SetInput(GPIO_Port **GPIO_x**, uint16_t **Bit_x**);
- ◆ void GPIO_SetOutputEnableReg(GPIO_Port **GPIO_x**, uint16_t **Bit_x**,
FunctionalState **NewState**)
- ◆ void GPIO_SetInputEnableReg(GPIO_Port **GPIO_x**, uint16_t **Bit_x**,
FunctionalState **NewState**)
- ◆ void GPIO_SetPullUp(GPIO_Port **GPIO_x**, uint16_t **Bit_x**,
FunctionalState **NewState**)
- ◆ void GPIO_SetPullDown(GPIO_Port **GPIO_x**, uint16_t **Bit_x**,
FunctionalState **NewState**)
- ◆ void GPIO_SetOpenDrain(GPIO_Port **GPIO_x**, uint16_t **Bit_x**,
FunctionalState **NewState**)
- ◆ void GPIO_EnableFuncReg(GPIO_Port **GPIO_x**, uint16_t **FuncReg_x**, uint16_t **Bit_x**)
- ◆ void GPIO_DisableFuncReg(GPIO_Port **GPIO_x**, uint16_t **FuncReg_x**, uint16_t **Bit_x**)

9.2.2 Detailed Description

Functions listed above can be divided into three parts:

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

9.2.3 Function Documentation

9.2.3.1 GPIO_ReadData

Read specified GPIO Data register.

Prototype:

```
uint16_t  
GPIO_ReadData(GPIO_Port GPIO_x)
```

Parameters:

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

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

Description:

This function will read specified GPIO Data register.

Return:

The value read from DATA register.

9.2.3.2 GPIO_ReadDataBit

Read specified GPIO pin.

Prototype:

```
uint16_t  
GPIO_ReadDataBit(GPIO_Port GPIO_x,  
uint16_t Bit_x)
```

Parameters:

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

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

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

- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,

- **GPIO_BIT_7**: GPIO pin 7,
- **GPIO_BIT_8**: GPIO pin 8,
- **GPIO_BIT_9**: GPIO pin 9,
- **GPIO_BIT_10**: GPIO pin 10,
- **GPIO_BIT_11**: GPIO pin 11,
- **GPIO_BIT_12**: GPIO pin 12,
- **GPIO_BIT_13**: GPIO pin 13,
- **GPIO_BIT_14**: GPIO pin 14,
- **GPIO_BIT_15**: GPIO pin 15.

Description:

This function will read specified GPIO pin.

Return:

The value read from GPIO pin as:

- **GPIO_BIT_VALUE_0**: Value 0,
- **GPIO_BIT_VALUE_1**: Value 1.

9.2.3.3 GPIO_WriteData

Write specified value to GPIO Data register.

Prototype:

```
void  
GPIO_WriteData(GPIO_Port GPIO_x,  
                uint16_t Data)
```

Parameters:

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

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

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

Description:

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

Return:

None

9.2.3.4 GPIO_WriteDataBit

Write specified value of single bit to GPIO pin.

Prototype:

```
void  
GPIO_WriteDataBit(GPIO_Port GPIO_x,  
                  uint16_t Bit_x,
```

uint16_t *BitValue*)

Parameters:

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

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

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

- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7.
- **GPIO_BIT_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15,
- **GPIO_BIT_ALL:** GPIO pin[0:15],
- Combination of the effective bits

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

- **GPIO_BIT_VALUE_0:** Clear GPIO pin,
- **GPIO_BIT_VALUE_1:** Set GPIO pin.

Description:

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

Return:

None

9.2.3.5 GPIO_Init

Initialize GPIO port function.

Prototype:

```
void  
GPIO_Init(GPIO_Port GPIO_x,  
          uint16_t Bit_x,  
          GPIO_InitTypeDef * GPIO_InitStruct)
```

Parameters:

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

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

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

- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7,
- **GPIO_BIT_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15,
- **GPIO_BIT_ALL:** GPIO pin[0:15],
- 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

9.2.3.6 GPIO_SetOutput

Set specified GPIO pin as output port.

Prototype:

```
void  
GPIO_SetOutput(GPIO_Port GPIO_x,  
               uint16_t Bit_x);
```

Parameters:

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

- **GPIO_PA:** GPIO port A.

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

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

- **GPIO_BIT_0**: GPIO pin 0,
- **GPIO_BIT_1**: GPIO pin 1,
- **GPIO_BIT_2**: GPIO pin 2,
- **GPIO_BIT_3**: GPIO pin 3,
- **GPIO_BIT_4**: GPIO pin 4,
- **GPIO_BIT_5**: GPIO pin 5,
- **GPIO_BIT_6**: GPIO pin 6,
- **GPIO_BIT_7**: GPIO pin 7,
- **GPIO_BIT_8**: GPIO pin 8,
- **GPIO_BIT_9**: GPIO pin 9,
- **GPIO_BIT_10**: GPIO pin 10,
- **GPIO_BIT_11**: GPIO pin 11,
- **GPIO_BIT_12**: GPIO pin 12,
- **GPIO_BIT_13**: GPIO pin 13,
- **GPIO_BIT_14**: GPIO pin 14,
- **GPIO_BIT_15**: GPIO pin 15,
- **GPIO_BIT_ALL**: GPIO pin[0:15],
- Combination of the effective bits.

Description:

This function will set specified GPIO pin as output port.

Return:

None

9.2.3.7 GPIO_SetInput

Set specified GPIO Pin as input port.

Prototype:

```
void  
GPIO_SetInput(GPIO_Port GPIO_x,  
              uint16_t Bit_x)
```

Parameters:

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

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

- **GPIO_PK** : GPIO port K.

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

- **GPIO_BIT_0**: GPIO pin 0,
- **GPIO_BIT_1**: GPIO pin 1,
- **GPIO_BIT_2**: GPIO pin 2,
- **GPIO_BIT_3**: GPIO pin 3,
- **GPIO_BIT_4**: GPIO pin 4,
- **GPIO_BIT_5**: GPIO pin 5,
- **GPIO_BIT_6**: GPIO pin 6,
- **GPIO_BIT_7**: GPIO pin 7,
- **GPIO_BIT_8**: GPIO pin 8,
- **GPIO_BIT_9**: GPIO pin 9,
- **GPIO_BIT_10**: GPIO pin 10,
- **GPIO_BIT_11**: GPIO pin 11,
- **GPIO_BIT_12**: GPIO pin 12,
- **GPIO_BIT_13**: GPIO pin 13,
- **GPIO_BIT_14**: GPIO pin 14,
- **GPIO_BIT_15**: GPIO pin 15,
- **GPIO_BIT_ALL**: GPIO pin[0:15],
- Combination of the effective bits.

Description:

This function will set specified GPIO pin as input port.

*To use the Port H as an analog input of the AD converter, disable input on PHIE and disable pull-up on PHPUP.

Return:

None

9.2.3.8 GPIO_SetOutputEnableReg

Enable or disable specified GPIO Pin output function.

Prototype:

```
void  
GPIO_SetOutputEnableReg(GPIO_Port GPIO_x,  
                        uint16_t Bit_x,  
                        FunctionalState NewState)
```

Parameters:

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

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

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

- **GPIO_BIT_0**: GPIO pin 0,

- **GPIO_BIT_1**: GPIO pin 1,
- **GPIO_BIT_2**: GPIO pin 2,
- **GPIO_BIT_3**: GPIO pin 3,
- **GPIO_BIT_4**: GPIO pin 4,
- **GPIO_BIT_5**: GPIO pin 5,
- **GPIO_BIT_6**: GPIO pin 6,
- **GPIO_BIT_7**: GPIO pin 7,
- **GPIO_BIT_8**: GPIO pin 8,
- **GPIO_BIT_9**: GPIO pin 9,
- **GPIO_BIT_10**: GPIO pin 10,
- **GPIO_BIT_11**: GPIO pin 11,
- **GPIO_BIT_12**: GPIO pin 12,
- **GPIO_BIT_13**: GPIO pin 13,
- **GPIO_BIT_14**: GPIO pin 14,
- **GPIO_BIT_15**: GPIO pin 15,
- **GPIO_BIT_ALL**: GPIO pin[0:15],
- Combination of the effective bits.

NewState:

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

Description:

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

Return:

None

9.2.3.9 GPIO_SetInputEnableReg

Enable or disable specified GPIO Pin input function.

Prototype:

```
void  
GPIO_SetInputEnableReg(GPIO_Port GPIO_x,  
                        uint16_t Bit_x,  
                        FunctionalState NewState)
```

Parameters:

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

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

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

- **GPIO_BIT_0**: GPIO pin 0,

- **GPIO_BIT_1**: GPIO pin 1,
- **GPIO_BIT_2**: GPIO pin 2,
- **GPIO_BIT_3**: GPIO pin 3,
- **GPIO_BIT_4**: GPIO pin 4,
- **GPIO_BIT_5**: GPIO pin 5,
- **GPIO_BIT_6**: GPIO pin 6,
- **GPIO_BIT_7**: GPIO pin 7,
- **GPIO_BIT_8**: GPIO pin 8,
- **GPIO_BIT_9**: GPIO pin 9,
- **GPIO_BIT_10**: GPIO pin 10,
- **GPIO_BIT_11**: GPIO pin 11,
- **GPIO_BIT_12**: GPIO pin 12,
- **GPIO_BIT_13**: GPIO pin 13,
- **GPIO_BIT_14**: GPIO pin 14,
- **GPIO_BIT_15**: GPIO pin 15,
- **GPIO_BIT_ALL**: GPIO pin[0:15],
- Combination of the effective bits.

NewState:

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

Description:

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

Return:

None

9.2.3.10 GPIO_SetPullUp

Enable or disable specified GPIO Pin pull-up function.

Prototype:

```
void  
GPIO_SetPullUp(GPIO_Port GPIO_x,  
                uint16_t Bit_x,  
                FunctionalState NewState)
```

Parameters:

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

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

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

- **GPIO_BIT_0**: GPIO pin 0,
- **GPIO_BIT_1**: GPIO pin 1,

- **GPIO_BIT_2**: GPIO pin 2,
- **GPIO_BIT_3**: GPIO pin 3,
- **GPIO_BIT_4**: GPIO pin 4,
- **GPIO_BIT_5**: GPIO pin 5,
- **GPIO_BIT_6**: GPIO pin 6,
- **GPIO_BIT_7**: GPIO pin 7,
- **GPIO_BIT_8**: GPIO pin 8,
- **GPIO_BIT_9**: GPIO pin 9,
- **GPIO_BIT_10**: GPIO pin 10,
- **GPIO_BIT_11**: GPIO pin 11,
- **GPIO_BIT_12**: GPIO pin 12,
- **GPIO_BIT_13**: GPIO pin 13,
- **GPIO_BIT_14**: GPIO pin 14,
- **GPIO_BIT_15**: GPIO pin 15,
- **GPIO_BIT_ALL**: GPIO pin[0:15],
- 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

9.2.3.11 GPIO_SetPullDown

Enable or disable specified GPIO Pin pull-down function.

Prototype:

```
void  
GPIO_SetPullDown(GPIO_Port GPIO_x,  
                 uint16_t Bit_x,  
                 FunctionalState NewState)
```

Parameters:

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

- **GPIO_PE**: GPIO port E.

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

- **GPIO_BIT_4**: GPIO pin 4,
- **GPIO_BIT_ALL**: GPIO pin[0:15],
- 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

9.2.3.12 GPIO_SetOpenDrain

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

Prototype:

```
void  
GPIO_SetOpenDrain(GPIO_Port GPIO_x,  
                  uint16_t Bit_x,  
                  FunctionalState NewState)
```

Parameters:

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

- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PK :** GPIO port K.

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

- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7,
- **GPIO_BIT_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15,
- **GPIO_BIT_ALL:** GPIO pin[0:15],
- Combination of the effective bits.

NewState:

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

Description:

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

Return:
None

9.2.3.13 GPIO_EnableFuncReg

Enable specified GPIO function.

Prototype:

```
void  
GPIO_EnableFuncReg(GPIO_Port GPIO_x,  
                    uint16_t FuncReg_x,  
                    uint16_t Bit_x);
```

Parameters:

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

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

FuncReg_x: The number of GPIO function register, which can be set as:

- **GPIO_FUNC_REG_1** for GPIO function register 1,
- **GPIO_FUNC_REG_2** for GPIO function register 2,
- **GPIO_FUNC_REG_3** for GPIO function register 3,
- **GPIO_FUNC_REG_4** for GPIO function register 4,
- **GPIO_FUNC_REG_5** for GPIO function register 5,

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

- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7,
- **GPIO_BIT_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15,
- **GPIO_BIT_ALL:** GPIO pin[0:15],
- Combination of the effective bits.

Description:

This function will enable GPIO pin specified function.

Return:
None

9.2.3.14 GPIO_DisableFuncReg

Disable specified GPIO function.

Prototype:

```
void  
GPIO_DisableFuncReg(GPIO_Port GPIO_x,  
                    uint16_t FuncReg_x,  
                    uint16_t Bit_x)
```

Parameters:

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

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

FuncReg_x: The number of GPIO function register, which can be set as:

- **GPIO_FUNC_REG_1** for GPIO function register 1,
- **GPIO_FUNC_REG_2** for GPIO function register 2,
- **GPIO_FUNC_REG_3** for GPIO function register 3,
- **GPIO_FUNC_REG_4** for GPIO function register 4,
- **GPIO_FUNC_REG_5** for GPIO function register 5,

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

- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7.
- **GPIO_BIT_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15,
- **GPIO_BIT_ALL:** GPIO pin[0:15],
- Combination of the effective bits.

Description:

This function will disable GPIO pin specified function.

Return:
None

9.2.4 Data Structure Description

9.2.4.1 GPIO_InitTypeDef

Data Fields:

uint16_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.

uint16_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.

uint16_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.

uint16_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.

9.2.4.2 GPIO_RegTypeDef

Data Fields:

uint16_t

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

uint16_t

PinCR Port x output control register.

- "0": output disable.
- "1": output enable.

uint16_t

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

uint16_t

PinOD Port x open drain control register.

- "0": CMOS
- "1": Open Drain

uint16_t

PinPUP Port x pull-up control register:

- "0": Pull-up disable.
- "1": Pull-up enable.

uint16_t

PinPDN Port x pull-down control register :

- "0": Pull-down disable.
- "1": Pull-down enable.

uint16_t

PinPIE Port x input control register:

- "0": Input disable.
- "1": Input enable.

9.2.4.3 TSB_Port_TypeDef

Data Fields:

__IO uint32_t

DATA The "DATA" can be read and written

__IO uint32_t

PinCR The "CR" can be read and written.

__IO uint32_t

PinFR[FRMAX] The "FR[FRMAX]" can be read and written

uint32_t

RESERVED0[RESER] Reserved

__IO uint32_t

PinOD The "OD" can be read and written

__IO uint32_t

PinPUP The "PUP" can be read and written

__IO uint32_t

PinPDN The "PDN" can be read and written:

uint32_t

RESERVED1[RESER] Reserved

__IO uint32_t

PinPIE Port x input control register:

10. I2C

10.1 Overview

The TMPM461x contains I2C Bus Interface with 5 channels(I2C0~4).

The I2C bus is connected to external devices via SCL and SDA, and it can communicate with multiple devices.

Data can be transferred in free data format by the I2C channels. In free data format, data is always sent by master-transmitter and received by slave-receiver.

The I2C driver APIs provide a set of functions to configure each channel such as setting self-address of the I2C 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 I2C channel.

All driver APIs are contained in /Libraries/TX04_Periph_Driver/src/tmpm461_i2c.c, with /Libraries/TX04_Periph_Driver/inc/tmpm461_i2c.h containing the macros, data types, structures and API definitions for use by applications.

10.2 API Functions

10.2.1 Function List

- ◆ void I2C_SetACK(TSB_I2C_TypeDef* **I2Cx**, FunctionalState **NewState**);
- ◆ void I2C_Init(TSB_I2C_TypeDef* **I2Cx**, I2C_InitTypeDef* **InitI2CStruct**);
- ◆ void I2C_SetBitNum(TSB_I2C_TypeDef* **I2Cx**, uint32_t **I2CBitNum**);
- ◆ void I2C_SWReset(TSB_I2C_TypeDef* **I2Cx**);
- ◆ void I2C_ClearINTReq(TSB_I2C_TypeDef* **I2Cx**);
- ◆ void I2C_GenerateStart(TSB_I2C_TypeDef* **I2Cx**);
- ◆ void I2C_GenerateStop(TSB_I2C_TypeDef* **I2Cx**);
- ◆ I2C_State I2C_GetState(TSB_I2C_TypeDef* **I2Cx**);
- ◆ void I2C_SetSendData(TSB_I2C_TypeDef* **I2Cx**, uint32_t **Data**);
- ◆ uint32_t I2C_GetReceiveData(TSB_I2C_TypeDef* **I2Cx**);
- ◆ void I2C_SetFreeDataMode(TSB_I2C_TypeDef* **I2Cx**, FunctionalState **NewState**);
- ◆ FunctionalState I2C_GetSlaveAddrMatchState(TSB_I2C_TypeDef * **I2Cx**);
- ◆ void I2C_SetPrescalerClock(TSB_I2C_TypeDef * **I2Cx**, uint32_t **PrescalerClock**);
- ◆ void I2C_SetINTReq(TSB_I2C_TypeDef * **I2Cx**,FunctionalState **NewState**);
- ◆ FunctionalState I2C_GetINTStatus(TSB_I2C_TypeDef * **I2Cx**);
- ◆ void I2C_ClearINTOutput(TSB_I2C_TypeDef * **I2Cx**);

10.2.2 Detailed Description

Functions listed above can be divided into four parts:

- 1) Configure and control the common functions of each I2C channel are handled by I2C_SetACK(), I2C_SetBitNum(),I2C_SetPrescalerClock()and I2C_Init().
- 2) Transfer control of each I2C channel is handled by I2C_ClearINTReq(), I2C_Generatstart(), I2C_Generatstop(),I2C_SetSendData(), I2C_GetReceiveData(),I2C_SetINTReq(),I2C_ClearINTOutput().
- 3) The status indication of each I2C channel is handled by I2C_GetState(),I2C_GetSlaveAddrMatchState() and I2C_GetINTStatus().
- 4) I2C_SWReset() and I2C_SetFreeDataMode() handle other specified functions.

10.2.3 Function Documentation

***Note:** in all of the following APIs, parameter “TSB_I2C_TypeDef* *I2Cx*” can be one of the following values:

TSB_I2C0, TSB_I2C1, TSB_I2C2, TSB_I2C3, TSB_I2C4

10.2.3.1 I2C_SetACK

Enable or disable the generation of ACK clock.

Prototype:

```
void  
I2C_SetACK(TSB_I2C_TypeDef* I2Cx,  
           FunctionalState NewState)
```

Parameters:

I2Cx is the specified I2C 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

10.2.3.2 I2C_Init

Initialize the specified I2C channel in I2C mode.

Prototype:

```
void  
I2C_Init(TSB_I2C_TypeDef* I2Cx,  
         I2C_InitTypeDef* InitI2CStruct)
```

Parameters:

I2Cx is the specified I2C channel.

InitI2CStruct is the structure containing I2C 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 I2C channel selected by *I2Cx*.

Return:

None

10.2.3.3 I2C_SetBitNum

Specify the number of bits per transfer.

Prototype:

```
void  
I2C_SetBitNum(TSB_I2C_TypeDef* I2Cx,  
              uint32_t I2CBitNum)
```

Parameters:

I2Cx is the specified I2C channel.

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

This parameter can be one of the following values:

- **I2C_DATA_LEN_8**, which means that the data length number of bits per transfer is 8;
- **I2C_DATA_LEN_1**, which means that the data length number of bits per transfer is 1;
- **I2C_DATA_LEN_2**, which means that the data length number of bits per transfer is 2;
- **I2C_DATA_LEN_3**, which means that the data length number of bits per transfer is 3;
- **I2C_DATA_LEN_4**, which means that the data length number of bits per transfer is 4;
- **I2C_DATA_LEN_5**, which means that the data length number of bits per transfer is 5;
- **I2C_DATA_LEN_6**, which means that the data length number of bits per transfer is 6;
- **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

10.2.3.4 I2C_SWReset

Reset the state of the specified I2C channel.

Prototype:

```
void  
I2C_SWReset(TSB_I2C_TypeDef* I2Cx)
```

Parameters:

I2Cx is the specified I2C 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

10.2.3.5 I2C_ClearINTReq

Clear I2C interrupt request in I2C bus mode.

Prototype:

void
I2C_ClearINTReq(TSB_I2C_TypeDef* **I2Cx**)

Parameters:

I2Cx is the specified I2C channel.

Description:

This function will clear the I2C interrupt, which has occurred, of the specified I2C channel.

Return:

None

10.2.3.6 I2C_GenerateStart

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

Prototype:

void
I2C_GenerateStart(TSB_I2C_TypeDef* **I2Cx**)

Parameters:

I2Cx is the specified I2C channel.

Description:

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

Return:

None

10.2.3.7 I2C_GenerateStop

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

Prototype:

void
I2C_GenerateStop(TSB_I2C_TypeDef* **I2Cx**)

Parameters:

I2Cx is the specified I2C channel.

Description:

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

Return:

None

10.2.3.8 I2C_GetState

Get the I2C channel state in I2C bus mode.

Prototype:

```
I2C_State  
I2C_GetState(TSB_I2C_TypeDef* I2Cx)
```

Parameters:

I2Cx is the specified I2C channel.

Description:

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

Return:

The state value of the I2C channel in I2C bus.

10.2.3.9 I2C_SetSendData

Set data to be sent and start transmitting from the specified I2C channel.

Prototype:

```
void  
I2C_SetSendData(TSB_I2C_TypeDef* I2Cx,  
                uint32_t Data)
```

Parameters:

I2Cx is the specified I2C 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 I2C channel selected by **I2Cx**. It is appropriate to call the function after the transmission of the start condition, which can be done by **I2C_Generatestart()**, or the reception of an ACK (usually causes an I2C interrupt), to send further data required by receiver.

Return:

None

10.2.3.10 I2C_GetReceiveData

Get data received from the specified I2C channel.

Prototype:

```
uint32_t  
I2C_GetReceiveData(TSB_I2C_TypeDef* I2Cx)
```

Parameters:

I2Cx is the specified I2C channel.

Description:

This function will set the data to be sent from the specified I2C channel selected by **I2Cx**. It is appropriate to call the function after the transmission of the start condition,

which can be done by **I2C_Generatestart()**, or the reception of an ACK (usually causes an I2C interrupt), to send further data required by receiver.

Return:

Data which has been received

10.2.3.11 I2C_SetFreeDataMode

Set I2C channel working in I2C free data mode.

Prototype:

```
void  
I2C_SetFreeDataMode(TSB_I2C_TypeDef* I2Cx,  
                    FunctionalState NewState)
```

Parameters:

I2Cx is the specified I2C channel.

NewState specifies the state of the I2C when system is idle mode, which can be

- **ENABLE:** enables the I2C channel.
- **DISABLE:** disables the I2C channel.

Description:

The specified I2C 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 I2C is needed to shift to transfer data in normal I2C format, call **I2C_Init()**.

Return:

None

10.2.3.12 I2C_GetSlaveAddrMatchState

Get slave address match detection state.

Prototype:

```
FunctionalState  
I2C_GetSlaveAddrMatchState(TSB_I2C_TypeDef* I2Cx)
```

Parameters:

I2Cx is the specified I2C channel.

Description:

Get slave address match detection state.

Return:

The state of match detection

ENABLE: Slave address is matched.

DISABLE: Slave address is unmatched.

10.2.3.13 I2C_SetPrescalerClock

Set prescaler clock of the specified I2C channel.

Prototype:

```
void  
I2C_SetPrescalerClock(TSB_I2C_TypeDef* I2Cx,  
                      uint32_t PrescalerClock)
```

Parameters:

I2Cx is the specified I2C channel.

PrescalerClock is the prescaler clock value.

This parameter can be one of the following values:

- **I2C_PRESCALER_DIV_1** to **I2C_PRESCALER_DIV_32**

Description:

This function will set prescaler clock of the specified I2C channel, The system clock(fsys) is divided according to **PrescalerClock** as the prescaler clock(fprsck), and the prescaler clock is further divided by **I2CClkDiv** (refer to Data Structure Description for details).and used as the serial clock for I2C transfer, Make sure the prescaler clock in the range between 50ns and 150ns.

Return:

None

10.2.3.14 I2C_SetINTReq

Enable or disable interrupt request of the I2C channel.

Prototype:

```
void  
I2C_SetINTReq(TSB_I2C_TypeDef* I2Cx,  
              FunctionalState NewState)
```

Parameters:

I2Cx is the specified I2C channel.

NewState: Specify I2C interrupt setting

This parameter can be one of the following values:

- **ENABLE** : Enable I2C interrupt
- **DISABLE** : Disable I2C interrupt

Description:

This function will enable or disable I2C interrupt request.

Return:

None

10.2.3.15 I2C_GetINTStatus

Get interrupt generation state.

Prototype:

```
FunctionalState  
I2C_GetINTStatus(TSB_I2C_TypeDef* I2Cx)
```

Parameters:

I2Cx is the specified I2C channel.

Description:

This function will get the state of I2C interrupt generation.

Return:

The state of interrupt generation

ENABLE: I2C interrupt has been generated

DISABLE: I2C has not interrupt

10.2.3.16 I2C_ClearINTReq

Clear the I2C interrupt output.

Prototype:

void

I2C_ClearINTOutput(TSB_I2C_TypeDef* *I2Cx*)

Parameters:

I2Cx is the specified I2C channel.

Description:

This function will clear the I2C interrupt output, which has occurred, of the specified I2C channel.

Return:

None

10.2.4 Data Structure Description

10.2.4.1 I2C_InitTypeDef

Data Fields:

uint32_t

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

uint32_t

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

- **I2C_DATA_LEN_8**, which means that the data length number of bits per transfer is 8;
- **I2C_DATA_LEN_1**, which means that the data length number of bits per transfer is 1;
- **I2C_DATA_LEN_2**, which means that the data length number of bits per transfer is 2;
- **I2C_DATA_LEN_3**, which means that the data length number of bits per transfer is 3;
- **I2C_DATA_LEN_4**, which means that the data length number of bits per transfer is 4;
- **I2C_DATA_LEN_5**, which means that the data length number of bits per transfer is 5;
- **I2C_DATA_LEN_6**, which means that the data length number of bits per transfer is 6;
- **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 prescaler clock for I2C transfer, which can be set as:

- **I2C_SCK_CLK_DIV_20**, which means that the frequency of the serial clock for I2C transfer is quotient of fprsck divided by 20;
- **I2C_SCK_CLK_DIV_24**, which means that the frequency of the serial clock for I2C transfer is quotient of fprsck divided by 24;
- **I2C_SCK_CLK_DIV_32**, which means that the frequency of the serial clock for I2C transfer is quotient of fprsck divided by 32;
- **I2C_SCK_CLK_DIV_48**, which means that the frequency of the serial clock for I2C transfer is quotient of fprsck divided by 48;
- **I2C_SCK_CLK_DIV_80**, which means that the frequency of the serial clock for I2C transfer is quotient of fprsck divided by 80;
- **I2C_SCK_CLK_DIV_144**, which means that the frequency of the serial clock for I2C transfer is quotient of fprsck divided by 144;
- **I2C_SCK_CLK_DIV_272**, which means that the frequency of the serial clock for I2C transfer is quotient of fprsck divided by 272;
- **I2C_SCK_CLK_DIV_528**, which means that the frequency of the serial clock for I2C transfer is quotient of fprsck divided by 528;

uint32_t

PrescalerClkDiv specifies the division of the system clock for generating the fprsck, which can be set as:

- **I2C_PRESCALER_DIV_1**, which means that the frequency of the prescaler clock is quotient of fsys divided by 1
- **I2C_PRESCALER_DIV_2**, which means that the frequency of the prescaler clock is quotient of fsys divided by 2
- **I2C_PRESCALER_DIV_3**, which means that the frequency of the prescaler clock is quotient of fsys divided by 3
- **I2C_PRESCALER_DIV_4**, which means that the frequency of the prescaler clock is quotient of fsys divided by 4
- **I2C_PRESCALER_DIV_5**, which means that the frequency of the prescaler clock is quotient of fsys divided by 5
- **I2C_PRESCALER_DIV_6**, which means that the frequency of the prescaler clock is quotient of fsys divided by 6
- **I2C_PRESCALER_DIV_7**, which means that the frequency of the prescaler clock is quotient of fsys divided by 7
- **I2C_PRESCALER_DIV_8**, which means that the frequency of the prescaler clock is quotient of fsys divided by 8
- **I2C_PRESCALER_DIV_9**, which means that the frequency of the prescaler clock is quotient of fsys divided by 9
- **I2C_PRESCALER_DIV_10**, which means that the frequency of the prescaler clock is quotient of fsys divided by 10
- **I2C_PRESCALER_DIV_11**, which means that the frequency of the prescaler clock is quotient of fsys divided by 11
- **I2C_PRESCALER_DIV_12**, which means that the frequency of the prescaler clock is quotient of fsys divided by 12
- **I2C_PRESCALER_DIV_13**, which means that the frequency of the prescaler clock is quotient of fsys divided by 13
- **I2C_PRESCALER_DIV_14**, which means that the frequency of the prescaler clock is quotient of fsys divided by 14
- **I2C_PRESCALER_DIV_15**, which means that the frequency of the prescaler clock is quotient of fsys divided by 15
- **I2C_PRESCALER_DIV_16**, which means that the frequency of the prescaler clock is quotient of fsys divided by 16

- **I2C_PRESCALER_DIV_17**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 17
- **I2C_PRESCALER_DIV_18**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 18
- **I2C_PRESCALER_DIV_19**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 19
- **I2C_PRESCALER_DIV_20**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 20
- **I2C_PRESCALER_DIV_21**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 21
- **I2C_PRESCALER_DIV_22**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 22
- **I2C_PRESCALER_DIV_23**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 23
- **I2C_PRESCALER_DIV_24**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 24
- **I2C_PRESCALER_DIV_25**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 25
- **I2C_PRESCALER_DIV_26**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 26
- **I2C_PRESCALER_DIV_27**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 27
- **I2C_PRESCALER_DIV_28**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 28
- **I2C_PRESCALER_DIV_29**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 29
- **I2C_PRESCALER_DIV_30**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 30
- **I2C_PRESCALER_DIV_31**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 31
- **I2C_PRESCALER_DIV_32**, which means that the frequency of the prescaler clock is quotient of f_{sys} divided by 32

***Note:** Make sure the prescaler clock in the range between 50ns and 150ns.

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.

10.2.4.2 I2C_State

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.

11. IGBT

11.1 Overview

TMPM461x contains 2 channels multi-purpose timer (MPT). MPT can operate in IGBT mode.

There are the following functions in IGBT mode,

- 1) 16-bit programmable square-wave output mode (PPG, two waves),
- 2) External trigger starting,
- 3) Period matching detection function,
- 4) Emergency stop function
- 5) Synchronous start mode

The IGBT driver APIs provide a set of functions to control IGBT module, including setting start mode, operation mode, counter state, source clock division, initial output level, trigger/EMG noise elimination division, changing output active/inactive timing, output wave period, EMG output and so on.

All driver APIs are contained in /Libraries/TX04_Periph_Driver/src/tmpm461_igbt.c, with /Libraries/TX04_Periph_Driver/inc/ tmpm461_igbt.h containing the macros, data types, structures and API definitions for use by applications.

11.2 API Functions

11.2.1 Function List

- ◆ void IGBT_Enable(TSB_MT_TypeDef* **IGBTx**)
- ◆ void IGBT_Disable(TSB_MT_TypeDef* **IGBTx**)
- ◆ void IGBT_SetClkInCoreHalt(TSB_MT_TypeDef* **IGBTx**, uint8_t **ClkState**)
- ◆ void IGBT_SetSWRunState(TSB_MT_TypeDef* **IGBTx**, uint8_t **Cmd**)
- ◆ uint16_t IGBT_GetCaptureValue(TSB_MT_TypeDef* **IGBTx**, uint8_t **CapReg**)
- ◆ void IGBT_Init(TSB_MT_TypeDef* **IGBTx**, IGBT_InitTypeDef* **InitStruct**)
- ◆ void IGBT_Recount(TSB_MT_TypeDef* **IGBTx**)
- ◆ void IGBT_ChangeOutputActiveTiming(TSB_MT_TypeDef* **IGBTx**, uint8_t **Output**, uint16_t **Timing**)
- ◆ void IGBT_ChangeOutputInactiveTiming(TSB_MT_TypeDef* **IGBTx**, uint8_t **Output**, uint16_t **Timing**)
- ◆ void IGBT_ChangePeriod(TSB_MT_TypeDef* **IGBTx**, uint16_t **Period**)
- ◆ WorkState IGBT_GetCntState(TSB_MT_TypeDef* **IGBTx**)
- ◆ Result IGBT_CancelEMGState(TSB_MT_TypeDef* **IGBTx**)
- ◆ IGBT_EMGStateTypeDef IGBT_GetEMGState(TSB_MT_TypeDef * **IGBTx**)
- ◆ void IGBT_ChangeTrgValue(TSB_MT_TypeDef* **IGBTx**, uint16_t **uTrgCnt**)
- ◆ void IGBT_CISynSlaveChCounter(TSB_MT_TypeDef * **IGBTx**)

11.2.2 Detailed Description

Functions listed above can be divided into five parts:

- 1) Initialize and configure the common functions of each IGBT channel are handled by IGBT_Enable(), IGBT_Disable() and IGBT_Init().
- 2) The counter state and control of each IGBT channel are handled by IGBT_SetClkInCoreHalt(), IGBT_SetSWRunState(), IGBT_Recount() and IGBT_GetCntState().

- 3) Changing Operation parameters and getting captured value of each IGBT channel are handled by IGBT_GetCaptureValue(), IGBT_ChangeOutputActiveTiming(), IGBT_ChangeOutputInactiveTiming() and IGBT_ChangePeriod().
- 4) Getting and cancel the EMG protection state is handled by IGBT_GetEMGState () and IGBT_CancelEMGState().
- 5) Change the Trigger value counter and clear synchronouse slave channel up counter are handled by IGBT_ChangeTrgValue() and IGBT_CISynSlaveChCounter().

11.2.3 Function Documentation

***Note:** in all of the following APIs, parameter “TSB_MT_TypeDef* *IGBTx*” can be one of the following values:
IGBT0 and **IGBT1**.

11.2.3.1 IGBT_Enable

Enable the specified MPT channel in IGBT mode.

Prototype:

```
void  
IGBT_Enable(TSB_MT_TypeDef* IGBTx)
```

Parameters:

IGBTx is the specified MPT channel in IGBT mode.

Description:

This function will enable the specified MPT channel selected by *IGBTx*. After calling this API, the MPT operates in IGBT mode and the specified channel should be initialized and configured by **IGBT_Init()**.

Return:

None

11.2.3.2 IGBT_Disable

Disable the specified MPT channel operating in IGBT mode.

Prototype:

```
void  
IGBT_Disable(TSB_MT_TypeDef* IGBTx)
```

Parameters:

IGBTx is the specified MPT channel in IGBT mode.

Description:

This function will disable the specified MPT channel selected by *IGBTx*.

Return:

None

11.2.3.3 IGBT_SetClkInCoreHalt

Set the clock stop or not in Core Halt when the specified MPT channel operates in IGBT mode.

Prototype:

```
void  
IGBT_SetClkInCoreHalt(TSB_MT_TypeDef* IGBTx,  
                      uint8_t ClkState)
```

Parameters:

IGBTx is the specified MPT channel in IGBT mode.

ClkState specify the control in Core Halt during debug mode, which can be one of the values below:

- **IGBT_RUNNING_IN_CORE_HALT**, clock does not stop and outputs are not controlled,
- **IGBT_STOP_IN_CORE_HALT**, clock stops and output are controlled in accordance with configuration.

Description:

In Core Halt during debug mode, the clock of IGBT mode can stop or keep running, which depends on **ClkState**. It's highly recommended to select **IGBT_STOP_IN_CORE_HALT** as **ClkState**.

Return:

None

11.2.3.4 IGBT_SetSWRunState

Start or stop the counter in IGBT mode by software command.

Prototype:

```
void  
IGBT_SetSWRunState(TSB_MT_TypeDef* IGBTx,  
                  uint8_t Cmd)
```

Parameters:

IGBTx is the specified MPT channel in IGBT mode.

Cmd is the command of controlling the counter, which can be one of the values below,

- **IGBT_RUN**, the command of starting the counter,
- **IGBT_STOP**, the command of stopping the counter.

Description:

This function can start or stop the counter by software.

Return:

None

***Note:**

1) The actual timing of counter starting or stopping depends on the configuration of IGBT mode. If **IGBT_CMD_START** or **IGBT_CMD_START_NO_START_INT** is selected as **StartMode** (refer to Data Structure Description for details), this API can fully control the counter. If **StartMode** is set as other values, trigger can also control the counter.

- 2) If **IGBT_FALLING_TRG_START** or **IGBT_RISING_TRG_START** is selected as **StartMode** (refer to Data Structure Description for details), after the initialization and configuration is complete, the software start command, **IGBT_SetSWRunState(IGBTx, IGBT_RUN)**, must be issued before the start trigger. Then the counter can be started by the trigger.
- 3) When EMG input is low level and EMG interrupt occurs, please use **IGBT_SetSWRunState(IGBTx, IGBT_STOP)** to stop the counter.

11.2.3.5 IGBT_GetCaptureValue

Get the captured counter value in IGBT mode.

Prototype:

```
uint16_t  
IGBT_GetCaptureValue(TSB_MT_TypeDef* IGBTx,  
                    uint8_t CapReg)
```

Parameters:

IGBTx is the specified MPT channel in IGBT mode.

CapReg selects the capture register, which can be one of the values below,

- **IGBT_CAPTURE_0**, capture register 0,
- **IGBT_CAPTURE_1**, capture register 1.

Description:

This function returns the value captured and stored in the specified capture register.

Return:

The value captured

***Note:**

Only when **IGBT_CMD_START** or **IGBT_CMD_START_NO_START_INT** is selected as **StartMode** (refer to Data Structure Description for details), the counter value can be captured and be got by calling this function. The timing of the first input edge will be captured and stored in capture register 0. And the timing of the second input edge will be captured and stored in the capture register 1.

11.2.3.6 IGBT_Init

Initialize and configure the specified MPT channel in IGBT mode.

Prototype:

```
void  
IGBT_Init(TSB_MT_TypeDef* IGBTx,  
         IGBT_InitTypeDef* InitStruct)
```

Parameters:

IGBTx is the specified MPT channel in IGBT mode.

InitStruct is the structure containing IGBT configuration including start mode, operation mode, output in stop state, start trigger acceptance mode, interrupt period, source clock division, initialization of output0/1, noise elimination time division for trigger input and EMG input, active and inactive timing of output0/1, the period of IGBT output wave and EMG function setting (refer to Data Structure Description for details).

Description:

After enabling the IGBT mode by calling **IGBT_Enable()**, this function can be used to initialize and configure the specified MPT channel in IGBT mode.

Return:

None

***Note:**

The corresponding I/O ports must be set as EMG input pins when MPTs operate in IGBT mode.

Call this function when **IGBT_CancelEMGState()** returns **SUCCESS** only, otherwise the initialization and configuration will not take effect.

11.2.3.7 IGBT_Recount

Clear and restart the counter.

Prototype:

```
void  
IGBT_Recount(TSB_MT_TypeDef* IGBTx)
```

Parameters:

IGBTx is the specified MPT channel in IGBT mode.

Description:

While the counter is running, call this function will make counter restart counting from 0.

Return:

None

11.2.3.8 IGBT_ChangeOutputActiveTiming

Change the active timing of IGBT output 0 or output 1.

Prototype:

```
void  
IGBT_ChangeOutputActiveTiming(TSB_MT_TypeDef* IGBTx,  
                               uint8_t Output,  
                               uint16_t Timing)
```

Parameters:

IGBTx is the specified MPT channel in IGBT mode.

Output selects the IGBT output port, which can be

- **IGBT_OUTPUT_0**, IGBT output port 0,
- **IGBT_OUTPUT_1**, IGBT output port 1.

Timing specifies the new output active timing. The value must be set between 0 and the output inactive timing.

Description:

This function is used to change the active timing of output. But the new active timing will take effect after the counter matches the period value.

Return:

None

11.2.3.9 IGBT_ChangeOutputInactiveTiming

Change the inactive timing of IGBT output 0 or output 1.

Prototype:

```
void  
IGBT_ChangeOutputInactiveTiming(TSB_MT_TypeDef* IGBTx,  
                                uint8_t Output,  
                                uint16_t Timing)
```

Parameters:

IGBTx is the specified MPT channel in IGBT mode.

Output selects the IGBT output port, which can be

- **IGBT_OUTPUT_0**, IGBT output port 0,
- **IGBT_OUTPUT_1**, IGBT output port 1.

Timing specifies the new output inactive timing. The value must be set between the output active timing and the **Period**.

Description:

This function is used to change the inactive timing of output. But the new inactive timing will take effect after the counter matches the period value.

Return:

None

11.2.3.10 IGBT_ChangePeriod

Change the period of IGBT output.

Prototype:

```
void  
IGBT_ChangePeriod(TSB_MT_TypeDef* IGBTx,  
                  uint16_t Period)
```

Parameters:

IGBTx is the specified MPT channel in IGBT mode.

Period specifies the new output period. The value must be set between the output inactive timing and 0xFFFF.

Description:

This function is used to change the period of output. But the new period will take effect after the counter matches the previous period value.

Return:

None

11.2.3.11 IGBT_GetCntState

Get the counter state.

Prototype:

WorkState
IGBT_GetCntState(TSB_MT_TypeDef* **IGBTx**)

Parameters:

IGBTx is the specified MPT channel in IGBT mode.

Description:

This function is used to get the counter state.

Return:

The counter state, which can be:

BUSY, the counter is running.

DONE, the counter stops.

11.2.3.12 IGBT_CancelEMGState

Cancel the EMG state of IGBT.

Prototype:

Result

IGBT_CancelEMGState(TSB_MT_TypeDef* **IGBTx**)

Parameters:

IGBTx is the specified MPT channel in IGBT mode.

Description:

This function is used to cancel the EMG state of IGBT. Before canceling the EMG state, call **IGBT_GetCntState()** to check the state of the counter and make sure that the EMG input level is H.

If the counter is running (**IGBT_GetCntState()** returns **BUSY**) or the EMG input is driven to L, it returns **ERROR** and the EMG state is not cancelled.

If the counter stops (**IGBT_GetCntState()** returns **DONE**) and the EMG input is driven to H, it cancels the EMG state and returns **SUCCESS**.

Return:

The result of EMG state canceling, which can be

SUCCESS, EMG state of IGBT is cancelled.

ERROR, EMG state of IGBT is not cancelled.

11.2.3.13 IGBT_GetEMGState

Get the EMG state of IGBT.

Prototype:

IGBT_EMGStateTypeDef

IGBT_GetEMGState(TSB_MT_TypeDef * **IGBTx**)

Parameters:

IGBTx is the specified MPT channel in IGBT mode.

Description:

This function is used to get the EMG state of IGBT, which includes EMG input pin status after noise elimination and EMG protection status.

Return:

The EMG status enumeration structure *IGBT_EMGStateTypeDef* of EMG state (refer to Data Structure Description for details).

11.2.3.14 IGBT_ChangeTrgValue

Change the Trigger value of IGBT output.

Prototype:

void

IGBT_ChangeTrgValue(TSB_MT_TypeDef* *IGBTx*, uint16_t *uTrgCnt*)

Parameters:

IGBTx is the specified MPT channel in IGBT mode.

uTrgCnt is the new IGBT up-counter.

Description:

Change the Trigger value of IGBT output.

Return:

None

11.2.3.15 IGBT_CISynSlaveChCounter

Clear synchronous counter clear, configure slave channels

Prototype:

void

IGBT_CISynSlaveChCounter(TSB_MT_TypeDef * *IGBTx*)

Parameters:

IGBTx is the specified MPT channel in IGBT mode.

Description:

Clear synchronous counter, configure of slave channels

Return:

None

11.2.4 Data Structure Description

11.2.4.1 IGBT_InitTypeDef

Data Fields:

uint8_t

StartMode selects start mode of counter, which could be

➤ **IGBT_CMD_START**, counter is controlled by software command and the timing of input edge can be captured.

➤ **IGBT_CMD_START_NO_START_INT**, counter is controlled by software command and the timing of input edge can be captured. No interrupt occurs when counter starts.

➤ **IGBT_CMD_FALLING_TRG_START**. There are 2 ways to start the counter. One is to issue the software start command during the trigger driven to low level. The other is a falling edge input to the trigger after the software start command is issued.

➤ **IGBT_CMD_FALLING_TRG_START_NO_START_INT**, the ways to start the counter is same as **IGBT_CMD_FALLING_TRG_START**, but no interrupt occurs when counter is started by software command.

➤ **IGBT_CMD_RISING_TRG_START**, There are 2 ways to start the counter. One is to issue the software start command during the trigger driven to high level. The other is a rising edge input to the trigger after the software start command is issued.

- **IGBT_CMD_RISING_TRG_START_NO_START_INT**, the ways to start the counter is same as **IGBT_CMD_RISING_TRG_START**, but no interrupt occurs when counter is started by software command.
- **IGBT_FALLING_TRG_START**, only falling trigger edge can start the counter. On the other hand a rising trigger edge can stop the counter (*See Note).
- **IGBT_RISING_TRG_START**, only rising trigger edge can start the counter. On the other hand a falling trigger edge can stop the counter (*See Note).
- **IGBT_SYNSLAVE_CHNL_START**, Synchronous start (sets only slave channels (*See Note)).

uint8_t

OperationMode selects IGBT operation mode, which can be set as:

- **IGBT_CONTINUOUS_OUTPUT**, IGBT operates in continuous output mode.
- **IGBT_ONE_TIME_OUTPUT**, IGBT operates in one-time output mode.

uint8_t

CntStopState specifies the output state when counter stops, which can be set as:

- **IGBT_OUTPUT_INACTIVE**, IGBT outputs inactive level.
- **IGBT_OUTPUT_MAINTAINED**, IGBT outputs do not change.
- **IGBT_OUTPUT_NORMAL**, counter does not stop until the end of the period, except a stop trigger, and outputs shift to inactive level when the counter stops.

FunctionalState

ActiveAcceptTrg selects whether the start trigger is accepted when output is active level. This parameter can be set as:

- **ENABLE**, trigger will always be accepted.
- **DISABLE**, trigger will be ignored during active output.

uint8_t

INTPeriod specifies the interrupt occurrence period, which can be set as:

- **IGBT_INT_PERIOD_1**, interrupt occurs every one IGBT output period.
- **IGBT_INT_PERIOD_2**, interrupt occurs every two IGBT output periods.
- **IGBT_INT_PERIOD_4**, interrupt occurs every four IGBT output periods.

uint8_t

ClkDiv selects the division of IGBT source clock, which can be set as:

- **IGBT_CLK_DIV_1**, the frequency of IGBT source clock equals to $\phi T0$.
- **IGBT_CLK_DIV_2**, the frequency of IGBT source clock is quotient of $\phi T0$ divided by 2.
- **IGBT_CLK_DIV_4**, the frequency of IGBT source clock is quotient of $\phi T0$ divided by 4.
- **IGBT_CLK_DIV_8**, the frequency of IGBT source clock is quotient of $\phi T0$ divided by 8.

uint8_t

Output0Init, initialize the IGBT output 0, which can be set as:

- **IGBT_OUTPUT_DISABLE**, disable IGBT output.
- **IGBT_OUTPUT_HIGH_ACTIVE**, initial output is low level and high level is the active output.
- **IGBT_OUTPUT_LOW_ACTIVE**, initial output is high level and low level is the active output.

uint8_t

Output1Init, initialize the IGBT output 1, which can be set as:

- **IGBT_OUTPUT_DISABLE**, disable IGBT output.

- **IGBT_OUTPUT_HIGH_ACTIVE**, initial output is low level and high level is the active output.
- **IGBT_OUTPUT_LOW_ACTIVE**, initial output is high level and low level is the active output.

uint8_t

TrgDenoiseDiv selects the division of noise elimination time for trigger input in IGBT mode. This parameter can be set as:

- **IGBT_NO_DENOISE**, no noise elimination.
- **IGBT_DENOISE_DIV_16**, eliminate pulses shorter than 16 / fsys.
- **IGBT_DENOISE_DIV_32**, eliminate pulses shorter than 32 / fsys.
- **IGBT_DENOISE_DIV_48**, eliminate pulses shorter than 48 / fsys.
- **IGBT_DENOISE_DIV_64**, eliminate pulses shorter than 64 / fsys.
- **IGBT_DENOISE_DIV_80**, eliminate pulses shorter than 80 / fsys.
- **IGBT_DENOISE_DIV_96**, eliminate pulses shorter than 96 / fsys.
- **IGBT_DENOISE_DIV_112**, eliminate pulses shorter than 112 / fsys.
- **IGBT_DENOISE_DIV_128**, eliminate pulses shorter than 128 / fsys.
- **IGBT_DENOISE_DIV_144**, eliminate pulses shorter than 144 / fsys.
- **IGBT_DENOISE_DIV_160**, eliminate pulses shorter than 160 / fsys.
- **IGBT_DENOISE_DIV_176**, eliminate pulses shorter than 176 / fsys.
- **IGBT_DENOISE_DIV_192**, eliminate pulses shorter than 192 / fsys.
- **IGBT_DENOISE_DIV_208**, eliminate pulses shorter than 208 / fsys.
- **IGBT_DENOISE_DIV_224**, eliminate pulses shorter than 224 / fsys.
- **IGBT_DENOISE_DIV_240**, eliminate pulses shorter than 240 / fsys.

uint16_t

Output0ActiveTiming specifies the active timing of output 0. The value must be set between 0 and Output0InactiveTiming.

uint16_t

Output0InactiveTiming specifies the active timing of output 0. The value must be set between Output0ActiveTiming and Period.

uint16_t

Output1ActiveTiming specifies the active timing of output 1. The value must be set between 0 and Output1InactiveTiming.

uint16_t

Output1InactiveTiming specifies the active timing of output 1. The value must be set between Output1ActiveTiming and Period.

uint16_t

Period specifies the IGBT output period, max. 0xFFFF.

uint8_t

EMGFunction specifies the EMG stop function. This parameter can be set as:

- **IGBT_DISABLE_EMG**, disable IGBT EMG stop function.
- **IGBT_EMG_OUTPUT_INACTIVE**, IGBT outputs inactive level during EMG state.
- **IGBT_EMG_OUTPUT_HIZ**, IGBT outputs Hi-z during EMG state.

uint8_t

EMGDenoiseDiv selects the division of noise elimination time for EMG input in IGBT mode. This parameter can be set as:

- **IGBT_NO_DENOISE**, no noise elimination.
- **IGBT_DENOISE_DIV_16**, eliminate pulses shorter than 16 / fsys.
- **IGBT_DENOISE_DIV_32**, eliminate pulses shorter than 32 / fsys.

- **IGBT_DENOISE_DIV_48**, eliminate pulses shorter than 48 / fsys.
- **IGBT_DENOISE_DIV_64**, eliminate pulses shorter than 64 / fsys.
- **IGBT_DENOISE_DIV_80**, eliminate pulses shorter than 80 / fsys.
- **IGBT_DENOISE_DIV_96**, eliminate pulses shorter than 96 / fsys.
- **IGBT_DENOISE_DIV_112**, eliminate pulses shorter than 112 / fsys.
- **IGBT_DENOISE_DIV_128**, eliminate pulses shorter than 128 / fsys.
- **IGBT_DENOISE_DIV_144**, eliminate pulses shorter than 144 / fsys.
- **IGBT_DENOISE_DIV_160**, eliminate pulses shorter than 160 / fsys.
- **IGBT_DENOISE_DIV_176**, eliminate pulses shorter than 176 / fsys.
- **IGBT_DENOISE_DIV_192**, eliminate pulses shorter than 192 / fsys.
- **IGBT_DENOISE_DIV_208**, eliminate pulses shorter than 208 / fsys.
- **IGBT_DENOISE_DIV_224**, eliminate pulses shorter than 224 / fsys.
- **IGBT_DENOISE_DIV_240**, eliminate pulses shorter than 240 / fsys.

***Note:**

To use trigger to start the counter, a software start command must be issued at first.

To use the synchronous start mode, set "11" to MTxIGCR<IGSTA[1:0]> on the slave channels and set other than "11" to the master channel.

11.2.4.2 IGBT_EMGStateTypeDef

Data Fields:

enum

IGBT_EMGInputState indicates the EMG input pin status after noise elimination, which could be

- **IGBT_EMG_INPUT_LOW**, EMG input pin after noise elimination is low.
- **IGBT_EMG_INPUT_HIGH**, EMG input pin after noise elimination is high.

enum

IGBT_EMGProtectState indicates the EMG protection status, which could be

- **IGBT_EMG_NORMAL**, EMG protection status is in normal operation.
- **IGBT_EMG_PROTECT**, EMG protection status is during in protection.

12. LVD

12.1 Overview

TMPM461x has Low voltage detection circuit (LVD). The voltage detection circuit generates a reset signal or an interrupt signal by detecting a decreasing/increasing voltage.

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

All driver APIs are contained in /Libraries/TX04_Periph_Driver/src/tmpm461_lvd.c, with /Libraries/TX04_Periph_Driver/inc/tmpm461_lvd.h containing the macros, data types, structures and API definitions for use by applications.

12.2 API Functions

12.2.1 Function List

- ◆ void LVD_EnableVD1(void)
- ◆ void LVD_DisableVD1(void)
- ◆ void LVD_SetVD1Level(uint32_t **VDLevel**)
- ◆ LVD_VDStatus LVD_GetVD1Status(void)
- ◆ void LVD_EnableVD2(void)
- ◆ void LVD_DisableVD2(void)
- ◆ void LVD_SetVD2Level(uint32_t **VDLevel**)
- ◆ LVD_VDStatus LVD_GetVD2Status(void)
- ◆ void LVD_SetVD1ResetOutput(FunctionalState **NewState**)
- ◆ void LVD_SetVD1INTOutput(FunctionalState **NewState**)
- ◆ void LVD_SetVD2ResetOutput(FunctionalState **NewState**)
- ◆ void LVD_SetVD2INTOutput(FunctionalState **NewState**)

12.2.2 Detailed Description

Functions listed above can be divided into two parts:

- 1) Configure LVD are handled by LVD_EnableVD1(), LVD_DisableVD1(), LVD_SetVD1Level(); LVD_EnableVD2(), LVD_DisableVD2(), LVD_SetVD2Level(); LVD_SetVD1ResetOutput(), LVD_SetVD1INTOutput(), LVD_SetVD2ResetOutput(), LVD_SetVD2INTOutput().
- 2) Get the power supply voltage detection status info by LVD_GetVD1Status() and LVD_GetVD2Status().

12.2.3 Function Documentation

12.2.3.1 LVD_EnableVD1

Enable the operation of voltage detection 1.

Prototype:

```
void  
LVD_EnableVD1(void)
```

Parameters:

None.

Description:

This function will enable the voltage detection 1 operation.

Return:

None.

12.2.3.2 LVD_DisableVD1

Disable the operation of voltage detection 1.

Prototype:

void
LVD_DisableVD1(void)

Parameters:

None.

Description:

This function will disable the voltage detection 1 operation.

Return:

None.

12.2.3.3 LVD_SetVD1Level

Select the level for detection voltage 1.

Prototype:

void
LVD_SetVD1Level(uint32_t *VDLevel*)

Parameters:

VDLevel is the level of voltage detection 1.

This parameter can be one of the following values:

- **LVD_VDLVL1_240:** Voltage detection level is from $2.40 \pm 0.1V$.
- **LVD_VDLVL1_250:** Voltage detection level is from $2.50 \pm 0.1V$.
- **LVD_VDLVL1_260:** Voltage detection level is from $2.60 \pm 0.1V$.
- **LVD_VDLVL1_270:** Voltage detection level is from $2.70 \pm 0.1V$.
- **LVD_VDLVL1_280:** Voltage detection level is from $2.80 \pm 0.1V$.
- **LVD_VDLVL1_290:** Voltage detection level is from $2.90 \pm 0.1V$.

Description:

This function will set the level of voltage detection 1.

Return:

None.

12.2.3.4 LVD_GetVD1Status

Get voltage detection 1 status.

Prototype:

LVD_VDStatus
LVD_GetVD1Status(void)

Parameters:

None.

Description:

This function will get voltage detection 1 status.

Return:

LVD_VDStatus: The voltage detection 1 status, which can be one of:

LVD_VD_UPPER: Power supply voltage is upper than the detection voltage.

LVD_VD_LOWER: Power supply voltage is lower than the detection voltage.

12.2.3.5 LVD_EnableVD2

Enable the operation of voltage detection 2.

Prototype:

void
LVD_EnableVD2(void)

Parameters:

None.

Description:

This function will enable the voltage detection 2 operation.

Return:

None.

12.2.3.6 LVD_DisableVD2

Disable the operation of voltage detection 2.

Prototype:

void
LVD_DisableVD2(void)

Parameters:

None.

Description:

This function will disable the voltage detection 2 operation.

Return:

None.

12.2.3.7 LVD_SetVD2Level

Select the detection voltage 2 level.

Prototype:

void
LVD_SetVD2Level(uint32_t **VDLevel**)

Parameters:

VDLevel is the voltage detection 2 level.

This parameter can be one of the following values:

- **LVD_VDLVL_280**: Voltage detection level is from $2.80 \pm 0.1V$.
- **LVD_VDLVL_285**: Voltage detection level is from $2.85 \pm 0.1V$.
- **LVD_VDLVL_290**: Voltage detection level is from $2.90 \pm 0.1V$.
- **LVD_VDLVL_295**: Voltage detection level is from $2.95 \pm 0.1V$.
- **LVD_VDLVL_300**: Voltage detection level is from $3.00 \pm 0.1V$.
- **LVD_VDLVL_305**: Voltage detection level is from $3.05 \pm 0.1V$.
- **LVD_VDLVL_310**: Voltage detection level is from $3.10 \pm 0.1V$.
- **LVD_VDLVL_315**: Voltage detection level is from $3.15 \pm 0.1V$.

Description:

This function will set the level of voltage detection 2.

Return:

None.

12.2.3.8 LVD_GetVD2Status

Get voltage detection 2 status.

Prototype:

LVD_VDStatus

LVD_GetVD2Status(void)

Parameters:

None.

Description:

This function will get voltage detection 2 status.

Return:

LVD_VDStatus: The voltage detection 2 status, which can be one of:

LVD_VD_UPPER: Power supply voltage is upper than the detection voltage.

LVD_VD_LOWER: Power supply voltage is lower than the detection voltage.

12.2.3.9 LVD_SetVD1ResetOutput

Enable or disable LVD reset output of voltage detection 1.

Prototype:

void

LVD_SetVD1ResetOutput(FunctionalState **NewState**)

Parameters:

NewState: new state of LVD reset output.

This parameter can be one of the following values:

ENABLE or **DISABLE**

Description:

This function enables or disables LVD reset output of voltage detection 1.

Return:

None.

12.2.3.10 LVD_SetVD1INTOutput

Enable or disable LVD interrupt output of voltage detection 1.

Prototype:

void
LVD_SetVD1INTOutput(FunctionalState **NewState**)

Parameters:

NewState: new state of LVD interrupt output.
This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:

This function enables or disables LVD interrupt output of voltage detection 1.

Return:

None.

12.2.3.11 LVD_SetVD2ResetOutput

Enable or disable LVD reset output of voltage detection 2.

Prototype:

void
LVD_SetVD2ResetOutput(FunctionalState **NewState**)

Parameters:

NewState: new state of LVD reset output.
This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:

This function enables or disables LVD reset output of voltage detection 2.

Return:

None.

12.2.3.12 LVD_SetVD2INTOutput

Enable or disable LVD interrupt output of voltage detection 2.

Prototype:

void
LVD_SetVD2INTOutput(FunctionalState **NewState**)

Parameters:

NewState: new state of LVD interrupt output.
This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:

This function enables or disables LVD interrupt output of voltage detection 2.

Return:

None.

12.2.4 Data Structure Description

None

13. OFD

13.1 Overview

TMPM461x has an oscillation frequency detector circuit (OFD), which can generate a reset signal when abnormal states of clock such as a harmonic, a sub harmonic or stopped state is detected.

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/TX04_Periph_Driver/src/tmpm461_ofd.c, with /Libraries/TX04_Periph_Driver/inc/tmpm461_ofd.h containing the macros, data types, structures and API definitions for use by applications.

13.2 API Functions

13.2.1 Function List

- ◆ void OFD_SetRegWriteMode(FunctionalState **NewState**);
- ◆ void OFD_Enable(void);
- ◆ void OFD_Disable(void);
- ◆ void OFD_SetDetectionFrequency(OFD_OSCSource **Source**,
uint32_t **HigherDetectionCount**,
uint32_t **LowerDetectionCount**);
- ◆ void OFD_Reset(FunctionalState **NewState**);
- ◆ OFD_Status OFD_GetStatus(void);
- ◆ void OFD_SetDetectionMonitor(OFD_MonitorMode **Mode**);

13.2.2 Detailed Description

Functions listed above can be divided into three parts:

- 1) Initialize and configure OFD function by OFD_SetRegWriteMode(), OFD_SetDetectionFrequency(), OFD_SetDetectionMonitor(), 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.

13.2.3 Function Documentation

13.2.3.1 OFD_SetRegWriteMode

Enable or disable the writing of all OFD registers except OFDCR1.

Prototype:

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

Parameters:

NewState: Set the interrupt source mask state, which can be set as:

- **ENABLE** : Enable the writing.

- **DISABLE:** Disable the writing.

Description:

This function will enable writing of all OFD registers except OFDCR1 when **NewState** is **ENABLE**, and disable writing of all OFD registers except OFDCR1 when **NewState** is **DISABLE**.

Return:

None

13.2.3.2 OFD_Enable

Enable the OFD function.

Prototype:

void
OFD_Enable(void)

Parameters:

None.

Description:

This function will enable the OFD function.

Return:

None

13.2.3.3 OFD_Disable

Disable the OFD function.

Prototype:

void
OFD_Disable(void)

Parameters:

None.

Description:

This function will disable the OFD function.

Return:

None

13.2.3.4 OFD_SetDetectionFrequency

Set the count value of detection frequency.

Prototype:

void
OFD_SetDetectionFrequency(OFD_OSCSource **Source**,
uint32_t **HigherDetectionCount**,
uint32_t **LowerDetectionCount**);

Parameters:

Source: Select internal or external oscillation source for the count value setting, which can be set as:

- **OFD_IHOSC:** Internal High Frequency Oscillation.
- **OFD_EHOSC:** External High Frequency Oscillation.

HigherDetectionCount: The count value of higher detection frequency, Maximum value is 0x1FFU.

LowerDetectionCount: The count value of lower detection frequency, Maximum value is 0x1FFU.

Description:

This function will set the count value of detection frequency for internal or external oscillation (both higher detection frequency and lower detection frequency)

Return:

None

13.2.3.5 OFD_Reset

Enable or disable the OFD reset.

Prototype:

void
OFD_Reset(FunctionalState **NewState**)

Parameters:

NewState: The new state of enable or disable, which can be set as:

- **ENABLE** : Enable OFD reset.
- **DISABLE:** Disable OFD reset.

Description:

This function will Enable or disable the OFD reset.

Return:

None

13.2.3.6 OFD_GetStatus

Get the OFD busy and frequency error info.

Prototype:

OFD_Status
OFD_GetStatus(void)

Parameters:

None

Description:

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

Return:

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

13.2.3.7 OFD_SetDetectionMonitor

Select intended detection mode by setting the external high frequency oscillation clock monitor register

Prototype:

void
OFD_SetDetectionMonitor(OFD_MonitorMode **Mode**)

Parameters:

Mode: Select intended detection mode, which can be set as:

- **OFD_NORMAL** : Normal Detection Mode
- **OFD_MONITOR**: Monitor Mode.

Description:

This function will select intended detection mode by setting the external high frequency oscillation clock monitor register.

Return:

None

13.2.4 Data Structure Description

13.2.4.1 OFD_Status

Data Fields:

uint32_t

All: Data.

Bit

uint32_t

FrequencyError: 1 Frequency Error status

uint32_t

OFDBusy: 1 OFD Busy status

14. RMC

14.1 Overview

TMPM461x's remote control signal preprocessor (here after referred to as RMC) receives a remote control signal of which carrier is removed.

TMPM461x has one RMC channel: TSB_RMC0.

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 each channel.

All driver APIs are contained in /Libraries/TX04_Periph_Driver/src/tmpm461_rmc.c, with /Libraries/TX04_Periph_Driver/inc/tmpm461_rmc.h containing the macros, data types, structures and API definitions for use by applications.

14.2 API Functions

14.2.1 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**)

14.2.2 Detailed Description

Functions listed above can be divided into three parts:

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

14.2.3 Function Documentation

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

14.2.3.1 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 RMCxEN<RMCEN>bit.(x can be 0,1)

Return:

None

14.2.3.2 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 RMCxEN<RMCEN>bit. (x can be 0,1)

Return:

None

14.2.3.3 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

14.2.3.4 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 RMCxREN<RMCREN> bit. (x can be 0,1)

Return:

None

14.2.3.5 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

RMCxRSTAT<RMCRNUM0-6> bits. (x can be 0,1)

Return:

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

14.2.3.6 RMC_SetLeaderDetection

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

Prototype:

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

Parameters:

RMCx is the specified RMC channel.

LeaderPara: The structure containing basic RMC leader detection configuration.

Data Fields:

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

ENABLE or **DISABLE**

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

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

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

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

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

Description:

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

This function handles the $RMCR1$ register and $RMCR2 < RMCLIEN > < RMCLD >$ two bits. (x can be 0,1)

See MCU datasheet for detail.

Return:

None

14.2.3.7 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 RMCxRCR2<RMCEDIEN> bit. (x can be 0,1)

Return:

None

14.2.3.8 RMC_SetSignalRxMethod

Select the method of receiving a remote control signal.

Prototype:

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

Parameters:

RMCx is the specified RMC channel.

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

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

Description:

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

This function handles the RMCxRCR2<RMCPHM> bit. (x can be 0,1)

Return:

None

14.2.3.9 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 RMCxRCR2<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 RMCxRCR2<RMCDMAX7:0> specifies a threshold for detecting a maximum data bit cycle. It is detected when a data bit cycle exceeds the threshold. It is not detected when <RMCDMAX7:0> = 11111111b.

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

This function handles the RMCxRCR2<RMCLL0-7> <RMCDMA0-7> bits. (x can be 0,1)

Return:

None

14.2.3.10 RMC_SetThreshold

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

Prototype:

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

Parameters:

RMCx is the specified RMC channel.

LargerThreshold: Specifies a larger threshold (within a range of 1.5T and 2T) to determine a pattern of remote control signal in a phase method. If the measured cycle exceeds the threshold, the bit is determined as "10". If not, the bit is determined as "01". Calculating formula of the threshold: $RMCDATHx1/fs[s]$.

The LargerThreshold should less than 0x80.

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

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

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

As for the determination of a remote control signal pattern in a phase method, if the measured cycle exceeds the threshold, the bit is determined as "01". If not, the bit is determined as "00". Calculating formula of the threshold to determine 0 or 1:

RMCDATLx1/fs[s].

This function handles the RMCxRCR3<RMCDATH0-6> <RMCDATL0-6> bits. (x can be 0,1)

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

14.2.3.11 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 RMCxRCR4<RMCP0> bit. (x can be 0,1)

Return:

None

14.2.3.12 RMC_SetNoiseCancellation

Set the noise cancellation time for the specified RMC channel.

Prototype:

void

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

Parameters:

RMCx is the specified RMC channel.

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

Description:

Specifies time that noises is cancelled by a noise canceller.

If <RMCNC3:0> = 0000b, noises are not cancelled.
Calculating formula of noise cancellation time: $RMCNC \times 1/fs[s]$.
This function handles the RMCxRCR4<RMCNC0-RMCNC3> bits. (x can be 0,1)

Return:
None

14.2.3.13 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 RMCxRSTAT<RMCRLIF>< RMCLOIF >< RMCDCMAX >< RMCEDIF > bits. (x can be 0,1)

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

14.2.3.14 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 RMCxRSTAT <RMCRLDR> bits. (x can be 0,1)

Return:
RMC_LeaderDetection: leader detection result, which can be one of:
RMC_LEADER_DETECTED: leader detected.
RMC_NO_LEADER: no leader detected.

14.2.3.15 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

14.2.3.16 RMC_SetSrcClk

Specifies that the sampling clock.

Prototype:

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

Parameters:

RMCx is the specified RMC channel.

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

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

Description:

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

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

.

Return:

None

14.2.4 Data Structure Description

14.2.4.1 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 <MCRBUF31:0>.

uint32_t

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

uint8_t

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

14.2.4.2 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**

14.2.4.3 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.

14.2.4.4 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

15. RTC

15.1 Overview

The Real Time Clock (RTC) in the TPM461x 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
- Clock correction function
- 1 Hz clock output

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\TX04_Periph_Driver\src\tpm461_rtc.c, with \Libraries\TX04_Periph_Driver\inc\tpm461_rtc.h containing the macros, data types, structures and API definitions for use by applications.

15.2 API Functions

15.2.1 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_ResetAlarm(void) ;
- ◆ void RTC_ResetClockSec(void);
- ◆ RTC_ReqState RTC_GetResetClockSecReq(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**);
- ◆ void RTC_SetProtectCtrl(FunctionalState **NewState**)
- ◆ void RTC_EnableCorrection(void)
- ◆ void RTC_DisableCorrection(void)
- ◆ void RTC_SetCorrectionTime(uint8_t **Time**)
- ◆ void RTC_SetCorrectionValue(RTC_CorrectionMode **Mode**,uint16_t **Cnt**)

15.2.2 Detailed Description

Functions listed above can be divided into six 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_GetResetClockSecReq(), RTC_SetClockValue() and RTC_GetClockValue() handle for RTC clock function only.
- 4) RTC_EnableAlarm(), RTC_DisableAlarm(), RTC_SetAlarmValue(),RTC_ResetAlarm() and RTC_GetAlarmValue() handle for RTC alarm function only.
- 5) RTC_EnableCorrection(),RTC_DisableCorrection(),RTC_SetCorrectionTime() and RTC_SetCorrectionValue() handle for RTC clock correction function.
- 6) RTC_SetAlarmOutput(),RTC_SetProtectCtrl() and RTC_SetRTCINT() handle other specified functions.

15.2.3 Function Documentation

15.2.3.1 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.

15.2.3.2 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

15.2.3.3 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

15.2.3.4 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

15.2.3.5 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.

15.2.3.6 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

15.2.3.7 RTC_SetHour12

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

Prototype:

```
void  
RTC_SetHour12(RTC_FuncMode NewMode,  
              uint8_t Hour,  
              uint8_t AmPm);
```

Parameters:

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

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

Hour: New hour value, max is 11.

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

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

Description:

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

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

Return:

None

15.2.3.8 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

15.2.3.9 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

15.2.3.10 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

15.2.3.11 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

15.2.3.12 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

15.2.3.13 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

15.2.3.14 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

15.2.3.15 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

15.2.3.16 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

15.2.3.17 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

15.2.3.18 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.

15.2.3.19 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

15.2.3.20 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.

15.2.3.21 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

15.2.3.22 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.

15.2.3.23 RTC_EnableClock

Enable RTC clock function.

Prototype:

```
void  
RTC_EnableClock(void);
```

Parameters:

None

Description:

This function will enable clock function.

Return:

None

15.2.3.24 RTC_DisableClock

Disable RTC clock function.

Prototype:

```
void  
RTC_DisableClock(void);
```

Parameters:

None

Description:

This function will disable clock function.

Return:

None

15.2.3.25 RTC_EnableAlarm

Enable RTC alarm function.

Prototype:

```
void  
RTC_EnableAlarm(void);
```

Parameters:

None

Description:

This function will enable alarm function.

Return:

None

15.2.3.26 RTC_DisableAlarm

Disable RTC alarm function.

Prototype:

```
void  
RTC_DisableAlarm(void);
```

Parameters:

None

Description:

This function will disable alarm function.

Return:

None

15.2.3.27 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**.

***Note:**

To set interrupt enable bits to <ENATMR>, <ENAALM> and <INTENA>, you must follow the order specified here. Make sure not to set them at the same time (make sure that there is time lag between interrupt enable and clock/alarm enable). To change the setting of <ENATMR> and <ENAALM>, <INTENA> must be disabled first.

Return:

None

15.2.3.28 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

15.2.3.29 RTC_ResetAlarm

Reset alarm.

Prototype:

void
RTC_ResetAlarm(void);

Parameters:

None

Description:

This function will reset alarm.

Return:

None

15.2.3.30 RTC_ResetClockSec

Reset RTC clock second counter.

Prototype:

void
RTC_ResetClockSec(void);

Parameters:

None

Description:

This function will reset sec counter.

Return:

None

15.2.3.31 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.

15.2.3.32 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

15.2.3.33 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

15.2.3.34 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

15.2.3.35 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

15.2.3.36 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

15.2.3.37 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

15.2.3.38 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

15.2.3.39 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

15.2.3.40 RTC_SetProtectCtrl

Enable or disable to protect RTC registers: RTCADJCTL and RTCADJDAT

Prototype:

```
void  
RTC_SetProtectCtrl(FunctionalState NewState);
```

Parameters:**NewState:**

- **ENABLE:** < RTCPROTECT>=0xC1 Register write enable.
- **DISABLE:** < RTCPROTECT>= Except 0xC1 Register write disable.

Description:

This function will enable or disable to protect RTC registers: RTCADJCTL and RTCADJDAT

Return:

None

15.2.3.41 RTC_EnableCorrection

Enable RTC correction function.

Prototype:

```
void  
RTC_EnableCorrection(void);
```

Parameters:

None

Description:

This function will enable RTC correction function.

Return:

None

15.2.3.42 RTC_DisableCorrection

Disable RTC correction function.

Prototype:

```
void  
RTC_DisableCorrection(void);
```

Parameters:

None

Description:

This function will disable RTC correction function.

Return:

None

15.2.3.43 RTC_SetCorrectionTime

Set correction reference time.

Prototype:

```
void  
RTC_SetCorrectionTime(uint8_t Time);
```

Parameters:

Time: the reference time of correction

This parameter can be one of the following values:

- **RTC_ADJ_TIME_1_SEC**: correction reference time is 1 second.
- **RTC_ADJ_TIME_10_SEC**: correction reference time is 10 seconds.
- **RTC_ADJ_TIME_20_SEC**: correction reference time is 20 seconds.
- **RTC_ADJ_TIME_30_SEC**: correction reference time is 30 seconds.
- **RTC_ADJ_TIME_1_MIN**: correction reference time is 1 minute.

Description:

This function will set correction reference time.

Return:

None

15.2.3.44 RTC_SetCorrectionValue

Set correction value.

Prototype:

```
void  
RTC_SetCorrectionValue(RTC_CorrectionMode Mode,uint16_t Cnt);
```

Parameters:

Mode: the mode of correction

This parameter can be one of the following values:

- **RTC_CORRECTION_PLUS** : a plus correction is applied.
- **RTC_CORRECTION_MINUS**: a minus correction is applied.

Cnt: a correction value per second.

For **RTC_CORRECTION_PLUS**, this parameter can only be 0~255.

For **RTC_CORRECTION_MINUS**, this parameter can only be 1~256.

Description:

This function will set correction value.

Return:

None

15.2.4 Data Structure Description

15.2.4.1 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.

15.2.4.2 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.

15.2.4.3 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.

16. SSP

16.1 Overview

TOSHIBA TMPM461x contains SSP (Synchronous Serial Port) module with 3 channels (SSP0, SSP1 and SSP2).

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

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

All driver APIs are contained in /Libraries/TX04_Periph_Driver/src/tmpm461_ssp.c, with /Libraries/TX04_Periph_Driver/inc/tmpm461_ssp.h containing the macros, data types, structures and API definitions for use by applications.

16.2 API Functions

16.2.1 Function List

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

16.2.2 Detailed Description

Functions listed above can be divided into six parts:

- 1) Configure the common functions of SSP are handled by `SSP_Init()`, which will call `SSP_SetClkPreScale()`, `SSP_SetFrameFormat()`, `SSP_SetClkPolarity()`, `SSP_SetClkPhase()`, `SSP_SetDataSize()`, `SSP_SetMSMode()`.
- 2) Data transmit and receive are handled by `SSP_SetTxData()`, `SSP_GetRxData()`.
- 3) SSP interrupt relative function are: `SSP_SetINTConfig()`, `SSP_GetINTConfig()`, `SSP_GetPreEnableINTState()`, `SSP_GetPostEnableINTState()`, `SSP_ClearINTFlag()`.
- 4) Get SSP status are handled by `SSP_GetWorkState()`, `SSP_GetFIFOState()`
- 5) Enable/Disable SSP module are handled by `SSP_Enable()`, `SSP_Disable()`.
- 6) `SSP_SetSlaveOutputCtrl()`, `SSP_SetLoopBackMode()` and `SSP_SetDMACtrl()` handle other specified functions.

16.2.3 Function Documentation

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

16.2.3.1 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

16.2.3.2 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

16.2.3.3 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

16.2.3.4 SSP_SetClkPreScale

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

Prototype:

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

Parameters:

SSPx: Select the SSP channel

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

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

Description:

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

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

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

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

Return:

None

16.2.3.5 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

16.2.3.6 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

16.2.3.7 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

16.2.3.8 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

16.2.3.9 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

16.2.3.10 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

16.2.3.11 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

16.2.3.12 SSP_SetTxData

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

Prototype:

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

Parameters:

SSPx: Select the SSP channel.

Data: 4~16bit data to be send

Description:

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

Return:

None

16.2.3.13 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

16.2.3.14 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.

16.2.3.15 SSP_GetFIFOState

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

Prototype:

SSP_FIFOState

SSP_GetFIFOState(TSB_SSP_TypeDef * **SSPx**
SSP_Direction **Direction**)

Parameters:

SSPx: Select the SSP channel.

Direction: The direction which means transmit or receive

This parameter can be one of the following values:

- **SSP_RX:** target is to check state of receive FIFO.
- **SSP_TX:** target is to check state of transmit FIFO.

Description:

This function will the specified SSP channel by **SSPx**, get the Rx/Tx FIFO state by **Direction**.

For example, data can be sent after judging Tx FIFO is available by the code below:

```
SSP_FIFOState fifoState;

fifoState = SSP_GetFIFOState(TSB_SSP0, SSP_TX);
if ((fifoState == SSP_FIFO_EMPTY) || (fifoState == SSP_FIFO_NORMAL))
    { SSP_SetTxData(SSP0, data_to_be_sent); }
```

Return:

The state of SSP FIFO, which can be

SSP_FIFO_EMPTY: FIFO is empty.

SSP_FIFO_NORMAL: FIFO is not full and not empty.

SSP_FIFO_INVALID: FIFO is invalid state.

SSP_FIFO_FULL: FIFO is full

16.2.3.16 SSP_SetINTConfig

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

Prototype:

void

SSP_SetINTConfig(TSB_SSP_TypeDef * **SSPx**,
uint32_t **IntSrc**)

Parameters:

SSPx: Select the SSP channel.

IntSrc: The interrupt source for SSP to be enabled or disabled.

To disable all interrupt sources, use the parameter:

- **SSP_INTCFG_NONE**

To enable the interrupt one by one, use the logical operator “ | ” with below parameter:

- **SSP_INTCFG_RX_OVERRUN:** Receive overrun interrupt.
- **SSP_INTCFG_RX_TIMEOUT:** Receive timeout interrupt.
- **SSP_INTCFG_RX:** Receive FIFO interrupt (at least half full).
- **SSP_INTCFG_TX:** Transmit FIFO interrupt (at least half empty).

To enable all the 4 interrupt above together, use the parameter:

- **SSP_INTCFG_ALL**

Description:

This function will specified SSP channel by **SSPx**, enable/disable interrupts by **IntSrc**.

For example, we can enable Tx and Rx interrupt by code like below:

```
SSP_SetINTConfig( SSP0, SSP_INTCFG_RX | SSP_INTCFG_TX )
```

Return:

None

16.2.3.17 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.

16.2.3.18 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.

16.2.3.19 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.

16.2.3.20 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

16.2.3.21 SSP_SetDMACtrl

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

Prototype:

```
void  
SSP_SetDMACtrl(TSB_SSP_TypeDef * SSPx,  
               SSP_Direction Direction,  
               FunctionalState NewState)
```

Parameters:

SSPx: Select the SSP channel.

Direction: The direction which means transmit or receive.

This parameter can be one of the following values:

- **SSP_RX**: target is to set receive DMA FIFO.
- **SSP_TX**: target is to set transmit DMA FIFO.

NewState: New state of DMA FIFO mode.

This parameter can be one of the following values:

- **ENABLE**: enables the DMA for FIFO.
- **DISABLE**: disables the DMA for FIFO.

Description:

This function will enable/disable the DMA FIFO Rx/Tx of the specified SSP channel by **SSPx**.

Return:

None

16.2.4 Data Structure Description

16.2.4.1 SSP_InitTypeDef

Data Fields for this structure:

SSP_FrameFormat

FrameFormat Set frame format of SSP.

Which can be:

- **SSP_FORMAT_SPI**: configure the SSP in SPI mode.
- **SSP_FORMAT_SSI**: configure the SSP in SSI mode.
- **SSP_FORMAT_MICROWIRE**: configure the SSP in Microwire mode

uint8_t

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

SSP_ClkPolarity

ClkPolarity SPI clock polarity, Specify the clock polarity in idle state of SCLK pin when the Frame Format is set as SPI.

Which can be:

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

SSP_ClkPhase

ClkPhase Specify the clock phase when the Frame Format is set as SPI.

Which can be:

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

uint8_t

DataSize Select data size From 4 to 16

SSP_MS_Mode

Mode SSP device mode.

Which can be:

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

16.2.4.2 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.

17. TMRB

17.1 Overview

TOSHIBA TPM461x contains 16 channels of multi-functional 16-bit timer/event counter (TMRB0 through TMRB15). Each channel can operate in the following modes:

- Interval timer mode
- Event counter mode
- Programmable pulse generation (PPG) mode
- Programmable pulse generation (PPG) external trigger mode

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

- Frequency measurement
- Pulse width measurement
- Time difference measurement

TPM461x also has 16-bit multi-purpose timer (MPT), when being operated in timer mode, they are the same as common timer channels.

The TMRB driver APIs provide a set of functions to configure each channel, such as setting the clock division, trailing timing and leading timing duration, capture timing and flip-flop function. And to control the running state of each channel such as controlling up-counter, the output of flip-flop and to indicate the status of each channel such as returning the factor of interrupt, value in capture registers and so on.

All driver APIs are contained in /Libraries/TX04_Periph_Driver/src/tmpm461_tmr.c, with /Libraries/TX04_Periph_Driver/inc/tmpm461_tmr.h containing the macros, data types, structures and API definitions for use by applications.

17.2 API Functions

17.2.1 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**)

17.2.2 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.

17.2.3 Function Documentation

***Note:** in all of the following APIs, unless otherwise specified, the parameter:

“TSB_TB_TypeDef* **TBx**” can be one of the following values:

TSB_TB0, TSB_TB1, TSB_TB2, TSB_TB3, TSB_TB4, TSB_TB5, TSB_TB6, TSB_TB7, TSB_TB8, TSB_TB9, TSB_TBA, TSB_TBB, TSB_TBC, TSB_TBD, TSB_TBE, TSB_TBF, TSB_TB_MPT0, TSB_TB_MPT1.

17.2.3.1 TMRB_Enable

Enable the specified TMRB channel.

Prototype:

void
TMRB_Enable(TSB_TB_TypeDef* **TBx**)

Parameters:

TBx is the specified TMRB channel.

Description:

This function will enable the specified TMRB channel selected by **TBx**.
If channel is MPT, this function will also select MPT channel as timer mode.

Return:

None

17.2.3.2 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

17.2.3.3 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

17.2.3.4 TMRB_Init

Initialize the specified TMRB channel.

Prototype:

```
void  
TMRB_Init(TSB_TB_TypeDef* TBx,  
          TMRB_InitTypeDef* InitStruct)
```

Parameters:

TBx is the specified TMRB channel.

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

Description:

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

Return:

None

17.2.3.5 TMRB_SetCaptureTiming

Configure the capture timing and up-counter clearing 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

When *TBx* = TSB_TB_MPT0 or TSB_TB_MPT1:

- **MPT_DISABLE_CAPTURE**: Capture is disabled.
- **MPT_CAPTURE_IN_RISING**: At the rising edge of MTxTBIN input, counter values are captured to the capture register 0 (MTxCP0)
- **MPT_CAPTURE_IN_RISING_FALLING**: At the rising edge of MTxTBIN input, counter values are captured to the capture register 0 (MTxCP0). At the falling edge of MTxTBIN input, counter values are captured to the capture register 1 (MTxCP1).

When *TBx* = TSB_TB0 to TSB_TBF:

- **TMRB_DISABLE_CAPTURE**: Capture is disabled.
- **TMRB_CAPTURE_TBIN0_TBIN1_RISING**: Captures a counter value on rising edge of TBxIN0 input into Capture register 0 (TBxCP0). Captures a counter value on rising edge of TBxIN1 input into Capture register 1 (TBxCP1). (**Only TSB_TB4 to TSB_TBD can choose TMRB_CAPTURE_TBIN0_TBIN1_RISING**)
- **TMRB_CAPTURE_TBIN0_RISING_FALLING**: Captures a counter value on rising edge of TBxIN0 input into Capture register 0 (TBxCP0). Captures a counter value on falling edge of TBxIN0 input into Capture register 1 (TBxCP1).
- **TMRB_CAPTURE_TBFF0_EDGE**: Captures a counter value on rising edge of TBxFF0 input into Capture register 0 (TBxCP0). Captures a counter value on falling edge of TBxFF0 input into Capture register 1 (TBxCP1).
- **TMRB_CLEAR_TBIN1_RISING**: Clears up-counter on rising edge of TBxIN1 input. (**Only TSB_TB4 to TSB_TBD can choose TMRB_CLEAR_TBIN1_RISING**)
- **TMRB_CAPTURE_TBIN0_RISING_CLEAR_TBIN1_RISING**:
Captures a counter value on rising edge of TBxIN0 input into Capture register 0 (TBxCP0); clears up-counter on rising edge of TBxIN1 input. If capture timing and up-counter clearing timing are same, capturing is performed first, and then up-counter is cleared. (**Only TSB_TB4 to TSB_TBD can choose TMRB_CAPTURE_TBIN0_RISING_CLEAR_TBIN1_RISING**)

Description:

This function will configure the capture timing and up-counter clearing timing.

Return:

None

17.2.3.6 TMRB_SetFlipFlop

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

Prototype:

```
void  
TMRB_SetFlipFlop(TSB_TB_TypeDef* TBx,  
                 TMRB_FFOutputTypeDef* FFStruct)
```

Parameters:

TBx is the specified TMRB channel.

FFStruct is the structure containing TMRB flip-flop function configuration including flip-flop output level and flip-flop-reverse trigger (refer to “Data Structure Description” for details).

Description:

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

Return:

None

17.2.3.7 TMRB_GetINTFactor

Indicate what causes the interrupt.

Prototype:

```
TMRB_INTFactor  
TMRB_GetINTFactor(TSB_TB_TypeDef* TBx)
```

Parameters:

TBx is the specified TMRB channel.

Description:

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

Return:

TMRB Interrupt factor. Each bit has the following meaning:

MatchLeadingTiming(Bit0): a match with the leadingtiming value is detected

MatchTrailingTiming(Bit1): a match with the trailingtiming value is detected

OverFlow(Bit2): an up-counter is overflow

***Note:**

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

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

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

17.2.3.8 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_TRAILING_INT**: Mask the interrupt the factor of which is that the value in up-counter and trailingtiming are match.
- **TMRB_MASK_MATCH_LEADING_INT**: Mask the interrupt the factor of which is that the value in up-counter and leadingtiming are match.
- **TMRB_MASK_OVERFLOW_INT**: Mask the interrupt the factor of which is the occurrence of overflow.
- **TMRB_NO_INT_MASK**: Unmask the interrupt.
- **TMRB_MASK_MATCH_LEADING_INT | TMRB_MASK_MATCH_TRAILING_INT**: Mask the interrupt the factor of which is that the value in up-counter and trailingtiming are match or mask the interrupt the factor of which is that the value in up-counter and leadingtiming are match.
- **TMRB_MASK_MATCH_LEADING_INT | TMRB_MASK_OVERFLOW_INT**: Mask the interrupt the factor of which is that the value in up-counter and leadingtiming are match or mask the interrupt the factor of which is the occurrence of overflow.
- **TMRB_MASK_MATCH_TRAILING_INT | TMRB_MASK_OVERFLOW_INT**: Mask the interrupt the factor of which is that the value in up-counter and trailingtiming are match or mask the interrupt the factor of which is the occurrence of overflow.
- **TMRB_MASK_MATCH_LEADING_INT | TMRB_MASK_MATCH_TRAILING_INT | TMRB_MASK_OVERFLOW_INT**: Mask the interrupt the factor of which is that the value in up-counter and trailingtiming are match or mask the interrupt the factor of which is that the value in up-counter and leadingtiming are match or mask the interrupt the factor of which is the occurrence of overflow.

Description:

If **TMRB_MASK_MATCH_TRAILING_INT** is selected, the interrupt of the specified TMRB channel will not happen when the value in up-counter and trailingtiming are match.

If **TMRB_MASK_MATCH_LEADING_INT** is selected, the interrupt of the specified TMRB channel will not happen when the value in up-counter and leadingtiming are match.

If **TMRB_MASK_OVERFLOW_INT** is selected, the interrupt of the specified TMRB channel will not happen even if there is an occurrence of overflow.

If **TMRB_NO_INT_MASK** is selected, all interrupt masks will be cleared.

If the combination of **TMRB_MASK_MATCH_TRAILING_INT** and **TMRB_MASK_MATCH_LEADING_INT** and **TMRB_MASK_OVERFLOW_INT** is selected, the interrupt of the specified TMRB channel will not happen even if the relevant situation happened.

Return:

None

17.2.3.9 TMRB_ChangeLeadingTiming

Change the value of leadingtiming for the specified channel.

Prototype:

void

TMRB_ChangeLeadingTiming(TSB_TB_TypeDef* **TBx**,
uint32_t **LeadingTiming**)

Parameters:

TBx is the specified TMRB channel.

LeadingTiming specifies the value of leadingtiming, max. is 0xFFFF.

Description:

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

Return:

None

***Note:**

LeadingTiming can not exceed **TrailingTiming**.

17.2.3.10 TMRB_ChangeTrailingTiming

Change the value of trailingtiming for the specified channel.

Prototype:

void

TMRB_ChangeTrailingTiming(TSB_TB_TypeDef* **TBx**,
uint32_t **TrailingTiming**)

Parameters:

TBx is the specified TMRB channel.

TrailingTiming specifies the value of trailingtiming, max. is 0xFFFF.

Description:

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

Return:
None

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

17.2.3.11 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

17.2.3.12 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

17.2.3.13 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

17.2.3.14 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

17.2.3.15 TMRB_SetSyncMode

Enable or disable the synchronous mode of specified TMRB channel.

Prototype:

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

Parameters:

TBx is the specified TMRB channel, which can be

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

NewState specifies the state of the synchronous mode of the TMRB, which can be

- **ENABLE**: enables the synchronous mode,

- **DISABLE:** disables the synchronous mode.

Description:

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

Return:

None

***Note:**

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

17.2.3.16 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

***Note:**

WriteRegMode is invalid for TMRB0~TMRB7. So when this API is used for these channels, 0 is recommended for **WriteRegMode**.

17.2.3.17 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

17.2.3.18 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

17.2.4 Data Structure Description

17.2.4.1 TMRB_InitTypeDef

Data Fields:

uint32_t

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

uint32_t

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

- **TMRB_CLK_DIV_2**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 2;
- **TMRB_CLK_DIV_8**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 8;
- **TMRB_CLK_DIV_32**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 32.
- **TMRB_CLK_DIV_64**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 64(TMRB0~TMRBF only).
- **TMRB_CLK_DIV_128**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 128(TMRB0~TMRBF only).
- **TMRB_CLK_DIV_256**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 256(TMRB0~TMRBF only).
- **TMRB_CLK_DIV_512**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 512(TMRB0~TMRBF only).

uint32_t

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

uint32_t

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

- **TMRB_FREE_RUN**, which means that the up-counter will not stop counting even when the value in it is match with trailingtiming, until it reaches 0xFFFF, then it will be cleared and starting counting from 0.(TMRB0~TMRBF only)
- **TMRB_AUTO_CLEAR**, which means that the up-counter will restart counting from 0 immediately when the value in up-counter matches **TrailingTiming**. (TMRB0~TMRBF only)
- **MPT_FREE_RUN**, which means that the up-counter will not stop counting even when the value in it is match with trailingtiming, until it reaches 0xFFFF, then it will be cleared and starting counting from 0. (MPT0~MPT1 only)
- **MPT_AUTO_CLEAR**, which means that the up-counter will restart counting from 0 immediately when the value in up-counter matches **TrailingTiming**. (MPT0~MPT1 only)

uint32_t

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

17.2.4.2 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_TRAILING**, which means that the reversing flip-flop output will be triggered when the up-counter matches the trailingtiming,
- **TMRB_FLIPFLOP_MATCH_LEADING**, which means that the reversing flip-flop output will be triggered when the up-counter matches the leadingtiming.

17.2.4.3 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 -

18. UART

18.1 Overview

TMPM461x has six serial I/O channels. Each channel can operate in both UART mode (asynchronous communication) and I/O Interface mode (synchronous 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/TX04_Periph_Driver/src/tmpm461_uart.c`, with `/Libraries/TX04_Periph_Driver/inc/tmpm461_uart.h` containing the macros, data types, structures and API definitions for use by applications.

18.2 API Functions

18.2.1 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_TRxAutoDisable *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**);
- ◆ void UART_TxBufferClear(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 UART_SetInputClock(TSB_SC_TypeDef * **UARTx**, uint32_t *clock*)

- ◆ void SIO_SetInputClock(TSB_SC_TypeDef * **SIOx**, uint32_t **Clock**)
- ◆ void SIO_Enable(TSB_SC_TypeDef* **SIOx**)
- ◆ void SIO_Disable(TSB_SC_TypeDef* **SIOx**)
- ◆ void SIO_Init(TSB_SC_TypeDef* **SIOx**,
uint32_t **IOClkSel**,
UART_InitTypeDef* **InitStruct**)
- ◆ uint8_t SIO_GetRxData(TSB_SC_TypeDef* **SIOx**)
- ◆ void SIO_SetTxData(TSB_SC_TypeDef* **SIOx**, uint8_t **Data**)

18.2.2 Detailed Description

Functions listed above can be divided into four parts:

- 1) Initialize and configure the common functions of each UART channel are handled by UART_Enable(), UART_Disable(), UART_SetInputClock(), UART_Init() and UART_DefaultConfig(), SIO_Enable(), SIO_Disable(), SIO_SetInputClock(), 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_SetWakeUpFunc() and UART_SetIdleMode() handle other specified functions.
- 4) FIFO operation functions are UART_FIFOConfig(), UART_SetFIFOTransferMode(), UART_TrxAutoDisable(), UART_RxFIFOINTCtrl(), UART_TxFIFOINTCtrl(), UART_RxFIFOByteSel(), UART_RxFIFOFillLevel(), UART_RxFIFOINTSel(), UART_RxFIFOClear(), UART_TxFIFOFillLevel(), UART_TxFIFOINTSel(), UART_TxFIFOClear(), UART_TxBufferClear(), UART_GetRxFIFOFillLevelStatus(), UART_GetRxFIFOOverRunStatus(), UART_GetTxFIFOFillLevelStatus(), and UART_GetTxFIFOUnderRunStatus(),

18.2.3 Function Documentation

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

UART0, UART1, UART2, UART3, UART4, UART5.

parameter “TSB_SC_TypeDef* **SIOx**” can be one of the following values:
SIO0, SIO1, SIO2, SIO3, SIO4, SIO5.

18.2.3.1 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

18.2.3.2 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

18.2.3.3 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.

18.2.3.4 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

18.2.3.5 UART_Init

Initialize and configure the specified UART channel.

Prototype:
void
UART_Init(TSB_SC_TypeDef* **UARTx**,
 UART_InitTypeDef* **InitStruct**)

Parameters:
UARTx is the specified UART channel.
InitStruct is the structure containing basic UART configuration including baud rate, data bits per transfer, stop bits, parity, transfer mode and flow control (refer to "Data Structure Description" for details).

Description:
This function will initialize and configure the baud rate, the number of bits per transfer, stop bit, parity, transfer mode and flow control for the specified UART channel selected by **UARTx**.

Return:
None

18.2.3.6 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

18.2.3.7 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

18.2.3.8 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

18.2.3.9 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.

18.2.3.10 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

18.2.3.11 UART_SetInputClock

Selects input clock for prescaler.

Prototype:

```
void  
UART_SetInputClock (TSB_SC_TypeDef * UARTx,  
                   uint32_t clock)
```

Parameters:

UARTx is the specified UART channel.

Clock is Selects input clock for prescaler as PhiT0/2 or PhiT0.

This parameter can be one of the following values:

0 :PhiT0/2

1 :PhiT0

Description:

This function will select the specified UART channel by **UARTx** and specified the input clock for prescaler by **clock**

Return:

None

18.2.3.12 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

18.2.3.13 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

18.2.3.14 UART_SetFIFOTransferMode

Transfer mode setting.

Prototype:

```
void  
UART_SetFIFOTransferMode (TSB_SC_TypeDef* UARTx,  
                          uint32_t TransferMode);
```

Parameters:

UARTx is the specified UART channel.

TransferMode Transfer mode.

This parameter can be one of the following values:

UART_TRANSFER_PROHIBIT, **UART_TRANSFER_HALFDPX_RX**, **UART_TRANSFER_HALFDPX_TX** or **UART_TRANSFER_FULLDPX**.

Description:

Transfer mode setting.

Return:

None

18.2.3.15 UART_TRxAutoDisable

Controls automatic disabling of transmission and reception.

Prototype:

```
void  
UART_TRxAutoDisable (TSB_SC_TypeDef* UARTx,  
                    UART_TRxAutoDisable TRxAutoDisable);
```

Parameters:

UARTx is the specified UART channel.

TRxAutoDisable Disabling transmission and reception or not

This parameter can be one of the following values:

UART_RTXCNT_NONE or **UART_RTXCNT_AUTODISABLE** .

Description:

Controls automatic disabling of transmission and reception.

Return:

None

18.2.3.16 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:

Enable or disable receive interrupt for receive FIFO.

Return:

None

18.2.3.17 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 transmit FIFO.

This parameter can be one of the following values:

ENABLE or DISABLE

Description:

Enable or disable transmit interrupt for transmit FIFO.

Return:

None

18.2.3.18 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:

Bytes used in receive FIFO.

Return:

None

18.2.3.19 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 or UART_RXFIFO4B_FLEVLE_3_1B.**

Description:

Receive FIFO fill level to generate receive interrupts.

Return:

None

18.2.3.20 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:

Select RX interrupt generation condition.

Return:

None

18.2.3.21 UART_RxFIFOClear

Receive FIFO clear.

Prototype:

void

UART_RxFIFOClear (TSB_SC_TypeDef* *UARTx*);

Parameters:

UARTx is the specified UART channel.

Description:

Receive FIFO clear.

Return:

None

18.2.3.22 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_FLEVLE_0_0B, **UART_TXFIFO4B_FLEVLE_1_1B**,
UART_TXFIFO4B_FLEVLE_2_0B or **UART_TXFIFO4B_FLEVLE_3_1B**.

Description:

Transmit FIFO fill level to generate transmit interrupts.

Return:

None

18.2.3.23 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:

Select TX interrupt generation condition.

Return:

None

18.2.3.24 UART_TxFIFOClear

TransmitFIFO clear.

Prototype:

```
void  
UART_TxFIFOClear (TSB_SC_TypeDef* UARTx);
```

Parameters:

UARTx is the specified UART channel.

Description:

Transmit FIFO clear.

Return:
None

18.2.3.25 UART_TxBufferClear

Transmit buffer clear.

Prototype:
void
UART_TxBufferClear (TSB_SC_TypeDef* **UARTx**);

Parameters:
UARTx is the specified UART channel.

Description:
Transmit buffer clear.

Return:
None

18.2.3.26 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.

18.2.3.27 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.

18.2.3.28 UART_GetTxFIFOFillLevelStatus

Status of transmit FIFO fill level.

Prototype:

uint32_t

UART_GetTxFIFOFillLevelStatus (TSB_SC_TypeDef* *UARTx*);

Parameters:

UARTx is the specified UART channel.

Description:

Status of transmit FIFO fill level.

Return:

UART_TRXFIFO_EMPTY: TX FIFO fill level is empty.

UART_TRXFIFO_1B: TX FIFO fill level is 1 byte.

UART_TRXFIFO_2B: TX FIFO fill level is 2 bytes.

UART_TRXFIFO_3B: TX FIFO fill level is 3 bytes.

UART_TRXFIFO_4B: TX FIFO fill level is 4 bytes.

18.2.3.29 UART_GetTxFIFOUnderRunStatus

Transmit FIFO under run

Prototype:

uint32_t

UART_GetTxFIFOUnderRunStatus (TSB_SC_TypeDef* *UARTx*);

Parameters:

UARTx is the specified UART channel.

Description:

Transmit FIFO under run

Return:

UART_TXFIFO_UNDERRUN: Flags for TX FIFO under-run.

18.2.3.30 SIO_SetInputClock

Selects input clock for prescaler.

Prototype:

void

SIO_SetInputClock (TSB_SC_TypeDef * *SIOx*,
uint32_t *Clock*)

Parameters:

SIOx is the specified SIO channel.

Clock is Selects input clock for prescaler as PhiT0/2 or PhiT0.

This parameter can be one of the following values:

SIO_CLOCK_T0_HALF :PhiT0/2

SIO_CLOCK_T0 :PhiT0

Description:

This function will select the specified SIO channel by **SIOx** and specified the input clock for prescaler by **clock**

Return:
None

18.2.3.31 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

18.2.3.32 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

18.2.3.33 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

18.2.3.34 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

18.2.3.35 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. (refer to "Data Structure Description" for details).

Description:

This function will initialize and configure the specified SIO channel selected by **SIOx**.

Return:
None

18.2.4 Data Structure Description

18.2.4.1 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

18.2.4.2 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

TIDLE The status of TXDx pin after output of the last bit, which can be set as:

- **SIO_TIDLE_LOW** Set the status of TXDx pin keep a low level output.
- **SIO_TIDLE_HIGH** Set the status of TXDx pin keep a high level output.
- **SIO_TIDLE_LAST** Set the status of TXDx pin keep a last bit.

uint32_t

TXDEMP The status of TXDx pin when an under run error is occurred in SCLK input mode, which can be set as:

- **SIO_TXDEMP_LOW** Set the status of TXDx pin is low level output.
- **SIO_TXDEMP_HIGH** Set the status of TXDx pin is high level output.

uint32_t

EHOLDTime The last bit hold time of TXDx pin in SCLK input mode, which can be set as:

- **SIO_EHOLD_FC_2** Set a last bit hold time is 2/fc.
- **SIO_EHOLD_FC_4** Set a last bit hold time is 4/fc.
- **SIO_EHOLD_FC_8** Set a last bit hold time is 8/fc.
- **SIO_EHOLD_FC_16** Set a last bit hold time is 16/fc.
- **SIO_EHOLD_FC_32** Set a last bit hold time is 32/fc.
- **SIO_EHOLD_FC_64** Set a last bit hold time is 64/fc.
- **SIO_EHOLD_FC_128** Set a last bit hold time is 128/fc.

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 Setting transfer mode, which can be set as:

- **SIO_LSB_FRIST** LSB first.
- **SIO_MSB_FRIST** MSB first.

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_TS0** Select the input clock to baud rate generator is TS0.
- **SIO_BR_CLOCK_TS2** Select the input clock to baud rate generator is TS2.
- **SIO_BR_CLOCK_TS8** Select the input clock to baud rate generator is TS8.
- **SIO_BR_CLOCK_TS32** Select the input clock to baud rate generator is TS32.

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.

TOSHIBA

- **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.

19. uDMAC

19.1 Overview

TMPM461x incorporates 3 units of built-in DMA controller.

The main functions for one unit are shown below:

| Functions | Features | | Descriptions |
|------------------------|--|---|---|
| Channels | 32 channels | | - |
| Start trigger | Start by Hardware | | DMA requests from peripheral functions |
| | Start by Software | | Specified by DMAxChnlSwRequest register |
| Priority | Between channels | ch0 (high priority) > ... > ch31 (high priority) > ch0 (Normal priority) > ... > ch31 (Normal priority) | High-priority can be configured by DMAxChnlPriority-Set register |
| Transfer data size | 8/16/32bit | | Can be specified source and destination independently |
| The number of transfer | 1 to 4095 times | | - |
| Address | Transfer source address | Increment / fixed | Transfer source address and destination address can be selected to increment or fixed. |
| | transfer destination address | Increment / fixed | |
| Endian | Little Endian | | - |
| Transfer type | Peripheral (register) → memory Memory → peripheral (register) Memory → memory | | If you select memory to memory, hardware start for DMA start up is not supported. Refer to the DMACxConfiguration register for more information. |
| Interrupt function | Transfer end interrupt Error interrupt | | Output for each unit |
| Transfer mode | Basic mode Automatic request mode Ping-pong mode Memory scatter / gather mode Peripheral scatter / gather mode | | - |

The uDMAC API provides a set of functions for using the TMPM461 uDMAC modules. It includes uDMAC transfer type set, channel set, mask set, primary/alternative data area set, channel priority, initialize data filling and so on.

This driver is contained in TX04_Periph_Driver\src\tmpm461_udmac.c, with TX04_Periph_Driver\incl\tmpm461_udmac.h containing the API definitions for use by applications.

***Note:** In this document, DMAC means uDMAC.

19.2 API Functions

19.2.1 Function List

- ◆ FunctionalState DMAC_GetDMACState(TSB_DMA_TypeDef * **DMACx**)
- ◆ void DMAC_Enable(TSB_DMA_TypeDef * **DMACx**)
- ◆ void DMAC_Disable(TSB_DMA_TypeDef * **DMACx**)
- ◆ void DMAC_SetPrimaryBaseAddr(TSB_DMA_TypeDef * **DMACx**, uint32_t **Addr**)
- ◆ uint32_t DMAC_GetBaseAddr(TSB_DMA_TypeDef * **DMACx**,

- ◆ void DMAC_SetSWReq(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel**)
- ◆ void DMACA_SetTransferType(DMACA_Channel **Channel**,
DMAC_TransferType **Type**)
- ◆ DMAC_TransferType DMACA_GetTransferType(DMACA_Channel **Channel**)
- ◆ void DMAC_SetMask(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel** ,
FunctionalState **NewState**)
- ◆ FunctionalState DMAC_GetMask(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel**)
- ◆ void DMAC_SetChannel(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel** ,
FunctionalState **NewState**)
- ◆ FunctionalState DMAC_GetChannelState(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel**)
- ◆ void DMAC_SetPrimaryAlt(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel**
DMAC_PrimaryAlt **PriAlt**)
- ◆ DMAC_PrimaryAlt DMAC_GetPrimaryAlt(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel**)
- ◆ void DMAC_SetChannelPriority(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel** ,
DMAC_Priority **Priority**)
- ◆ DMAC_Priority DMAC_GetChannelPriority(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel**)
- ◆ void DMAC_ClearBusErr(TSB_DMA_TypeDef * **DMACx**)
- ◆ Result DMAC_GetBusErrState(TSB_DMA_TypeDef * **DMACx**)
- ◆ void DMAC_FillInitData(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel** ,
DMAC_InitTypeDef * **InitStruct**)
- ◆ DMACA_Flag DMACA_GetINTFlag(void)
- ◆ DMACB_Flag DMACB_GetINTFlag(void)
- ◆ DMACC_Flag DMACC_GetINTFlag(void)

19.2.2 Detailed Description

Functions listed above can be divided into six parts:

- 1) uDMAC configuration by DMACA_SetTransferType(), DMACA_GetTransferType(), DMAC_SetMask(), DMAC_GetMask(), DMAC_SetChannel(), DMAC_GetChannelState(), DMAC_SetPrimaryAlt(), DMAC_GetPrimaryAlt(), DMAC_SetChannelPriority(), DMAC_GetChannelPriority().
- 2) uDMAC enable/disable by DMAC_GetDMACState(), DMAC_Enable(), DMAC_Disable().
- 3) uDMAC software trigger by DMAC_SetSWReq().
- 4) uDMAC bus error by DMAC_ClearBusErr(), DMAC_GetBusErrState().
- 5) uDMAC control data area filled by: DMAC_FillInitData(), DMAC_SetPrimaryBaseAddr(), DMAC_GetBaseAddr().
- 6) uDMAC factor flag by DMACA_GetINTFlag(), DMACB_GetINTFlag(), DMACC_GetINTFlag(),

19.2.3 Function Documentation

NOTE: For the parameter ‘DMACx’ and ‘Channel’ of all functions, if there isn’t special explanation, the sentence ‘DMACx: Select DMAC unit.’ and ‘Channel: Select channel’ will follow the content below:

DMACx: Select DMAC unit.

This parameter can be one of the following values:

- **DMAC_UNIT_A :** DMAC unit A
- **DMAC_UNIT_B :** DMAC unit B
- **DMAC_UNIT_C :** DMAC unit C

Channel: Select channel.

The parameter can be one of the following values:

For DMAC_UNIT_A:

- **DMACA_ADC_COMPLETION :** ADC conversion completion
- **DMACA_SSP0_RX :** SSP0 reception
- **DMACA_SSP0_TX :** SSP0 transmission
- **DMACA_SSP1_RX :** SSP1 reception
- **DMACA_SSP1_TX :** SSP1 transmission
- **DMACA_SSP2_RX :** SSP2 reception
- **DMACA_SSP2_TX :** SSP2 transmission
- **DMACA_UART0_RX :** UART0 reception
- **DMACA_UART0_TX :** UART0 transmission
- **DMACA_UART1_RX :** UART1 reception
- **DMACA_UART1_TX :** UART1 transmission
- **DMACA_I2C0_TX_RX :** I2C0 ransmission/reception
- **DMACA_I2C1_TX_RX :** I2C1 ransmission/reception
- **DMACA_I2C2_TX_RX :** I2C2 ransmission/reception
- **DMACA_I2C3_TX_RX :** I2C3 ransmission/reception
- **DMACA_I2C4_TX_RX :** I2C4 ransmission/reception
- **DMACA_DMAREQA :** DMA UNITA request pin DMAREQA

For DMAC_UNIT_B:

- **DMACB_SIO0_UART0_RX :** SIO/UART0 reception
- **DMACB_SIO0_UART0_TX :** SIO/UART0 transmission
- **DMACB_SIO1_UART1_RX :** SIO/UART1 reception
- **DMACB_SIO1_UART1_TX :** SIO/UART1 transmission
- **DMACB_SIO2_UART2_RX :** SIO/UART2 reception
- **DMACB_SIO2_UART2_TX :** SIO/UART2 transmission
- **DMACB_TMRB0_CMP_MATCH :** TMRB0 compare match
- **DMACB_TMRB1_CMP_MATCH :** TMRB1 compare match
- **DMACB_TMRB2_CMP_MATCH :** TMRB2 compare match
- **DMACB_TMRB3_CMP_MATCH :** TMRB3 compare match
- **DMACB_TMRB4_CMP_MATCH :** TMRB4 compare match
- **DMACB_TMRB5_CMP_MATCH :** TMRB5 compare match
- **DMACB_TMRB6_CMP_MATCH :** TMRB6 compare match
- **DMACB_TMRB7_CMP_MATCH :** TMRB7 compare match
- **DMACB_TMRBF_CMP_MATCH :** TMRBF compare match
- **DMACB_TMRB0_INPUT_CAP0 :** TMRB0 input capture 0
- **DMACB_TMRB0_INPUT_CAP1 :** TMRB0 input capture 1
- **DMACB_TMRB1_INPUT_CAP0 :** TMRB1 input capture 0
- **DMACB_TMRB1_INPUT_CAP1 :** TMRB1 input capture 1
- **DMACB_TMRB2_INPUT_CAP0 :** TMRB2 input capture 0
- **DMACB_TMRB2_INPUT_CAP1 :** TMRB2 input capture 1
- **DMACB_TMRB3_INPUT_CAP0 :** TMRB3 input capture 0
- **DMACB_TMRB3_INPUT_CAP1 :** TMRB3 input capture 1
- **DMACB_TMRB4_INPUT_CAP0 :** TMRB4 input capture 0

- **DMACB_TMRB4_INPUT_CAP1** : TMRB4 input capture 1
- **DMACB_TMRB5_INPUT_CAP0** : TMRB5 input capture 0
- **DMACB_TMRB5_INPUT_CAP1** : TMRB5 input capture 1
- **DMACB_DMAREQB** : DMA UNITB request pin DMAREQB

For DMAC_UNIT_C:

- **DMACC_SIO3_UART3_RX** : SIO/UART3 reception
- **DMACC_SIO3_UART3_TX** : SIO/UART3 transmission
- **DMACC_SIO4_UART4_RX** : SIO/UART4 reception
- **DMACC_SIO4_UART4_TX** : SIO/UART4 transmission
- **DMACC_SIO5_UART5_RX** : SIO/UART5 reception
- **DMACC_SIO5_UART5_TX** : SIO/UART5 transmission
- **DMACC_TMRB8_CMP_MATCH** : TMRB8 compare match
- **DMACC_TMRB9_CMP_MATCH** : TMRB9 compare match
- **DMACC_TMRBA_CMP_MATCH** : TMRBA compare match
- **DMACC_TMRBB_CMP_MATCH** : TMRBB compare match
- **DMACC_TMRBC_CMP_MATCH** : TMRBC compare match
- **DMACC_TMRBD_CMP_MATCH** : TMRBD compare match
- **DMACC_TMRBE_CMP_MATCH** : TMRBE compare match
- **DMACC_TMRB7_CMP_MATCH** : TMRB7 compare match
- **DMACC_TMRBF_CMP_MATCH** : TMRBF compare match
- **DMACC_TMRB8_INPUT_CAP0** : TMRB8 input capture 0
- **DMACC_TMRB8_INPUT_CAP1** : TMRB8 input capture 1
- **DMACC_TMRB9_INPUT_CAP0** : TMRB9 input capture 0
- **DMACC_TMRB9_INPUT_CAP1** : TMRB9 input capture 1
- **DMACC_TMRBA_INPUT_CAP0** : TMRBA input capture 0
- **DMACC_TMRBA_INPUT_CAP1** : TMRBA input capture 1
- **DMACC_TMRBB_INPUT_CAP0** : TMRBB input capture 0
- **DMACC_TMRBB_INPUT_CAP1** : TMRBB input capture 1
- **DMACC_TMRBC_INPUT_CAP0** : TMRBC input capture 0
- **DMACC_TMRBC_INPUT_CAP1** : TMRBC input capture 1
- **DMACC_TMRBD_INPUT_CAP0** : TMRBD input capture 0
- **DMACC_TMRBD_INPUT_CAP1** : TMRBD input capture 1
- **DMACC_DMAREQC** : DMA UNITC request pin DMAREQC

19.2.3.1 DMAC_GetDMACState

Get the state of specified DMAC unit.

Prototype:

FunctionalState

DMAC_GetDMACState(TSB_DMA_TypeDef * **DMACx**)

Parameters:

DMACx: Select DMAC unit.

Description:

This function will get the state of specified DMAC unit.

Return:

- **DISABLE** : The DMAC unit is disabled
- **ENABLE** : The DMAC unit is enabled

19.2.3.2 DMAC_Enable

Enable the specified DMAC unit.

Prototype:

void
DMAC_Enable(TSB_DMA_TypeDef * **DMACx**)

Parameters:

DMACx: Select DMAC unit.

Description:

This function will enable the specified DMAC unit.

Return:

None

19.2.3.3 DMAC_Disable

Disable the specified DMAC unit.

Prototype:

void
DMAC_Disable(TSB_DMA_TypeDef * **DMACx**)

Parameters:

DMACx: Select DMAC unit.

Description:

This function will disable the specified DMAC unit.

Return:

None

19.2.3.4 DMAC_SetPrimaryBaseAddr

Set the base address of the primary data of the specified DMAC unit.

Prototype:

void
DMAC_SetPrimaryBaseAddr(TSB_DMA_TypeDef * **DMACx**,
uint32_t **Addr**)

Parameters:

DMACx: Select DMAC unit.

Addr: The base address of the primary data, bit0 to bit9 must be 0.

Description:

This function will set the base address of the primary data of the specified DMAC unit.

Return:

None

19.2.3.5 DMAC_GetBaseAddr

Get the primary/alternative base address of the specified DMAC unit.

Prototype:

```
uint32_t  
DMAC_GetBaseAddr(TSB_DMA_TypeDef * DMACx,  
                 DMAC_PrimaryAlt PriAlt)
```

Parameters:

DMACx: Select DMAC unit.

PriAlt: Select base address type

This parameter can be one of the following values:

- **DMAC_PRIMARY** : Get primary base address
- **DMAC_ALTERNATE** : Get alternative base address

Description:

This function will get the primary/alternative base address of the specified DMAC unit.

Return:

The base address of primary/alternative data

19.2.3.6 DMAC_SetSWReq

Set software transfer request to the specified channel of the specified DMAC unit.

Prototype:

```
void  
DMAC_SetSWReq(TSB_DMA_TypeDef * DMACx ,  
              uint8_t Channel)
```

Parameters:

DMACx: Select DMAC unit.

Channel: Select channel.

Description:

This function will set software transfer request to the specified channel by **Channel** of the specified DMAC unit.

Return:

None

19.2.3.7 DMACA_SetTransferType

Set transfer type to the specified channel of the DMAC UNITA.

Prototype:

```
void  
DMACA_SetTransferType(uint8_t Channel,  
                     DMAC_TransferType Type)
```

Parameters:

Channel: Select UNITA channel.

This parameter can be one of the following values:

When Type is DMAC_BURST:

- **DMACA_ADC_COMPLETION** : ADC conversion completion
- **DMACA_SSP0_RX** : SSP0 reception
- **DMACA_SSP0_TX** : SSP0 transmission
- **DMACA_SSP1_RX** : SSP1 reception
- **DMACA_SSP1_TX** : SSP1 transmission
- **DMACA_SSP2_RX** : SSP2 reception
- **DMACA_SSP2_TX** : SSP2 transmission
- **DMACA_UART0_RX** : UART0 reception
- **DMACA_UART0_TX** : UART0 transmission
- **DMACA_UART1_RX** : UART1 reception
- **DMACA_UART1_TX** : UART1 transmission
- **DMACA_I2C0_TX_RX** : I2C0 ransmission/reception
- **DMACA_I2C1_TX_RX** : I2C1 ransmission/reception
- **DMACA_I2C2_TX_RX** : I2C2 ransmission/reception
- **DMACA_I2C3_TX_RX** : I2C3 ransmission/reception
- **DMACA_I2C4_TX_RX** : I2C4 ransmission/reception
- **DMACA_DMAREQA** : DMA UNITA request pin DMAREQA

When Type is DMAC_SINGLE:

- **DMACA_SSP0_RX** : SSP0 reception
- **DMACA_SSP0_TX** : SSP0 transmission
- **DMACA_SSP1_RX** : SSP1 reception
- **DMACA_SSP1_TX** : SSP1 transmission
- **DMACA_SSP2_RX** : SSP2 reception
- **DMACA_SSP2_TX** : SSP2 transmission
- **DMACA_UART0_RX** : UART0 reception
- **DMACA_UART0_TX** : UART0 transmission
- **DMACA_UART1_RX** : UART1 reception
- **DMACA_UART1_TX** : UART1 transmission

Type: Select transfer type.

This parameter can be one of the following values:

- **DMAC_BURST** : Single transfer is disabled, only burst transfer request can be used
- **DMAC_SINGLE** : Single transfer is enabled

Description:

This function will set transfer type to the specified channel of the DMAC UNITA.

Return:

None

19.2.3.8 DMACA_GetTransferType

Get transfer type setting for the specified channel of the DMAC UNITA

Prototype:

DMAC_TransferType

DMACA_GetTransferType(uint8_t **Channel**)

Parameters:

Channel: Select UNITA channel.

The parameter can be one of the following values:

- **DMACA_ADC_COMPLETION** : ADC conversion completion
- **DMACA_SSP0_RX** : SSP0 reception
- **DMACA_SSP0_TX** : SSP0 transmission
- **DMACA_SSP1_RX** : SSP1 reception
- **DMACA_SSP1_TX** : SSP1 transmission
- **DMACA_SSP2_RX** : SSP2 reception
- **DMACA_SSP2_TX** : SSP2 transmission
- **DMACA_UART0_RX** : UART0 reception
- **DMACA_UART0_TX** : UART0 transmission
- **DMACA_UART1_RX** : UART1 reception
- **DMACA_UART1_TX** : UART1 transmission
- **DMACA_I2C0_TX_RX** : I2C0 transmission/reception
- **DMACA_I2C1_TX_RX** : I2C1 transmission/reception
- **DMACA_I2C2_TX_RX** : I2C2 transmission/reception
- **DMACA_I2C3_TX_RX** : I2C3 transmission/reception
- **DMACA_I2C4_TX_RX** : I2C4 transmission/reception
- **DMACA_DMAREQA** : DMA UNITA request pin DMAREQA

Description:

This function will get transfer type setting for the specified channel of the DMAC UNITA.

Return:

The transfer type with DMAC_TransferType type:

- **DMAC_BURST** : Single transfer is disabled, only burst transfer request can be used
- **DMAC_SINGLE** : Single transfer is enabled

19.2.3.9 DMAC_SetMask

Set mask for the specified channel of the specified DMAC unit.

Prototype:

```
void  
DMAC_SetMask(TSB_DMA_TypeDef * DMACx ,  
              uint8_t Channel ,  
              FunctionalState NewState)
```

Parameters:

DMACx: Select DMAC unit.

Channel: Select channel.

NewState: Clear or set the mask to enable or disable the DMA channel.

This parameter can be one of the following values:

- **ENABLE** : The DMA channel mask is cleared, DMA request is enable(valid)
- **DISABLE** : The DMA channel is masked, DMA request is disable(invalid)

Description:

This function will set mask for the specified channel of the specified DMAC unit.

Return:

None

19.2.3.10 DMAC_GetMask

Get mask setting for the specified channel of the specified DMAC unit.

Prototype:

FunctionalState
DMAC_GetMask(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel**)

Parameters:

DMACx: Select DMAC unit.

Channel: Select channel.

Description:

This function will get mask setting for the specified channel of the specified DMAC unit.

Return:

The inverted mask setting:

- **ENABLE :** The DMA channel mask is cleared, DMA request is enable(valid)
- **DISABLE :** The DMA channel is masked, DMA request is disable(invalid)

19.2.3.11 DMAC_SetChannel

Enable or disable the specified channel of the specified DMAC unit.

Prototype:

void
DMAC_SetChannel(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel** ,
FunctionalState **NewState**)

Parameters:

DMACx: Select DMAC unit.

Channel: Select channel.

NewState: Enable or disable the DMA channel.

This parameter can be one of the following values:

- **ENABLE :** The DMA channel will be enabled
- **DISABLE :** The DMA channel will be disabled

Description:

This function will enable or disable the specified channel of the specified DMAC unit. by **NewState**.

Return:

None

19.2.3.12 DMAC_GetChannelState

Get the enable/disable setting for specified channel of the specified DMAC unit.

Prototype:

FunctionalState
DMAC_GetChannelState(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel**)

Parameters:

DMACx: Select DMAC unit.

Channel: Select channel.

Description:

This function will get the enable/disable setting for specified channel of the specified DMAC unit.

Return:

The enable/disable setting for channel:

- **ENABLE** : The DMA channel is enabled
- **DISABLE** : The DMA channel is disabled

19.2.3.13 DMAC_SetPrimaryAlt

Set to use primary data or alternative data for specified channel of the specified DMAC unit.

Prototype:

void
DMAC_SetPrimaryAlt(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel** ,
DMAC_PrimaryAlt **PriAlt**)

Parameters:

DMACx: Select DMAC unit.

Channel: Select channel.

PriAlt: Select primary data or alternative data for channel specified by 'ChannelA' above.

This parameter can be one of the following values:

- **DMAC_PRIMARY**: Channel will use primary data
- **DMAC_ALTERNATE**: Channel will use alternative data

Description:

This function will set to use primary data or alternative data for specified channel of the specified DMAC unit.

Return:

None

19.2.3.14 DMAC_GetPrimaryAlt

Get the setting of the using of primary data or alternative data for specified channel of the specified DMAC unit.

Prototype:

DMAC_PrimaryAlt

DMAC_GetPrimaryAlt(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel**)

Parameters:

DMACx: Select DMAC unit.

Channel: Select channel.

Description:

This function will get the setting of the using of primary data or alternative data for specified channel of the specified DMAC unit.

Return:

The setting of the using of primary data or alternative data:

- **DMAC_PRIMARY:** Channel is using primary data
- **DMAC_ALTERNATE:** Channel is using alternative data

19.2.3.15 DMAC_SetChannelPriority

Set the priority for specified channel of the specified DMAC unit.

Prototype:

void
DMAC_SetChannelPriority(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel** ,
DMAC_Priority **Priority**)

Parameters:

DMACx: Select DMAC unit.

Channel: Select channel.

Priority: Select Priority.

This parameter can be one of the following values:

- **DMAC_PRIOTIRY_NORMAL:** Normal priority.
- **DMAC_PRIOTIRY_HIGH:** High priority.

Description:

This function will set the priority for specified channel of the specified DMAC unit.

Return:

None

19.2.3.16 DMAC_GetChannelPriority

Get the priority setting for specified channel of the specified DMAC unit.

Prototype:

DMAC_Priority
DMAC_GetChannelPriority(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel**)

Parameters:

DMACx: Select DMAC unit.

Channel: Select channel.

Description:

This function will get the priority setting for specified channel of the specified DMAC unit

Return:

The priority setting of channel:

- **DMAC_PRIOTIRY_NORMAL:** Normal priority.
- **DMAC_PRIOTIRY_HIGH:** High priority.

19.2.3.17 DMAC_ClearBusErr

Clear the bus error of the specified DMAC unit.

Prototype:

void

DMAC_ClearBusErr(TSB_DMA_TypeDef * **DMACx**)

Parameters:

DMACx: Select DMAC unit.

Description:

This function will clear the bus error of the specified DMAC unit.

Return:

None

19.2.3.18 DMAC_GetBusErrState

Get the bus error state of the specified DMAC unit.

Prototype:

Result

DMAC_GetBusErrState(TSB_DMA_TypeDef * **DMACx**)

Parameters:

DMACx: Select DMAC unit.

Description:

This function will get the bus error state of the specified DMAC unit.

Return:

The bus error state:

- **SUCCESS:** No bus error.
- **ERROR :** There is error in bus.

19.2.3.19 DMAC_FillInitData

Fill the DMA setting data of specified channel of the DMAC UNITA to RAM.

Prototype:

void

DMAC_FillInitData(TSB_DMA_TypeDef * **DMACx** ,
uint8_t **Channel** ,

DMAC_InitTypeDef * **InitStruct**)

Parameters:

DMACx: Select DMAC unit.

Channel: Select channel.

InitStruct: The structure contains the DMA setting values.

Description:

This function will fill the DMA setting data of specified channel of the DMAC UNITA to RAM.

Return:

None

19.2.3.20 DMACA_GetINTFlag

Get the DMA factor flag of the DMAC UNITA

Prototype:

DMACA_Flag

DMACA_GetINTFlag(void)

Parameters:

None

Description:

This function will get the DMA factor flag of the DMAC UNITA

Return:

A union with DMA factor flag of DMAC UNITA(refer to Data Structure Description of DMACA_Flag for details)

19.2.3.21 DMACB_GetINTFlag

Get the DMA factor flag of the DMAC UNITB

Prototype:

DMACB_Flag

DMACB_GetINTFlag(void)

Parameters:

None

Description:

This function will get the DMA factor flag of the DMAC UNITB

Return:

A union with DMA factor flag of DMAC UNITB(refer to Data Structure Description of DMACB_Flag for details)

19.2.3.22 DMACC_GetINTFlag

Get the DMA factor flag of the DMAC UNITC

Prototype:

DMACC_Flag

DMACC_GetINTFlag(void)

Parameters:

None

Description:

This function will get the DMA factor flag of the DMAC UNITC

Return:

A union with DMA factor flag of DMAC UNITC(refer to Data Structure Description of DMACC_Flag for details)

19.2.4 Data Structure Description

19.2.4.1 DMAC_InitTypeDef

Data fields:

uint32_t

SrcEndPoint: The final address of data source.

uint32_t

DstEndPoint: The final address of data destination.

DMAC_CycleCtrl

Mode: Set operation mode,

which can be:

- **DMAC_INVALID:** Invalid, DMA will stop the operation
- **DMAC_BASIC:** Basic mode
- **DMAC_AUTOMATIC:** Automatic request mode
- **DMAC_PINGPONG:** Ping-pong mode
- **DMAC_MEM_SCATTER_GATHER_PRI:** Memory scatter/gather mode (primary data)
- **DMAC_MEM_SCATTER_GATHER_ALT:** Memory scatter/gather mode (alternative data)
- **DMAC_PERI_SCATTER_GATHER_PRI:** Peripheral memory scatter/gather mode (primary data)
- **DMAC_PERI_SCATTER_GATHER_ALT:** Peripheral memory scatter/gather mode (alternative data)

DMAC_Next_UseBurst

NextUseBurst: Specifies whether to set "1" to the register DMAxChnlUseburstSet<chnl_useburst_set> bit to use burst transfer at the end of the DMA transfer using alternative data in the peripheral scatter/gather mode.

which can be:

- **DMAC_NEXT_NOT_USE_BURST:** Do not change the value of <chnl_useburst_set>.
- **DMAC_NEXT_USE_BURST:** Sets <chnl_useburst_set> to "1"

uint32_t

TxNum: Set the actual number of transfers. Maximum is 1024.

DMAC_Arbitration

ArbitrationMoment: Specifies the arbitration moment(R_Power).

After the specified numbers of transfers, an existence of a transfer request is

checked. If there is a high-priority request, the control is switched to high-priority channel.

DMAC_BitWidth

SrcWidth: Set source bit width,
which can be:

- **DMAC_BYTE:** Data size of transfer is 1 byte.
- **DMAC_HALF_WORD:** Data size of transfer is 2 bytes.
- **DMAC_WORD:** Data size of transfer is 4 bytes

DMAC_IncWidth

SrcInc: Set increment of the source address,
which can be:

- **DMAC_INC_1B:** Address increment 1 byte.
- **DMAC_INC_2B:** Address increment 2 bytes.
- **DMAC_INC_4B:** Address increment 4 bytes.
- **DMAC_INC_0B:** Address does not increase

DMAC_BitWidth

DstWidth: Set destination bit width,
which can be:

- **DMAC_BYTE:** Data size of transfer is 1 byte
- **DMAC_HALF_WORD:** Data size of transfer is 2 bytes
- **DMAC_WORD:** Data size of transfer is 4 bytes

DMAC_IncWidth

DstInc: Set increment of the destination address,
which can be:

- **DMAC_INC_1B:** Address increment 1 byte
- **DMAC_INC_2B:** Address increment 2 bytes
- **DMAC_INC_4B:** Address increment 4 bytes
- **DMAC_INC_0B:** Address does not increase

19.2.4.2 DMACA_Flag

Data Fields for this union:

uint32_t

All The flag of DMA UINTA interrupt.

Bit Fields:

uint32_t

ADCCompletion (Bit 0) The flag of ADC completion occurs an interrupt.
'1' means occurs an interrupt

uint32_t

SSP0Reception (Bit 1) The flag of SSP0 reception occurs an interrupt
'1' means occurs an interrupt

uint32_t

SSP0Transmission (Bit 2) The flag of SSP0 transmission occurs an interrupt
'1' means occurs an interrupt

uint32_t

SSP1Reception (Bit 3) The flag of SSP1 reception occurs an interrupt
'1' means occurs an interrupt

uint32_t

SSP1Transmission (Bit 4) The flag of SSP1 transmission occurs an interrupt
'1' means occurs an interrupt

| | |
|--|--|
| uint32_t SSP2Reception (Bit 5) | The flag of SSP2 reception occurs an interrupt '1' means occurs an interrupt |
| uint32_t SSP2Transmission (Bit 6) | The flag of SSP2 transmission occurs an interrupt '1' means occurs an interrupt |
| uint32_t UART0Reception (Bit 7) | The flag of UART0 reception occurs an interrupt '1' means occurs an interrupt |
| uint32_t UART0Transmission (Bit 8) | The flag of UART0 transmission occurs an interrupt '1' means occurs an interrupt |
| uint32_t UART1Reception (Bit 9) | The flag of UART1 reception occurs an interrupt '1' means occurs an interrupt |
| uint32_t UART1Transmission (Bit 10) | The flag of UART1 transmission occurs an interrupt '1' means occurs an interrupt |
| uint32_t I2C0RxorTx (Bit 11) | The flag of I2C0 reception/transmission occurs an interrupt '1' means occurs an interrupt |
| uint32_t I2C1RxorTx (Bit 12) | The flag of I2C1 reception/transmission occurs an interrupt '1' means occurs an interrupt |
| uint32_t I2C2RxorTx (Bit 13) | The flag of I2C2 reception/transmission occurs an interrupt '1' means occurs an interrupt |
| uint32_t I2C3RxorTx (Bit 14) | The flag of I2C3 reception/transmission occurs an interrupt '1' means occurs an interrupt |
| uint32_t I2C4RxorTx (Bit 15) | The flag of I2C4 reception/transmission occurs an interrupt '1' means occurs an interrupt |
| uint32_t Reserved (Bit 16 to Bit 30) | reserved. |
| uint32_t DMAREQA (Bit 31) | The flag of pin DMAREQA occurs an interrupt '1' means occurs an interrupt |

19.2.4.3 DMACB_Flag

Data Fields for this union:

uint32_t
All The flag of DMA UINTB interrupt.

Bit Fields:

uint32_t

SIO_UART0Reception (Bit 0) The flag of SIO/UART0 reception occurs an interrupt.

'1' means occurs an interrupt

uint32_t

SIO_UART0Transmission (Bit 1) The flag of SIO/UART0 transmission occurs an interrupt.

'1' means occurs an interrupt

uint32_t

SIO_UART1Reception (Bit 2) The flag of SIO/UART1 reception occurs an interrupt.

'1' means occurs an interrupt

uint32_t

SIO_UART1Transmission (Bit 3) The flag of SIO/UART1 transmission occurs an interrupt.

'1' means occurs an interrupt

uint32_t

SIO_UART2Reception (Bit 4) The flag of SIO/UART2 reception occurs an interrupt.

'1' means occurs an interrupt

uint32_t

SIO_UART2Transmission (Bit 5) The flag of SIO/UART2 transmission occurs an interrupt.

'1' means occurs an interrupt

uint32_t

Reserved0 (Bit 6 to Bit 9) reserved.

uint32_t

TMRB0CompareMatch (Bit 10) The flag of TMRB0 compare match occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB1CompareMatch (Bit 11) The flag of TMRB1 compare match occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB2CompareMatch (Bit 12) The flag of TMRB2 compare match occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB3CompareMatch (Bit 13) The flag of TMRB3 compare match occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB4CompareMatch (Bit 14) The flag of TMRB4 compare match occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB5CompareMatch (Bit 15) The flag of TMRB5 compare match occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB6CompareMatch (Bit 16) The flag of TMRB6 compare match occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB7CompareMatch (Bit 17) The flag of TMRB7 compare match occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRBFCompareMatch (Bit 18) The flag of TMRBF compare match occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB0InputCapture0 (Bit 19) The flag of TMRB0 input capture 0 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB0InputCapture1 (Bit 20) The flag of TMRB0 input capture 1 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB1InputCapture0 (Bit 21) The flag of TMRB1 input capture 0 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB1InputCapture1 (Bit 22) The flag of TMRB1 input capture 1 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB2InputCapture0 (Bit 23) The flag of TMRB2 input capture 0 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB2InputCapture1 (Bit 24) The flag of TMRB2 input capture 1 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB3InputCapture0 (Bit 25) The flag of TMRB3 input capture 0 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB3InputCapture1 (Bit 26) The flag of TMRB3 input capture 1 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB4InputCapture0 (Bit 27) The flag of TMRB4 input capture 0 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB4InputCapture1 (Bit 28) The flag of TMRB4 input capture 1 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB5InputCapture0 (Bit 29) The flag of TMRB5 input capture 0 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRB5InputCapture1 (Bit 30) The flag of TMRB5 input capture 1 occurs an interrupt

'1' means occurs an interrupt

uint32_t

DMAREQB (Bit 31) The flag of pin DMAREQB occurs an interrupt

'1' means occurs an interrupt

19.2.4.4 DMACC_Flag

Data Fields for this union:

uint32_t

All The flag of DMA UINTC interrupt.

Bit Fields:

uint32_t

SIO_UART3Reception (Bit 0)The flag of SIO/UART3 reception occurs an interrupt.

'1' means occurs an interrupt

uint32_t

SIO_UART3Transmission (Bit 1)The flag of SIO/UART3 transmission occurs an interrupt.

'1' means occurs an interrupt

uint32_t

SIO_UART4Reception (Bit 2)The flag of SIO/UART4 reception occurs an interrupt.

'1' means occurs an interrupt

uint32_t

SIO_UART4Transmission (Bit 3)The flag of SIO/UART4 transmission occurs an interrupt.

'1' means occurs an interrupt

uint32_t

SIO_UART5Reception (Bit 4)The flag of SIO/UART5 reception occurs an interrupt.

'1' means occurs an interrupt

uint32_t

SIO_UART5Transmission (Bit 5)The flag of SIO/UART5 transmission occurs an interrupt.

'1' means occurs an interrupt

uint32_t

Reserved0 (Bit 6 to Bit 9) reserved.

uint32_t

TMRB8CompareMatch (Bit 10) The flag of TMRB8 compare match occurs an interrupt
'1' means occurs an interrupt

uint32_t

TMRB9CompareMatch (Bit 11) The flag of TMRB9 compare match occurs an interrupt
'1' means occurs an interrupt

uint32_t

TMRBACompareMatch (Bit 12) The flag of TMRBA compare match occurs an interrupt
'1' means occurs an interrupt

uint32_t

TMRBBCompareMatch (Bit 13) The flag of TMRBB compare match occurs an interrupt
'1' means occurs an interrupt

uint32_t

TMRBCCompareMatch (Bit 14) The flag of TMRBC compare match occurs an interrupt
'1' means occurs an interrupt

uint32_t

TMRBDCompareMatch (Bit 15) The flag of TMRBD compare match occurs an interrupt
'1' means occurs an interrupt

uint32_t

TMRBECompareMatch (Bit 16) The flag of TMRBE compare match occurs an interrupt
'1' means occurs an interrupt

uint32_t

TMRB7CompareMatch (Bit 17) The flag of TMRB7 compare match occurs an interrupt
'1' means occurs an interrupt

uint32_t

TMRBFCompareMatch (Bit 18) The flag of TMRBF compare match occurs an interrupt
'1' means occurs an interrupt

uint32_t

TMRB8InputCapture0 (Bit 19) The flag of TMRB8 input capture 0 occurs an interrupt
'1' means occurs an interrupt

uint32_t

TMRB8InputCapture1 (Bit 20) The flag of TMRB8 input capture 1 occurs an interrupt
'1' means occurs an interrupt

uint32_t

TMRB9InputCapture0 (Bit 21) The flag of TMRB9 input capture 0 occurs an interrupt
'1' means occurs an interrupt

uint32_t

TMRB9InputCapture1 (Bit 22) The flag of TMRB9 input capture 1 occurs an interrupt
'1' means occurs an interrupt

uint32_t

TMRBAInputCapture0 (Bit 23) The flag of TMRBA input capture 0 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRBAInputCapture1 (Bit 24) The flag of TMRBA input capture 1 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRBBInputCapture0 (Bit 25) The flag of TMRBB input capture 0 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRBBInputCapture1 (Bit 26) The flag of TMRBB input capture 1 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRBCInputCapture0 (Bit 27) The flag of TMRBC input capture 0 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRBCInputCapture1 (Bit 28) The flag of TMRBC input capture 1 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRBDInputCapture0 (Bit 29) The flag of TMRBD input capture 0 occurs an interrupt

'1' means occurs an interrupt

uint32_t

TMRBDInputCapture1 (Bit 30) The flag of TMRBD input capture 1 occurs an interrupt

'1' means occurs an interrupt

uint32_t

DMAREQC (Bit 31) The flag of pin DMAREQC occurs an interrupt

'1' means occurs an interrupt

20. WDT

20.1 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\TX04_Periph_Driver\src\tmpm461_wdt.c, with \Libraries\TX04_Periph_Driver\inc\tmpm461_wdt.h containing the API definitions for use by applications.

20.2 API Functions

20.2.1 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)

20.2.2 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().

20.2.3 Function Documentation

20.2.3.1 WDT_SetDetectTime

Set detection time for WDT.

Prototype:

```
void  
WDT_SetDetectTime(uint32_t DetectTime)
```

Parameters:

DetectTime: Set the detection time

This parameter can be one of the following values:

- **WDT_DETECT_TIME_EXP_15:** **DetectTime** is $2^{15}/f_{sys}$
- **WDT_DETECT_TIME_EXP_17:** **DetectTime** is $2^{17}/f_{sys}$
- **WDT_DETECT_TIME_EXP_19:** **DetectTime** is $2^{19}/f_{sys}$
- **WDT_DETECT_TIME_EXP_21:** **DetectTime** is $2^{21}/f_{sys}$

- **WDT_DETECT_TIME_EXP_23:** *DetectTime* is $2^{23}/f_{sys}$
- **WDT_DETECT_TIME_EXP_25:** *DetectTime* is $2^{25}/f_{sys}$

Description:

This function will set detection time for WDT.

Return:

None

20.2.3.2 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

20.2.3.3 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 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

20.2.3.4 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

20.2.3.5 WDT_Enable

Enable the WDT function.

Prototype:

void
WDT_Enable(void)

Parameters:

None

Description:

This function will enable WDT.

Return:

None

20.2.3.6 WDT_Disable

Disable the WDT function.

Prototype:

void
WDT_Disable(void)

Parameters:

None

Description:

This function will disable WDT.

Return:
None

20.2.3.7 WDT_WriteClearCode

Write the clear code.

Prototype:
void
WDT_WriteClearCode (void)

Parameters:
None

Description:
This function will clear the WDT counter.

Return:
None

20.2.4 Data Structure Description

20.2.4.1 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_WDOOUT:** Set WDT to generate reset when the counter overflows.
- **WDT_NMIINT:** Set WDT to generate NMI interrupt when the counter overflows.