TOSHIBA

# TOSHIBA TX00 Peripheral Driver User Guide (TMPM061)

Ver 1
Sep, 2017

TOSHIBA ELECTRONIC DEVICES & STORAGE CORPORATION

CMDR-M061UG-01E

## RESTRICTIONS ON PRODUCT USE

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

# Index

# 1. Introduction

TOSHIBA TX00 Peripheral Driver is a set of drivers for all peripherals found on the TOSHIBA TX00 series microcontrollers. TMPM061 Peripheral Driver is an important part of TOSHIBA TX00 Peripheral Driver, which are designed for TMPM061 series MCUs.

TOSHIBA TX00 Peripheral Driver contains a collection of macros, data types, and structures for each peripheral.

The design goals of TOSHIBA TMPM061 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 TX00 Peripheral Driver

**/Libraries**
This folder contains all CMSIS files and TMPM061 Peripheral Drivers.

**/Libraries/ TX00_CMSIS**
This folder contains the TMPM061 CMSIS files: device peripheral access layer and core peripheral access layer.

**/Libraries/TX00_Periph_Driver**
This folder contains all the source code of the drivers, the core of TOSHIBA TMPM061 Peripheral Driver.

**/Libraries/TX00_Periph_Driver/inc**
This folder contains all the header files of TMPM061 Peripheral Drivers for each peripheral.

**/Libraries/TX00_Periph_Driver/src**
This folder contains all the source files of TMPM061 Peripheral Drivers for each peripheral.

**/Project**
This folder contains template project and examples for using TMPM061 Peripheral Driver.

**/Project/Template**
This folder contains template project of TOSHIBA TMPM061 Peripheral Driver.

**/Project/Examples**
This folder contains a set of examples for using TMPM061 Peripheral Driver

**/Utilities/TMPM061-EVAL**
This folder contains the configuration and driver files for hardware resources (e.g. led, key) on Toshiba TMPM061-EVAL board.

# 3. ADC

## 3.1 Overview

A 10-bit, sequential-conversion analog/digital converter (AD converter) is built into the TMPM061FWFG.

In the TMPM061FWFG, the following ADC functions cannot be used. Do not set the related registers.

| Function | Register |
|---|---|
| Highest priority conversion | ADMOD2, ADREGSP |
| AD monitoring function | ADMOD3, ADMOD5, ADCMP0, ADCMP1 |
| AD start-up by hardware | ADMOD4 <ADHTG> <ADHS> <HADHTG> <HADHS> |

In the TMPM061FWFG, 3 channels from 0 to 2 are used as input channels of the AD converter. Analog signals input to each channel are as follows.

| Channel | Input |
|---|---|
| Channel 0 | AIN0 pin (PF0/98pin) |
| Channel 1 | Channel 1 AIN1 pin (PF1/99pin) |
| Channel 2 | Channel 2 Temperature sensor output |

Conversion channels are specified with ADMOD0<SCAN>, ADMOD1<ADSCN> and <ADCH>. refer to table below for available settings.

| | | ADMOD1<ADCH[3:0]> | | | |
|---|---|---|---|---|---|
| | | 0000 | 0001 | 0010 | 0011 to 1111 |
| ADMOD0<SCAN>=0 | Fixed channel | AIN0 | AIN1 | AIN2 | Not available |
| ADMOD0<SCAN>=1 | ADMOD1<ADSCN>=00 4-channel scan | AIN0 | AIN0 ~ AIN1 | AIN0 ~ AIN2 | |
| | ADMOD1<ADSCN>=01 8-channel scan | AIN0 | AIN0 ~ AIN1 | AIN0 ~ AIN2 | |
| | ADMOD1<ADSCN>=02 12-channel scan | AIN0 | AIN0 ~ AIN1 | AIN0 ~ AIN2 | |

The ADC API provides a set of functions for using the TMPM061 ADC modules. It includes ADC channel set, mode set, interrupt set, ADC status read, ADC result value read and so on.

This driver is contained in TX00_Periph_Driver\src\tmpm061_adc.c, with TX00_Periph_Driver\inc\tmpm061_adc.h containing the API definitions for use by applications.

## 3.2 API Functions

### 3.2.1 Function List

- void ADC_SWReset(void)
- void ADC_SetClk(uint32_t ***Conversion_Time***, uint32_t ***Prescaler_Output***)
- void ADC_Start(void)
- void ADC_SetScanMode(FunctionalState ***NewState***)
- void ADC_SetRepeatMode(FunctionalState ***NewState***)
- void ADC_SetINTMode(uint8_t ***INTMode***)

- WorkState ADC_GetConvertState(void)
- void ADC_SetInputChannel(uint8_t ***InputChannel***)
- void ADC_SetChannelScanMode(ADC_ChannelScanMode ***ScanMode***)
- void ADC_SetIdleMode(FunctionalState ***NewState***)
- void ADC_SetVref(FunctionalState ***NewState***)
- ADC_Result ADC_GetConvertResult(uint8_t ***ADREGx***)

## 3.2.2 Detailed Description

Functions listed above can be divided into four parts:

1) ADC setting by ADC_SetClk(), ADC_SetScanMode(), ADC_SetRepeatMode(), ADC_SetINTMode(), ADC_SetInputChannel(), ADC_SetChannelScanMode(), ADC_SetVref().
2) ADC function start by ADC_Start().
3) ADC state or data read functions by ADC_GetConvertState(), ADC_GetConvertResult().
4) ADC_SWReset() and ADC_SetIdleMode() handle other specified functions.

## 3.2.3 Function Documentation

### 3.2.3.1 ADC_SWReset

Software reset ADC.

**Prototype:**
void
ADC_SWReset(void)

**Parameters:**
None

**Description:**
This function will software reset ADC to initializes all the registers except ADCLK<ADCLK>.

**Return:**
None

### 3.2.3.2 ADC_SetClk

Set ADC sample hold time and prescaler output.

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

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

- **ADC_CONVERSION_35_CLOCK:** conversion need 35.5 clock
- **ADC_CONVERSION_42_CLOCK:** conversion need 42 clock
- **ADC_CONVERSION_68_CLOCK:** conversion need 68 clock
- **ADC_CONVERSION_81_CLOCK:** conversion need 81 clock

***Prescaler_Output***: Select ADC prescaler output(ADCLK).

This parameter can be one of the following values:

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

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

**Notes:**
Please do not use this function to change the analog to digital conversion clock setting during the conversion is going. After power up or called ADC_SWReset(), or calling **ADC_GetConvertState()** to check AD conversion state is not **BUSY**, user can call this function.

**Return:**
None

### 3.2.3.3 ADC_Start

Start ADC conversion.

**Prototype:**
void
ADC_Start(void)

**Parameters:**
None

**Description:**
This function will start AD conversion.

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

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

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

Before starting AD conversion, Vref must be enabled by calling **ADC_SetVref(ENABLE)**, wait at least 3 us to let the internal reference voltage becomes stable.

**Return:**
None

### 3.2.3.4 ADC_SetScanMode

Set ADC scan mode.

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

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

**Description:**
This function will enable or disable ADC scan mode by ***NewState*** setting.

**Return:**
None

### 3.2.3.5 ADC_SetRepeatMode

Set ADC repeat mode.

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

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

**Description:**
This function will enable or disable ADC repeat mode by ***NewState*** setting.

**Return:**
None

### 3.2.3.6 ADC_SetINTMode

Set ADC interrupt mode in fixed channel repeat conversion mode.

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

**Parameters:**
***INTMode***: Specify AD conversion interrupt mode.
The parameter can be one of the following values:
- **ADC_INT_SINGLE:** Generate in interrupt once every single conversion.
- **ADC_INT_CONVERSION_4:** Generate interrupt once every 4 conversions.
- **ADC_INT_CONVERSION_8:** Generate interrupt once every 8 conversions.

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

**Notes:**
This function is valid only in fixed channel repeat conversion mode.
Examples for setting fixed channel repeat conversion mode:

call **ADC_SetScanMode(DISABLE)** then call **ADC_SetRepeatMode(ENABLE)**.

**Return:**
None

### 3.2.3.7 ADC_GetConvertState

Read normal ADC completion flag.

**Prototype:**
WorkState
ADC_GetConvertState(void)

**Parameters:**
None

**Description:**
This function will read normal AD conversion state. User can use this function to check AD conversion completed or not.

**Return:**
The state of normal AD conversion, which can be:
**DONE**: Conversion is complete.
**BUSY**: Conversion is going.

### 3.2.3.8 ADC_SetInputChannel

Set ADC input channel.

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

**Parameters:**
***InputChannel***: Analog input channel, and the input channel also related with other settings.
This parameter can be one of the following values:
- **ADC_AN_0:** AIN0 pin
- **ADC_AN_1:** AIN1 pin
- **ADC_AN_2:** Inner temperature sensor output

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

**Return:**
None

### 3.2.3.9 ADC_SetChannelScanMode

Set ADC operation for scanning.

**Prototype:**
void
ADC_SetChannelScanMode(ADC_ChannelScanMode ***ScanMode***)

**Parameters:**
***ScanMode***: Specify operation mode for channel scanning.
The parameter can be one of the following values:
**ADC_SCAN_4CH, ADC_SCAN_8CH** or **ADC_SCAN_12CH.**

**Description:**
This function will set ADC operation for scanning
Refer to table below for detail scan range information.

| | | ADC_SetInputChannel() | | | |
|---|---|---|---|---|---|
| | | ADC_AN_0 | ADC_AN_1 | ADC_AN_2 | |
| ADC_SetScanMode(DISABLE) | Fixed channel | AIN0 | AIN1 | AIN2 | |
| ADC_SetScanMode(ENABLE) | ADC_SetChannelScanMode(ADC_SCAN_4CH) 4-channel scan | AIN0 | AIN0 ~ AIN1 | AIN0 ~ AIN2 | |
| | ADC_SetChannelScanMode(ADC_SCAN_8CH) 8-channel scan | AIN0 | AIN0 ~ AIN1 | AIN0 ~ AIN2 | |
| | ADC_SetChannelScanMode(ADC_SCAN_12CH) 12-channel scan | AIN0 | AIN0 ~ AIN1 | AIN0 ~ AIN2 | |

**Return:**
None

### 3.2.3.10 ADC_SetIdleMode

Set ADC operation in IDLE mode.

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

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

**Description:**
This function will specify ADC module is enabled or disabled when system is in IDLE mode by ***NewState*** setting.
This function is necessary to be called before system enter IDLE mode.

**Return:**
None

### 3.2.3.11 ADC_SetVref

Set ADC Vref application control on or off.

**Prototype:**
void

ADC_SetVref(FunctionalState ***NewState***)

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

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

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

**Return:**
None

#### 3.2.3.12 ADC_GetConvertResult

Read ADC register's result storage flag state, overrun state and result value.

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

**Parameters:**
***ADREGx***: Select ADC result register.
The parameter can be one of the following values:
**ADC_REG_0**, **ADC_REG_1**, **ADC_REG_2**,

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

**Return:**
The AD result in ADC_Result Structure:

### 3.2.4 Data Structure Description

#### 3.2.4.1 ADC_ResultTypeDef

**Data Fields for this structure:**

uint32_t
***All:*** AD Conversion Result.
***Bit***

| | | |
|---|---|---|
| uint32_t ***Stored***: | 1 | '1' means AD result has been stored. |
| uint32_t ***OverRun***: | 1 | Overrun flag. |
| uint32_t ***Reserved1***: | 4 | Rerved1. |
| uint32_t ***ADResult***: | 10 | store ADC result |
| uint32_t ***Reserved2***: | 16 | Reserved2. |

# 4. CG

## 4.1 Overview

TOSHIBA TMPM061 has a Clock Generator (CG), which control the clock and core mode of TMPM061.

Feature of Clock Generator:

- The clock/mode control block enables to select clock gear, prescaler clock and warm-up of the oscillator. There is also the low power consumption mode which can reduce power consumption by mode transitions.
  This chapter describes how to control clock operating modes and mode transitions.
  The clock/mode control block has the following functions:
  - Controls the system clock
  - Controls the prescaler clock
  - Controls the warm-up timer

  In addition to NORMAL mode, the TMPM061FWFG can operate in variety of low power modes to reduce powerconsumption according to its usage conditions.

- Clock system
  fEHOSC : External high-speed oscillator clock input from oscillator connected to X1, X2
  fEHCLKIN : External high-speed clock input from X1
  fIHOSC : Internal high-speed oscillator clock input from internal oscillator
  fs : External low-speed oscillator clock input from oscillator connected to XT1, XT2
  fELCLKIN : External low-speed clock input from PJ5 (33pin)
  fosc : High-speed clock specified by CGOSCCR<OSCSEL>
  fc : Clock specified by CGEHCLKSEL<EHCLKSEL> (high-speed clock)
  fgear : Clock specified by CGSYSCR<GEAR[2:0]>
  fsys : Clock specified by CGCKSEL<SYSCK> (system clock)
  fperiph : Clock specified by CGSYSCR<FPSEL[1:0]>
  ΦT0 : Clock specified by CGSYSCR<PRCK[2:0]> (prescaler clock)

  The high-speed clock fc and the prescaler clock ΦT0 are dividable.
  High-speed clock gear: fc, fc/2, fc/4, fc/8, fc/16.
  Prescaler clock: fperiph, fperiph/2, fperiph/4, fperiph/8, fperiph/16, fperiph/32.

## 4.2 API Functions

### 4.2.1 Function List

- void CG_SetFgearLevel(CG_DivideLevel ***DivideFgearFromFc***)
- CG_DivideLevel CG_GetFgearLevel(void)
- void CG_SetPhiT0Src(CG_PhiT0Src ***PhiT0Src***)
- CG_PhiT0Src CG_GetPhiT0Src(void)
- Result CG_SetPhiT0Level(CG_DivideLevel ***DividePhiT0FromFc***)

- CG_DivideLevel CG_GetPhiT0Level(void)
- void CG_SetSCOUTSrc(CG_SCOUTSrc ***Source***)
- CG_SCOUTSrc CG_GetSCOUTSrc(void)
- void CG_SetWarmUpTime(CG_WarmUpSrc ***Source***, uint16_t ***Time***)
- void CG_StartWarmUp(void)
- WorkState CG_GetWarmUpState(void)
- Result CG_SetFosc(CG_FoscSrc ***Source***, FunctionalState ***NewState***)
- void CG_SetFoscSrc(CG_FoscSrc ***Source***)
- CG_FoscSrc CG_GetFoscSrc(void)
- FunctionalState CG_GetFoscState(CG_FoscSrc ***Source***)
- Result CG_SetFs(FunctionalState ***NewState***)
- FunctionalState CG_GetFsState(void)
- void CG_SetPortM(CG_PortMMode ***Mode***)
- void CG_SetLowOscSrc(CG_LoscSrc ***Source***)
- void CG_SetExtHighClk(FunctionalState ***NewState***)
- void CG_SetSTBYMode(CG_STBYMode ***Mode***)
- CG_STBYMode CG_GetSTBYMode(void)
- void CG_SetExitStopModeFosc(FunctionalState ***NewState***)
- FunctionalState CG_GetExitStopModeFoscState(void)
- void CG_SetExitStopModeFs(FunctionalState ***NewState***)
- FunctionalState CG_GetExitStopModeFsState(void)
- void CG_SetPinStateInStopMode(FunctionalState ***NewState***)
- FunctionalState CG_GetPinStateInStopMode(void)
- void CG_SelExtHighClk(CG_EHClkSrc ***Source***)
- Result CG_SetFsysSrc(CG_FsysSrc ***Source***)
- CG_FsysSrc CG_GetFsysSrc(void)
- void CG_SetSTBYReleaseINTSrc(CG_INTSrc ***INTSource***,
  CG_INTActiveState ***ActiveState***,
  FunctionalState ***NewState***)
- CG_INTActiveState CG_GetSTBYReleaseINTState(CG_INTSrc ***INTSource***)
- void CG_ClearINTReq(CG_INTSrc ***INTSource***)
- CG_ResetFlag CG_GetResetFlag(void)

### 4.2.2 Detailed Description

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

1) One group of APIs are in charge of clock selection, such as:
   CG_SetFgearLevel(), CG_GetFgearLevel(),CG_SetPhiT0Src(),CG_GetPhiT0Src(),
   CG_SetPhiT0Level(), CG_GetPhiT0Level(), CG_SetSCOUTSrc(),
   CG_GetSCOUTSrc(), CG_SetWarmUpTime(), CG_StartWarmUp(),
   CG_GetWarmUpState(),CG_SetFosc(),CG_SetFoscSrc(), CG_GetFoscSrc(),
   CG_GetFoscState(), CG_SetFs(),
   CG_GetFsState(),CG_SetFsysSrc(),CG_GetFsysSrc(),CG_SetPortM().

2) The 2nd group of APIs handle settings of standby modes:
   CG_SetSTBYMode(), CG_GetSTBYMode( ), CG_SetExitStopModeFosc(),
   CG_GetExitStopModeFoscState(), CG_SetExitStopModeFs(),
   CG_GetExitStopModeFsState(), CG_SetPinStateInStopMode(),
   CG_GetPinStateInStopMode().

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

## 4.2.3 Function Documentation

### 4.2.3.1 CG_SetFgearLevel

Set the dividing level between clock fgear and fc.

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

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

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

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

**Return:**
None

### 4.2.3.2 CG_GetFgearLevel

Get the dividing level between fgear and fc.

**Prototype:**
CG_DivideLevel
CG_GetFgearLevel (void)

**Parameters:**
None

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

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

### 4.2.3.3 CG_SetPhiT0Src

Select the PhiT0(ΦT0) source between fperiph and fc or fperiph and fs.

**Prototype:**

void
CG_SetPhiT0Src(CG_PhiT0Src ***PhiT0Src***)

**Parameters:**
***PhiT0Src***: Select PhiT0 source.
This parameter can be one of the following values:
- **CG_PHIT0_SRC_FGEAR** means PhiT0 source is fgear.
- **CG_PHIT0_SRC_FC** means PhiT0 source is fc.
- **CG_PHIT0_SRC_FS** means PhiT0 source is fs.

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

**Return:**
None

### 4.2.3.4 CG_GetPhiT0Src

Get the PhiT0 (ΦT0) source.

**Prototype:**
CG_PhiT0Src
CG_GetPhiT0Src (void)

**Parameters:**
None

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

**Return:**
**CG_PHIT0_SRC_FGEAR** means PhiT0 source is fgear.
**CG_PHIT0_SRC_FC** means PhiT0 source is fc.
**CG_PHIT0_SRC_FS** means PhiT0 source is fs.

### 4.2.3.5 CG_SetPhiT0Level

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

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

**Parameters:**
***DividePhiT0FromFc***: divide level between PhiT0(ΦT0) and fc or PhiT0(ΦT0) and fs.
This parameter can be one of the following values:
- **CG_DIVIDE_1**: ΦT0 **=** fc or fs
- **CG_DIVIDE_2**: ΦT0 = fc/2 or fs/2
- **CG_DIVIDE_4**: ΦT0 = fc/4 or fs/2
- **CG_DIVIDE_8**: ΦT0 = fc/8 or fs/8
- **CG_DIVIDE_16**: ΦT0 = fc/16 or fs/16
- **CG_DIVIDE_32**: ΦT0 = fc/32 or fs/32
- **CG_DIVIDE_64**: ΦT0 = fc/64
- **CG_DIVIDE_128**: ΦT0 = fc/128

- **CG_DIVIDE_256**: ΦT0 = fc/256
- **CG_DIVIDE_512**: ΦT0 = fc/512

**Description:**
This function will set the dividing level of prescaler clock.

**Return:**
**SUCCESS** means the setting has been written to registers successfully.
**ERROR** means the setting has not been written to registers**.**

### 4.2.3.6 CG_GetPhiT0Level

Get the dividing level between clock ΦT0 and fc orΦT0 and fs.

**Prototype:**
CG_DivideLevel
CG_GetPhiT0Level(void)

**Parameters:**
None

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

**Return:**
Dividing level between clock ΦT0 and fc orΦT0 and fs, the value will be one of the following:
**CG_DIVIDE_1**: ΦT0 = fc or fs
**CG_DIVIDE_2**: ΦT0 = fc/2 or fs/2
**CG_DIVIDE_4**: ΦT0 = fc/4 or fs/4
**CG_DIVIDE_8**: ΦT0 = fc/8 or fs/8
**CG_DIVIDE_16**: ΦT0 = fc/16 or fs/16
**CG_DIVIDE_32**: ΦT0 = fc/32 or fs/32
**CG_DIVIDE_64**: ΦT0 = fc/64
**CG_DIVIDE_128** : ΦT0 = fc/128
**CG_DIVIDE_256** : ΦT0 = fc/256
**CG_DIVIDE_512** : ΦT0 = fc/512
**CG_DIVIDE_UNKNOWN :** invalid data is read.

### 4.2.3.7 CG_SetSCOUTSrc

Set the clock source of SCOUT output.

**Prototype:**
void
CG_SetSCOUTSrc(CG_SCOUTSrc ***Source***)

**Parameters:**
***Source***: select clock source of SCOUT.
This parameter can be one of the following values:
- **CG_SCOUT_SRC_FS**: SCOUT source is set to fs.
- **CG_SCOUT_SRC_HALF_FSYS**: SCOUT source is set to fsys/2.
- **CG_SCOUT_SRC_FSYS**: SCOUT source is set to fsys.

- **CG_SCOUT_SRC_PHIT0**: SCOUT source is set to ΦT0.

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

**Return:**
None

### 4.2.3.8 CG_GetSCOUTSrc

Get the clock source of SCOUT output.

**Prototype:**
SCOUTSrc
CG_GetSCOUTSrc(void)

**Parameters:**
None

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

**Return:**
The clock source of SCOUT output:
**CG_SCOUT_SRC_FS**: SCOUT source is fs
**CG_SCOUT_SRC_HALF_FSYS**: SCOUT source is set to fsys/2
**CG_SCOUT_SRC_FSYS**: SCOUT source is fsys
**CG_SCOUT_SRC_PHIT0**: SCOUT source is ΦT0

### 4.2.3.9 CG_SetWarmUpTime

Set the warm up time.

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

**Parameters:**
***Source***: select source of warm-up counter.
- **CG_WARM_UP_SRC_OSC1**: fosc1 is selected as timer source,
- **CG_WARM_UP_SRC_OSC2**: fosc2 is selected as timer source,
- **CG_WARM_UP_SRC_XT1**: fs is selected as timer source.

***Time:*** If ***Source*** is **CG_WARM_UP_SRC_OSC1** or **CG_WARM_UP_SRC_OSC2,** Time value range is 0U to 0x1000U.
If ***Source*** is **CG_WARM_UP_SRC_XT1,** Time value range is 0U to 0x4000U.

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

Example of calculating register value for warm-up time:

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

**Return**:
None.

### 4.2.3.10 CG_StartWarmUp

Start warm up timer.

**Prototype:**
void
CG_StartWarmUp (void)

**Parameters:**
None

**Description:**
This function will start the warm up timer.

**Return:**
None

### 4.2.3.11 CG_GetWarmUpState

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

**Prototype:**
WorkState
CG_GetWarmUpState (void)

**Parameters:**
None

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

Example of using warm-up timer:

```
CG_SetWarmUpTime(CG_WARM_UP_SRC_OSC1, 0x32);
/* start warm up */
CG_StartWarmUp();
/* check warm up is finished or not*/
While( CG_GetWarmUpState() == BUSY);
```

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

### 4.2.3.12 CG_SetLowOscSrc

Select the Low-speed clock.

**Prototype:**
void

CG_SetLowOscSrc(CG_LoscSrc ***Source***)

**Parameters:**
***Source***:
- CG_LOSC_OSC: Low-speed oscillator (fELOSC)
- CG_LOSC_CLK : Low-speed cllock (fELCLKIN)

**Description:**
Select the Low-speed clock or Low-speed oscillator

**Return:**
None

### 4.2.3.13 CG_SetExtHighClk

External high-speed clock input.

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

**Parameters:**
***NewState***:
- **ENABLE**: to enable the External high-speed clock input.
- **DISABLE**: to disable the External high-speed clock input.

**Description:**
This function will enable or disable external high-speed clock input.

**Return:**
None

### 4.2.3.14 CG_SetFosc

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

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

**Parameters:**
***Source***: select source for fosc.
- **CG_FOSC_OSC1**: fosc1 is selected,
- **CG_FOSC_OSC2**: fosc2 is selected.

***NewState***: select source for fosc.
- **ENABLE**: to enable the high-speed oscillator.
- **DISABLE**: to disable the high-speed oscillator.

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

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

### 4.2.3.15 CG_SetFoscSrc

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

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

**Parameters:**
***Source***: select source for fosc.
- **CG_FOSC_OSC1**: fosc1 is selected,
- **CG_FOSC_OSC2**: fosc2 is selected.

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

**Return:**
None

### 4.2.3.16 CG_GetFoscSrc

Get the source of the high-speed oscillator.

**Prototype:**
CG_FoscSrc
CG_GetFoscSrc(void)

**Parameters:**
None

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

**Return:**
The source of fosc
**CG_FOSC_OSC1**: fosc1 is selected.
**CG_FOSC_OSC2**: fosc2 is selected.

### 4.2.3.17 CG_GetFoscState

Get the state of the high-speed oscillator.

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

**Parameters:**
***Source***: select source for fosc.
- **CG_FOSC_OSC1**: fosc1 is selected,
- **CG_FOSC_OSC2**: fosc2 is selected.

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

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

### 4.2.3.18 CG_SetFs

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

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

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

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

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

### 4.2.3.19 CG_GetFsState

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

**Prototype:**
FunctionalState
CG_GetFsState (void)

**Parameters:**
None

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

**Return:**
The state of XT1
**ENABLE**: XT1 is enabled.
**DISABLE**: XT1 is disabled.

### 4.2.3.20 CG_SetPortM

Set portM for X1/X2 or general port.

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

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

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

**Return:**
None

### 4.2.3.21 CG_SetSTBYMode

Set the standby mode.

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

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

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

**Return:**
None

### 4.2.3.22 CG_GetSTBYMode

Get the standby mode.

**Prototype:**
CG_STBYMode
CG_GetSTBYMode (void)

**Parameters:**
None

**Description:**
This function will get the setting of standby mode.

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

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

### 4.2.3.23 CG_SetExitStopModeFosc

Enable or disable fosc after releasing stop mode

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

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

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

**Return:**
None

### 4.2.3.24 CG_GetExitStopModeFoscState

Get the state of X1 after releasing stop mode

**Prototype:**
FunctionalState
CG_GetExitStopModeFoscState (void)

**Parameters:**
None

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

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

### 4.2.3.25 CG_SetExitStopModeFs

Enable or disable XT1 after releasing stop mode

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

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

**Description:**
This function will enable or disable XT1 after releasing stop mode

**Return:**
None

### 4.2.3.26 CG_GetExitStopModeFsState

Get the state of XT1 after releasing stop mode

**Prototype:**
FunctionalState
CG_GetExitStopModeFsState (void)

**Parameters:**
None

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

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

### 4.2.3.27 CG_SetPinStateInStopMode

Specify the pin status in stop mode

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

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

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

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

**Return:**
None

### 4.2.3.28 CG_GetPinStateInStopMode

Get the pin status in stop mode

**Prototype:**
FunctionalState
CG_GetPinStateInStopMode (void)

**Parameters:**
None

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

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

### 4.2.3.29 CG_SetFsysSrc

Set the clock source of fsys.

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

**Parameters:**
***Source***: select the source of system clock (fsys)
This parameter can be one of the following values:
- **CG_FSYS_SRC_FGEAR:** source of fsys will be set to fgear
- **CG_FSYS_SRC_FS**: source of fsys will be set to fs.

**Description:**
This function will set the clock source of system clock (fsys).
If **CG_FSYS_SRC_FGEAR** is specified, the high-speed oscillator (X1) should be enabled earlier; if **CG_FSYS_SRC_FS** is specified, the low-speed oscillator (XT1) should be enabled earlier; otherwise, calling of this API will return **ERROR**.

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

### 4.2.3.30 CG_GetFsysSrc

Get the clock source of fsys

**Prototype:**
CG_FsysSrc
CG_GetFsysSrc (void)

**Parameters:**
None

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

**Return:**

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

### 4.2.3.31 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_INTLVD** : INTLVD
- **CG_INT_SRC_0** : INT0
- **CG_INT_SRC_1** : INT1
- **CG_INT_SRC_2** : INT2
- **CG_INT_SRC_3** : INT3

- **CG_INT_SRC_RTC**: RTC interrupt.

***ActiveState***: select the active state for release trigger.
This parameter can be one of the following values:
- **CG_INT_ACTIVE_STATE_L**: active on low level
- **CG_INT_ACTIVE_STATE_H**: active on high level
- **CG_INT_ACTIVE_STATE_FALLING**: active on falling edge
- **CG_INT_ACTIVE_STATE_RISGING**: active on rising edge
- **CG_INT_ACTIVE_STATE_BOTH_EDGES**: active on both edges

***NewState***: enable or disable this release trigger
This parameter can be one of the following values:
- **ENABLE**: clear standby mode when the interrupt occurs and the condition of active state is matched.
- **DISABLE:** do not clear standby mode even though the interrupt occurs and the condition of active state is matched.

**Description:**
This function will set the INT source for releasing standby mode.
For **CG_INT_SRC_INTLVD**, only "rising" state (**CG_INT_ACTIVE_STATE_RISING)** will be set to the register, no matter what the value of ***ActiveState*** is. For **CG_INT_SRC_RTC**, only "Falling " state (**CG_INT_ACTIVE_STATE_FALLING**) will be set to the register, no matter what the value of ***ActiveState*** is.

**Return:**
None

### 4.2.3.32 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_INTLVD CG_INT_SRC_0, CG_INT_SRC_1, CG_INT_SRC_2, CG_INT_SRC_3, 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

### 4.2.3.33 CG_ClearINTReq

Clear the INT request for releasing standby mode.

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

**Parameters:**
***INTSource***: select the release INT source.
This parameter can be one of the following values:
**CG_INT_SRC_INTLVD CG_INT_SRC_0, CG_INT_SRC_1, CG_INT_SRC_2, CG_INT_SRC_3, CG_INT_SRC_RTC.**

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

**Return:**
None

### 4.2.3.34 CG_SelExtHighClk

External high-speed clock select.

**Prototype**:
void
CG_SelExtHighClk (CG_EHClkSrc ***Source***)

**Parameters:**
***Source***:

- CG_EHCLK_OSCSEL: Use the clock selected by CGOSCCR<OSCSEL>
- CG_EHCLK_OSC2: Use the external high-speed clock input

**Description:**
External high-speed clock select.

**Return:**
None

#### 4.2.3.35 CG_GetResetFlag

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

**Prototype**:
CG_ResetFlag
CG_GetResetFlag(void)

**Parameters**:
None

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

**Return:**
Reset flag:
**ResetPin** (Bit 0) means reset from power-on.
**WDTReset** (Bit 2) means reset from WDT.
**DebugReset** (Bit 4) means reset from SYSRESETREQ.

### 4.2.4 Data Structure Description

#### 4.2.4.1 CG_ResetFlag

**Data Fields**:
uint32_t
***All*** specifies CG reset source.

**Bit Fields**:
uint32_t
***ResetPin*** 1 means reset from Reset pin.
uint32_t
***Reserved*** 1 means reserved.
uint32_t
***WDTReset*** 1 means reset from WDT.
uint32_t
***Reserved2*** 1 means reserved.
uint32_t
***DebugReset*** 1 means reset from SYSRESETREQ.
uint32_t
***Reserverd3*** 27 means reserved.

# 5. DSADC

## 5.1 Overview

TMPM061FWFG contains 3 units of a 24-bit Delta-Sigma Analog/Digital Converter (DSADC).
In the synchronous start function of DSADC, the following table is an assignment of a master unit and slave unit.

Master/slave assignment

| Master | Slave |
|---|---|
| Unit A | Unit B<br>Unit C |

A reference voltage circuit (BGR) used in the DSADC is shared with a temperature sensor and needs to set the control register (TEMPEN) of temperature sensor.

**Features**
DSADC has the following features:
・Conversion start
Conversion started by software
・Conversion modes
- Single conversion
- Repeat conversion
・Status flags
- Conversion result store flag
- Overrun flag
- Conversion end flag
- Conversion flag
・Conversion clock can be divided by below ratios.
fc/1, fc/2, fc/4, fc/8
・Conversion end interrupt output
・Conversion start correct function
・Synchronous start function for multiple units

When DSADC is used, provide pin treatments as follows:
・Do not connect VREFINx to a reference voltage.
・Connect AGNDREFx to DVSS.
・Connect a 1 μF capacitor to between VREFINx and AGNDREFx.

When DSADC is not used, below settings are required.
・Connect VREFINx to DVDD3.
・Connect AGNDREFx to DVSS.

When a temperature sensor is also not used, a reference voltage circuit requires below settings.
・Connect DSRVDD3 and SRVDD to DVDD3.
・Connect DSRVSS to DVSS.

# 5.2 API Functions

## 5.2.1 Function List

- void DSADC_SetClk(TSB_DSAD_TypeDef *DSADCx,uint32_t Clk)
- void DSADC_SWReset(TSB_DSAD_TypeDef *DSADCx)
- void DSADC_Start(TSB_DSAD_TypeDef *DSADCx)
- void DSADC_ChangeMode(TSB_DSAD_TypeDef *DSADCx,uint8_t SyncMode,uint8_t ConvMode)
- void DSADC_SetAmplifier(TSB_DSAD_TypeDef *DSADCx,uint32_t Amplifier)
- uint32_t DSADC_GetConvertResult(TSB_DSAD_TypeDef *DSADCx)
- void DSADC_Init(TSB_DSAD_TypeDef *DSADCx,DSADC_InitTypeDef * InitStruct);
- DSAD_status DSADC_GetStatus(TSB_DSAD_TypeDef *DSADCx)
- void DSADC_SetClkSupply(TSB_DSAD_TypeDef * DSADCx, FunctionalState NewState)

## 5.2.2 Detailed Description

Functions listed above can be divided into four parts:

1) ADC setting by DSADC_SetClk(),DSADC_ChangeMode (),DSADC_Init (), DSADC_SetClkSupply () and DSADC_SetAmplifier ().
2) ADC function start by DSADC_Start().
3) ADC state or data read functions by DSADC_GetConvertResult (), DSADC_GetStatus ().
4) DSADC_SWReset() handle other specified functions.

## 5.2.3 Function Documentation

### 5.2.3.1 DSADC_SetClk

Set AD conversion clock.

**Prototype:**

Void
DSADC_SetClk(TSB_DSAD_TypeDef *DSADCx,uint32_t Clk)

**Parameters:**

***DSADCx***: Select the DSADC channel.
This parameter can be one of the following values:

- **TSB_DSAD0:** DSADC module channel 0
- **TSB_DSAD1:** DSADC module channel 1
- **TSB_DSAD2:** DSADC module channel 2

***Clk:*** AD conversion clock selection.
This parameter can be one of the following values:

- **DSADC_FC_DIVIDE_LEVEL_1:** fc / 1
- **DSADC_FC_DIVIDE_LEVEL_2:** fc / 4
- **DSADC_FC_DIVIDE_LEVEL_4:** fc / 4
- **DSADC_FC_DIVIDE_LEVEL_8:** fc / 8

**Description:**

This function will set DSADC prescaler output by ***Clk***.

**Notes:**

During the analog to digital conversion, do not call this function to change the conversion clock setting.
Before calling this function, use **DSADC_GetStatus ()** to check DSADC conversion state is not **BUSY**.

**Return:**
None

### 5.2.3.2 DSADC_SWReset

Software reset DSADC

**Prototype:**
void
DSADC_SWReset(TSB_DSAD_TypeDef *DSADCx)

**Parameters:**
***DSADCx***: Select the DSADC channel.
This parameter can be one of the following values:
- **TSB_DSAD0:** DSADC module channel 0
- **TSB_DSAD1:** DSADC module channel 1
- **TSB_DSAD2:** DSADC module channel 2

**Description:**
This function will software reset DSADC.

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

**Return:**
None

### 5.2.3.3 DSADC_Start

Start DSADC function.

**Prototype:**
void
DSADC_Start(TSB_DSAD_TypeDef *DSADCx)

**Parameters:**
***DSADCx***: Select the DSADC channel.
This parameter can be one of the following values:
- **TSB_DSAD0:** DSADC module channel 0
- **TSB_DSAD1:** DSADC module channel 1
- **TSB_DSAD2:** DSADC module channel 2

**Description:**
This function will start DSADC conversion.

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

Repeat conversion mode

Please refer to the description of **DSADC_ChangeMode ()** for the details.

**NOTE: There is timing restrictions in setting DSADC, before starting AD conversion, please refer to part "Start Sequence" in chapter DSADC in datasheet.**

**Return:**
None

### 5.2.3.4 DSADC_ChangeMode

Change DSADC Synchronous mode and Conversion mode.

**Prototype:**
void
DSADC_ChangeMode(TSB_DSAD_TypeDef *DSADCx,uint8_t SyncMode,uint8_t ConvMode)

**Parameters:**
***DSADCx***: Select the DSADC channel.
This parameter can be one of the following values:
- **TSB_DSAD0:** DSADC module channel 0
- **TSB_DSAD1:** DSADC module channel 1
- **TSB_DSAD2:** DSADC module channel 2

***SyncMode***: Select the DSADC Synchronous mode.
This parameter can be one of the following values:
- **DSADC_A_SYNC_MODE**
- **DSADC_SYNC_MODE**

***ConvMode***: Select the ConvMode mode.
This parameter can be one of the following values:
- **DSADC_SINGLE_MODE**
- **DSADC_REPEAT_MODE**

**Description:**
This function will change DSADC Synchronous mode and Conversion mode.

**Return:**
None

### 5.2.3.5 DSADC_SetAmplifier

Set DSADC Amplifier.

**Prototype:**
void
DSADC_SetAmplifier(TSB_DSAD_TypeDef *DSADCx,uint32_t Amplifier)

**Parameters:**
***DSADCx***: Select the DSADC channel.
This parameter can be one of the following values:
- **TSB_DSAD0:** DSADC module channel 0
- **TSB_DSAD1:** DSADC module channel 1

- **TSB_DSAD2:** DSADC module channel 2

***Gain***: Amplifier gain setting for the specified Channel.
This parameter can be one of the following values:
- **DSADC_GAIN_1x:** Amplifier gain is 1
- **DSADC_GAIN_2x:** Amplifier gain is 2
- **DSADC_GAIN_4x:** Amplifier gain is 4
- **DSADC_GAIN_8x:** Amplifier gain is 8
- **DSADC_GAIN_16x:** Amplifier gain is 16

**Description:**
Set gains for the specified channel of DSADC, the input range will become 1/Gain.

**Return:**
None

### 5.2.3.6 DSADC_GetConvertResult

Get DSADC convert result.

**Prototype:**
uint32_t
DSADC_GetConvertResult(TSB_DSAD_TypeDef *DSADCx)

**Parameters:**
***DSADCx***: Select the DSADC channel.
This parameter can be one of the following values:
- **TSB_DSAD0:** DSADC module channel 0
- **TSB_DSAD1:** DSADC module channel 1
- **TSB_DSAD2:** DSADC module channel 2

**Description:**
This function will read DSADC register's result storage flag state, overrun state, and result value by ***DSADRES*** setting.

**Return:**
Result

### 5.2.3.7 DSADC_Init

Initialize the specified DSADC channel.

**Prototype:**
void
DSADC_Init(TSB_DSAD_TypeDef *DSADCx,DSADC_InitTypeDef * InitStruct)

**Parameters:**
***DSADCx***: Select the DSADC channel.
This parameter can be one of the following values:
- **TSB_DSAD0:** DSADC module channel 0
- **TSB_DSAD1:** DSADC module channel 1
- **TSB_DSAD2:** DSADC module channel 2

***InitStruct***: The structure containing basic DSADC configuration.
The parameter can be one of the following values:

- **InitStruct->Clk:** Set transfer clock.
- **InitStruct->BiasEn:** Enable Bias control.
- **InitStruct->ModulatorEn:** Enable Modulator control.
- **InitStruct->SyncMode:** Set synchronous mode.
- **InitStruct->Repeatmode:** Set Conversion mode.
- **InitStruct->Amplifier:** Set amplifier setting.
- **InitStruct->Offset:** Set conversion start correction.
- **InitStruct->CorrectEn:** Enable conversion start correction.

**Description:**
This function will initialize the specified DSADC channel.

**Return:**
None

### 5.2.3.8 DSADC_GetStatus

Indicate DSADC Convertor status and result.

**Prototype:**
DSAD_status
DSADC_GetStatus (TSB_DSAD_TypeDef *DSADCx)

**Parameters:**
***DSADCx***: Select the DSADC channel.
This parameter can be one of the following values:

- **TSB_DSAD0:** DSADC module channel 0
- **TSB_DSAD1:** DSADC module channel 1
- **TSB_DSAD2:** DSADC module channel 2

**Description:**
This function will read AD conversion busy/completion flag and start or not flag. This function is used to check whether AD conversion has completed or not and started or not.

**Return:**
A union with the state of AD conversion:
**retval.F_ResultStore** (Bit 0): '1' means AD conversion result is stored.
**retval.F_Overrun** (Bit 1): '1' means AD is Overrunning.
**retval. F_Convert** (Bit 2): '1' means top-priority AD is converting.
**retval. F_ConvertEnd** (Bit 3): 1' means normal AD conversion is complete.
**retval. ConversionResult**(Bit 8 to 31): Conversion result is stored.

### 5.2.3.9 DSADC_SetClkSupply

Set DSADC clock enable or disable.

**Prototype:**
void
DSADC_SetClkSupply(TSB_DSAD_TypeDef *DSADCx,FunctionalState NewState)

**Parameters:**
***InputChannel***: Analog input channel.
This parameter can be one of the following values:

- **DSADC_AN_02:** DSADC module channel 2
- **DSADC_AN_03:** DSADC module channel 3
- **DSADC_AN_04:** DSADC module channel 4
- **DSADC_AN_05:** DSADC module channel 5

***NewState***: Specify DSADC clock.
This parameter can be one of the following values:

- **ENABLE :** Enable DSADC clock
- **DISABLE:** Disable DSADC clock

**Description:**
This function will set DSADC clock enable or disable.

**Return:**
None

## 5.2.4 Data Structure Description

### 5.2.4.1 DSADC_InitTypeDef

**Data Fields for this struct:**

**Bit Fields**:

| | |
|---|---|
| uint8_t<br>***Clk*** (Bit 0) | Set transfer clock. |
| uint8_t<br>***BiasEn*** (Bit 1) | Enable Bias control flag (BIASEN).<br>'1' means Operation |
| uint8_t<br>***ModulatorEn*** (Bit 2) | Enable Modulator control flag (MODEN)<br>'1' means Control |
| uint8_t<br>***SyncMode*** (Bit 3) | Set synchronous mode flag (ADSYNC)<br>'1' means Synchronous operation |
| uint8_t<br>***Repeatmode*** (Bit 4) | Set conversion mode (REPEAT)<br>'1' means Repeat conversion |
| uint8_t<br>***Amplifier*** (Bit 5 to Bit 6) | set amplifier setting (DSGAIN [2:0]) |
| Uint16_t<br>***Offset*** (Bit 7) | Set conversion start correction time (OFFSET) |
| Uint8_t<br>***CorrectEn*** (Bit 8) | Enable conversion start correction (ADJ) |

### 5.2.4.2 DSAD_status

**Data Fields for this union:**

uint32_t
***All*** specifies AD conversion result.

**Bit Fields**:
uint32_t
***F_ResultStore*** (Bit 0) '1' means conversion result store.

uint32_t
***F_Overrun*** (Bit 1) '1' means overrun.

uint32_t
***F_Convert*** (Bit 2) '1' means conversion starts.

uint32_t
***F_ConvertEnd*** (Bit 3) '1' means conversion end.

uint32_t
***Reserved*** (Bit4 to Bit7) reserved.

uint32_t
***ConversionResult*** (Bit8 to Bit31) means AD result value.

# 6. FC

## 6.1 Overview

TMPM061 device contains flash memory.
For TMPM061FWFG, the size of flash is 128Kbyte.

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 (TMPM061), please refer to the MCU data sheet.

This driver is contained in \Libraries\TX00_Periph_Driver\src\tmpm061_fc.c with \Libraries\TX00_Periph_Driver \inc\tmpm061_fc.h containing the API definitions for use by applications.

## 6.2 API Functions

### 6.2.1 Function List

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

### 6.2.2 Detailed Description

Functions listed above can be divided into four parts:
1) The security function restricts flash ROM data readout and debugging.
   FC_SetSecurityBit(), FC_GetSecurityBit().
2) The functions get the automatic operation status and each block protection status:
   FC_GetBusyState(), FC_GetBlockProtectState(),
3) The functions change the protection status of each block:
   FC_ProgramBlockProtectState(), FC_EraseBlockProtectState().
4) Use automatic operation command to write or erase the content of flash.
   FC_WritePage(), FC_EraseBlock(), FC_EraseChip().

### 6.2.3 Function Documentation

#### 6.2.3.1 FC_SetSecurityBit

Set the value of SECBIT register.

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

**Parameters:**
***NewState***: Select the state of SECBIT register.
This parameter can be one of the following values:
- **DISABLE**: Protection function is not available.
- **ENABLE**: Protection function is available.

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

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

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

**Return:**
None

### 6.2.3.2 FC_GetSecurityBit

Get the value of SECBIT register.

**Prototype:**
FunctionalState
FC_GetSecurityBit(void)

**Parameters:**
None

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

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

### 6.2.3.3 FC_GetBusyState

Get the status of the flash auto operation.

**Prototype:**
WorkState
FC_GetBusyState (void)

**Parameters:**

None

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

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

### 6.2.3.4 FC_GetBlockProtectState

Get the block protection status.

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

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

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

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

### 6.2.3.5 FC_ProgramBlockProtectState

Program the protection bits

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

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

**Description:**

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

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

### 6.2.3.6 FC_EraseBlockProtectState

Erase the protection bits

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

**Parameters**:
***BlockGroup***:The flash block group
- **FC_BLOCK_GROUP_0** for block 0, block 1, block 2, block 3

**Description:**
This API is used to erase the protection bits (clear them to”0”) so that the corresponding blocks will not be protected.
One group of protection bits will be erased when this API is executed each time.

**Return:**
Result of the operation to erase the protection bits
**FC_SUCCESS**: Erase the protection bits successfully.
**FC_ERROR_OVER_TIME**: Erase block protection bits operation over time error.

### 6.2.3.7 FC_WritePage

Write data to the specified page

**Prototype:**
FC_Result
FC_WritePage(uint32_t ***PageAddr***, uint32_t * ***Data***)

**Parameters**:
***PageAddr***:The page start address
***Data***: The pointer to data buffer to be written into the page. The data size should be 128Byte.

**Description:**
This API is used to write data to specified page.
The TMPM061 contains 32 words in a page. The flash can only be written page by page.
The automatic page programming is allowed only once for a page already erased. No programming can be performed twice or more time irrespective of data value whether it is “1” or “0”.
***Note**: An attempt to rewrite a page two or more times without erasing the content can cause damages to the device.

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

### 6.2.3.8 FC_EraseBlock

Erase the content of specified block.

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

**Parameters**:
***BlockAddr***: The block start address.

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

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

### 6.2.3.9 FC_EraseChip

Erase the content of the entire chip.

**Prototype:**
FC_Result
FC_EraseChip(void)

**Parameters**:
None

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

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

# 7. LCD

## 7.1 Overview

The TMPM061FWFG has a driver and control circuit to directly drive a liquid crystal display (LCD) device.

The pins to be connected to the LCD are as follows:
Segment output pins : 40 pins (SEG39 to SEG0)
Common output pins : 4 pins (COM3 to COM0)

In addition, the VLC pin is provided as a drive power supply pin, and the LV1 and LV2 pins are provided as external bleeder resistor connection pins.

**Note**: When the static, 1/3 or 1/2 duties are selected, unused common output pins should be opened. (It outputs bias voltage)

The LCD driver can directly drive the following five types of LCD:

1. 1/4 duty (1/3 bias) LCD Max. 160 pixels (8 segments × 20 digits)
2. 1/3 duty (1/3 bias) LCD Max. 120 pixels (8 segments × 15 digits)
3. 1/3 duty (1/2 bias) LCD Max. 120 pixels (8 segments × 15 digits)
4. 1/2 duty (1/2 bias) LCD Max. 80 pixels (8 segments × 10 digits)
5. Static LCD Max. 40 pixels (8 segments × 5 digits)

The LCD API provides a set of functions for using the TMPM061 LCD modules. It includes LCD bias set, duty set, write data to display buffer, and other setting relative with LCD display.

This driver is contained in TX00_Periph_Driver\src\tmpm061_lcd.c, with TX00_Periph_Driver\inc\tmpm061_lcd.h containing the API definitions for use by applications.

## 7.2 API Functions

### 7.2.1 Function List

- void LCD_Enable(void)
- void LCD_Disable(void)
- void LCD_SetDisplay(FunctionalState ***NewState***)
- void LCD_SetDutyBias(LCD_DutyBias ***DutyBias***)
- void LCD_SetBaseFreq(LCD_BaseFreq ***Freq***)
- void LCD_SetLowBleederTime(LCD_LowResistorConnectionTime ***Time***)
- void LCD_SetInternalBleeder(LCD_BleederResistorValue ***Value***)
- void LCD_SetBleederSource(LCD_BleederResistorSource ***Source***)
- void LCD_WriteBuf(LCD_BufIndex ***TargetBuf***, uint8_t ***Data***)

### 7.2.2 Detailed Description

Functions listed above can be divided into three parts:

1) LCD setting by LCD_SetDisplay(), LCD_SetDutyBias(), LCD_SetBaseFreq(), LCD_SetLowBleederTime(), LCD_SetInternalBleeder(), LCD_SetBleederSource().
2) LCD module work or stop by LCD_Enable(), LCD_Disable().
3) Write display data to LCD buffer by LCD_WriteBuf().

## 7.2.3 Function Documentation

### 7.2.3.1 LCD_Enable

Enable operation for LCD module.

**Prototype:**
void
LCD_Enable(void)

**Parameters:**
None

**Description:**
This function will enable the operation for LCD module.

**Return:**
None

### 7.2.3.2 LCD_Disable

Disable operation for LCD module.

**Prototype:**
void
LCD_Disable(void)

**Parameters:**
None

**Description:**
This function will disable operation for LCD module

**Return:**
None

### 7.2.3.3 LCD_SetDisplay

Enable or disable the LCD display while LCD module is enabled.

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

**Parameters:**
***NewState***: Enable or disable the LCD display
This parameter can be one of the following values:
- **ENABLE:** Enable LCD display
- **DISABLE:** Blank LCD display

**Description:**
This function will enable or disable the LCD display while LCD module is enabled. When it is disabled, the LCD display will be blank.

**Return:**
None

### 7.2.3.4 LCD_SetDutyBias

Set duty and bias for LCD driving method.

**Prototype:**
void
LCD_SetDutyBias(LCD_DutyBias ***DutyBias***)

**Parameters:**
***DutyBias***: Duty and bias setting value for LCD driving method
This parameter can be one of the following values:
- **LCD_DUTY4_BIAS3 :** 1/4 duty, 1/3 bias
- **LCD_DUTY3_BIAS3 :** 1/3 duty, 1/3 bias
- **LCD_DUTY3_BIAS2 :** 1/3 duty, 1/2 bias
- **LCD_DUTY2_BIAS2 :** 1/2 duty, 1/2 bias
- **LCD_STATIC :** static

**Description:**
This function will set duty and bias for LCD driving method which is depended by the parameter of the actual LCD glass.

**Return:**
None

### 7.2.3.5 LCD_SetBaseFreq

Set base frequency which is relative with fsys(gear clock frequency).

**Prototype:**
void
LCD_SetBaseFreq(LCD_BaseFreq ***Freq***)

**Parameters:**
***Freq***: Base frequency selection
This parameter can be one of the following values:
- **LCD_FSYS_DIVIDE_2_POWER_18 :** F_base = fsys / (2^18)
- **LCD_FSYS_DIVIDE_2_POWER_17 :** F_base = fsys / (2^17)
- **LCD_FSYS_DIVIDE_2_POWER_16 :** F_base = fsys / (2^16)
- **LCD_FSYS_DIVIDE_2_POWER_15 :** F_base = fsys / (2^15)
- **LCD_FSYS_DIVIDE_2_POWER_14 :** F_base = fsys / (2^14)
- **LCD_FS_DIVIDE_2_POWER_9 :** F_base = fs / (2^9)
- **LCD_FS_DIVIDE_2_POWER_8 :** F_base = fs / (2^8)

**Description:**
This function will set base frequency which is relative with fsys(gear clock frequency). it will decide the “frame frequency”

The relationship of this base frequency and the frame frequency is listed below:

| SLF | Base frequency [Hz] | Frame frequency [Hz] | | | |
|---|---|---|---|---|---|
| | | 1/4 Duty | 1/3 Duty | 1/2 Duty | Static |
| 0000 | fsys / $2^{18}$ | fsys / $2^{18}$ | ( 4 / 3 ) x fsys / $2^{18}$ | ( 4 / 2 ) x fsys / $2^{18}$ | fsys / $2^{18}$ |
| | (fsys = 16 MHz) | 61 | 81 | 122 | 61 |
| 0001 | fsys / $2^{17}$ | fsys / $2^{17}$ | ( 4 / 3 ) x fsys / $2^{17}$ | ( 4 / 2 ) x fsys / $2^{17}$ | fsys / $2^{17}$ |
| | (fsys = 16 MHz) | 122 | 163 | 244 | 122 |
| | (fsys= 8 MHz) | 61 | 81 | 122 | 61 |
| 0010 | fsys / $2^{16}$ | fsys / $2^{16}$ | ( 4 / 3 ) x fsys / $2^{16}$ | ( 4 / 2 ) x fsys / $2^{16}$ | fsys / $2^{16}$ |
| | (fsys = 8 MHz) | 122 | 163 | 244 | 122 |
| | (fsys = 4 MHz) | 61 | 81 | 122 | 61 |
| 0011 | fsys / $2^{15}$ | fsys / $2^{15}$ | ( 4 / 3 ) x fsys / $2^{15}$ | ( 4 / 2 ) x fsys / $2^{15}$ | fsys / $2^{15}$ |
| | (fsys = 4 MHz) | 122 | 163 | 244 | 122 |
| | (fsys = 2 MHz) | 61 | 81 | 122 | 61 |
| 0100 | fsys / $2^{14}$ | fsys / $2^{14}$ | ( 4 / 3 ) x fsys / $2^{14}$ | ( 4 / 2 ) x fsys / $2^{14}$ | fsys / $2^{14}$ |
| | (fsys = 2 MHz) | 122 | 163 | 244 | 122 |
| | (fsys = 1 MHz) | 61 | 81 | 122 | 61 |
| 1000 | fs / $2^{9}$ | fs / $2^{9}$ | ( 4 / 3 ) x fs / $2^{9}$ | ( 4 / 2 ) x fs / $2^{9}$ | fs / $2^{9}$ |
| | (fs = 32.768 kHz) | 64 | 85 | 128 | 64 |
| 1001 | fs / $2^{8}$ | fs / $2^{8}$ | ( 4 / 3 ) x fs / $2^{8}$ | ( 4 / 2 ) x fs / $2^{8}$ | fs / $2^{8}$ |
| | (fs = 32.768 kHz) | 128 | 171 | 256 | 128 |

**Return:**
None

### 7.2.3.6 LCD_SetLowBleederTime

Set low internal bleeder resistor connection time.

**Prototype:**
void
LCD_SetLowBleederTime(LCD_LowResistorConnectionTime ***Time***)

**Parameters:**
***Time***: Low internal bleeder resistor connection time selection.
The parameter can be one of the following values:

- **LCD_NOT_CONNECT :** Low internal bleeder resistor is never connected
- **LCD_2_POWER_7_DEVIDE_BASE_FREQ :** (2^7)/F_base
- **LCD_2_POWER_6_DEVIDE_BASE_FREQ :** (2^6)/F_base
- **LCD_2_POWER_5_DEVIDE_BASE_FREQ :** (2^5)/F_base
- **LCD_2_POWER_4_DEVIDE_BASE_FREQ :** (2^4)/F_base
- **LCD_2_POWER_3_DEVIDE_BASE_FREQ :** (2^3)/F_base
- **LCD_ALWAYS_CONNECT :** Low internal bleeder resistor is always connected

**Description:**
This function will set low internal bleeder resistor connection time

Internal bleeder resistor is comprised of two parts: high resistor and low resistor, they are connected in parallel for each bias voltage.

The low bleeder resistor is controlled with an analog switch, the time to turn on the low bleeder resistor can be adjusted by LCDCR2<LRSE>(this function will set this value).

While the analog switch is turned on, the low resistor is connected in parallel to the high resistor , this reduces the total amount of resistance, allowing the drive capability of the LCD driver to be increased. Typically, the longer the period of connecting the low resistor, the higher the drive capability of the LCD panel, but the higher the power consumption.

Conversely, the shorter the period of connecting the low resistor, the lower the drive capability, and the lower the power consumption.

Insufficient drive capability will cause adverse effects on the LCD display, such as blurring.

Choose the optimum drive capability for the LCD panel to be used.

**Return:**
None

### 7.2.3.7 LCD_SetInternalBleeder

Set high internal bleeder resistor value.

**Prototype:**
void
LCD_SetInternalBleeder(LCD_BleederResistorValue ***Value***)

**Parameters:**
***Value***: high internal bleeder resistor value selection.
The parameter can be one of the following values:
- **LCD_BLEEDER_RESISTOR_200K** : 200K ohm
- **LCD_BLEEDER_RESISTOR_500K** : 500K ohm

**Description:**
Internal bleeder resistor is comprised of two parts: high resistor and low resistor, they are connected in parallel for each bias voltage.

This function will set the value for the high internal bleeder resistor as 200K ohm or 500K ohm.

**Return:**
None

### 7.2.3.8 LCD_SetBleederSource

Set bleeder resistor source (Internal or external ).

**Prototype:**
void
LCD_SetBleederSource(LCD_BleederResistorSource ***Source***)

**Parameters:**

***Source***: Internal/external bleeder resistor switching control
This parameter can be one of the following values:

- **LCD_BLEEDER_RESISTOR_EXTERNAL:** Use external bleeder resistor
- **LCD_BLEEDER_RESISTOR_INTERNAL :** Use internal bleeder resistor

**Description:**
This function will set bleeder resistor source (Internal or external ).

**Return:**
None

### 7.2.3.9 LCD_WriteBuf

Write data to LCD buffer, 1 Seg or 2 Seg a time for a LCD buffer register..

**Prototype:**
void
LCD_WriteBuf(LCD_BufIndex ***TargetBuf***, uint8_t ***Data***)

**Parameters:**
***TargetBuf***: Index of target LCD buffer.
The parameter can be one of the following values:
Note: refer to members in enum type: LCD_BufIndex in tmpm061_lcd.h
**For one Seg a time:**

- **LCD_BUF_SEG00, LCD_BUF_SEG01, LCD_BUF_SEG02**
- **.....**
- **LCD_BUF_SEG37, LCD_BUF_SEG38, LCD_BUF_SEG39**

**For two Seg a time:**

- **LCD_BUF_SEG0100, LCD_BUF_SEG0302, LCD_BUF_SEG0504**
- **.....**
- **LCD_BUF_SEG3534, LCD_BUF_SEG3736, LCD_BUF_SEG3938**

***Data***: the data will be wrote to LCD buffer. When ***TargetBuf*** is used for two Seg a time, the high nibble is for odd Seg, the low nibble is for even Seg.

**Description:**
This function will write data to LCD buffer, 1 Seg or 2 Seg a time for a LCD data buffer register.
For example:
Called as LCD_WriteBuf(**LCD_BUF_SEG35, 0x04**) will write 0x04 to data buffer for Seg35. Called as LCD_WriteBuf(**LCD_BUF_SEG3534, 0x56**) will write 0x05 to data buffer for Seg35 and 0x06 to data buffer for Seg34.

**Return:**
None

## 7.2.4 Data Structure Description

None

# 8. LVD

## 8.1 Overview

TMPM061 has a voltage detection circuit, which can detect decreasing/increasing voltage for the supply voltage DVDD3 and generate a NMI named INTLVD.

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/TX00_Periph_Driver/src/tmpm061_lvd.c, with /Libraries/TX00_Periph_Driver/inc/tmpm061_lvd.h containing the macros, data types, structures and API definitions for use by applications.

## 8.2 API Functions

### 8.2.1 Function List

- void LVD_Enable(void)
- void LVD_Disable(void)
- void LVD_SetVoltage(uint32_t ***Voltage***)
- LVD_SupplyStatus LVD_GetStatus(void)
- void LVD_SetINTOutput(FunctionalState ***NewState***)
- void LVD_SetINTCondition(uint32_t ***Condition***)

### 8.2.2 Detailed Description

Functions listed above can be divided into two parts:

1) Configure of LVD are handled by LVD_Enable(), LVD_Disable(), LVD_SetVoltage(); LVD_SetINTOutput(), LVD_SetINTCondition().
2) Get the power supply voltage detection status info by LVD_GetStatus() .

### 8.2.3 Function Documentation

#### 8.2.3.1 LVD_Enable

Enable the operation of voltage detection.

**Prototype:**
void
LVD_Enable(void)

**Parameters:**
***None.***

**Description:**
This function will enable the voltage detection operation.

**Return:**
None.

### 8.2.3.2 LVD_Disable

Disable the operation of voltage detection .

**Prototype:**
void
LVD_Disable(void)

**Parameters:**
***None.***

**Description:**
This function will disable the voltage detection operation.

**Return:**
None.

### 8.2.3.3 LVD_SetVoltage

Set threshold voltage for detection.

**Prototype:**
void
LVD_SetVoltage(uint32_t ***Voltage***)

**Parameters:**
***Voltage*** is the threshold voltage for detection.
This parameter can be one of the following values:

- **LVD_DETECT_VOLTAGE_280:** Voltage detection ragne is 2.80 ± 0.2V.
- **LVD_DETECT_VOLTAGE_285:** Voltage detection ragne is 2.85 ± 0.2V.
- **LVD_DETECT_VOLTAGE_290:** Voltage detection ragne is 2.90 ± 0.2V.
- **LVD_DETECT_VOLTAGE_295:** Voltage detection ragne is 2.95 ± 0.2V.
- **LVD_DETECT_VOLTAGE_300:** Voltage detection ragne is 3.00 ± 0.2V.
- **LVD_DETECT_VOLTAGE_305:** Voltage detection ragne is 3.05 ± 0.2V.
- **LVD_DETECT_VOLTAGE_310:** Voltage detection ragne is 3.10 ± 0.2V.
- **LVD_DETECT_VOLTAGE_315:** Voltage detection ragne is 3.15 ± 0.2V.

**Description:**
Set threshold voltage for detection.

**Return:**
None.

### 8.2.3.4 LVD_GetStatus

Get current low voltage detection status.

**Prototype:**
LVD_SupplyStatus
LVD_GetStatus(void)

**Parameters:**
***None.***

**Description:**

Get current low voltage detection status.

**Return:**
***LVD_SUPPLY_HIGH***: Power supply voltage is higher than the detection voltage specified by function LVD_SetVoltage().
***LVD_SUPPLY_LOW***: Power supply voltage is lower than the detection voltage specified by function LVD_SetVoltage().

### 8.2.3.5 LVD_SetINTOutput

Enable or disable LVD interrupt output when LVD module is enabled.

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

**Parameters:**
***NewState***: To enable or disable the output for INTLVD
This parameter can be one of the following values:
**ENABLE** or **DISABLE**

**Description:**
This function will enable or disable LVD interrupt output when LVD module is enabledn.

**Return:**
None.

### 8.2.3.6 LVD_SetINTCondition

Set the INTLVD generation condition.

**Prototype:**
void
LVD_SetINTCondition(uint32_t ***Condition***)

**Parameters:**
***Condition***: Select INTLVD generation condition when supply power is varying
This parameter can be one of the following values:

- **LVD_INTSEL_LOWER :** When the supply power is decreasing and lower than setting voltage.
- **LVD_INTSEL_LOWER_UPPER**: When the supply power is decreasing and lower than setting voltage, or, after that, when the supply power is rising and higher than setting voltage.

**Description:**
This function will set the INTLVD generation condition

**Return:**
None.

## 8.2.4 Data Structure Description

None

# 9. RTC

## 9.1 Overview

The Real Time Clock (RTC) in the TMPM061 have 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 error correction

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/TX00_Periph_Driver/src/tmpm061_rtc.c, with /Libraries/ TX00_Periph_Driver/inc/tmpm061_rtc.h containing the macros, data types, structures and API definitions for use by applications.

## 9.2 API Functions

### 9.2.1 Function List

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

- void RTC_ResetClockSec(void)
- RTC_ReqState RTC_GetResetClockSecReq(void)
- void RTC_ResetAlarm(void)
- void RTC_SetDateValue(RTC_DateTypeDef * ***DateStruct***)
- void RTC_GetDateValue(RTC_DateTypeDef * ***DateStruct***)
- void RTC_SetTimeValue(RTC_TimeTypeDef * ***TimeStruct***)
- void RTC_GetTimeValue(RTC_TimeTypeDef * ***TimeStruct***)
- void RTC_SetAlarmValue(RTC_AlarmTypeDef * ***AlarmStruct***)
- void RTC_GetAlarmValue(RTC_AlarmTypeDef * ***AlarmStruct***)
- void RTC_SetCorrectionProtect(FunctionalState ***WriteEnableState***)
- void RTC_EnableErrCorrection(void)
- void RTC_DisableErrCorrection(void)
- void RTC_ConfigErrCorrection(RTC_AdjustInterval ***Interval***, int8_t ***Value***)

## 9.2.2 Detailed Description

Functions listed above can be divided into five parts:

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

## 9.2.3 Function Documentation

### 9.2.3.1 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.
After calling this function, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None.

### 9.2.3.2 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

### 9.2.3.3 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**.
After calling this function with the parameter ***NewMode* = RTC_CLOCK_MODE**, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None

### 9.2.3.4 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

### 9.2.3.5 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.

### 9.2.3.6 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**.

After calling this function with the parameter ***NewMode* = RTC_CLOCK_MODE**, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

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

**Return:**
None

### 9.2.3.7 RTC_SetHour12

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

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

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

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

**Description:**
This function will set new hour value and AM/PM mode for RTC clock when ***NewMode*** is **RTC_CLOCK_MODE**, and set new hour value and AM/PM mode for RTC alarm when ***NewMode*** is **RTC_ALARM_MODE**.
After calling this function with the parameter ***NewMode* = RTC_CLOCK_MODE**, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

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

**Return:**
None

### 9.2.3.8 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

### 9.2.3.9 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**.
After calling this function with the parameter ***NewMode* = RTC_CLOCK_MODE**, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None

### 9.2.3.10 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

### 9.2.3.11 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**.
After calling this function with the parameter ***NewMode*** **= RTC_CLOCK_MODE**, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None

### 9.2.3.12 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

### 9.2.3.13 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.
After calling this function, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None

### 9.2.3.14 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

### 9.2.3.15 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.
After calling this function, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None

### 9.2.3.16 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

### 9.2.3.17 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**.
After calling this function, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

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

**Return:**
None

### 9.2.3.18 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.

### 9.2.3.19 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.
After calling this function with the parameter ***NewMode* = RTC_CLOCK_MODE**, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None

### 9.2.3.20 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.

### 9.2.3.21 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".
After calling this function, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None

### 9.2.3.22 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.

### 9.2.3.23 RTC_EnableClock

Enable RTC clock function.

**Prototype:**
void
RTC_EnableClock(void)

**Parameters:**
None

**Description:**
This function will enable clock function.
After calling this function, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None

### 9.2.3.24 RTC_DisableClock

Disable RTC clock function.

**Prototype:**
void
RTC_DisableClock(void)

**Parameters:**
None

**Description:**
This function will disable clock function.
After calling this function, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None

### 9.2.3.25 RTC_EnableAlarm

Enable RTC alarm function.

**Prototype:**
void
RTC_EnableAlarm(void)

**Parameters:**
None

**Description:**
This function will enable alarm function.

**Return:**
None

### 9.2.3.26 RTC_DisableAlarm

Disable RTC alarm function.

**Prototype:**
void
RTC_DisableAlarm(void)

**Parameters:**
None

**Description:**
This function will disable alarm function.

**Return:**
None

### 9.2.3.27 RTC_SetRTCINT

Enable or disable INTRTC.

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

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

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

**Return:**
None

### 9.2.3.28 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

**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 or 16)

**Return:**
None

### 9.2.3.29 RTC_ResetClockSec

Reset RTC clock second counter.

**Prototype:**
void
RTC_ResetClockSec(void)

**Parameters:**
None

**Description:**
This function will reset sec counter.
After calling this function, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None

### 9.2.3.30 RTC_GetResetClockSecReq

Get reset RTC clock second counter request state.

**Prototype:**
RTC_ReqState
RTC_GetResetClockSecReq(void)

**Parameters:**
None

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

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

### 9.2.3.31 RTC_ResetAlarm

Reset RTC alarm.

**Prototype:**
void
RTC_ResetAlarm(void)

**Parameters:**
None

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

**Return:**
None

### 9.2.3.32 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.
After calling this function, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None

### 9.2.3.33 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

### 9.2.3.34 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.
After calling this function, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None

### 9.2.3.35 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

### 9.2.3.36 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

### 9.2.3.37 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

### 9.2.3.38 RTC_SetCorrectionProtect

Enable or disable write to Clock correction function register.

**Prototype:**
void
RTC_SetCorrectionProtect (FunctionalState WriteEnableState)

**Parameters:**

***WriteEnableState***: New state of write.
- **ENABLE**: Enable write function.
- **DISABLE**: Disable write function.

**Description:**
Enable or disable write to Clock correction function register.

**Notes:**
In the initial state, write function is enabled.
When write is disabled, RTCADJCTL and RTCADJDAT will be written disabled.

**Return:**

None

### 9.2.3.39 RTC_EnableErrCorrection

Enable clock error correction function.

**Prototype:**
void
RTC_EnableErrCorrection(void)

**Parameters:**

None

**Description:**
This function will enable clock error correction function.
After calling this function, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None

### 9.2.3.40 RTC_DisableErrCorrection

Disable clock error correction function.

**Prototype:**
void
RTC_DisableErrCorrection(void)

**Parameters:**

None

**Description:**
This function will disable clock error correction function.
After calling this function, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None

### 9.2.3.41 RTC_ConfigErrCorrection

Configure clock error correction function.

**Prototype:**
void
RTC_ConfigErrCorrection(RTC_AdjustInterval ***Interval***,
int8_t ***Value***)

**Parameters:**

***Interval***: Adjustment interval,
- **RTC_ADJUST_20_SEC**: corrections by every 20sec.
- **RTC_ADJUST_30_SEC**: corrections by every 30sec.

***Value***: Adjustment value, this parameter can be -128 to +127.

**Description:**
This function will configure clock error correction function.

**Notes:**
***Value*** is an 8-bit signed number. The lowest bit of ***Value*** has no effect.
For example: ***Value*** = +101 is actually the same as ***Value*** = +100.

If ***Interval*** = RTC_ADJUST_20_SEC, please use the follow formula to calculate the
shifting amount.
Shifting amount = 24 * 3600 / 20 * (Value / 32768) ≈ 0.132 * Value (sec / day)

If ***Interval*** = RTC_ADJUST_30_SEC, please use the follow formula to calculate the
shifting amount.
Shifting amount = 24 * 3600 / 30 * (Value / 32768) ≈ 0.088 * Value (sec / day)

After calling this function, **RTC_UpdateData()** should be called to update RTC registers, then wait until **RTC_GetAccessStatus()** returns DONE.

**Return:**
None

## 9.2.4 Data Structure Description

### 9.2.4.1 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.

#### 9.2.4.2 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.

#### 9.2.4.3 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.

# 10. SBI

## 10.1 Overview

This device contains some Serial Bus Interface channels. Each channel can operate in I2C bus mode with multi-master capability.
In I2C bus mode, the SBI is connected to external devices via SCL and SDA.
Data can be transferred in free data format by the SBI channels. In free data format, data is always sent by master-transmitter and received by slave-receiver.

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

All driver APIs are contained in /Libraries/TX00_Periph_Driver/src/tmpm061_sbi.c(*), with /Libraries/TX00_Periph_Driver/inc/tmpm061_sbi.h containing the macros, data types, structures and API definitions for use by applications.

## 10.2 API Functions

### 10.2.1 Function List

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

### 10.2.2 Detailed Description

Functions listed above can be divided into four parts:

1) Configure and control the common functions of each SBI channel are handled by SBI_Enable(), SBI_Disable(), SBI_SetI2CACK(), SBI_SetI2CBitNum(), and SBI_InitI2C().
2) Transfer control of each SBI channel is handled by SBI_ClearI2CINTReq(), SBI_GenerateI2Cstart(), SBI_GenerateI2Cstop(),SBI_SetSendData(), SBI_GetReceiveData().
3) The status indication of each SBI channel is handled by SBI_GetI2CState().

4) SBI_SWReset(), SBI_SetIdleMode() and SBI_EnableI2CfreeDataMode() handle other specified functions.

## 10.2.3 Function Documentation

**Note**: in all of the following APIs, parameter "TSB_SBI_TypeDef* ***SBIx***" can be one of the following values: **TSB_SBI0**.

### 10.2.3.1 SBI_Enable

Enable the specified SBI channel.

**Prototype:**
void
SBI_Enable(TSB_SBI_TypeDef* ***SBIx***)

**Parameters:**
***SBIx*** is the specified SBI channel.

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

**Return:**
None

### 10.2.3.2 SBI_Disable

Disable the specified SBI channel.

**Prototype:**
void
SBI_Disable(TSB_SBI_TypeDef* ***SBIx***)

**Parameters:**
***SBIx*** is the specified SBI channel.

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

**Return:**
None

### 10.2.3.3 SBI_SetI2CACK

Enable or disable the generation of ACK clock.

**Prototype:**
void
SBI_SetI2CACK(TSB_SBI_TypeDef* ***SBIx***,
FunctionalState ***NewState***)

**Parameters:**
***SBIx*** is the specified SBI channel.
***NewState*** sets the generation of ACK clock, which can be:
- **ENABLE** for generating of ACK clock

- **DISABLE** for no ACK clock

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

**Return:**
None

### 10.2.3.4 SBI_InitI2C

Initialize the specified SBI channel in I2C mode.

**Prototype:**
void
SBI_InitI2C(TSB_SBI_TypeDef* ***SBIx***,
SBI_InitI2CTypeDef* ***InitI2CStruct***)

**Parameters:**
***SBIx*** is the specified SBI channel.
***InitI2CStruct*** is the structure containing SBI configuration (refer to Data Structure Description for details).

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

**Return:**
None

### 10.2.3.5 SBI_SetI2CBitNum

Specify the number of bits per transfer.

**Prototype:**
void
SBI_SetI2CBitNum(TSB_SBI_TypeDef* ***SBIx***,
uint32_t ***I2CBitNum***)

**Parameters:**
***SBIx*** is the specified SBI channel.
***I2CBitNum*** specifies the number of bits per transfer, max. 8.
This parameter can be one of the following values:

- **SBI_I2C_DATA_LEN_8**, which means that the data length number of bits per transfer is 8;
- **SBI_I2C_DATA_LEN_1**, which means that the data length number of bits per transfer is 1;
- **SBI_I2C_DATA_LEN_2**, which means that the data length number of bits per transfer is 2;
- **SBI_I2C_DATA_LEN_3**, which means that the data length number of bits per transfer is 3;
- **SBI_I2C_DATA_LEN_4**, which means that the data length number of bits per transfer is 4;

- **SBI_I2C_DATA_LEN_5**, which means that the data length number of bits per transfer is 5;
- **SBI_I2C_DATA_LEN_6**, which means that the data length number of bits per transfer is 6;
- **SBI_I2C_DATA_LEN_7**, which means that the data length number of bits per transfer is 7.

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

**Return:**
None

### 10.2.3.6 SBI_SWReset

Reset the state of the specified SBI channel.

**Prototype:**
void
SBI_SWReset(TSB_SBI_TypeDef* ***SBIx***)

**Parameters:**
***SBIx*** is the specified SBI channel.

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

**Return:**
None

### 10.2.3.7 SBI_ClearI2CINTReq

Clear SBI interrupt request in I2C bus mode.

**Prototype:**
void
SBI_ClearI2CINTReq(TSB_SBI_TypeDef* ***SBIx***)

**Parameters:**
***SBIx*** is the specified SBI channel.

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

**Return:**
None

### 10.2.3.8 SBI_GenerateI2CStart

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

**Prototype:**
void
SBI_GenerateI2CStart(TSB_SBI_TypeDef* ***SBIx***)

**Parameters:**
***SBIx*** is the specified SBI channel.

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

**Return:**
None

### 10.2.3.9 SBI_GenerateI2CStop

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

**Prototype:**
void
SBI_GenerateI2CStop(TSB_SBI_TypeDef* ***SBIx***)

**Parameters:**
***SBIx*** is the specified SBI channel.

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

**Return:**
None

### 10.2.3.10 SBI_GetI2CState

Get the SBI channel state in I2C bus mode.

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

**Parameters:**
***SBIx*** is the specified SBI channel.

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

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

### 10.2.3.11 SBI_SetIdleMode

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

**Prototype:**
void
SBI_SetIdleMode(TSB_SBI_TypeDef* ***SBIx***,
FunctionalState ***NewState***)

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

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

**Return:**
None

### 10.2.3.12 SBI_SetSendData

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

**Prototype:**
void
SBI_SetSendData(TSB_SBI_TypeDef* ***SBIx***,
uint32_t ***Data***)

**Parameters:**
***SBIx*** is the specified SBI channel.
***Data*** is a byte-data to be sent. The maximum value is 0xFF.

**Description:**
This function will set the data to be sent from the specified SBI channel selected by ***SBIx***. It is appropriate to call the function after the transmission of the start condition, which can be done by **SBI_GenerateI2Cstart()**, or the reception of an ACK (usually causes an SBI interrupt), to send further data required by receiver.

**Return:**
None

### 10.2.3.13 SBI_GetReceiveData

Get data received from the specified SBI channel.

**Prototype:**
uint32_t
SBI_GetReceiveData(TSB_SBI_TypeDef* ***SBIx***)

**Parameters:**
***SBIx*** is the specified SBI channel.

**Description:**
This function will set the data to be sent from the specified SBI channel selected by ***SBIx***. It is appropriate to call the function after the transmission of the start condition, which can be done by **SBI_GenerateI2Cstart()**, or the reception of an ACK (usually causes an SBI interrupt), to send further data required by receiver.

**Return:**
Data which has been received

#### 10.2.3.14 SBI_SetI2CFreeDataMode

Set SBI channel working in I2C free data mode.

**Prototype:**
void
SBI_SetI2CFreeDataMode(TSB_SBI_TypeDef* ***SBIx***,
FunctionalState ***NewState***)

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

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

**Return:**
None

## 10.2.4 Data Structure Description

### 10.2.4.1 SBI_InitI2CTypeDef

**Data Fields:**

uint32_t
***I2CSelfAddr*** specifies self-address of the SBI channel in I2C mode, the last bit of which can not be 1 and max. 0xFE.

uint32_t
***I2CDataLen*** Specify data length of the SBI channel in I2C mode, which can be set as:
- **SBI_I2C_DATA_LEN_8**, which means that the data length number of bits per transfer is 8;
- **SBI_I2C_DATA_LEN_1**, which means that the data length number of bits per transfer is 1;
- **SBI_I2C_DATA_LEN_2**, which means that the data length number of bits per transfer is 2;
- **SBI_I2C_DATA_LEN_3**, which means that the data length number of bits per transfer is 3;

- **SBI_I2C_DATA_LEN_4**, which means that the data length number of bits per transfer is 4;
- **SBI_I2C_DATA_LEN_5**, which means that the data length number of bits per transfer is 5;
- **SBI_I2C_DATA_LEN_6**, which means that the data length number of bits per transfer is 6;
- **SBI_I2C_DATA_LEN_7**, which means that the data length number of bits per transfer is 7.

uint32_t
***I2CClkDiv*** specifies the division of the source clock for I2C transfer, which can be set as:

- **SBI_I2C_CLK_DIV_104**, which means that the frequency of source clock for I2C transfer is quotient of fsys divided by 104;
- **SBI_I2C_CLK_DIV_136**, which means that the frequency of source clock for I2C transfer is quotient of fsys divided by 136;
- **SBI_I2C_CLK_DIV_200**, which means that the frequency of source clock for I2C transfer is quotient of fsys divided by 200;
- **SBI_I2C_CLK_DIV_328**, which means that the frequency of source clock for I2C transfer is quotient of fsys divided by 328;
- **SBI_I2C_CLK_DIV_584**, which means that the frequency of source clock for I2C transfer is quotient of fsys divided by 584;
- **SBI_I2C_CLK_DIV_1096**, which means that the frequency of source clock for I2C transfer is quotient of fsys divided by 1096;
- **SBI_I2C_CLK_DIV_2120**, which means that the frequency of source clock for I2C transfer is quotient of fsys divided by 2120.

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

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

### 10.2.4.2 SBI_I2CState

**Data Fields:**

uint32_t
***All*** specifies state data in I2C mode

**Bit Fields:**

uint32_t
***LastRxBit*** specifies last received bit monitor.

uint32_t
***GeneralCall*** specifies general call detected monitor.

uint32_t
***SlaveAddrMatch*** specifies slave address match monitor.

uint32_t
***ArbitrationLost*** specifies arbitration last detected monitor.

uint32_t
***INTReq*** specifies Interrupt request monitor.

uint32_t
***BusState*** specifies bus busy flag.

uint32_t
***TRx*** specifies transfer or Receive selection monitor.

uint32_t
***MasterSlave*** specifies master or slave selection monitor.

# 11. TMR16A

## 11.1 Overview

TOSHIBA TMPM061 has 7 channels TMR16A contains the following functions:

- Match interrupt
- Square waveform output
- Read capture.

All driver APIs are contained in /Libraries/TX00_Periph_Driver/src/tmpm061_tmr16a.c, with /Libraries/TX00_Periph_Driver/inc/tmpm061_tmr16a.h containing the macros, data types, structures and API definitions for use by applications.

## 11.2 API Functions

### 11.2.1 Function List

- void TMR16A_SetIdleMode(TSB_T16A_TypeDef * ***T16Ax***, FunctionalState ***NewState***);
- void TMR16A_SetClkInCoreHalt(TSB_T16A_TypeDef * ***T16Ax***, uint8_t ***ClkState***);
- void TMR16A_SetRunState(TSB_T16A_TypeDef * ***T16Ax***, uint32_t ***Cmd***);
- void TMR16A_SetSrcClk(TSB_T16A_TypeDef * ***T16Ax***, uint32_t ***SrcClk***);
- void TMR16A_SetFlipFlop(TSB_T16A_TypeDef * ***T16Ax***, TMR16A_FFOutputTypeDef * ***FFStruct***);
- void TMR16A_ChangeCycle(TSB_T16A_TypeDef * ***T16Ax***, uint32_t ***Cycle***);
- uint16_t TMR16A_GetCaptureValue(TSB_T16A_TypeDef* ***T16Ax***);

### 11.2.2 Detailed Description

Functions listed above can be divided into four parts:

1) Configure and control the common functions of each TMR16A channel are handled by TMR16A_SetSrcClk(),TMR16A_SetRunState(),and TMR16A_ChangeCycle().
2) The status indication of each TMR16A channel is handled by TMR16A_GetCaptureValue().
3) TMR16A_SetFlipFlop(),TMR16A_SetClkInCoreHalt (),TMR16A_SetIdleMode() handle other specified functions.

### 11.2.3 Function Documentation

**Note**: in all of the following APIs, unless otherwise specified, the parameter:
"TSB_T16A_TypeDef * ***T16Ax***" can be one of the following values:
**TSB_T16A0, TSB_T16A1, TSB_T16A2, TSB_T16A3, TSB_T16A4, TSB_T16A5 TSB_T16A6.**

#### 11.2.3.1 TMR16A_SetIdleMode

Enable or disable the specified TMR16A channel when system is in idle mode.

**Prototype:**
void
TMR16A _SetIdleMode(TSB_ T16A _TypeDef* ***T16Ax***,
FunctionalState ***NewState***)

**Parameters:**
***T16Ax*** is the specified TMR16A channel.
***NewState*** specifies the state of the TMR16A when system is idle mode, which can be

- **ENABLE:** enables the TMR16A channel,
- **DISABLE:** disables the TMR16A channel.

**Description:**
The specified TMR16A channel can still be running if ***NewState*** is **ENABLE** even if system enters idle mode. **DISABLE** can stop the running TMR16A if system enters idle mode.

**Return:**
None

### 11.2.3.2 TMR16A_SetClkInCoreHalt

Enable or disable clock operation in Core HALT during debug mode.

**Prototype:**
void
TMR16A_SetClkInCoreHalt (TSB_T16A_TypeDef* ***T16Ax***,
uint8_t ***ClkState***)

**Parameters:**
***T16Ax*** is the specified TMR16A channel.
***ClkState*** specifies timer state in HALT mode, which can be

- **TMR16A_RUNNING_IN_CORE_HALT:** clock not stops in Core HALT
- **TMR16A_STOP_IN_CORE_HALT:** clock stops in Core HALT.

**Description:**
This function will set enable or disable clock operation in Core HALT during debug mode.

**Return:**
None

### 11.2.3.3 TMR16A_SetRunState

Start or stop counter of the specified T16A channel.

**Prototype:**
void
TMR16A_SetRunState(TSB_T16A_TypeDef* ***T16Ax***,
uint32_t ***Cmd***)

**Parameters:**
***T16Ax*** is the specified TMR16A channel.
***Cmd*** sets the state of up-counter, which can be:

- **TMR16A_RUN:** starting counting
- **TMR16A_STOP:** stopping counting

**Description:**

The up-counter of the specified TMR16A channel starts counting if ***Cmd*** is **TMR16A_RUN** and up-counter stops counting and the value in up-counter register is clear if ***Cmd*** is **TMR16A_STOP**.

**Return:**
None

### 11.2.3.4 TMR16A_SetIdleMode

Enable or disable the specified TMR16A channel when system is in idle mode.

**Prototype:**
void
TMR16A_SetIdleMode(TSB_T16A_TypeDef* ***T16Ax***,
FunctionalState ***NewState***)

**Parameters:**
***T16Ax*** is the specified TMR16A channel.
***NewState*** specifies the state of the TMR16A when system is idle mode, which can be

- **ENABLE:** enables the TMR16A channel,
- **DISABLE:** disables the TMR16A channel.

**Description:**
The specified TMR16A channel can still be running if ***NewState*** is **ENABLE** even if system enters idle mode. **DISABLE** can stop the running TMR16A if system enters idle mode.

**Return:**
None

### 11.2.3.5 TMR16A_SetFlipFlop

Configure the flip-flop function of the specified TMR16A channel.

**Prototype:**
void
TMR16A_SetFlipFlop(TSB_T16A_TypeDef* ***T16Ax***,
TMR16A_FFOutputTypeDef* ***FFStruct***)

**Parameters:**
***T16Ax*** is the specified TMR16A channel.
***FFStruct*** is the structure containing TMR16A 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 TMR16A channel. Also the level of the output can be controlled by this API.

**Return:**
None

#### 11.2.3.6 TMR16A_ChangeCycle

Change the value of cycle for the specified channel.

**Prototype:**
void
TMR16A_ChangeCycle(TSB_T16A_TypeDef* ***T16Ax***,
uint32_t ***Cycle***)

**Parameters:**
***T16Ax*** is the specified TMR16A channel.
***Cycle*** specifies the value of cycle, max is 0xFFFF.

**Description:**
This function will specify the absolute value of cycle for the specified TMR16A. The actual interval of cycle depends on the configuration of CG and the value of ***ClkDiv***

**Return:**
None

#### 11.2.3.7 TMR16A_GetCaptureValue

Get the value of capture register of the specified TMR16A channel.

**Prototype:**
uint16_t
TMR16A_GetCaptureValue(TSB_T16A_TypeDef* ***T16Ax***)

**Parameters:**
***T16Ax*** is the specified TMR16A channel.

**Description:**
This function will return the value of capture register of the specified TMR16A channel.

**Return:**
The captured value.

### 11.2.4 Data Structure Description

#### 11.2.4.1 TMR16A_FFOutputTypeDef

**Data Fields:**

uint32_t
**TMR16AFlipflopCtrl**selects the level of flip-flop output which can be

- **TMR16A_FLIPFLOP_INVERT:** setting output reversed by using software.
- **TMR16A_FLIPFLOP_SET:** setting output to be high level.
- **TMR16A_FLIPFLOP_CLEAR:** setting output to be low level.

uint32_t
**TMR16AFlipflopReverseTrg**specifies the reverse trigger of the flip-flop output, which can be set as:

- **TMR16A_DISALBE_FLIPFLOP**, which disables the flip-flop output reverse trigger.

➢ **TMR16A_FLIPFLOP_MATCH_CYCLE**, which means that the reversing flip-flop output will be triggered when the up-counter matches the cycle.

# 12. TMRB

## 12.1 Overview

TOSHIBA TMPM061 contains 2 channels of a 16-bit up-counter, two 16-bit timer registers, two 16-bit capture registers, two comparators, a capture input control, a timer flip-flop and its associated control circuit. (TMRB0 through TMRB1). Each channel can operate in the following modes:

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

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

- Frequency measurement
- Pulse width measurement
- Time difference measurement

The TMRB driver APIs provide a set of functions to configure each channel, such as setting the clock division, trailing timing and leading timing duration, capture timing and flip-flop function. And to control the running state of each channel such as controlling up-counter, the output of flip-flop and to indicate the status of each channel such as returning the factor of interrupt, value in capture registers and so on.

All driver APIs are contained in /Libraries/TX00_Periph_Driver/src/tmpm061_tmrb.c, with /Libraries/TX00_Periph_Driver/inc/tmpm061_tmrb.h containing the macros, data types, structures and API definitions for use by applications.

## 12.2 API Functions

### 12.2.1 Function List

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

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

## 12.2.2 Detailed Description

Functions listed above can be divided into four parts:

1) Configure and control the common functions of each TMRB channel are handled by TMRB_Enable(), TMRB_Disable(), TMRB_Init(), TMRB_SetRunState(), TMRB_ChangeLeadingTiming() and TMRB_ChangeTrailingTiming().
2) Capture function of each TMRB channel is handled by TMRB_SetCaptureTiming(), and TMRB_ExecuteSWCapture().
3) The status indication of each TMRB channel is handled by TMRB_GetINTFactor(), TMRB_GetUpCntValue() and TMRB_GetCaptureValue().
4) TMRB_SetFlipFlop(), TMRB_SetINTMask(), TMRB_SetIdleMode(), TMRB_SetSyncMode(), TMRB_SetDoubleBuf(), TMRB_SetExtStartTrg() and TMRB_SetClkInCoreHalt (), TMRB_SetExtInput() handle other specified functions.

## 12.2.3 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.**

### 12.2.3.1 TMRB_Enable

Enable the specified TMRB channel.

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

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

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

**Return:**
None

### 12.2.3.2 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

### 12.2.3.3 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

### 12.2.3.4 TMRB_Init

Initialize the specified TMRB channel.

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

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

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

**Return:**
None

### 12.2.3.5 TMRB_SetCaptureTiming

Configure the capture timing.

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

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

***CaptureTiming*** specifies TMRB capture timing, which can be

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

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

**Return:**
None

### 12.2.3.6 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

### 12.2.3.7 TMRB_GetINTFactor

Indicate what causes the interrupt.

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

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

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

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

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

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

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

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

### 12.2.3.8 TMRB_SetINTMask

Mask the specified TMRB interrupt.

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

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

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

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

**Return:**
None

### 12.2.3.9 TMRB_ChangeLeadingTiming

Change the value of leading timing for the specified channel.

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

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

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

**Return:**
None

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

### 12.2.3.10 TMRB_ChangeTrailingTiming

Change the value of trailing timing for the specified channel.

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

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

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

**Return:**
None

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

### 12.2.3.11 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

### 12.2.3.12 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

### 12.2.3.13 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

### 12.2.3.14 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

### 12.2.3.15 TMRB_SetSyncMode

Enable or disable the synchronous mode of specified TMRB channel.

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

**Parameters:**
***TBx*** is the specified TMRB channel, which can be
**TSB_TB0, TSB_TB1**.

***NewState*** specifies the state of the synchronous mode of the TMRB, which can be

- **ENABLE:** enables the synchronous mode,
- **DISABLE:** disables the synchronous mode.

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

**Return:**
None

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

### 12.2.3.16 TMRB_SetDoubleBuf

Enable or disable double buffering for the specified TMRB channel.

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

**Parameters:**
***TBx*** is the specified TMRB channel.
***NewState*** specifies the state of double buffering of the TMRB, which can be

- **ENABLE:** enables double buffering,
- **DISABLE:** disables double buffering.

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

### 12.2.3.17 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

### 12.2.3.18 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

### 12.2.3.19 TMRB_SetExtInput

Set the external input source

**Prototype:**
void

TMRB_SetExtInput (TSB_TB_TypeDef* ***TBx***)

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

**Description:**
This function will set TBxIN0/1 as the external input for the specified TMRB channel.

**Return:**
None

## 12.2.4 Data Structure Description

### 12.2.4.1 TMRB_InitTypeDef

**Data Fields:**

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

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

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

uint32_t
**TrailingTiming** specifies the trailing timing value to be written into TBnRG1, max. 0xFFFF.

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

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

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

### 12.2.4.2 TMRB_FFOutputTypeDef

**Data Fields:**

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

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

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

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

### 12.2.4.3 TMRB_INTFactor

**Data Fields:**

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

# 13. SIO/UART

## 13.1 Overview

This device has several serial I/O channels. Each channel can operate in I/O Interface mode(synchronous communication) and UART mode (asynchronous communication), which can be 7-bit length, 8-bit length and 9-bit length.
In 9-bit UART mode, a wakeup function can be used when the master controller can start up slave controllers via the serial link (multi-controller system).

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

All driver APIs are contained in /Libraries/TX00_Periph_Driver/src/tmpm061_uart.c, with /Libraries/TX00_Periph_Driver/inc/tmpm061_uart.h containing the macros, data types, structures and API definitions for use by applications.

## 13.2 API Functions

### 13.2.1 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 SIO_Enable(TSB_SC_TypeDef * SIOx)
- void SIO_Disable(TSB_SC_TypeDef * SIOx)
- uint8_t SIO_GetRxData(TSB_SC_TypeDef * SIOx)
- void SIO_SetTxData(TSB_SC_TypeDef * SIOx, uint8_t Data)
- void SIO_Init(TSB_SC_TypeDef * SIOx, uint32_t IOClkSel, SIO_InitTypeDef * InitStruct)

### 13.2.2 Detailed Description

Functions listed above can be divided into three parts:

1) Initialize and configure the common functions of each UART channel are handled by UART_Enable(), UART_Disable(), UART_Init() and UART_DefaultConfig(), SIO_Enable(), SIO_Disable() and SIO_Init().
2) Transfer control and error check of each UART channel are handled by UART_GetBufState(), UART_GetRxData(), UART_SetTxData(), UART_GetErrState() and SIO_GetRxData() and SIO_SetTxData().
3) UART_SWReset(), UART_SetWakeUpFunc() and UART_SetIdleMode() handle other specified functions.

## 13.2.3 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**.
parameter "TSB_SC_TypeDef* ***SIOx***" can be one of the following values:
**SIO0, SIO1, SIO2, SIO3**.

### 13.2.3.1 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

### 13.2.3.2 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

### 13.2.3.3 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.

### 13.2.3.4 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

### 13.2.3.5 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

### 13.2.3.6 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

### 13.2.3.7 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

### 13.2.3.8 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

### 13.2.3.9 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.

### 13.2.3.10 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

### 13.2.3.11 UART_SetIdleMode

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

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

**Parameters:**
***UARTx*** is the specified UART channel.
***NewState*** is the new state of the UART channel in system idle mode.

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

**Description:**
This function will enable the specified UART channel selected by ***UARTx*** in system idle mode when ***NewState*** is **ENABLE**, and disable the channel when ***NewState*** is **DISABLE**.

**Return:**
None

### 13.2.3.1 SIO_Enable

Enable the specified SIO channel.

**Prototype:**
void
SIO_Enable (TSB_SC_TypeDef* ***SIOx***)

**Parameters:**
***SIOx*** is the specified SIOx channel.

**Description:**
This function will enable the specified SIO channel selected by ***SIOx***.

**Return:**
None

### 13.2.3.2 SIO_Disable

Disable the specified SIO channel.

**Prototype:**
void
SIO _Disable(TSB_SC_TypeDef* ***SIOx***)

**Parameters:**
***SIOTx*** is the specified SIO channel.

**Description:**
This function will disable the specified SIO channel selected by ***SIOx***.

**Return:**
None

### 13.2.3.3 SIO_GetRxData

Get data received from the specified SIO channel.

**Prototype:**
uint32_t
SIO_GetRxData(TSB_SC_TypeDef* ***SIOx***)

**Parameters:**
***SIOx*** is the specified SIO channel.

**Description:**
This function will get the data received from the specified SIO channel selected by ***SIOx***.

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

### 13.2.3.4 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

#### 13.2.3.5 SIO_Init

Initialize and configure the specified SIO channel.

**Prototype:**
void
SIO_Init(TSB_SC_TypeDef* ***SIOx***,
uint32_t ***IOClkSel,***
SIO_InitTypeDef* ***InitStruct***)

**Parameters:**
***SIOx*** is the specified SIO channel.
***IOClkSel*** is the work clock
***InitStruct*** is the structure containing basic SIO configuration including baud rate, transmission direction, transfer mode.

**Description:**
This function will initialize and configure the baud rate, transmission direction, transfer mode for the specified SIO channel selected by ***SIOx***.

**Return:**
None

## 13.2.4 Data Structure Description

### 13.2.4.1 UART_InitTypeDef

**Data Fields:**

uint32_t
***BaudRate*** configures the UART communication baud rate ranging from 2400(bps) to 115200(bps) (*).

uint32_t
***DataBits*** specifies data bits per transfer, which can be set as:
- **UART_DATA_BITS_7** for 7-bit mode
- **UART_DATA_BITS_8** for 8-bit mode
- **UART_DATA_BITS_9** for 9-bit mode

uint32_t
***StopBits*** specifies the length of stop bit transmission in UART mode, which can be set as:
- **UART_STOP_BITS_1** for 1 stop bit
- **UART_STOP_BITS_2** for 2 stop bits

uint32_t
***Parity*** specifies the parity mode, which can be set as:
- **UART_NO_PARITY** for no parity
- **UART_EVEN_PARITY** for even parity
- **UART_ODD_PARITY** for odd parity

uint32_t
***Mode*** enables or disables reception, transmission or both, which can be set as one of the followings or both by using a logical OR operation:
- **UART_ENABLE_TX** for enabling transmission
- **UART_ENABLE_RX** for enabling reception

uint32_t

***FlowCtrl*** specifies whether the hardware flow control mode is enabled or disabled (**). It can be set as:

- **UART_NONE_FLOW_CTRL** for no flow control

*: If the frequency of fperiph (refer to CG for details) is set too low or too high, the baud rate can not be configured correctly.
**: Only UART_NONE_FLOW_CTRL is included in this version.

### 13.2.4.2 SIO_InitTypeDef

**Data Fields:**

uint32_t
***InputClkEdge*** Select the input clock edge.on the SCLK output mode this bit only can set to be 0(SIO_SCLKS_TXDF_RXDR).

uint32_t
***IntervalTime*** Setting interval time of continuous transmission, which could be set as:

- **SIO_SINT_TIME_NONE** for none
- **SIO_SINT_TIME_SCLK_1** for 1*SCLK
- **SIO_SINT_TIME_SCLK_2** for 2*SCLK
- **SIO_SINT_TIME_SCLK_4** for 4*SCLK
- **SIO_SINT_TIME_SCLK_8** for 8*SCLK
- **SIO_SINT_TIME_SCLK_16** for 16*SCLK
- **SIO_SINT_TIME_SCLK_32** for 32*SCLK
- **SIO_SINT_TIME_SCLK_64** for 64*SCLK

uint32_t
***TransferMode*** Setting transfer mode which could be transfer prohibited, which can be set as:

- **SIO_TRANSFER_PROHIBIT** for transfer prohibited.
- **SIO_TRANSFER_HALFDPX_RX** for half duplex(Receive).
- **SIO_TRANSFER_HALFDPX_TX** for half duplex(Transmit).
- **SIO_TRANSFER_FULLDPX** for full duplex.

uint32_t
***TransferDir*** sets transfer direction which could be set as:

- **SIO_LSB_FRIST** for LSB FRIST in transmission
- **SIO_MSB_FRIST** for MSB FRIST in transmission.

uint32_t
***Mode*** enables or disables receive, transmission or both, which can be set as one of the followings or both by using a logical OR operation:

- **SIO_ENABLE_TX** for enabling transmission
- **SIO_ENABLE_RX** for enabling reception

uint32_t
***DoubleBuffer*** Double Buffer mode is enabled or disabled.

uint32_t
***BaudRateClock*** Select the input clock for baud rate generator, which can be set as:

- **SIO_BR_CLOCK_T1**
- **SIO_BR_CLOCK_T4**
- **SIO_BR_CLOCK_T16**
- **SIO_BR_CLOCK_T64**

uint32_t
***Divider*** division ratio "N",which can be set as:

- **SIO_BR_DIVIDER_1**
- **SIO_BR_DIVIDER_2**

- **SIO_BR_DIVIDER_3**
- **SIO_BR_DIVIDER_4**
- **SIO_BR_DIVIDER_5**
- **SIO_BR_DIVIDER_6**
- **SIO_BR_DIVIDER_7**
- **SIO_BR_DIVIDER_8**
- **SIO_BR_DIVIDER_9**
- **SIO_BR_DIVIDER_10**
- **SIO_BR_DIVIDER_11**
- **SIO_BR_DIVIDER_12**
- **SIO_BR_DIVIDER_13**
- **SIO_BR_DIVIDER_14**
- **SIO_BR_DIVIDER_15**
- **SIO_BR_DIVIDER_16**

# 14. WDT

## 14.1 Overview

The watchdog timer (WDT) is for detecting malfunctions (runaways) of the CPU caused by noises or other disturbances and remedying them to return the CPU to normal operation.

The WDT drivers API provide a set of functions to configure WDT, including such parameters as detection time, output if counter overflows, the state of WDT when enter IDLE mode and so on.

This driver is contained in \Libraries\TX00_Periph_Driver\src\tmpm061_wdt.c, with \Libraries/TX00_Periph_Driver\inc\tmpm061_wdt.h containing the API definitions for use by applications.

## 14.2 API Functions

### 14.2.1 Function List

- void WDT_SetDetectTime(uint32_t ***DetectTime***)
- void WDT_SetIdleMode(FunctionalState ***NewState***)
- void WDT_SetOverflowOutput(uint32_t ***OverflowOutput***)
- void WDT_Init(WDT_InitTypeDef * ***InitStruct***)
- void WDT_Enable(void)
- void WDT_Disable(void)
- void WDT_WriteClearCode(void)

### 14.2.2 Detailed Description

Functions listed above can be divided into three parts:

1) The Watchdog Timer basic function are handled by the WDT_SetDetectTime(),WDT_SetOverflowOutput(), WDT_Init(), WDT_Enable(), WDT_Disable(), and WDT_WriteClearCode() functions.
2) Run or stop the WDT counter when enter IDLE mode is handled by the WDT_SetIdleMode().

### 14.2.3 Function Documentation

#### 14.2.3.1 WDT_SetDetectTime

Set detection time for WDT.

**Prototype:**
void
WDT_SetDetectTime(uint32_t ***DetectTime***)

**Parameters:**
***DetectTime*:** Set the detection time
This parameter can be one of the following values:

- **WDT_DETECT_TIME_EXP_15:** ***DetectTime*** is 2^15/fsys
- **WDT_DETECT_TIME_EXP_17:** ***DetectTime*** is 2^17/fsys

- **WDT_DETECT_TIME_EXP_19:** ***DetectTime*** is 2^19/fsys
- **WDT_DETECT_TIME_EXP_21:** ***DetectTime*** is 2^21/fsys
- **WDT_DETECT_TIME_EXP_23:** ***DetectTime*** is 2^23/fsys
- **WDT_DETECT_TIME_EXP_25:** ***DetectTime*** is 2^25/fsys

**Description:**
This function will set detection time for WDT.

**Return:**
None

### 14.2.3.2 WDT_SetIdleMode

Run or stop the WDT counter when the system enters IDLE mode.

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

**Parameters:**
***NewState***: Run or stop WDT counter.
This parameter can be one of the following values:
- **ENABLE**: Run the WDT counter.
- **DISABLE**. Stop the WDT counter.

**Description:**
This function will run the WDT counter when the system enters IDLE mode when ***NewState*** is **ENABLE**, and stop the WDT counter when the system enters IDLE mode when ***NewState*** is **DISABLE**.

**Notes:**
If CPU needs to enter the IDLE mode, this function must be called with appropriate parameter.

**Return:**
None

### 14.2.3.3 WDT_SetOverflowOutput

Set WDT to generate NMI interrupt or 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

### 14.2.3.4 WDT_Init

Initialize and configure WDT.

**Prototype:**
void
WDT_Init (WDT_InitTypeDef* ***InitStruct***)

**Parameters:**
***InitStruct***: The structure containing basic WDT configuration including detect time and WDT output when counter overflow. (Refer to "Data structure Description" for details)

**Description:**
This function will initialize and configure the WDT detection time and the output of WDT when the counter overflows. **WDT_SetDetectTime()** and **WDT_SetOverflowOutput()** will be called by it.

**Return:**
None

### 14.2.3.5 WDT_Enable

Enable the WDT function.

**Prototype:**
void
WDT_Enable(void)

**Parameters:**
None

**Description:**
This function will enable WDT.

**Return:**
None

### 14.2.3.6 WDT_Disable

Disable the WDT function.

**Prototype:**
void
WDT_Disable(void)

**Parameters:**

None

**Description:**
This function will disable WDT.

**Return:**
None

#### 14.2.3.7 WDT_WriteClearCode

Write the clear code.

**Prototype:**
void
WDT_WriteClearCode (void)

**Parameters:**
None

**Description:**
This function will clear the WDT counter.

**Return:**
None

### 14.2.4 Data Structure Description

#### 14.2.4.1 WDT_InitTypeDef

**Data Fields:**

uint32_t
***DetectTime*** Set WDT detection time, which can be set as:

- **WDT_DETECT_TIME_EXP_15: *DetectTime*** is 2^15/fsys
- **WDT_DETECT_TIME_EXP_17: *DetectTime*** is 2^17/fsys
- **WDT_DETECT_TIME_EXP_19: *DetectTime*** is 2^19/fsys
- **WDT_DETECT_TIME_EXP_21: *DetectTime*** is 2^21/fsys
- **WDT_DETECT_TIME_EXP_23: *DetectTime*** is 2^23/fsys
- **WDT_DETECT_TIME_EXP_25: *DetectTime*** is 2^25/fsys

uint32_t
***OverflowOutput*** Select the action when the WDT counter overflows, which can be set as:

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