TOSHIBA

TOSHIBA TX03 Peripheral Driver User Guide (TMPM330/332/333)

Ver 1
Sep, 2017

TOSHIBA ELECTRONIC DEVICES & STORAGE CORPORATION

CMDR-M330UG-01E

RESTRICTIONS ON PRODUCT USE

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

Index

1. Introduction

TOSHIBA TX03 Peripheral Driver is a set of drivers for all peripherals found on the TOSHIBA TX03 series microcontrollers. TMPM33x(*) Peripheral Driver is an important part of TOSHIBA TX03 Peripheral Driver, which are designed for TMPM33x series MCUs.

TOSHIBA TX03 Peripheral Driver contains a collection of macros, data types, structures, functions plus examples for each peripheral, which makes it quite understandable and convenient to be used in the user application.

The design goals of TOSHIBA TMPM33x Peripheral Driver:

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

*Note: here TMPM33x stands for TMPM330/332/333.

2. Organization of TOSHIBA TX03 Peripheral Driver

/Libraries
This folder contains all CMSIS files and TMPM33x Peripheral Drivers.

/Libraries/ TX03_CMSIS
This folder contains the TMPM33x CMSIS files: device peripheral access layer and core peripheral access layer.

/Libraries/TX03_Periph_Driver
This folder contains all the source code of the drivers, the core of TOSHIBA TMPM33x Peripheral Driver.

/Libraries/TX03_Periph_Driver/inc
This folder contains all the header files of TMPM33x Peripheral Drivers for each peripheral.

/Libraries/TX03_Periph_Driver/src
This folder contains all the source files of TMPM33x Peripheral Drivers for each peripheral.

/Project
This folder contains template project and examples for using TMPM33x Peripheral Driver.

/Project/Template

This folder contains template project of TOSHIBA TMPM33x Peripheral Driver.

/Project/Examples

This folder contains a set of examples for using TMPM33x Peripheral Driver

/Utilities/TMPM33x-SK

This folder contains the configuration and driver files for hardware resource except for MCU on TMPM330-SK board.

3. ADC

3.1 Overview

This device has several channels 10-bit A/D converter, the main functions include:
• Start by an internal or external timer trigger
• Fixed channel/scan mode
• Single/repeat mode
• AD monitoring 2ch
• Conversion speed 1.15usec(@fsys = 40MHz)

The ADC API provides a set of functions for using the TMPM33x ADC modules. It includes ADC channel set, mode set, monitor function set, interrupt set, ADC status read, ADC result value set and so on.

This driver is contained in TX03_Periph_Driver\src\tmpm33**x**_adc.c(*), with TX03_Periph_Driver\inc\tmpm33**x**_adc.h(*) containing the API definitions for use by applications.

*__Note__: "**x**" can be 0,2,3

3.2 Difference among TMPM330, TMPM332 and TMPM333 for ADC

TMPM330/M333 have 12 channels (AN0~AN11);
TMPM332 only has 8 channels (AN4~AN11).

3.3 API Functions

3.3.1 Function List

- void ADC_SWReset(void)
- void ADC_SetAccuracy(void)
- void ADC_SetClk(uint32_t ***Sample_HoldTime***, uint32_t ***Prescaler_Output***)
- void ADC_Start(void)
- void ADC_SetScanMode(FunctionalState ***NewState***)
- void ADC_SetRepeatMode(FunctionalState ***NewState***)
- void ADC_SetINTMode(uint8_t ***INTMode***)
- 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***)
- void ADC_SetInputChannelTop(uint8_t ***TopInputChannel***)
- void ADC_StartTopConvert(void)
- WorkState ADC_GetTopConvertState(void)
- void ADC_SetMonitor(uint16_t ***ADCMPx***, FunctionalState ***NewState***)
- void ADC_SetResultCmpReg(uint16_t ***ADCMPx***, uint8_t ***ResultComparison***)
- void ADC_SetMonitorINT(uint16_t ***ADCMPx***, ADC_ComparisonState ***NewState***)
- void ADC_SetHWTrg(uint8_t ***HwSource***, FunctionalState ***NewState***)
- void ADC_SetHWTrgTop(uint8_t ***HwSource***, FunctionalState ***NewState***)
- ADC_ResultTypeDef ADC_GetConvertResult (uint8_t ***ADREGx***)
- void ADC_SetCmpValue(uint16_t ***ADCMPx***, uint16_t ***value***)

3.3.2 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(), ADC_SetInputChannelTop(), ADC_SetMonitor(), ADC_SetResultCmpReg(), ADC_SetMonitorINT(), ADC_SetHWTrg(), ADC_SetHWTrgTop(), ADC_SetCmpValue().
2) ADC function start by ADC_Start(), ADC_StartTopConvert().
3) ADC state or data read functions by ADC_GetConvertState(), ADC_GetTopConvertState(), ADC_GetConvertResult ().
4) ADC_SWReset(), ADC_SetAccuracy(), and ADC_SetIdleMode() handle other specified functions.

3.3.3 Function Documentation

3.3.3.1 ADC_SWReset

Software reset ADC function.

Prototype:
void
ADC_SWReset(void)

Parameters:
None

Description:
This function will software reset ADC.

Notes:
A software reset initializes other bits. Re-setting a mode register is needed.

Return:
None

3.3.3.2 ADC_SetAccuracy

Set ADC accuracy.

Prototype:
void
ADC_SetAccuracy(void)

Parameters:
None

Description:
This function will set ADC accuracy to assure conversion accuracy.

Notes:
This function should be called before ADC setting if ADC is reset.

Return:
None

3.3.3.3 ADC_SetClk

Set ADC sample hold time and prescaler output.

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

Parameters:
Sample_HoldTime: Select ADC sample hold time.
This parameter can be one of the following values:
ADC_HOLD_CLK_8, **ADC_HOLD_CLK_16**,
ADC_HOLD_CLK_24, **ADC_HOLD_CLK_32**,
ADC_HOLD_CLK_64, **ADC_HOLD_CLK_128**,
ADC_HOLD_CLK_512.

Prescaler_Output: Select ADC prescaler output(ADCLK).
This parameter can be one of the following values:
ADC_FC_DIVIDE_LEVEL_1, **ADC_FC_DIVIDE_LEVEL_2**,
ADC_FC_DIVIDE_LEVEL_4, **ADC_FC_DIVIDE_LEVEL_8**,
ADC_FC_DIVIDE_LEVEL_16

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

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

Return:
None

3.3.3.4 ADC_Start

Start ADC.

Prototype:
void
ADC_Start(void)

Parameters:
None

Description:
This function will start AD conversion.

Notes:
This function should be called after specifying the mode.
Before starting AD conversion, Vref should be enabled by calling **ADC_SetVref(ENABLE)**, wait for 3 us during which time the internal reference voltage is stable, and then **ADC_Start()**.

Return:
None

3.3.3.5 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

3.3.3.6 ADC_SetRepeatMode

Set ADC repeat mode.

Prototype:
void
ADC_SetRepeatMode(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 repeat mode by ***NewState*** setting.

Return:
None

3.3.3.7 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_SIGNLE, ADC_INT_CONVERSION_4
Or **ADC_INT_CONVERSION_8**

Description:
This function will specify ADC interrupt mode by ***INTMode*** setting only in fixed channel repeat conversion mode.

Notes:
Examples for setting fixed channel repeat conversion mode:
1. **ADC_SetScanMode(DISABLE)**.
2. **ADC_SetRepeatMode(ENABLE)**.

Return:
None

3.3.3.8 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: During conversion.

3.3.3.9 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, ADC_AN_1, ADC_AN_2, ADC_AN_3(Invalid for TMPM332),
ADC_AN_4, ADC_AN_5, ADC_AN_6, ADC_AN_7,
ADC_AN_8, ADC_AN_9, ADC_AN_10, ADC_AN_11;
ADC_AN_0_1, ADC_AN_0_2, ADC_AN_0_3, ADC_AN_0_4,
ADC_AN_0_5, ADC_AN_0_6, ADC_AN_0_7,(ADC_AN_0_1~ADC_AN_0_7 is invalid for TMPM332)
ADC_AN_4_5, ADC_AN_4_6, ADC_AN_4_7, ADC_AN_8_9, ADC_AN_8_10, ADC_AN_8_11.

Description:
This function will specify ADC input channel by ***InputChannel*** setting. And the input channels also relate with mode setting.
In fixed channel mode(**ADC_SetScanMode(DISABLE)**), user can only select one channel of the 12 channels.

In channel scan mode (**ADC_SetScanMode(ENABLE)**), the input channels is different for 4 channel scan mode(**ADC_SetChannelScanMode(ADC_SCAN_4CH)**) and 8 channel scan mode(**ADC_SetChannelScanMode(ADC_SCAN_8CH)**).

Notes:
Set channel scan mode: **ADC_SetScanMode(ENABLE)**.
Set 4 channel scan mode mode: **ADC_SetChannelScanMode(ADC_SCAN_4CH)**.
Set 8 channel scan mode mode: **ADC_SetChannelScanMode(ADC_SCAN_8CH)**.

Return:
None

3.3.3.10 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 or **ADC_SCAN_8CH**

Description:
This function will specify different channel scan mode by ***ScanMode*** setting.

Notes:
This function setting will change the input channel setting **ADC_SetInputChannel()**, please refer to **ADC_SetInputChannel()** description for details.

Return:
None

3.3.3.11 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 enable or disable in system IDLE mode by ***NewState*** setting.
This function is necessary to be called before system enter IDLE mode.

Return:
None

3.3.3.12 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 Vref on or off by ***NewState***.

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

Return:
None

3.3.3.13 ADC_SetInputChannelTop

Set ADC top-priority conversion analog input channel select.

Prototype:
void

ADC_SetInputChannelTop(uint8_t ***TopInputChannel***)

Parameters:
TopInputChannel: Analog input channel for top-priority conversion.
This parameter can be one of the following values:
ADC_AN_0, ADC_AN_1, ADC_AN_2, ADC_AN_3(Invalid for TMPM332),
ADC_AN_4, ADC_AN_5, ADC_AN_6, ADC_AN_7,
ADC_AN_8, ADC_AN_9, ADC_AN_10, ADC_AN_11.

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

Notes:
Only one channel of **ADC_AN_0~ADC_AN_11** can be selected as Top-priority conversion input each time.

Return:
None

3.3.3.14 ADC_StartTopConvert

Start ADC top-priority conversion.

Prototype:
void
ADC_StartTopConvert(void)

Parameters:
None

Description:
This function will start top-priority conversion.

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

Return:
None

3.3.3.15 ADC_GetTopConvertState

Read Top-priority AD conversion state.

Prototype:
WorkState
ADC_GetTopConvertState(void)

Parameters:
None

Description:
This function read top-priority conversion state.

Return:
The state of top-priority AD conversion, which can be
DONE: Conversion completed.
BUSY: Before or during conversion.

3.3.3.16 ADC_SetMonitor

Set ADC monitor function.

Prototype:
void
ADC_SetMonitor(uint16_t ***ADCMPx***,
FunctionalState ***NewState***)

Parameters:
ADCMPx: Select AD compare register
The parameter can be one of the following values:
ADC_CMP_0 or **ADC_CMP_1**

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

Description:
This device has 2ch AD monitor channels: channel 0 and channel 1.
This function will specify ADC monitor channel by ***ADCMPx*** setting and specify ADC monitor function enable or disable by ***NewState*** setting.

Return:

None

3.3.3.17 ADC_SetResultCmpReg

Set ADC result comparison register.

Prototype:
void
ADC_SetResultCmpReg(uint16_t ***ADCMPx***,
uint8_t ***ResultComparison***)

Parameters:
ADCMPx: Select AD compare register
The parameter can be one of the following values:
ADC_CMP_0 or **ADC_CMP_1**

ResultComparison: Select the AD conversion result storage register that is to be compared with the comparison register if ADC monitor function is enabled.
The parameter can be one of the following values:
ADC_REG_08, **ADC_REG_19**, **ADC_REG_2A**, **ADC_REG_3B**
ADC_REG_4C, **ADC_REG_5D**, **ADC_REG_6E**, **ADC_REG_7F**
ADC_REG_SP

Description:
This device has 2ch AD monitor channels: channel 0 and channel 1.
This function will specify ADC monitor channel by ***ADCMPx*** setting and specify AD conversion result storage register that is to be compared with the comparison register by ***ResultComparison*** setting.

Return:
None

3.3.3.18 ADC_SetMonitorINT

Set ADC monitor interrupt.

Prototype:
void
ADC_SetMonitorINT(uint16_t ***ADCMPx***,
ADC_ComparisonState ***NewState***)

Parameters:

ADCMPx: Selet AD compare register
The parameter can be one of the following values:
ADC_CMP_0 or **ADC_CMP_1**

NewState: Specify ADC monitor function interrupt condition
This parameter can be one of the following values:
ADC_COMPARISON_SMALLER or **ADC_COMPARISON_LARGER**.

Description:
This device has 2ch AD monitor channels: channel 0 and channel 1.
This function will specify ADC monitor interrupt by ***ADCMPx*** setting and specify ADC monitor function interrupt condition by ***NewState*** setting.

Return:
None

3.3.3.19 ADC_SetHWTrg

Hardware trigger for normal ADC enable or disable and Hardware Source for activating normal ADC setting.

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

Parameters:
HwSource: Hardware source for activating normal ADC.
This parameter can be one of the following values:
ADC_EXT_TRG or **ADC_MATCH_TB6RG0**

NewState: enable or disable hardware source for activating normal ADC
This parameter can be one of the following values:
ENABLE or **DISABLE**

Description:
This function will specify hardware Source for activating normal ADC setting by ***HwSource*** setting and specify hardware trigger for normal ADC monitor function enable or disable by ***NewState*** setting.
This function also has relation with TB6 setting.

Notes:

The TMPM330 disables the external trigger used for hardware activation. Therefore do not use the function as **ADC_SetHWTrg(ADC_EXT_TRG, NewState)**.
If AD conversion is executed with the match triggers <ADHTG>(Refer to TMPM330 datasheet) and <HADHTG>(Refer to TMPM330 datasheet) of a 16-bit timer set to "1" by using a source for triggering hardware, A/D conversion can be activated at specified intervals by performing three steps shown below when the timer is idle:

- Select a source for triggering hardware.
- Enable hardware activation of AD conversion.
- Start the timer

Do not make a top-priority AD conversion setting and a normal AD conversion setting simultaneously. Do not use functions
ADC_SetHWTrg(ADC_MATCH_TB6RG0, ENABLE) and
ADC_SetHWTrgTop(ADC_MATCH_TB5RG0, ENABLE) simultaneously.

Return:
None

3.3.3.20 ADC_SetHWTrgTop

Hardware trigger for top-priority ADC enable or disable and Hardware Source for activating top-priority ADC setting.

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

Parameters:
HwSource: Hardware source for activating top-priority ADC.
This parameter can be one of the following values:
ADC_EXT_TRG or **ADC_MATCH_TB5RG0**

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

Description:
This function will specify hardware Source for activating top-priority ADC setting by ***HwSource*** setting and specify hardware trigger for top-priority ADC monitor function enable or disable by ***NewState*** setting.
This function also has relation with TB5 setting.

Notes:

The TMPM330 disables the external trigger used for hardware activation. Therefore do not use the function as
ADC_SetHWTrgTop(ADC_EXT_TRG, NewState).
Do not make a top-priority AD conversion setting and a normal AD conversion setting simultaneously. Do not use functions
ADC_SetHWTrg(ADC_MATCH_TB6RG0, ENABLE) and
ADC_SetHWTrgTop(ADC_MATCH_TB5RG0, ENABLE) simultaneously.

Return:
None

3.3.3.21 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_08, **ADC_REG_19**, **ADC_REG_2A**, **ADC_REG_3B**
ADC_REG_4C, **ADC_REG_5D**, **ADC_REG_6E**, **ADC_REG_7F**
ADC_REG_SP

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

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

The ***ADREGx*** overrun state will set to **ADC_OVERRUN** if a conversion result is overwritten before both conversion result storage registers (ADREGxH and ADREGxL) are read. The overrun state will be cleared after overrun state is read by this function.

AD conversion result is stored in ***ADRGEx*** in different ADC mode as below table.

Table Analog Input Channels and Related A/D Conversion Result Registers

Analog input	A/D conversion result register

channel (port A)	Conversion modes other than shown to the right	Fixed channel repeat conversion mode (every one conversion)	Fixed channel repeat conversion mode (every four conversions)	Fixed channel repeat conversion mode (every eight conversions)
ADC_AN_0	ADC_REG_08	ADC_REG_08 fixed	ADC_REG_08--> ADC_REG_3B	ADC_REG_08--> ADC_REG_7F
ADC_AN_1	ADC_REG_19			
ADC_AN_2	ADC_REG_2A			
ADC_AN_3	ADC_REG_3B			
ADC_AN_4	ADC_REG_4C			
ADC_AN_5	ADC_REG_5D			
ADC_AN_6	ADC_REG_6E			
ADC_AN_7	ADC_REG_7F			
ADC_AN_8	ADC_REG_08			
ADC_AN_9	ADC_REG_19			
ADC_AN_10	ADC_REG_2A			
ADC_AN_11	ADC_REG_3B			

The ADC mode setting, please refer to relate APIs.

For high-priority ADC, the result is stored in ADC_REG_SP.

Return:

ADC result structure:

1. The state of ADC result stored state, which can be
   - **DONE**: AD conversion complete and result stored.
   - **BUSY**: During conversion.
2. The state of normal AD conversion complete, which can be
   - **ADC_NO_OVERRUN**: No Conversion overrun.
   - **ADC_OVERRUN**: Conversion overrun.
3. ADC result value.

3.3.3.22 ADC_SetCmpValue

Set ADC comparison register value.

Prototype:

void
ADC_SetCmpValue(uint16_t ***ADCMPx***,
uint16_t ***value***)

Parameters:

ADCMPx: Select AD compare register

The parameter can be one of the following values:

ADC_CMP_0 or **ADC_CMP_1**

value: The value set to ADC compare register

Description:

This device has 2ch AD monitor channels: channel 0 and channel 1.
This function will set the ADC compare register value of ADC monitor channel which specify by ***ADCMPx*** setting.
The max setting value should not be larger than 0x03ff for ADC is only 10-bit.

Notes:

ADC monitor function setting process:
1. **ADC_SetResultCmpReg(ADCMPx, ResultComparison)**
2. **ADC_SetCmpValue(ADCMPx, value)**
3. **ADC_SetMonitorINT(ADCMPx, ResultComparison)**
4. **ADC_SetMonitor(ADCMPx, ENABLE)**

After AD conversion finished, if the condition match **ADC_SetMonitorINT()** setting, ADC monitor interrupt will occurs(The interrupt enable, please refer to interrupt chapter of TMPM330 datasheet).

Return:

None

3.3.4 Data Structure Description

3.3.4.1 ADC_ResultTypeDef

Data Fields:

WorkState

ADCResultStored specifies ADC result storage flag, which can be set as:

- **BUSY**, which means that ADC result has not been stored to the result register;
- **DONE**, which means that which ADC result has been stored to the result register.

ADC_OverrunState

ADCOverrunState specifies ADC overrun flag, which can be set as:

- **ADC_NO_OVERRUN**, which means that ADC is not overrun;
- **ADC_OVERRUN**, which means that ADC is overrun.

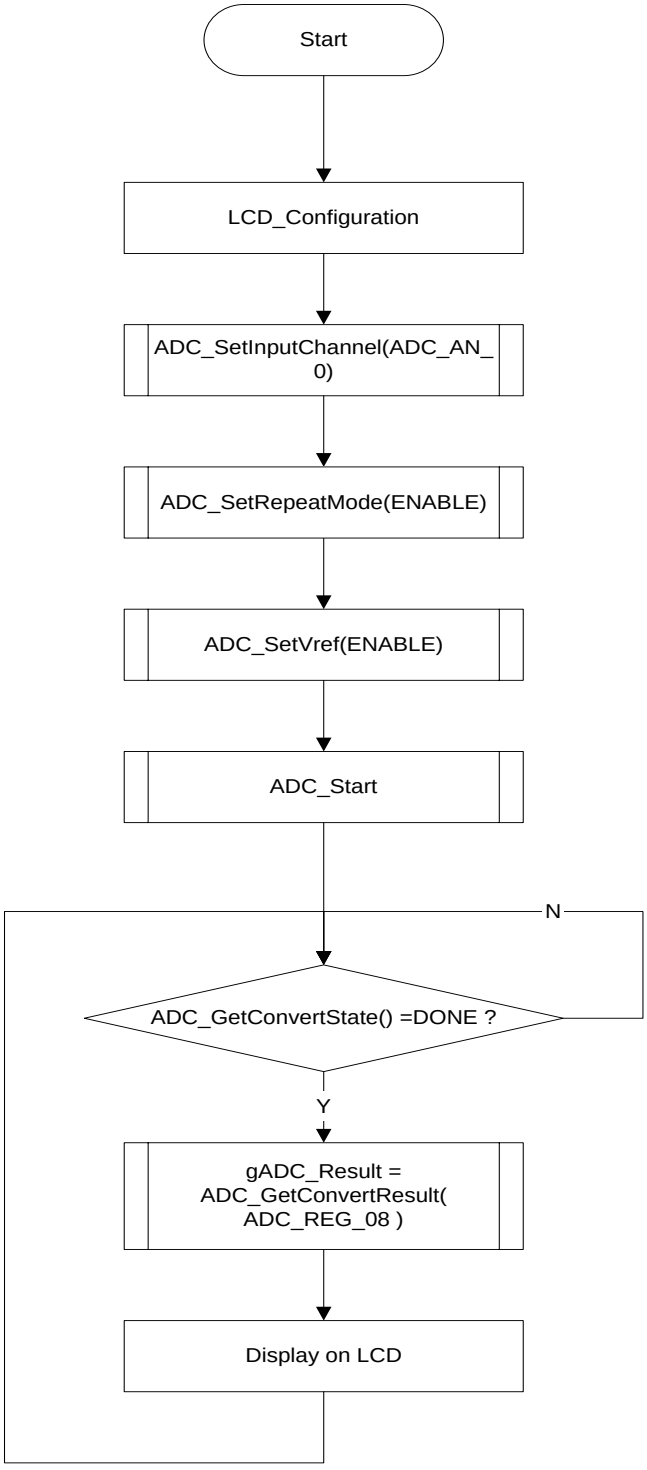
uint32_t

ADCResultValue specifies ADC result value,

3.3.5 Programming Example

This is a simple application based on the TMPM33x Peripheral Driver (ADC), which read ADC data from AN0.

3.3.5.1 Flow chart



3.3.5.2 Code and Explanation for the Example

At first, initial ADC setting for AN0, repeat mode, ADC Vref ON. For example,

```
ADC_SetInputChannel(ADC_AN_0);
ADC_SetRepeatMode(ENABLE);
ADC_SetVref(ENABLE);
```

Then start ADC.

```
ADC_Start();
```

After ADC start, check ADC convert end or not, if ADC is finished, the read the ADC result data.

```
while(DONE == ADC_GetConvertState());
gADC_Result = ADC_GetConvertResult (ADC_REG_08);
```

After ADC result data is read, display on LCD.

4. CEC

4.1 Overview

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

Reception

- Clock sampling at 32KHz (1 Cycle)
    1. Adjustable noise canceling time
    2. Data reception per 1byte
- Flexible data sampling point
    1. Data reception is available even when an address discrepancy is detected.
- Error detection
    1. Cycle error（min. / max.）
    2. ACK collision
    3. Waveform error

Transmission

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

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm33**x**_cec.c(*), with /Libraries/TX03_Periph_Driver/inc/tmpm33**x**_cec.h(*) containing the macros, data types, structures and API definitions for use by applications.

***Note**: "**x**" can be 0, 2.

4.2 Difference among TMPM330, TMPM332 and TMPM333 in CEC

There's no CEC module in TMPM333.

4.3 API Functions

4.3.1 Function List

- void CEC_Enable(void)
- void CEC_Disable(void)
- void CEC_SWReset(void)
- Result CEC_DefaultConfig(void)
- Result CEC_SetLogicalAddr(CEC_LogicalAddr ***LogicalAddr***)
- Result CEC_AddLogicalAddr(CEC_LogicalAddr ***LogicalAddr***)
- Result CEC_RemoveLogicalAddr(CEC_LogicalAddr ***LogicalAddr***)
- CEC_AddrListTypeDef CEC_GetLogicalAddr(void)
- void CEC_SetRxCtrl(FunctionalState ***NewState***)
- Result CEC_StartTx(void)
- void CEC_StopTx(void)
- CEC_DataTypeDef CEC_GetRxData(void)
- void CEC_SetTxData(uint8_t ***Data***,CEC_EOMBit ***EOM_Flag***)
- void CEC_SetIdleMode(FunctionalState ***NewState***)
- FunctionalState CEC_GetRxState(void)
- WorkState CEC_GetTxState(void)
- CEC_RxINTState CEC_GetRxINTState(void)
- CEC_TxINTState CEC_GetTxINTState(void)
- Result CEC_SetACKResponseMode(FunctionalState ***NewState***)
- Result CEC_SetNoiseCancellation(CEC_LowCancellation ***LowCancellation***,
  CEC_HighCancellation ***HighCancellation***)
- Result CEC_SetCycleConfig(CEC_CycleMin ***CycleMin***, CEC_CycleMax ***CycleMax***)
- Result CEC_SetDataValidTime(CEC_ValidTime ***ValidTime***)
- Result CEC_SetTimeOutMode(CEC_TimeOut ***TimeOut***);
- Result CEC_SetRxErrINTSuspend(FunctionalState ***NewState***)
- Result CEC_SetSnoopMode(FunctionalState ***NewState***)
- Result CEC_SetRxDetectWaveConfig (
  CEC_Logical1RisingTimeMin ***Logical1RisingTimeMin***,
  CEC_Logical1RisingTimeMax ***Logical1RisingTimeMax***,
  CEC_Logical0RisingTimeMin ***Logical0RisingTimeMin***,
  CEC_Logical0RisingTimeMax ***Logical0RisingTimeMax***)
- Result CEC_SetRxStartBitWaveConfig(
  CEC_StartBitRisingTimeMin ***RisingTimeMin***,
  CEC_StartBitRisingTimeMax ***RisingTimeMax***,
  CEC_StartBitCycleMin ***CycleMin***,
  CEC_StartBitCycleMax ***CycleMax***)
- Result CEC_SetRxWaveErrDetect(FunctionalState ***NewState***)
- Result CEC_SetTxWaveConfig(
  CEC_TxDataBitCycle ***DataBitCycle***,

CEC_TxDataBitRisingTime ***DataBitRisingTime***,
CEC_TxStartBitCycle ***StartBitCycle***,
CEC_TxStartBitRisingTime ***StartBitRisingTime***)

- Result CEC_SetTxBroadcast(FunctionalState ***NewState***)
- Result CEC_SetBusFreeTime(CEC_BusFree ***BusFree***)

4.3.2 Detailed Description

Functions listed above can be divided into three parts:

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

4.3.3 Function Documentation

4.3.3.1 CEC_Enable

Enable CEC module.

Prototype:
void
CEC_Enable(void)

Parameters:
None

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

Return:
None

4.3.3.2 CEC_Disable

Disable CEC module.

Prototype:
void
CEC_Disable(void)

Parameters:
None

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

Return:
None

4.3.3.3 CEC_SWReset

Reset the CEC module.

Prototype:
void
CEC_SWReset(void)

Parameters:
None

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

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

Return:
None

4.3.3.4 CEC_DefaultConfig

Initialize the CEC in the default configuration.

Prototype:
Result
CEC_DefaultConfig(void)

Parameters:
None

Description:
Initialize the CEC in the default configuration:
Idle Mode: on
Noise Cancellation Time: H: 1 cycle L: 1 cycle
Cycle Range: 2.05ms~2.75ms
Data Valid Time: 1.05ms
Time Out: 1 Bit
Rx Start Wave configure: Min of start: 3.5ms; Max of start: 3.9ms
Min of cycle: 4.3ms; Max of cycle: 4.7ms
Receive Bit Wave configure: Min of “1”: 0.4ms; Max of “1”: 0.8ms
Min of “0”:1.3ms; Max of “0”: 1.7
Send Bit Wave configure: RV
Bus free configure: 5 bit cycle
Snoop mode: On
When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.
Registers CECEN, CECADD, CECRCR1, CECRCR2, CECRCR3 and CECTCR are modified by this function.

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

4.3.3.5 CEC_SetLogicalAddr

Specify the logical address assigned to CEC.

Prototype:
Result
CEC_SetLogicalAddr(CEC_LogicalAddr ***LogicalAddr***)

Parameters:
LogicalAddr: the logical address of CEC
This parameter can be one of the following values:
CEC_TV, **CEC_RECORDING_DEVICE_1**, **CEC_RECORDING_DEVICE_2**, **CEC_STB_1**, **CEC_DVD_1**,**CEC_AUDIO_SYSTEM**, **CEC_STB_2**, **CEC_STB_3**, **CEC_DVD_2**, **CEC_RECORDING_DEVICE_3**, **CEC_FREE_USE**, **CEC_BROADCAST**

Description:
Specify the logical address assigned to CEC. After call this function, the old logical address setting will be cleared and set the new one in the register.
When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.
Register CECADD is modified by this function.

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

4.3.3.6 CEC_AddLogicalAddr

Add new logical address to CEC device.

Prototype:
Result
CEC_AddLogicalAddr(CEC_LogicalAddr ***LogicalAddr***)

Parameters:
LogicalAddr: the logical address of CEC.
This parameter can be one of the following values:
CEC_TV, **CEC_RECORDING_DEVICE_1**, **CEC_RECORDING_DEVICE_2**, **CEC_STB_1**, **CEC_DVD_1**, **CEC_AUDIO_SYSTEM**, **CEC_STB_2**, **CEC_STB_3**, **CEC_DVD_2**, **CEC_RECORDING_DEVICE_3**, **CEC_FREE_USE**, **CEC_BROADCAST**

Description:
Add one new logical address to CEC. After calling this function, the old logical address setting will be remain and just add a new address.
When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.
Register CECADD is modified by this function.

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

4.3.3.7 CEC_RemoveLogicalAddr

Remove one of logical address from CEC device.

Prototype:
Result
CEC_RemoveLogicalAddr(CEC_LogicalAddr ***LogicalAddr***)

Parameters:
LogicalAddr is the logical address of CEC.
This parameter can be one of the following values:
CEC_TV, **CEC_RECORDING_DEVICE_1**, **CEC_RECORDING_DEVICE_2**, **CEC_STB_1**, **CEC_DVD_1**, **CEC_AUDIO_SYSTEM**, **CEC_STB_2**, **CEC_STB_3**, **CEC_DVD_2**, **CEC_RECORDING_DEVICE_3**, **CEC_FREE_USE**, **CEC_BROADCAST**

Description:
Remove one of logical address from CEC device.
After calling this function, other old logical address setting will be remained.
When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.
Register CECADD is modified by this function.

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

4.3.3.8 CEC_GetLogicalAddr

Get the logical address of device.

Prototype:
CEC_AddrListTypeDef
CEC_GetLogicalAddr(void)

Parameters:
None

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

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

4.3.3.9 CEC_SetRxCtrl

Enable/Disable data reception of CEC.

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

Parameters:
NewState: New state of the data reception function.

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

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

Return:
None

4.3.3.10 CEC_StartTx

Start data transmission of CEC.

Prototype:
Result

CEC_StartTx(void)

Parameters:
None

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

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

4.3.3.11 CEC_StopTx

Stop data transmission of CEC.

Prototype:
void
CEC_StopTx(void)

Parameters:
None

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

Return:
None

4.3.3.12 CEC_GetRxData

Read the data received.

Prototype:
CEC_DataTypeDef
CEC_GetRxData (void)

Parameters:
None

Description:
Read the received data. The data include the one byte of received data. The bit 7 is the MSB, the received ACK state and the received EOM state. The function should be called as soon as a reception interrupt is generated. The reception continues from the first data block until the final data block which has the EOM bit indicating “1”. After detecting the final data block, CEC waits for the next start bit.

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

4.3.3.13 CEC_SetTxData

Set the data to be sent.

Prototype:
void
CEC_SetTxData(uint8_t ***Data***,
CEC_EOMBit ***EOM_Flag***)

Parameters:
Data: The 1 byte data to be sent.
EOM_Flag: Specify the EOM bit to transmit, which can be:
- **CEC_EOM** means last data of the frame.
- **CEC_NO_EOM** means not last data of the frame.

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

Return:
None

4.3.3.14 CEC_SetIdleMode

Control the CEC operation at the IDLE mode.

Prototype:

void
CEC_SetIdleMode(FunctionalState ***NewState***)

Parameters:
NewState: New state of the Idle Mode.
- **ENABLE** : using CEC at the IDLE mode
- **DISABLE** : not using CEC at the IDLE mode

Description:
Control the CEC operation at the IDLE mode.
Set ***NewState*** with **ENABLE** when using CEC at the IDLE mode.
Register CECEN< I2CEC> is modified by this function.

Return:
None

4.3.3.15 CEC_GetRxState

Get the Receive state of CEC.

Prototype:
FunctionalState
CEC_GetRxState (void)

Parameters:
None

Description:
Get the Receive state of CEC.

Return:
ENABLE means CEC reception is enabled.
DISABLE means CEC reception is disabled.

4.3.3.16 CEC_GetTxState

Get the transmission state of CEC

Prototype:
WorkState
CEC_GetTxState(void)

Parameters:

None

Description:
Get the transmission state of CEC.

Return:
BUSY means transmission is in progress.
DONE means transmission is completed or an interrupt is generated.

4.3.3.17 CEC_GetRxINTState

Get the receive interrupt state of CEC

Prototype:
CEC_RxINTState
CEC_GetRxINTState(void)

Parameters:
None

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

Return:
State of Receive Interrupt, Each bit is described as below:
RxEnd(Bit 0) means 1 byte of data reception is completed,
RxStartBit(Bit 1) means a start bit is detected,
MAXCycleErr(Bit 2) means the maximum cycle error detected,
MINCycleErr(Bit 3) means the minimum cycle error detected,
ACKCollision(Bit 4) means ACK collision detected,
BufOverrun(Bit 5) means receive buffer overrun detected,
WaveformErr(Bit 6) means wave form error detected,

4.3.3.18 CEC_GetTxINTState

Get the transmission interrupt state of CEC.

Prototype:
CEC_TxINTState
CEC_GetTxINTState(void)

Parameters:
None

Description:
Get the transmission interrupt state of CEC and return the result. Each bit is described as follow:
TxStart(Bit 0) means 1 byte of data transmission is started, **TxEnd**(Bit 1) means data transmission including the EOM bit is completed, **ArbitrationLost**(Bit 2) means arbitration lost occurs, **ACKErr**(Bit 3) means ACK error detected and **BufUnderrun**(Bit 4) means transmit buffer underrun detected.

Return:
State of Transmit Interrupt, Each bit is described as follow:
TxStart(Bit 0) means 1 byte of data transmission is started,
TxEnd(Bit 1) means data transmission including the EOM bit is completed,
ArbitrationLost(Bit 2) means arbitration lost occurs,
ACKErr(Bit 3) means ACK error detected,
BufUnderrun(Bit 4) means transmit buffer underrun detected,

4.3.3.19 CEC_SetACKResponseMode

Set the ACK response mode of CEC.

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

Parameters:
NewState: New state of the ACK Response Mode.
- **ENABLE** : enable ACK Response Mode
- **DISABLE** : disable ACK Response Mode

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

Register CECRCR1< CECACKDIS> is modified by this function.

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

4.3.3.20 CEC_SetNoiseCancellation

Set the Noise Cancellation Mode of CEC.

Prototype:
Result
CEC_SetNoiseCancellation(CEC_LowCancellation ***LowCancellation***,
CEC_HighCancellation ***HighCancellation***)

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

- **CEC_LOW_CANCELLATION_1**: 1 cycle,
- **CEC_LOW_CANCELLATION_2**: 2 cycles,
- **CEC_LOW_CANCELLATION_3**: 3 cycles,
- **CEC_LOW_CANCELLATION_4**: 4 cycles,
- **CEC_LOW_CANCELLATION_5**: 5 cycles,
- **CEC_LOW_CANCELLATION_6**: 6 cycles,
- **CEC_LOW_CANCELLATION_7**: 7 cycles,
- **CEC_LOW_CANCELLATION_8**: 8 cycles.

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

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

Description:
The noise cancellation time is configurable with the ***LowCancellation*** and ***HighCancellation*** by this function. You can configure the time to detect "1" and "0" respectively. It is considered as noise if "1"s or "0"s of the same number as the specified value are not sampled. Please refer to the "Preconfiguration (2) Noise Cancellation Time of CEC" in the datasheet.
When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.
Registers CECRCR1< CECHNC>, CECRCR1< CECLNC> are modified by this function.

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

4.3.3.21 CEC_SetCycleConfig

Set the cycle error detected configuration of CEC

Prototype:
Result
CEC_SetCycleConfig(CEC_CycleMin ***CycleMin***,
CEC_CycleMax ***CycleMax***)

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

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

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

- **CEC_CYCLE_MAX_0**: 2.75ms,
- **CEC_CYCLE_MAX_1**: 2.75ms+1cycles,
- **CEC_CYCLE_MAX_2**: 2.75ms+2cycles,
- **CEC_CYCLE_MAX_3**: 2.75ms+3cycles,
- **CEC_CYCLE_MAX_4**: 2.75ms-1cycles,
- **CEC_CYCLE_MAX_5**: 2.75ms-2cycles,
- **CEC_CYCLE_MAX_6**: 2.75ms-3cycles,
- **CEC_CYCLE_MAX_7**: 2.75ms-4cycles.

Description:
Call this function to detect a cycle error.
You can specify the time to detect a cycle error for each sampling clock cycle between the ranges of -4 to +3 cycles from the maximum or minimum time set in the CEC standard. Detecting an error during data reception causes an error interrupt, and CEC waits for the next start bit. The received data is discarded.
When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.

Registers CECRCR1< CECMIN>, CECRCR1< CECMAX> are modified by this function.

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

4.3.3.22 CEC_SetDataValidTime

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

Prototype:
Result
CEC_SetDataValidTime(CEC_ValidTime ***ValidTime***)

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

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

Description:
Call this function for configuring the point of determining the data as “0” or “1”.
You can specify it per two sampling clock cycles between the ranges of + or - 6 cycles with approx. 1.05 ms from the bit start point. Please refer to the “Preconfiguration (4) Point of Determining Data Time of CEC” in the datasheet.
When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.
Register CECRCR1< CECDAT> is modified by this function.

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

4.3.3.23 CEC_SetTimeOutMode

Specify the time to determine a timeout.

Prototype:

Result
CEC_SetTimeOutMode(CEC_TimeOut ***TimeOut***)

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

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

Description:
Call this function to specify the time to determine a time out.
This is used when the setting of a receive error interrupt suspension, which is specified in set error interrupt is suspended.
When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.
Register CECRCR1< CECTOUT> is modified by this function.

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

4.3.3.24 CEC_SetRxErrINTSuspend

Specify if a receive error interrupt is suspended or not.

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

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

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

Description:
Call this function to specify if a reception error interrupt (maximum cycle error, buffer overrun and waveform error) is suspended or not. Setting **ENABLE** generates no interrupt at the error detection.
When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.
Register CECRCR1< CECRIHLD> is modified by this function.

Return:

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

4.3.3.25 CEC_SetSnoopMode

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

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

Parameters:
NewState: The state of snoop mode enable or not.

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

Description:
By calling this function, you can specify whether the data should be received or not when destination address does not correspond with the address set in the logical address register. In this case, the data is received as usual, and an interrupt is generated by detecting an error. However, an ACK response of neither the header block nor the data block is sent. A broadcast message is received regardless of the Snoop Mode.
When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.
Register CECRCR1< CECOTH> is modified by this function.

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

4.3.3.26 CEC_SetRxDetectWaveConfig

Specify waveform error detection range

Prototype:
Result
CEC_SetRxDetectWaveConfig(
CEC_Logical1RisingTimeMin ***Logical1RisingTimeMin***,
CEC_Logical1RisingTimeMax ***Logical1RisingTimeMax***,
CEC_Logical0RisingTimeMin ***Logical0RisingTimeMin***,
CEC_Logical0RisingTimeMax ***Logical0RisingTimeMax***)

Parameters:

Logical1RisingTimeMin: The fastest rising timing of logical "1" determined as proper. This parameter can be one of the following values:

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

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

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

Logical0RisingTimeMin: The fastest rising timing of logical "0" determined as proper. This parameter can be one of the following values:

- **CEC_LOGICAL_0_RISING_TIME_MIN_0**: 1.3ms,
- **CEC_LOGICAL_0_RISING_TIME_MIN_1**: 1.3ms -1cycle,
- **CEC_LOGICAL_0_RISING_TIME_MIN_2**: 1.3ms -2cycles,
- **CEC_LOGICAL_0_RISING_TIME_MIN_3**: 1.3ms -3cycles,
- **CEC_LOGICAL_0_RISING_TIME_MIN_4**: 1.3ms -4cycles,
- **CEC_LOGICAL_0_RISING_TIME_MIN_5**: 1.3ms -5cycles,
- **CEC_LOGICAL_0_RISING_TIME_MIN_6**: 1.3ms -6cycles,
- **CEC_LOGICAL_0_RISING_TIME_MIN_7**: 1.3ms -7cycles.

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

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

Description:

Call this function to detect an error when a received waveform is out of the defined tolerance range,
You can specify the detection time with ***Logical1RisingTimeMin***, ***Logical1RisingTimeMax***, ***Logical0RisingTimeMin*** and ***Logical0RisingTimeMax***.
Please refer to the “Preconfiguration (10) Waveform Error Detection of CEC” in the datasheet.
When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.
Register CECRCR3 is modified by this function.

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

4.3.3.27 CEC_SetRxStartBitWaveConfig

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

Prototype:
Result
CEC_SetRxStartBitWaveConfig(
CEC_StartBitRisingTimeMin ***RisingTimeMin***,
CEC_StartBitRisingTimeMax ***RisingTimeMax***,
CEC_StartBitCycleMin ***CycleMin***,
CEC_StartBitCycleMax ***CycleMax***)

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

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

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

- **CEC_START_BIT_RISING_TIME_MAX_0**: 3.9ms,
- **CEC_START_BIT_RISING_TIME_MAX_1**: 3.9ms+1cycle,
- **CEC_START_BIT_RISING_TIME_MAX_2**: 3.9ms+2cycles,
- **CEC_START_BIT_RISING_TIME_MAX_3**: 3.9ms+3cycles,

- **CEC_START_BIT_RISING_TIME_MAX_4**: 3.9ms+4cycles,
- **CEC_START_BIT_RISING_TIME_MAX_5**: 3.9ms+5cycles,
- **CEC_START_BIT_RISING_TIME_MAX_6**: 3.9ms+6cycles,
- **CEC_START_BIT_RISING_TIME_MAX_7**: 3.9ms+7cycles.

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

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

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

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

Description:

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

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

Please refer to the “Preconfiguration (9) Start Bit Detection of CEC” in the datasheet.

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

Register CECRCR2 is modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.3.3.28 CEC_SetRxWaveErrDetect

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

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

Parameters:
NewState: The state of Waveform error detection Mode.
- **ENABLE:** enable the waveform error detection mode.
- **DISABLE:** disable the waveform error detection mode.

Description:
Enable or disable the function that detects a received waveform does not identical to the one defined and generates waveform error interrupt.
When the CEC reception is enabled or the transmission is in progress, the function will return **ERROR** and do nothing.
Register CECRCR3< CECWAVEN> is modified by this function.

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

4.3.3.29 CEC_SetTxWaveConfig

Specify the waveform parameter for transmit

Prototype:
Result
CEC_SetTxWaveConfig(CEC_TxDataBitCycle ***DataBitCycle*** ,
CEC_TxDataBitRisingTime ***DataBitRisingTime***,
CEC_TxStartBitCycle ***StartBitCycle***,
CEC_TxStartBitRisingTime ***StartBitRisingTime***)

Parameters:
DataBitCycle: the cycle of a data bit. parameter can be one of the following values:
- **CEC_TX_DATA_BIT_CYCLE_0**: RV (Reference value: 2.4ms approx.),
- **CEC_TX_DATA_BIT_CYCLE_1**: RV-1cycle,
- **CEC_TX_DATA_BIT_CYCLE_2**: RV-2cycles,
- **CEC_TX_DATA_BIT_CYCLE_3**: RV-3cycles,
- **CEC_TX_DATA_BIT_CYCLE_4**: RV-4cycles,
- **CEC_TX_DATA_BIT_CYCLE_5**: RV-5cycles,
- **CEC_TX_DATA_BIT_CYCLE_6**: RV-6cycles,
- **CEC_TX_DATA_BIT_CYCLE_7**: RV-7cycles,
- **CEC_TX_DATA_BIT_CYCLE_8**: RV-8cycles,
- **CEC_TX_DATA_BIT_CYCLE_9**: RV-9cycles,
- **CEC_TX_DATA_BIT_CYCLE_10**: RV-10cycles,

- **CEC_TX_DATA_BIT_CYCLE_11**: RV-11cycles,
- **CEC_TX_DATA_BIT_CYCLE_12**: RV-12cycles,
- **CEC_TX_DATA_BIT_CYCLE_13**: RV-13cycles,
- **CEC_TX_DATA_BIT_CYCLE_14**: RV-14cycles,
- **CEC_TX_DATA_BIT_CYCLE_15**: RV-15cycles.

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

- **CEC_TX_DATA_BIT_RISING_TIME_0**: RV (logical "1": 0.6 ms approx., logical "0": 1.5 ms approx),
- **CEC_TX_DATA_BIT_RISING_TIME_1**: RV-1cycle,
- **CEC_TX_DATA_BIT_RISING_TIME_2**: RV-2cycles,
- **CEC_TX_DATA_BIT_RISING_TIME_3**: RV-3cycles,
- **CEC_TX_DATA_BIT_RISING_TIME_4**: RV-4cycles,
- **CEC_TX_DATA_BIT_RISING_TIME_5**: RV-5cycles,
- **CEC_TX_DATA_BIT_RISING_TIME_6**: RV-6cycles,
- **CEC_TX_DATA_BIT_RISING_TIME_7**: RV-7cycles.

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

- **CEC_TX_START_BIT_CYCLE_0**: RV (4.5ms approx.),
- **CEC_TX_START_BIT_CYCLE_1**: RV-1cycle,
- **CEC_TX_START_BIT_CYCLE_2**: RV-2cycles,
- **CEC_TX_START_BIT_CYCLE_3**: RV-3cycles,
- **CEC_TX_START_BIT_CYCLE_4**: RV-4cycles,
- **CEC_TX_START_BIT_CYCLE_5**: RV-5cycles,
- **CEC_TX_START_BIT_CYCLE_6**: RV-6cycles,
- **CEC_TX_START_BIT_CYCLE_7**: RV-7cycles.

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

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

Description:

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

Please refer to the “Preconfiguration (3) Adjusting Transmission Waveform of CEC” in the datasheet.
When the transmission is in progress, the function will return **ERROR** and do nothing.
Register CECTCR is modified by this function.

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

4.3.3.30 CEC_SetTxBroadcast

Set message as a broadcast message or not.

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

Parameters:
NewState: The state of Broadcast mode.
- **ENABLE:** enable the Broadcast mode.
- **DISABLE:** disable the Broadcast mode.

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

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

4.3.3.31 CEC_SetBusFreeTime

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

Prototype:
Result
CEC_SetBusFreeTime (CEC_BusFree ***BusFree***)

Parameters:

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

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

Description:

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

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

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

Register CECTCR< CECFREE> is modified by this function.

Return:

SUCCESS means set successful.

ERROR means set failed and do nothing.

4.3.4 Data Structure Description

4.3.4.1 CEC_DataTypeDef

Data Fields:

uint8_t

Data Reads one byte of data received. The bit 7 is the MSB.

CEC_ACKState

ACKBit The received ACK bit which can be:

- **CEC_ACK** means the ACK bit is “1”
- **CEC_NO_ACK** means the ACK bit is “0”

CEC_EOMBit

EOMBit The received EOM bit which can be:

- **CEC_EOM** means the EOM bit is “1”
- **CEC_NO_EOM** means the EOM bit is “0”

4.3.4.2 CEC_AddrListTypeDef

Data Fields:

uint8_t

AddrNumber is the number of logical address.

CEC_LogicalAddr

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

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

4.3.5 Programming Example

This is a simple application based on the TMPM33x Peripheral Driver (CEC).

The application works in the follow Environment:

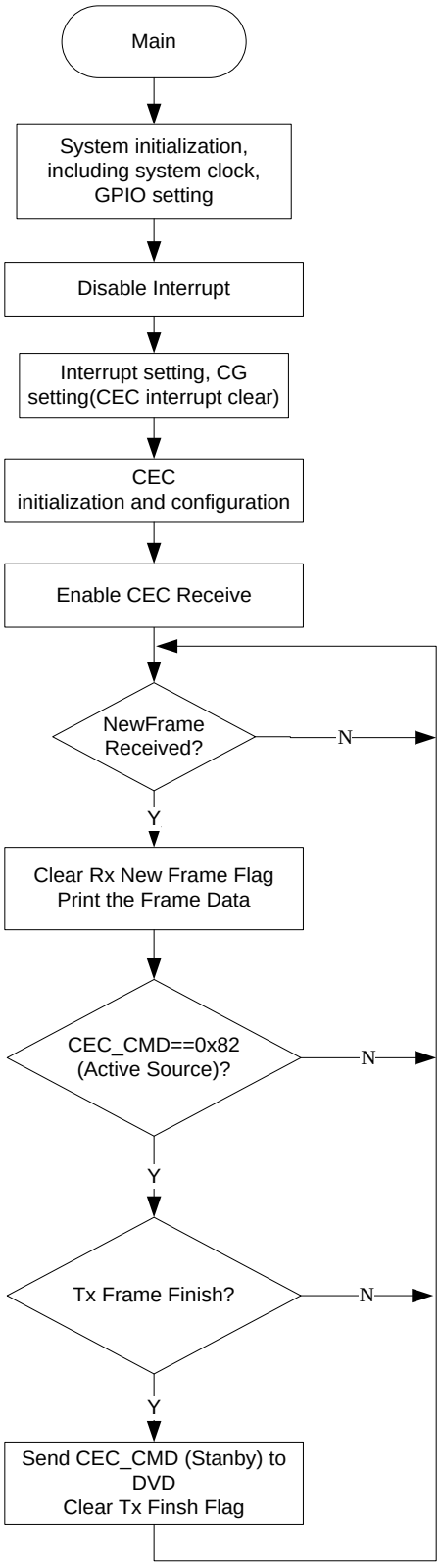
- IAR Embedded Workbench for ARM 5.40
- TMPM330 Demo board(M4M330B-2)
- DVD Player which support CEC (Philips DVP5996K/93)
- PC (OS: Windows XP)

And the process described as follow:

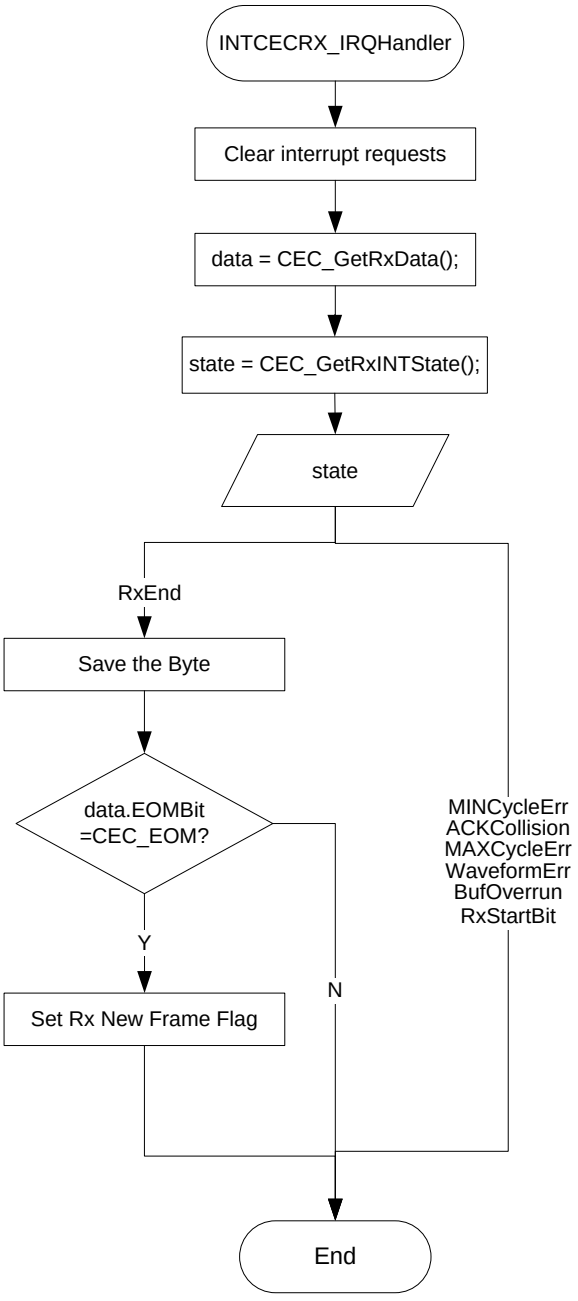
1. The TMPM330 Demo board receives the CEC message from DVD player
2. Print the CEC message data on the Terminal I/O of IAR Embedded Workbench
3. When the TMPM330 Demo board receives the “Active_Source” (CEC command) from DVD player (When DVD player turn on or DVD play a source, the command will be sent), the TMPM330 Demo board send “Standby” command to DVD player.
4. DVD player should be change to standby mode.

4.3.5.1 Flow Chart

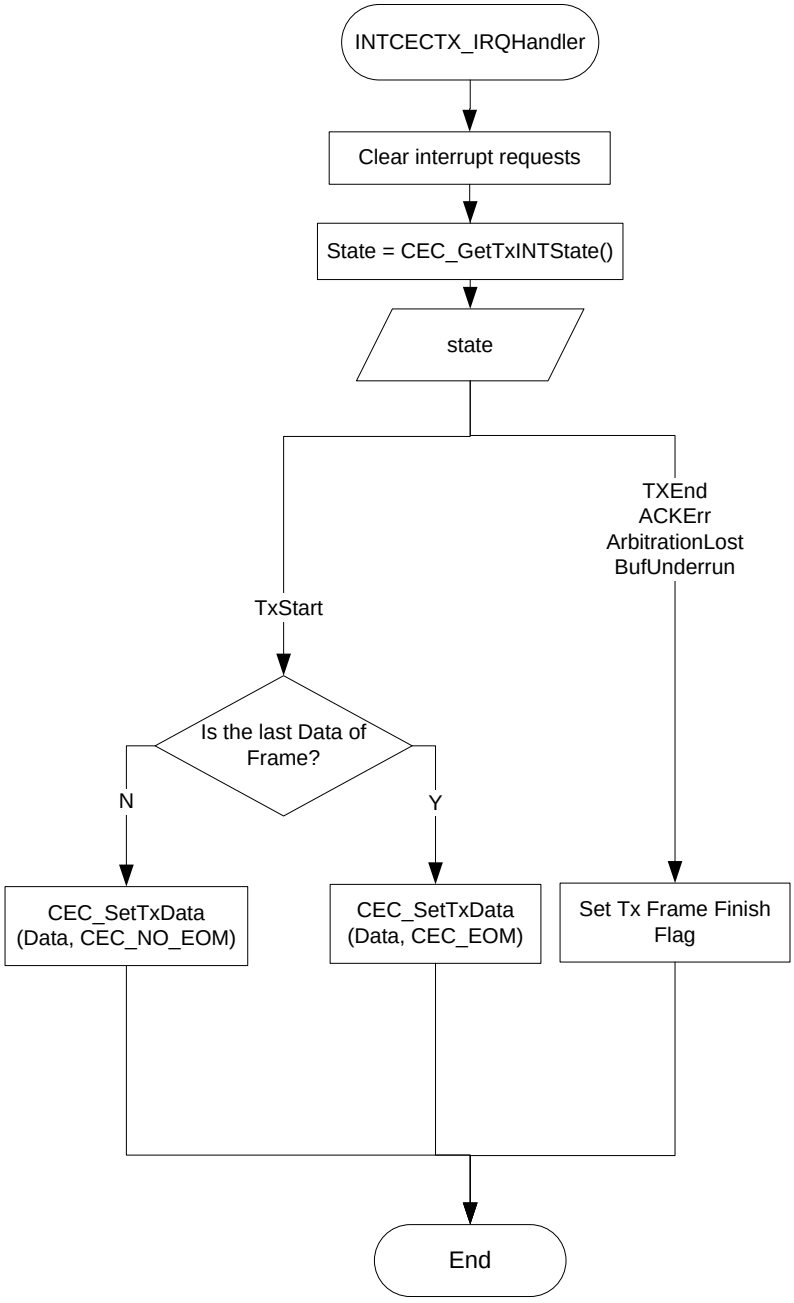
4.3.5.1.1 Main Function



4.3.5.1.2 CEC Receive Interrupt



4.3.5.1.3 CEC Transmission Interrupt



4.3.5.2 Code and Explanation for the Example

4.3.5.2.1 Main Function

At first, initialize and configure CEC. For example:

```
CEC_Enable();
CEC_SWReset()
CEC_DefaultConfig()
```

Then Set the logical address and enable reception of CEC. For example:

```
CEC_SetLogicalAddr(CEC_TV);
CEC_SetRxCtrl(ENABLE);
```

After the setting above send the first byte of a frame. For example

```
if(CEC_BROADCAST==Destination){
      CEC_SetTxBroadcast(ENABLE);
}else{
      CEC_SetTxBroadcast(DISABLE);
}
CEC_SetTxData(CEC_Data[0], CEC_NO_EOM);
CEC_StartTx();
```

4.3.5.2.2 CEC Receive Interrupt

In the interrupt function: INTCECRX_IRQHandler

First get interrupt state:

```
CEC_RxINTState state;
state = CEC_GetRxINTState();
```

Then get Receive data:

```
CEC_DataTypeDef data;
data = CEC_GetRxData();
```

4.3.5.2.3 CEC Transmission Interrupt

In the interrupt function: INTCECTX_IRQHandler

First get interrupt state:

```
CEC_TxINTState state;
state = CEC_GetTxINTState ();
```

It the next data isn't the last data of a frame then send next data:

```
CEC_SetTxData(CEC_Data[current_num],CEC_NO_EOM);
```

It the next data is the last data of a frame then send last data as follow:

```
CEC_SetTxData(CEC_Data[current_num],CEC_EOM);
```

5. CG

5.1 Overview

The CG API provides a set of functions for using the TMPM33**x** CG modules as the following:

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

This driver is contained in TX03_Periph_Driver\src\tmpm33**x**_cg.c(*), with TX03_Periph_Driver\inc\tmpm33**x**_cg.h(*) containing the API definitions for use by applications.
***Note: x** can be 0,2,3

The following symbols fosc, fs, fpll, fc, fgear, fsys, fperiph, ΦT0 are used for kinds of clock in CG. Please refer to the clock system diagram in section “6.3.3 Clock System Block Diagram” of the datasheet for their meaning.

fosc : Clock input from the X1 and X2 pins
fs : Clock input from the XT1 and XT2 (low-speed clock)
fpll : Clock quadrupled by PLL
fc : Clock specified by PLLSEL<PLLSEL> (high-speed clock)
fgear : Clock specified by SYSCR1<GEAR2:0>
fsys : Clock specified by CKSEL<SYSCK> (system clock)
fperiph : Clock specified by SYSCR1<FPSEL2:0>
ΦT0 : Clock specified by SYSCR1<PRCK2:0> (prescaler clock)

5.2 API Functions

5.2.1 Function List

- void CG_SetFgearLevel(CG_DivideLevel ***DivideFgearFromFc***)
- CG_DivideLevel CG_GetFgearLevel(void)
- Result CG_SetPhiT0Level(CG_DivideLevel ***DividePhiT0FromFc***)
- CG_DivideLevel CG_GetPhiT0Level(void)
- void CG_SetSCOUTSrc(CG_SCOUTSrc ***Source***)
- CG_SCOUTSrc CG_GetSCOUTSrc(void)
- Result CG_SetWarmUpTime(CG_WarmUpSrc ***Source***, CG_WarmUpTime ***Time***)
- void CG_StartWarmUp(void)
- WorkState CG_GetWarmUpState(void)
- Result CG_SetPLL(FunctionalState ***NewState***)
- FunctionalState CG_GetPLLState(void)
- Result CG_SetFosc(FunctionalState ***NewState***)
- FunctionalState CG_GetFoscState(void)
- Result CG_SetFs(FunctionalState ***NewState***)
- FunctionalState CG_GetFsState(void)
- void CG_SetSTBYMode(CG_STBYMode ***Mode***)
- CG_STBYMode CG_GetSTBYMode(void)
- void CG_SetExitStopModeFosc(FunctionalState ***NewState***)
- FunctionalState CG_GetExitStopModeFoscState(void)
- void CG_SetExitStopModeFs(FunctionalState ***NewState***)
- FunctionalState CG_GetExitStopModeFsState(void)
- void CG_SetPinStateInStopMode(FunctionalState ***NewState***)
- FunctionalState CG_GetPinStateInStopMode(void)
- Result CG_SetFcSrc(CG_FcSrc ***Source***)
- CG_FcSrc CG_GetFcSrc(void)
- Result CG_SetFsysSrc(CG_FsysSrc ***Source***)
- CG_FsysSrc CG_GetFsysSrc(void)
- void CG_SetSTBYReleaseINTSrc(CG_INTSrc ***INTSource***,
  CG_INTActiveState ***ActiveState***,
  FunctionalState ***NewState***)
- CG_INTActiveState CG_GetSTBYReleaseINTState(CG_INTSrc ***INTSource***)
- void CG_ClearINTReq(CG_INTSrc ***INTSource***)
- CG_NMIFactor CG_GetNMIFlag(void)
- CG_ResetFlag CG_GetResetFlag(void)

◆ 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_SetPhiT0Level(), CG_GetPhiT0Level(), CG_SetSCOUTSrc(), CG_GetSCOUTSrc(), CG_SetWarmUpTime(), CG_StartWarmUp(), CG_GetWarmUpState(), CG_SetPLL(), CG_GetPLLState(), CG_SetFosc(), CG_GetFoscState(), CG_SetFs(), CG_GetFsState(), CG_SetFcSrc(), CG_GetFcSrc(), CG_SetFsysSrc(),CG_GetFsysSrc(),

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

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

5.2.2 Function Documentation

5.2.2.1 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

Description :
This function will set the dividing level between clock fgear and fc. If the value of input parameter ***DivideFgearFromFc*** is beyond the above ones, it will enter error parameter process.

Return:
None.

5.2.2.2 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_UNKNOWN: invalid data is read

5.2.2.3 CG_SetPhiT0Level

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

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

Parameters:
DividePhiT0FromFc: divide level between PhiT0(ΦT0) and fc.
This parameter can be one of the following values:
- **CG_DIVIDE_1**: ΦT0 = fc
- **CG_DIVIDE_2**: ΦT0 = fc/2
- **CG_DIVIDE_4**: ΦT0 = fc/4

- **CG_DIVIDE_8**: ΦT0 = fc/8
- **CG_DIVIDE_16**: ΦT0 = fc/16
- **CG_DIVIDE_32**: ΦT0 = fc/32
- **CG_DIVIDE_64**: ΦT0 = fc/64
- **CG_DIVIDE_128**: ΦT0 = fc/128
- **CG_DIVIDE_256**: ΦT0 = fc/256

Description:

This function will set the dividing level between clock PhiT0(ΦT0) and fc

This function will not change the value of fc and fgear. And the setting of clock must match the following condition:

DividePhiT0FromFc <= ***DivideFgearFromFc*** + **CG_DIVIDE_32**

Otherwise the API will return **ERROR**

Return:

SUCCESS means the setting has been written to registers successfully.

ERROR means the setting has not been written to registers**.**

5.2.2.4 CG_GetPhiT0Level

Get the dividing level between clock ΦT0 and fc

Prototype:

CG_DivideLevel

CG_GetPhiT0Level(void)

Parameters:

None

Description:

This function will get the dividing level between clock ΦT0 and fc.

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

Return:

Dividing level between clock ΦT0 and fc, the value will be one of the following:

CG_DIVIDE_1: ΦT0 = fc

CG_DIVIDE_2: ΦT0 = fc/2

CG_DIVIDE_4: ΦT0 = fc/4

CG_DIVIDE_8: ΦT0 = fc/8

CG_DIVIDE_16: ΦT0 = fc/16

CG_DIVIDE_32: ΦT0 = fc/32

CG_DIVIDE_64: ΦT0 = fc/64

CG_DIVIDE_128 : ΦT0 = fc/128

CG_DIVIDE_256 : ΦT0 = fc/256
CG_DIVIDE_UNKNOWN : invalid data is read

5.2.2.5 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

5.2.2.6 CG_GetSCOUTSrc

Get the clock source of SCOUT output.

Prototype:
CG_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

5.2.2.7 CG_SetWarmUpTime

Set the warm up time

Prototype:
Result
CG_SetWarmUpTime (CG_WarmUpSrc ***Source***,
CG_WarmUpTime ***Time***)

Parameters:
Source: select source of warm-up counter
- **CG_WARM_UP_SRC_X1**: fosc is selected as timer source
- **CG_WARM_UP_SRC_XT1**: fs is selected as timer source

Time: select count time of warm-up timer

Description:
This parameter ***Time*** can be one of the following values:

For ***Source*** = **CG_WARM_UP_SRC_X1** (fosc):
CG_WARM_UP_TIME_NONE, CG_WARM_UP_TIME_EXP_10,
CG_WARM_UP_TIME_EXP_11, CG_WARM_UP_TIME_EXP_12,
CG_WARM_UP_TIME_EXP_13, CG_WARM_UP_TIME_EXP_14,
CG_WARM_UP_TIME_EXP_15 or CG_WARM_UP_TIME_EXP_16,

For ***Source*** = **CG_WARM_UP_SRC_XT1** (fs):
CG_WARM_UP_TIME_NONE, CG_WARM_UP_TIME_EXP_6,
CG_WARM_UP_TIME_EXP_7, CG_WARM_UP_TIME_EXP_8,
CG_WARM_UP_TIME_EXP_15, CG_WARM_UP_TIME_EXP_16,
CG_WARM_UP_TIME_EXP_17 or CG_WARM_UP_TIME_EXP_18

If the value of parameter ***Time*** is not set correctly, the API will return **ERROR**
Please refer to the following table for the detailed warm up time

Table Warm-up Time(fosc=10MHz, fs=32.768kHz)

Warm-up time	***Source*** = **CG_WARM_UP_SRC_X1**	***Source*** = **CG_WARM_UP_SRC_XT1**
CG_WARM_UP_TIME_NONE	Without warm-up	Without warm-up
CG_WARM_UP_TIME_EXP_6	-	1.953 ms
CG_WARM_UP_TIME_EXP_7	-	3.906 ms
CG_WARM_UP_TIME_EXP_8	-	7.813 ms

CG_WARM_UP_TIME_EXP_10	102.4 us	-
CG_WARM_UP_TIME_EXP_11	204.8 us	-
CG_WARM_UP_TIME_EXP_12	409.6 us	-
CG_WARM_UP_TIME_EXP_13	819.2 us	-
CG_WARM_UP_TIME_EXP_14	1.638 ms	-
CG_WARM_UP_TIME_EXP_15	3.277 ms	1.0 s
CG_WARM_UP_TIME_EXP_16	6.554 ms	2.0 s
CG_WARM_UP_TIME_EXP_17	-	4.0 s
CG_WARM_UP_TIME_EXP_18	-	8.0 s

Return:
SUCCESS: the setting has been written to the register successfully.
ERROR: the setting has not been written to the register.

5.2.2.8 CG_StartWarmUp

Start warm-up timer

Prototype:
void
CG_StartWarmUp (void)

Parameters:
None

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

Return:
None

5.2.2.9 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 middle or completed

Example of using warm up timer:

```
/* set up warm time 100us*/
CG_SetWarmUpTime(CG_WARM_UP_SRC_X1,
                        CG_WARM_UP_TIME_EXP_10);
/* 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

5.2.2.10 CG_SetPLL

Enable or disable the PLL circuit.

Prototype:

Result

CG_SetPLL (FunctionalState ***NewState***)

Parameters:

NewState:

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

Description:

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

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

Return:

SUCCESS: operation is finished successfully

ERROR: operation is not done

5.2.2.11 CG_GetPLLState

Get the state of PLL circuit

Prototype:

FunctionalState

CG_GetPLLState (void)

Parameters:
None

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

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

5.2.2.12 CG_SetFosc

Enable or disable the high-speed oscillator (X1)

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

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

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

5.2.2.13 CG_GetFoscState

Get the state of the high-speed oscillator(X1)

Prototype:
FunctionalState
CG_GetFoscState (void)

Parameters:
None

Description:
This function will get the state of the high-speed oscillator (X1).

Return:
The state of X1
ENABLE: X1 is enabled.
DISABLE: X1 is not enabled.

5.2.2.14 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) as the input parameter.
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

5.2.2.15 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 not enabled.

5.2.2.16 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, CEC (only for reception) and RMC operate
- **CG_STBY_MODE_IDLE**: IDLE mode. Only the CPU is stopped in this mode

Description:
This function will change the setting of the standby mode to enter when using standby instruction.
Please refer to the section “CG Programming Example” for the information of how to enter and exit low power consumption mode.

Return:
None

5.2.2.17 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.

5.2.2.18 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

5.2.2.19 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

5.2.2.20 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

5.2.2.21 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

5.2.2.22 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 6-5 Pin States in the STOP Mode” in the TMPM33X datasheet.

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

Return:
None

5.2.2.23 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

5.2.2.24 CG_SetFcSrc

Set the clock source of fc

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

Parameters:

Source: the source for fc

This parameter can be one of the following values:

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

Description:

This function will set the clock source of fc.

The following conditions should be matched before calling this API

a) high-speed oscillator is set to on

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

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

Return:

SUCCESS: set clock souce for fc successfully

ERROR: clock source of fc is not changed.

5.2.2.25 CG_GetFcSrc

Get the clock source of fc

Prototype:

CG_FcSrc

CG_GetFosc (void)

Parameters:

None

Description:

This function will get the clock source of fc.

Return:

The clock source of fc

The value returned can be one of the following values:

CG_FC_SRC_FOSC: fc source is set to fosc

CG_FC_SRC_FPLL: fc source is set to fpll

5.2.2.26 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

5.2.2.27 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.

5.2.2.28 CG_SetSTBYReleaseINTSrc

Set the INT source for releasing low power mode

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

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

- **CG_INT_SRC_0** : INT0
- **CG_INT_SRC_1** : INT1
- **CG_INT_SRC_2** : INT2
- **CG_INT_SRC_3** : INT3
- **CG_INT_SRC_4** : INT4
- **CG_INT_SRC_5** : INT5(For M330/333)
- **CG_INT_SRC_CEC_RX** : CEC reception interrupt (For M330/M332)
- **CG_INT_SRC_RMC_RX_0** : reception interrupt of RMC Channel 0 (For M330/332)
- **CG_INT_SRC_RTC** : RTC interrupt
- **CG_INT_SRC_6** : INT6(For M330/333)
- **CG_INT_SRC_7** : INT7(For M330/333)
- **CG_INT_SRC_RMC_RX_1** : reception interrupt of RMC channel 1(For M330)
- **CG_INT_SRC_CEC_TX** : CEC transmission interrupt(For M330/332)

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_CEC_RX**, **CG_INT_SRC_RMC_RX_0** and **CG_INT_SRC_RMC_RX_1**, 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 and **CG_INT_SRC_CEC_TX**, only "falling" state (**CG_INT_ACTIVE_STATE_FALLING**) will be set to the register, no matter what the value of ***ActiveState*** is.

Return:
None

5.2.2.29 CG_GetSTBYReleaseINTState

Get the active state of INT source for standby clear request

Prototype:
CG_INTActiveState
CG_GetSTBYReleaseINTState(CG_INTSrc ***INTSource***)

Parameters:
INTSource: select the release INT source
This parameter can be one of the following values:
CG_INT_SRC_0, CG_INT_SRC_1, CG_INT_SRC_2,
CG_INT_SRC_3, CG_INT_SRC_4,
CG_INT_SRC_5(For M330/333)**,**
CG_INT_SRC_CEC_RX (For M330/332)**,**
CG_INT_SRC_RMC_RX_0 (For M330/332)**,**
CG_INT_SRC_RTC, CG_INT_SRC_6 (For M330/333)**,**
CG_INT_SRC_7 (For M330/333)**,**
CG_INT_SRC_RMC_RX_1 (For M330)
or CG_INT_SRC_CEC_TX (For M330/332)

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

5.2.2.30 CG_ClearINTReq

Clear the INT request of releasing standby mode

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

Parameters:
INTSource: select the release INT source
This parameter can be one of the following values:
CG_INT_SRC_0, CG_INT_SRC_1, CG_INT_SRC_2,
CG_INT_SRC_3, CG_INT_SRC_4,
CG_INT_SRC_5(For M330/333)**,**
CG_INT_SRC_CEC_RX (For M330/332)**,**
CG_INT_SRC_RMC_RX_0 (For M330/332)**,**
CG_INT_SRC_RTC, CG_INT_SRC_6 (For M330/333)**,**
CG_INT_SRC_7 (For M330/333)**,**
CG_INT_SRC_RMC_RX_1 (For M330)
or CG_INT_SRC_CEC_TX (For M330/332)

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

Return:
None

5.2.2.31 CG_GetNMIFlag

Get the NMI flag, which shows what triggered NMI

Prototype:
CG_NMIFactor
CG_ GetNMIFlag (void)

Parameters:
None

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

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

5.2.2.32 CG_GetResetFlag

Get the type of reset reason 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 , each bit is described as the following:
PowerOn(Bit0): In initial reset state right after power-on.
ResetPin(Bit1): Reset from RESET pin
WDTReset(Bit2): Reset from WDT
DebugReset(Bit4): Reset from SYSRESETREQ

5.2.3 Data Structure Description

None

5.2.4 Programming Example

This is a simple application based on the TMPM33x Peripheral Driver (CG), which will switch MCU among different modes.

5.2.4.1 Normal setup for CG (after Reset)

The following codes are just an example for setting CG in Normal mode. It is supposed that the high-speed oscillation is 10MHz,.

Example code is as the following:

```
/* set fgear = fc/2 */
CG_SetFgearLevel(CG_DIVIDE_2);

/* set fperi = fgear,fpreclk = fperi/32 */
CG_SetPhiT0Level(CG_DIVIDE_64);
```

```
/* set SCOUT source to ΦT0*/
CG_SetSCOUTSrc(CG_SCOUT_SRC_PHIT0);

/* enable high-speed oscillation*/
CG_SetFosc(ENABLE);

/* enable low-speed oscillation*/
CG_SetFs(ENABLE);

/* set low power consumption mode Sleep*/
CG_SetSTBYMode (CG_STBY_MODE_SLEEP);

/* set high-speed oscillation to be enabled after releasing stop mode*/
CG_SetExitStopModeFosc(ENABLE);

/* set pin status in stop mode to "active"*/
CG_SetPinStateInStopMode(ENABLE);

/* set up pll and wait 200us for pll to warm up , set fc source to fpll*/
CG_EnableClkMulCircuit();    /*this module refer to    Programming Example */

/* set system clock to fgear */
CG_SetFsysSrc(CG_FSYS_SRC_FGEAR);
```

5.2.4.2 Setup to enter slow mode

```
/* Set CG module: Normal ->Slow mode */
/*... Add code here to disable modules (except CPU,RTC,I/O,CEC and RMC) if it has been
enabled earlier...*/
/* then switch system clock to fs*/
CG_SetFsysSrc(CG_FSYS_SRC_FS);

/* Then stop high-speed oscillation*/
CG_SetFosc(DISABLE);
```

5.2.4.3 Setup to enter sleep mode

```
/* Set CG Module : Slow ->Sleep mode*/
_disable_irq();
/* set source (RTC interrupt) to exit sleep mode*/
CG_SetSTBYReleaseINTSrc(CG_INT_SRC_RTC,
                        CG_INT_ACTIVE_STATE_FALLING,
                        ENABLE);
```

```
   / * enable RTC interrupt */
NVIC_ClearPendingIRQ(INTRTC_IRQn);
NVIC_EnableIRQ(INTRTC_IRQn);

   /*clear interrupt request*/
   CG_ClearINTReq(CG_INT_SRC_RTC);

/* The following code will set up real-time clock and alarm, and generate RTC interrupt after 2 minutes*/
   /* set current time*/
TSB_RTC->RESTR = 0xf0U;
TSB_RTC->PAGER = 0x00U;
TSB_RTC->SECR = 0x00U;
TSB_RTC->MINR = 0x00U;
TSB_RTC->HOURR = 0x00U;
TSB_RTC->DAYR = 0x01U;
TSB_RTC->DATER = 0x01U;
TSB_RTC->MONTHR = 0x01U;
TSB_RTC->YEARR = 0x01U;

/* alarm disable, page =1*/
TSB_RTC->PAGER = 0x1U;
/* set alarm time = clock+2min */
TSB_RTC->MINR = 0x02U;
TSB_RTC->HOURR = 0x00U;
TSB_RTC->DAYR = 0x01U;
TSB_RTC->DATER = 0x01U;
TSB_RTC->MONTHR = 0x01U;

/* enable alarm*/
TSB_RTC->PAGER = 0x0cU;

/* enable RTC interrupt */
TSB_RTC->PAGER = 0x8cU;
_enable_irq();

/* enter sleep mode */
_WFI();
/* Interrupt is needed to exit sleep mode*/
/* wait 2 min to generate RTC interrupt */
```

5.2.4.4 Setup to switch from Slow Mode to Normal Mode

```
/* set CG module: Slow -> Normal */
```

```
/* start high-speed oscillation*/
CG_SetFosc(ENABLE);

/* wait 1.953ms for fosc to warm up*/
CG_SetWarmUpTime(CG_WARM_UP_SRCXT1, CG_WARM_UP_TIME_EXP_6);

/* start warm up*/
CG_StartWarmUp();

/* check whether warm up ends or not*/
while(CG_GetWarmUpState() == BUSY);

/* set system clock to clock-gear*/
CG_SetFsysSrc(CG_FSYS_SRC_FGEAR);
```

5.2.4.5 Sample module CG_EnableClkMulCircuit()

The following codes introduce how to enable multiple clock circuit

```
Result CG_EnableClkMulCircuit(void)
{
    Result retval = ERROR;
    retval = CG_SetPLL(ENABLE);
    WorkState st = BUSY;
    if (retval == SUCCESS) {
        /*set warm up time to about 200us */
        retval = CG_SetWarmUpTime(CG_WARM_UP_SRC_X1,
                                  CG_WARM_UP_TIME_EXP_11);
        if (retval == SUCCESS) {
            CG_StartWarmUp();
            /*wait warm up to end */
            do {
                st = CG_GetWarmUpState();
            } while (st != DONE);
            retval = CG_SetFcSrc(CG_FC_SRC_FPLL);
        } else {
            /*Do nothing */
        }
    } else {
        /*Do nothing */
    }
    return retval;
}
```

6. FC

6.1 Overview

TMPM330FDFG, TMPM333FDFG and TMPM332FWUG device contain one NANO flash and the size of TMPM330FDFG and TMPM333FDFG flash is 512Kbyte. The size of TMPM332FWUG 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 resisters is 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 (TMPM330FDFG, TMPM333FDFG and TMPM332FWUG) please refers to the MCU data sheet.

This driver is contained in \Libraries\TX03_Periph_Driver\src\tmpm33**x**_fc.c (*), with \Libraries\TX03_Periph_Driver \inc\tmpm33**x**_fc.h (*) containing the API definitions for use by applications.

***Note**: **x** can be 0,2,3.

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(FC_BlockNum ***BlockNum***)

6.2.2 Detailed Description

Functions listed above can be divided into two parts:

1) The security function restricts flash ROM data readout and debugging.
   FC_SetSecurityBit (), FC_GetSecurityBit().

2) The write/erase-protection function enables the internal flash to prohibit the writing and erasing operation for each block.
FC_GetBusyState (), FC_GetBlockProtectState().

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:
All the protection bits (the FLCS<BLPRO> bits) used for the write/erase-protection function are set to “1”, and the SECBIT <SECBIT> bit is set to”1”, this function is available.

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:

If the value of SECBIT <SECBIT> bit is"1" (**ENABLE**), then this function is available, if the value of SECBIT <SECBIT> bit is"0" (**DISABLE**), and then this function is not available.

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 terminated, it returns to the ready state and outputs "1" to accept the next command.

Return:
Status of the flash auto operation:
BUSY: Flash memory is in automatic operation.
DONE: Automatic operation is terminated.

6.2.3.4 FC_GetBlockProtectState

Get the block protection status.

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

Parameters:
BlockNum:The flash block number
- **FC_BLOCK_0** for block 0,
- **FC_BLOCK_1** for block 1,

- **FC_BLOCK_2** for block 2,
- **FC_BLOCK_3** for block 3,
- **FC_BLOCK_4** for block 4, this parameter is for M330FDFG and M333 FDFG,
- **FC_BLOCK_5** for block 5, this parameter is for M330FDFG and M333 FDFG.

Description:
Each of the protection bits (6 bits) 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, data cannot be written to it. About the block configuration of the flash memory, please refer to overview.

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

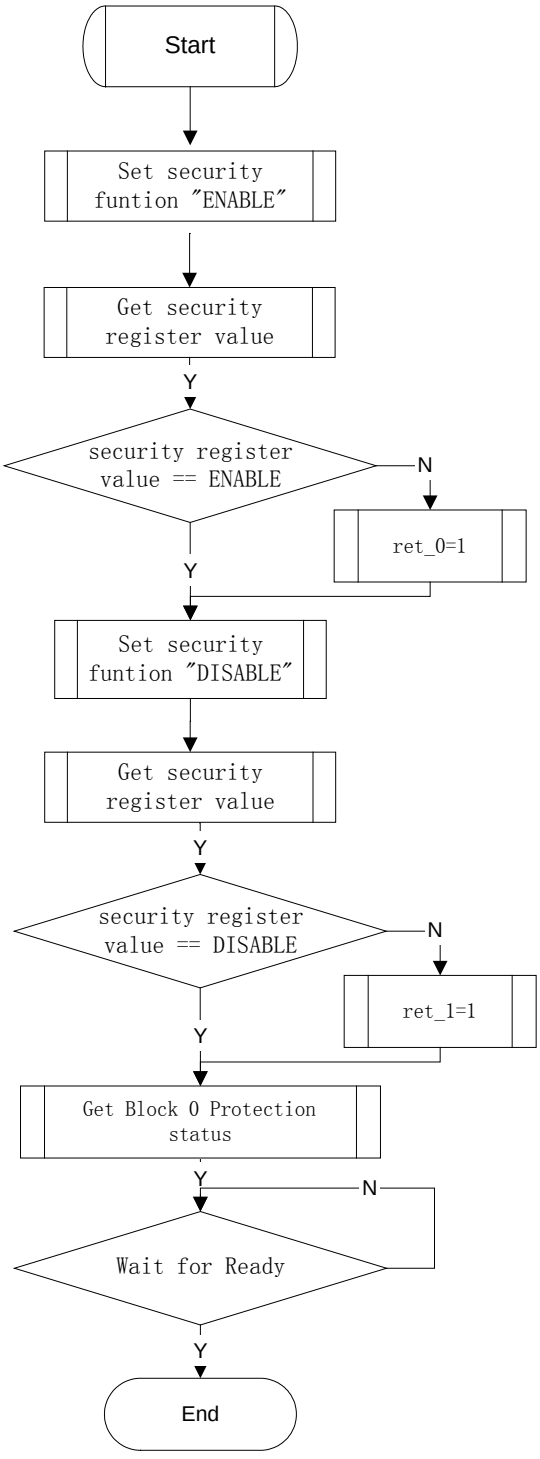
6.2.4 Data Structure Description

None

6.2.5 Programming Example

This is a simple application based on the TMPM33x Peripheral Driver (FC), and used for write and read the FC registers

6.2.5.1 Flow Chart



6.2.5.2 Code and Explanation for the Example

The sample source is shown as below:

```
#include "tmpm330_fc.h"

int main(void)
{
	FunctionalState tmp_value_sec0, tmp_value_sec1, tmp_value_block;
	int ret_0 = 0U, ret_1 = 0U;

	while (1) {
		/* Set security function "ENABLE" */
		FC_SetSecurityBit(ENABLE);

		/* Get security register value */
		tmp_value_sec0 = FC_GetSecurityBit();
		if (tmp_value_sec0 != ENABLE) {
			ret_0 = 1U;
		}

		/* Set security function "DISABLE" */
		FC_SetSecurityBit(DISABLE);

		/* Get security register value */
		tmp_value_sec1 = FC_GetSecurityBit();
		if (tmp_value_sec1 != DISABLE) {
			ret_1 = 1U;
		}

		/* Get Block 0 Protection status */
		tmp_value_block = FC_GetBlockProtectState(FC_BLOCK_0);

		/* Wait for Ready */
		while (FC_GetBusyState() != DONE) {
		/* Do nothing */
		};
	}
}
```

7. GPIO

7.1 Overview

For TOSHIBA TMPM330/TMPM332/TMPM333 general-purpose I/O ports, inputs and outputs can be specified in units of bits. Besides the general-purpose input/output function, all ports perform specified function.

The GPIO driver APIs provide a set of functions to configure each port, including such common parameters as input, output, pull-down, pull-up, open-drain, CMOS and so on.

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm33**x**_gpio.c(*), with /Libraries/TX03_Periph_Driver/inc/tmpm33**x**_gpio.h(*) containing the macros, data types, structures and API definitions for use by applications.

***Note**: "**x**" can be 0, 2, 3.

7.2 Difference among TMPM330, TMPM332 and TMPM333 in GPIO

Input/Output Ports for TMPM330/TMPM333: Port A-Port K.
Input/Output Ports for TMPM332: Port A-Port B, Port D-Port K.

7.3 API Functions

7.3.1 Function List

- uint8_t GPIO_ReadData(GPIO_Port ***GPIO_x***)
- uint8_t GPIO_ReadDataBit(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***)
- void GPIO_WriteData(GPIO_Port ***GPIO_x***, uint8_t ***Data***)
- void GPIO_WriteDataBit(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***, uint8_t ***BitValue***)
- void GPIO_Init(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***,
  GPIO_InitTypeDef ****GPIO_InitStruct***)
- void GPIO_SetOutput(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***)
- void GPIO_SetInput(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***)
- void GPIO_SetOutputEnableReg(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***,
  FunctionalState ***NewState***)
- void GPIO_SetInputEnableReg(GPIO_Port **GPIO_x**, uint8_t **Bit_x**,
  FunctionalState ***NewState***)

- void GPIO_SetPullUp(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***,
  FunctionalState ***NewState***)
- void GPIO_SetPullDown(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***,
  FunctionalState ***NewState***)
- void GPIO_SetOpenDrain(GPIO_Port ***GPIO_x***, uint8_t ***Bit_x***,
  FunctionalState ***NewState***)
- void GPIO_EnableFuncReg(GPIO_Port ***GPIO_x***, uint8_t ***FuncReg_x***,uint8_t ***Bit_x***)
- void GPIO_DisableFuncReg(GPIO_Port ***GPIO_x***, uint8_t ***FuncReg_x***, uint8_t ***Bit_x***)

7.3.2 Detailed Description

Functions listed above can be divided into three parts:

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

7.3.3 Function Documentation

7.3.3.1 GPIO_ReadData

Read specified GPIO Data register.

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

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

- **GPIO_PA:** GPIO port A,
- **GPIO_PB:** GPIO port B,
- **GPIO_PC:** GPIO port C (for TMPM330/TMPM333),
- **GPIO_PD:** GPIO port D,
- **GPIO_PE:** GPIO port E,
- **GPIO_PF:** GPIO port F,
- **GPIO_PG:** GPIO port G,
- **GPIO_PH:** GPIO port H,
- **GPIO_PI:** GPIO port I,
- **GPIO_PJ:** GPIO port J,
- **GPIO_PK:** GPIO port K.

Description:
This function will read specified GPIO Data register.

Return:
The value read from DATA register.

7.3.3.2 GPIO_ReadDataBit

Read specified GPIO pin.

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

Parameters:
GPIO_x: Select GPIO port, which can be set as:
- **GPIO_PA:** GPIO port A,
- **GPIO_PB:** GPIO port B,
- **GPIO_PC:** GPIO port C (for TMPM330/TMPM333),
- **GPIO_PD:** GPIO port D,
- **GPIO_PE:** GPIO port E,
- **GPIO_PF:** GPIO port F,
- **GPIO_PG:** GPIO port G,
- **GPIO_PH:** GPIO port H,
- **GPIO_PI:** GPIO port I,
- **GPIO_PJ:** GPIO port J,
- **GPIO_PK:** GPIO port K.

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

Description:
This function will read specified GPIO pin.

Return:

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

7.3.3.3 GPIO_WriteData

Write specified value to GPIO Data register.

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

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

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

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

Return:
None

7.3.3.4 GPIO_WriteDataBit

Write specified value to GPIO pin.

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

Parameters:

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

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

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

- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7.

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

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

Description:

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

Return:

None

7.3.3.5 GPIO_Init

Initialize GPIO port function.

Prototype:

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

Parameters:

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

- **GPIO_PA:** GPIO port A,
- **GPIO_PB:** GPIO port B,
- **GPIO_PC:** GPIO port C (for TMPM330/TMPM333),
- **GPIO_PD:** GPIO port D,
- **GPIO_PE:** GPIO port E,
- **GPIO_PF:** GPIO port F,
- **GPIO_PG:** GPIO port G,
- **GPIO_PH:** GPIO port H,
- **GPIO_PI:** GPIO port I,
- **GPIO_PJ:** GPIO port J,
- **GPIO_PK:** GPIO port K.

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

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

GPIO_InitStruct: The structure containing basic GPIO configuration. (Refer to 7.2.4 Data structure Description for details)

Description:

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

Return:

None

7.3.3.6 GPIO_SetOutput

Set specified GPIO pin as output port.

Prototype:

```
void
GPIO_SetOutput(GPIO_Port    GPIO_x,
                    uint8_t Bit_x)
```

Parameters:

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

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

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

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

Description:

This function will set specified GPIO pin as output port.

Return:

None

7.3.3.7 GPIO_SetInput

Set specified GPIO Pin as input port.

Prototype:

```
void
GPIO_SetInput(GPIO_Port GPIO_x,
              uint8_t Bit_x)
```

Parameters:

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

- **GPIO_PA:** GPIO port A,
- **GPIO_PB:** GPIO port B,
- **GPIO_PC:** GPIO port C (for TMPM330/TMPM333),
- **GPIO_PD:** GPIO port D,

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

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

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

Description:

This function will set specified GPIO pin as input port.

Return:

None

7.3.3.8 GPIO_SetOutputEnableReg

Enable or disable specified GPIO Pin output function.

Prototype:

```
void
GPIO_SetOutputEnableReg(GPIO_Port GPIO_x,
                        uint8_t Bit_x,
                        FunctionalState NewState)
```

Parameters:

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

- **GPIO_PA:** GPIO port A,
- **GPIO_PB:** GPIO port B,
- **GPIO_PE:** GPIO port E,
- **GPIO_PF:** GPIO port F,
- **GPIO_PG:** GPIO port G,
- **GPIO_PH:** GPIO port H,
- **GPIO_PI:** GPIO port I,

- **GPIO_PJ:** GPIO port J,
- **GPIO_PK:** GPIO port K.

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

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

NewState:

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

Description:

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

Return:

None

7.3.3.9 GPIO_SetInputEnableReg

Enable or disable specified GPIO Pin input function.

Prototype:

```
void
GPIO_SetInputEnableReg(GPIO_Port GPIO_x,
                       uint8_t Bit_x,
                       FunctionalState NewState)
```

Parameters:

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

- **GPIO_PA:** GPIO port A,
- **GPIO_PB:** GPIO port B,
- **GPIO_PC:** GPIO port C (for TMPM330/TMPM333),
- **GPIO_PD:** GPIO port D,
- **GPIO_PE:** GPIO port E,

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

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

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

NewState:

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

Description:

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

Return:

None

7.3.3.10 GPIO_SetPullUp

Enable or disable specified GPIO Pin pull-up function.

Prototype:

```
void
GPIO_SetPullUp(GPIO_Port GPIO_x,
               uint8_t Bit_x,
               FunctionalState NewState)
```

Parameters:

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

- **GPIO_PA:** GPIO port A,

- **GPIO_PB:** GPIO port B,
- **GPIO_PC:** GPIO port C (for TMPM330/TMPM333),
- **GPIO_PD:** GPIO port D,
- **GPIO_PE:** GPIO port E,
- **GPIO_PF:** GPIO port F,
- **GPIO_PG:** GPIO port G,
- **GPIO_PH:** GPIO port H,
- **GPIO_PI:** GPIO port I,
- **GPIO_PJ:** GPIO port J,
- **GPIO_PK:** GPIO port K.

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

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

NewState:

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

Description:

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

Return:

None

7.3.3.11 GPIO_SetPullDown

Enable or disable pull-down function for the specified GPIO Pin.

Prototype:

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

Parameters:

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

- **GPIO_PA:** GPIO port A.

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

- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_ALL:** All GPIO pins can be set.

NewState:

- **ENABLE :** enable pulldown state
- **DISABLE :** disable pulldown state

Description:

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

Return:

None

7.3.3.12 GPIO_SetOpenDrain

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

Prototype:

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

Parameters:

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

- **GPIO_PE:** GPIO port E,
- **GPIO_PF:** GPIO port F,
- **GPIO_PG:** GPIO port G.

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

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

- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7,
- **GPIO_BIT_ALL:** All GPIO pins can be set.

NewState:

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

Description:

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

Return:

None

7.3.3.13 GPIO_EnableFuncReg

Enable specified GPIO function.

Prototype:

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

Parameters:

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

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

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

- **GPIO_FUNC_REG_1** for GPIO function register 1,
- **GPIO_FUNC_REG_2** for GPIO function register 2.

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

- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7.

Description:
This function will enable GPIO pin specified function. For TMPM330/TMPM333, except that PE, PF and PK have two function registers, the other GPIO ports have only one (Refer to TMPM330/TMPM333 datasheet for details); for TMPM332, except that PE and PF have two function registers, the other GPIO ports have only one (Refer to TMPM332 datasheet for details).

Return:
None

7.3.3.14 GPIO_DisableFuncReg

Disable specified GPIO function.

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

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

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

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

- **GPIO_FUNC_REG_1** for GPIO function register 1,
- **GPIO_FUNC_REG_2** for GPIO function register 2.

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

- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7.

Description:

This function will disable GPIO pin specified function. For TMPM330/TMPM333, except that PE, PF and PK have two function registers, the other GPIO ports have only one (Refer to TMPM330/TMPM333 datasheet for details); for TMPM332, except that PE and PF have two function registers, the other GPIO ports have only one (Refer to TMPM332 datasheet for details).

Return:

None

7.3.4 Data Structure Description

7.3.4.1 GPIO_InitTypeDef

Data Fields:

uint8_t

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

- **GPIO_INPUT:** Set GPIO pin as input port
- **GPIO_OUTPUT:** Set GPIO pin as output port
- **GPIO_IO_MODE_NONE:** Don't change GPIO pin I/O mode.

uint8_t

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

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

uint8_t

PullDown Enable or disable specified GPIO Pin pull-down function, which can be set as:

- **GPIO_PULLDOWN_ENABLE:** Enable specified GPIO pin pull-down function**.**
- **GPIO_PULLDOWN_DISABLE:** Disable specified GPIO pin pull-down function**.**
- **GPIO_PULLDOWN_NONE:** Don't have pull-down function.

uint8_t

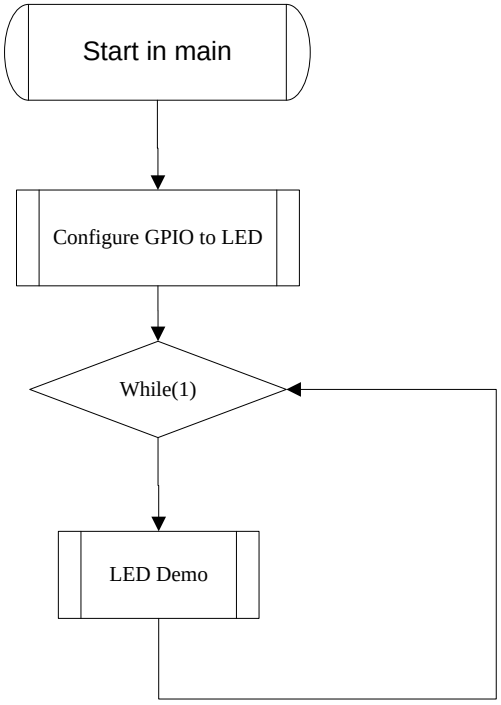
OpenDrain Set specified GPIO Pin as open drain port or CMOS port, which can be set as:

- **GPIO_OPEN_DRAIN_ENABLE:** Set specified GPIO pin as open drain port**.**
- **GPIO_OPEN_DRAIN_DISABLE:** Set specified GPIO pin as CMOS port**.**
- **GPIO_OPEN_DRAIN_NONE:** Don't have open-drain function.

7.3.5 Programming Example

This is a simple application based on the TMPM33x Peripheral Driver (GPIO), use GPIO functions to configure GPIO to LED, then turn on the LED, or turn off the LED.

7.3.5.1 Flow Chart



7.3.5.2 Code and Explanation for the Example

At first, use GPIO_Init() to configure GPIO to LED. Create a GPIO_InitTypeDef struct, then fill all the data fields. For example,

```
GPIO_InitTypeDef led_io;
led_io.IOMode = GPIO_OUTPUT_MODE;
led_io.PullUp = GPIO_PULLUP_ENABLE;
led_io.PullDown = GPIO_PULLDOWN_NONE;
led_io.OpenDrain = GPIO_OPEN_DRAIN_NONE;
```

Then call GPIO_Init() function to initialize LED.

```
GPIO_Init(GPIO_PG, GPIO_BIT_0, &led_io);
```

In the While process, do the LED demo: LED on and LED off.

Set LED on by Using GPIO_WriteData () to write 1 to GPIO DATA register.

```
GPIO_WriteDataBit(GPIO_PG, GPIO_BIT_0, GPIO_BIT_VALUE_1);
```

Set LED off by Using GPIO_WriteData () to write 0 to GPIO DATA register.

```
GPIO_WriteDataBit(GPIO_PG, GPIO_BIT_0, GPIO_BIT_VALUE_0);
```

8. RMC

8.1 Overview

TOSHIBA TMPM33x has remote control signal preprocessor.
TMPM330 has remote control signal preprocessor: RMC0 and RMC1.
TMPM332 has remote control signal preprocessor: RMC0.

Reception of Remote Control Signal:
• Sampled by 32KHz clock
• Noise canceller
• Leader detection
• Batch reception up to 72bit of data

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

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm33x_rmc.c(*), with /Libraries/TX03_Periph_Driver/inc/tmpm33x_rmc.h(*) containing the macros, data types, structures and API definitions for use by applications.

***Note: x** can be 0, 2

8.2 Difference among TMPM330, TMPM332 and TMPM333 in RMC

TMPM330 has two channel of RMC(RMC0 and RMC1) and TMPM332 only has one channel of RMC(RMC0) while TMPM333 doesn’t support RMC.

8.3 API Functions

8.3.1 Function List

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

8.3.2 Detailed Description

Functions listed above can be divided into three parts:

1) Reset and set each RMC channel are handled by RMC_Enable(), RMC_Disable(), RMC_Init() and RMC_SetRxCtrl().
2) RMC basic function are handled by SetLeaderDetection (), SetFallingEdgeINT (), RMC_SetSignalRxMethod(), RMC_ SetRxTrg(), RMC_ SetThreshold(), RMC_SetInputSignalReversed() and RMC_ SetNoiseCancellation().
3) RMC_GetINTFactor(), RMC_GetLeader() and RMC_GetRxData() to get the receive status and save the received data from buffer.

8.3.3 Function Documentation

8.3.3.1 RMC_Enable

Enable the specified RMC channel.

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

Parameters:
RMCx is the specified RMC channel.
This parameter can be one of the following values:
TSB_RMC0, **TSB_RMC1** (For TMPM330 only)

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

Return:
None

8.3.3.2 RMC_Disable

Disable the function of specified RMC channel.

Prototype:
void
RMC_Disable(TSB_RMC_TypeDef * ***RMCx***);
Parameters:
RMCx is the specified RMC channel.
This parameter can be one of the following values:
TSB_RMC0, **TSB_RMC1** (For TMPM330 only)

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

Return:
None

8.3.3.3 RMC_Init

RMC registers initial.

Prototype:
void

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

Parameters:

RMCx is the specified RMC channel.

This parameter can be one of the following values:

TSB_RMC0, **TSB_RMC1** (For TMPM330 only)

RMC_InitStruct : The structure containing the basic RMC configuration.

(For details, please refer to section “8.3.4 Data Structure Description”)

Description:

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

Return:

None

8.3.3.4 RMC_SetRxCtrl

Enable or disable reception of the specified RMC channel.

Prototype:

void

RMC_SetRxCtrl(TSB_RMC_TypeDef * ***RMCx***, FunctionalState ***NewState***);

Parameters:

RMCx is the specified RMC channel.

This parameter can be one of the following values:

TSB_RMC0, **TSB_RMC1** (For TMPM330 only)

NewState is the new state for reception of RMC.

This parameter can be one of the following values:

ENABLE or **DISABLE**

Description:

This function will enable or disable reception of the specified RMC channel selected by ***RMCx***.

This function handles the RMCREN<RMCREN> bit.

Return:

None

8.3.3.5 RMC_GetRxData

Get the received data from the specified RMC channel.

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

Parameters:
RMCx is the specified RMC channel.
This parameter can be one of the following values:
TSB_RMC0, **TSB_RMC1** (For TMPM330 only)

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

Return:
RMC_RxDataDef: Structure to read data from the RMC receive buffer.
(Refer to "8.3.4 Data Structure Description" for details).

8.3.3.6 RMC_ SetLeaderDetection

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

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

Parameters:
RMCx is the specified RMC channel.
This parameter can be one of the following values:
TSB_RMC0, **TSB_RMC1** (For TMPM330 only)

LeaderPara: The structure containing basic RMC leader detection configuration.

Data Fields:

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

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

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

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

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

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

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

Return:
None

8.3.3.7 RMC_SetFallingEdgeINT

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

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

Parameters:

RMCx is the specified RMC channel.

This parameter can be one of the following values:

TSB_RMC0, **TSB_RMC1** (For TMPM330 only)

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

This parameter can be one of the following values:

ENABLE or **DISABLE**

Description:

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

This function handles the RMCRCR2<RMCEDIEN> bit.

Return:

None

8.3.3.8 RMC_SetSignalRxMethod

Select the method of receiving a remote control signal.

Prototype:

void

RMC_SetSignalRxMethod(TSB_RMC_TypeDef * ***RMCx***,

RMC_RxMethod ***Method***);

Parameters:

RMCx is the specified RMC channel.

This parameter can be one of the following values:

TSB_RMC0, **TSB_RMC1** (For TMPM330 only)

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

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

Description:

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

This function handles the RMCRCR2<RMCPHM> bit.

Return:
None

8.3.3.9 RMC_SetRxTrg

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

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

Parameters:
RMCx is the specified RMC channel.
This parameter can be one of the following values:
TSB_RMC0, **TSB_RMC1** (For TMPM330 only)

LowWidth: Excess low width that triggers reception completion and interrupt generation.
MaxDataBitCycle: Maximum data bit cycle that triggers reception completion and interrupt generation.

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

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

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

Return:
None

8.3.3.10 RMC_SetThreshold

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

Prototype:

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

Parameters:

RMCx is the specified RMC channel.
This parameter can be one of the following values:
TSB_RMC0, **TSB_RMC1** (For TMPM330 only)

LargerThreshold: Specifies a larger threshold (within a range of 1.5T and 2T) to determine a pattern of remote control signal in a phase method. If the measured cycle exceeds the threshold, the bit is determined as “10”. If not, the bit is determined as “01”. Calculating formula of the threshold: RMCDATHx1/fs[s].
The LargerThreshold should less than 0x80.

SmallerThreshold: Specifies two kinds of thresholds: a threshold to determine whether a data bit is 0 or 1; a smaller threshold (within a range of 1T and 1.5T) to determine a pattern of remote control signal in a phase method.
As for the determination of data bit, if the measured cycle exceeds the threshold, the bit is determined as “1”. If not, the bit is determined as “0”.
Calculating-formula of the threshold: RMCDATL×1/fs[s].
As for the determination of a remote control signal pattern in a phase method, if the measured cycle exceeds the threshold, the bit is determined as “01”. If not, the bit is determined as “00”. Calculating formula of the threshold to determine 0 or 1: RMCDATLx1/fs[s].
This function handles the RMCRCR3<RMCDATH0-6> <RMCDATL0-6> bits.
The SmallerThreshold should less than 0x80.

Description:

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

Return:

None

8.3.3.11 RMC_SetInputSignalReversed

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

Prototype:

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

Parameters:

RMCx is the specified RMC channel.
This parameter can be one of the following values:
TSB_RMC0, **TSB_RMC1** (For TMPM330 only)

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

Description:

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

Return:

None

8.3.3.12 RMC_SetNoiseCancellation

Set the noise cancellation time for the specified RMC channel.

Prototype:

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

Parameters:

RMCx is the specified RMC channel.
This parameter can be one of the following values:
TSB_RMC0, **TSB_RMC1** (For TMPM330 only)

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

Description:

Specifies time that noises is cancelled by a noise canceller.
If <RMCNC3:0> = 0000b, noises are not cancelled.
Calculating formula of noise cancellation time: RMCNC x 1/fs[s].
This function handles the RMCRCR4<RMCNC0-RMCNC3> bits.

Return:
None

8.3.3.13 RMC_GetINTFactor

Get the interrupt factor for the specified RMC channel.

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

Parameters:
RMCx is the specified RMC channel.
This parameter can be one of the following values:
TSB_RMC0, **TSB_RMC1** (For TMPM330 only)

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

Return:
RMC_INTFactor: Interrupt factor structure.
(Refer to “8.3.4 Data Structure Description” for details).

8.3.3.14 RMC_GetLeader

Get the leader detection result for the specified RMC channel.

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

Parameters:

RMCx is the specified RMC channel.

This parameter can be one of the following values:

TSB_RMC0, **TSB_RMC1** (For TMPM330 only)

Description:

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

This info is updated every time an interrupt is generated.

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

Return:

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

RMC_LEADER_DETECTED: leader detected.

RMC_NO_LEADER: no leader detected.

8.3.4 Data Structure Description

8.3.4.1 RMC_RxDataDef

Data Fields:

uint8

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

uint32_t

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

uint32_t

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

uint8_t

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

8.3.4.2 RMC_LeaderParameterTypeDef

Data Fields:

FunctionalState

***LeaderDetectionState*:** ENABLE or DISABLE the leader detection.

Parameter can be one of the following values:

ENABLE or **DISABLE**

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

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

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

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

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

8.3.4.3 RMC_InitTypeDef

Data Fields:

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

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

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

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

uint8_t

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

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

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

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

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

8.3.4.4 RMC_INTFactor

Data Fields:

uint32_t
All: Data.
Bit

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

8.3.5 Programming Example

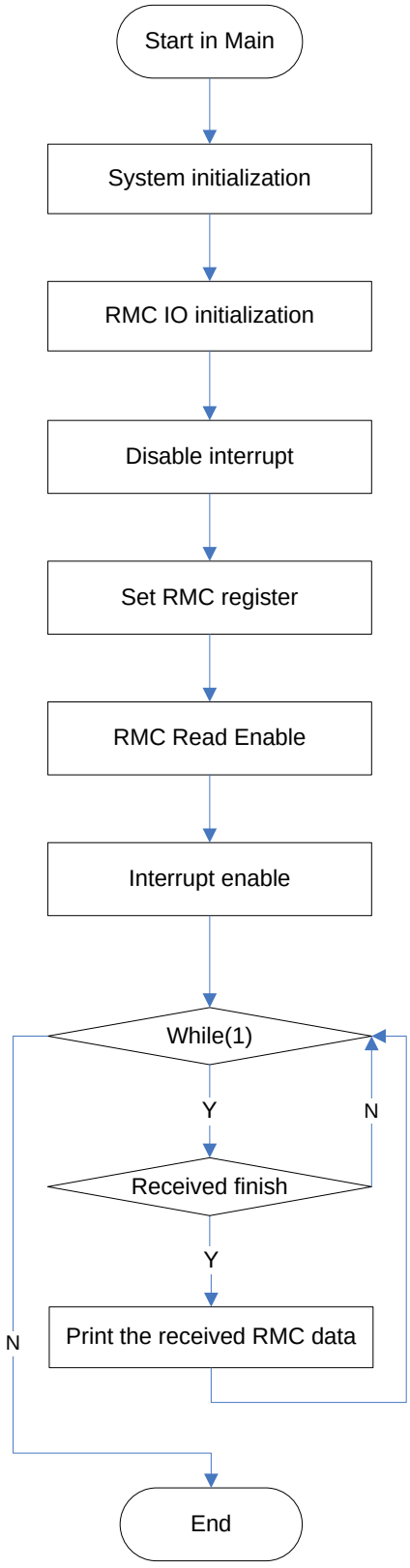
This is a simple application based on the TMPM33x Peripheral Driver (RMC), which received the remote control signal from RMC channel 0.

The application works in the follow Environment:

- IAR Embedded Workbench for ARM 5.40
- TMPM330 Demo board(M4M330B-2)
- PC (OS: Windows XP)

TOSHIBA format or RC5 format remote control device is available in this application; user can change the definition on header file to select one remote control format. In this application, if the M330 board receives the TOSHIBA format or RC5 format remote control signal, then it will print the received data on the Terminal I/O of IAR Embedded Workbench, just as ”RMC DATA: 0x2fd807f ”(TOSHIBA format).

8.3.5.1 Flow Chart



8.3.5.2 Code and Explanation for the Example

At first, in main(), create a myRMC structure, then fill all the data fields.
For example

```
RMC_InitTypeDef myRMC;

myRMC.LeaderPara.MaxCycle = RMC_MAX_CYCLE;
myRMC.LeaderPara.MinCycle = RMC_MIN_CYCLE;
myRMC.LeaderPara.MaxLowWidth = RMC_MAX_LOW_WIDTH;
myRMC.LeaderPara.MinLowWidth = RMC_MIN_LOW_WIDTH;
myRMC.LeaderPara.LeaderDetectionState = ENABLE;
myRMC.LeaderPara.LeaderINTState = DISABLE;
myRMC.FallingEdgeINTState = DISABLE;
myRMC.SignalRxMethod = RMC_RX_IN_CYCLE_METHOD;
myRMC.LowWidth = RMC_TRG_LOW_WIDTH;
myRMC.MaxDataBitCycle = RMC_TRG_MAX_DATA_BIT_CYCLE;
myRMC.LargerThreshold = RMC_LARGER_THRESHOLD;
myRMC.SmallerThreshold = RMC_SMALLER_THRESHOLD;
myRMC.InputSignalReversedState = DISABLE;
myRMC.NoiseCancellationTime = RMC_NOISE_CANCELLATION_TIME;
```

Then, enable and initialize the RMC channel 0.

```
RMC_Enable(TSB_RMC0);
```

Initial the specified RMC channel with the structure which includes the basic RMC configuration.

```
RMC_Init(TSB_RMC0, &myRMC);
```

Enable the reception of the specified RMC0 channel.

```
RMC_SetRxCtrl(TSB_RMC0, ENABLE);
```

In interrupt INTRMCRX0_IRQHandler ():
Get the interrupt factor for the specified RMC0 channel.

```
RMC_INTFactor myRMC_INTFactor;
myRMC_INTFactor = RMC_GetINTFactor(TSB_RMC0);
```

Get the leader detection result for the specified RMC0 channel.

```
RMC_LeaderDetection myRMC_LeaderDetection;
myRMC_LeaderDetection = RMC_GetLeader(TSB_RMC0);
```

Get the received data from the specified RMC0 channel.

```
RMC_RxDataTypeDef myRMC_RxDataDef;
myRMC_RxDataDef = RMC_GetRxData(TSB_RMC0);
```

9. RTC

9.1 Overview

The Real Time Clock (RTC) in the TMPM33**x** (*) has such functions as follow:

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

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

All driver APIs are contained in/Libraries/TX03_Periph_Driver/src/tmpm33**x**_rtc.c (*), with /Libraries/ TX03_Periph_Driver/inc/tmpm33**x**_rtc.h (*) containing the macros, data types, structures and API definitions for use by applications.

***Note**: **x** can be 0, 2, 3.

9.2 Difference among TMPM330, TMPM332 and TMPM333 in RTC

None

9.3 API Functions

9.3.1 Function List

- void RTC_SetSec(uint8_t ***Sec***)
- uint8_t RTC_GetSec(void)
- void RTC_SetMin(RTC_FuncMode ***NewMode***, uint8_t ***Min***)
- uint8_t RTC_GetMin(RTC_FuncMode ***NewMode***)
- uint8_t RTC_GetAMPM(RTC_FuncMode ***NewMode***)
- void RTC_SetHour24(RTC_FuncMode ***NewMode***, uint8_t ***Hour***)
- void RTC_SetHour12(RTC_FuncMode ***NewMode***, uint8_t ***Hour***, uint8_t ***AmPm***)
- uint8_t RTC_GetHour(RTC_FuncMode ***NewMode***)
- void RTC_SetDay(RTC_FuncMode ***NewMode***, uint8_t ***Day***)

- uint8_t RTC_GetDay(RTC_FuncMode ***NewMode***)
- void RTC_SetDate(RTC_FuncMode ***NewMode***, uint8_t ***Date***)
- uint8_t RTC_GetDate(RTC_FuncMode ***NewMode***)
- void RTC_SetMonth(uint8_t ***Month***)
- uint8_t RTC_GetMonth(void)
- void RTC_SetYear(uint8_t ***Year***)
- uint8_t RTC_GetYear(void)
- void RTC_SetHourMode(uint8_t ***HourMode***)
- uint8_t RTC_GetHourMode(void)
- void RTC_SetLeapYear(uint8_t ***LeapYear***)
- uint8_t RTC_GetLeapYear(void)
- void RTC_SetTimeAdjustReq(void)
- RTC_ReqState RTC_GetTimeAdjustReq(void)
- void RTC_EnableClock(void)
- void RTC_DisableClock(void)
- void RTC_EnableAlarm(void)
- void RTC_DisableAlarm(void)
- void RTC_SetRTCINT(FunctionalState ***NewState***)
- void RTC_SetAlarmOutput(uint8_t ***Output***)
- void RTC_ResetClockSec(void)
- RTC_ReqState RTC_GetResetClockSecReq(void)
- void RTC_ResetAlarm(void)
- void RTC_SetDateValue(RTC_DateTypeDef * ***DateStruct***)
- void RTC_GetDateValue(RTC_DateTypeDef * ***DateStruct***)
- void RTC_SetTimeValue(RTC_TimeTypeDef * ***TimeStruct***)
- void RTC_GetTimeValue(RTC_TimeTypeDef * ***TimeStruct***)
- void RTC_SetClockValue(RTC_DateTypeDef * ***DateStruct***, RTC_TimeTypeDef * ***TimeStruct***)
- void RTC_GetClockValue(RTC_DateTypeDef * ***DateStruct***, RTC_TimeTypeDef * ***TimeStruct***)
- void RTC_SetAlarmValue(RTC_AlarmTypeDef * ***AlarmStruct***)
- void RTC_GetAlarmValue(RTC_AlarmTypeDef * ***AlarmStruct***)

9.3.2 Detailed Description

Functions listed above can be divided into five parts:

1) Configure the common functions of RTC date are handled by RTC_SetDay(), RTC_GetDay(), RTC_SetDate(), RTC_GetDate(), RTC_SetMonth(), RTC_GetMonth(), RTC_SetYear(), RTC_GetYear(), RTC_SetLeapYear(), RTC_GetLeapYear(), RTC_SetDateValue(), RTC_GetDateValue(),
2) Configure the common functions of RTC time are handled by RTC_SetSec(), RTC_GetSec(), RTC_SetMin(),RTC_GetMin(),RTC_SetHour24(), RTC_SetHour12(),

RTC_GetHour(), RTC_SetHourMode(), RTC_GetHourMode, RTC_GetAMPM(), RTC_SetTimeValue(), RTC_GetTimeValue().

3) RTC_EnableClock(), RTC_DisableClock(), RTC_SetTimeAdjustReq(), RTC_GetTimeAdjustReq(), RTC_ResetClockSec(), RTC_GetResetClockSec(), RTC_SetClockValue() and RTC_GetClockValue() handle for RTC clock function only.
4) RTC_EnableAlarm(), RTC_DisableAlarm(), RTC_ResetAlarm(), RTC_SetAlarmValue() and RTC_GetAlarmValue() handle for RTC alarm function only.
5) RTC_SetAlarmOutput() and RTC_SetRTCINT() handle other specified functions.

9.3.3 Function Documentation

9.3.3.1 RTC_SetSec

Set second value for RTC clock.

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

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

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

*How to use RTC 1Hz interrupt, please refer to “9.3.5.2 Code and Explanation for the Example” for details.

Return:
None

9.3.3.2 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

9.3.3.3 RTC_SetMin

Set minute value for RTC clock or alarm.

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

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

Min: New min value, max 59

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

Return:
None

9.3.3.4 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

9.3.3.5 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.

9.3.3.6 RTC_SetHour24

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

Prototype:
void
RTC_SetHour24(RTC_FuncMode ***NewMode***,

uint8_t ***Hour***)

Parameters:

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

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

Hour: New hour value, max. is 23.

Description:

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

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

Return:

None

9.3.3.7 RTC_SetHour12

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

Prototype:

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

Parameters:

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

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

Hour: New hour value, max. is 11.

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

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

Description:

This function will set new hour value and AM/PM mode for RTC clock when ***NewMode*** is **RTC_CLOCK_MODE**, and set new hour value and AM/PM mode for RTC alarm when ***NewMode*** is **RTC_ALARM_MODE**. RTC register are updated

synchronizing with the timing of INTRTC, so after calling this function, it should wait for RTC 1HZ interrupt occurs.

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

Return:
None

9.3.3.8 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

9.3.3.9 RTC_SetDay

Set day value for RTC clock or alarm.

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

Parameters:

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

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

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

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

Description:

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

Return:

None

9.3.3.10 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

9.3.3.11 RTC_SetDate

Set date value for RTC clock or alarm.

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

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

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

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

Return:
None

9.3.3.12 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

9.3.3.13 RTC_SetMonth

Set month value for RTC clock.

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

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

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

Return:
None

9.3.3.14 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

9.3.3.15 RTC_SetYear

Set year value for RTC clock.

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

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

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

Return:
None

9.3.3.16 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

9.3.3.17 RTC_SetHourMode

Select 24-hour clock or 12-hour clock.

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

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

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

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

* Before call this function, **RTC_DisableClock()** function should be called firstly. (See “9.3.3.24 RTC_DisableClock” for details)

Return:
None

9.3.3.18 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.

9.3.3.19 RTC_SetLeapYear

Set leap year state.

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

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

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

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

Return:
None

9.3.3.20 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.

9.3.3.21 RTC_SetTimeAdjustReq

Set time adjustment + or – 30 seconds.

Prototype:
void
RTC_SetTimeAdjustReq(void)

Parameters:
None

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

Return:
None

9.3.3.22 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.

9.3.3.23 RTC_EnableClock

Enable RTC clock function.

Prototype:
void
RTC_EnableClock(void)

Parameters:
None

Description:
This function will enable clock function.

Return:
None

9.3.3.24 RTC_DisableClock

Disable RTC clock function.

Prototype:
void
RTC_DisableClock(void)

Parameters:
None

Description:
This function will disable clock function.

Return:
None

9.3.3.25 RTC_EnableAlarm

Enable RTC alarm function.

Prototype:
void
RTC_EnableAlarm(void)

Parameters:
None

Description:
This function will enable alarm function.

Return:
None

9.3.3.26 RTC_DisableAlarm

Disable RTC alarm function.

Prototype:
void
RTC_DisableAlarm(void)

Parameters:
None

Description:
This function will disable alarm function.

Return:
None

9.3.3.27 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

9.3.3.28 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_1_HZ,** Alarm pin output is 1Hz cycle “0” pulse. If ***Output*** is **RTC_PULSE_16_HZ,** Alarm pin output is 16Hz cycle “0” pulse.

Return:
None

9.3.3.29 RTC_ResetClockSec

Reset RTC clock second counter.

Prototype:
void
RTC_ResetClockSec(void)

Parameters:
None

Description:
This function will reset sec counter.

Return:
None

9.3.3.30 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.

9.3.3.31 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

9.3.3.32 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 "9.3.4 Data structure Description" for details)

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

Return:
None

9.3.3.33 RTC_GetDateValue

Get the RTC clock date.

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

Parameters:

DateStruct: The structure containing basic date configuration. (Refer to "9.3.4 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

9.3.3.34 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 "9.3.4 Data structure Description" for details)

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

Return:
None

9.3.3.35 RTC_GetTimeValue

Get the RTC time.

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

Parameters:

TimeStruct: The structure containing basic Time configuration. (Refer to "9.3.4 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

9.3.3.36 RTC_SetClockValue

Set the RTC clock date and time.

Prototype:

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

Parameters:

DateStruct: The structure containing basic Date configuration including leap year state, year, month, date and day.
TimeStruct: The structure containing basic Time configuration including hour mode, hour, AM/PM mode in 12H mode, minute and second. (Refer to "9.3.4 Data structure Description" for details)

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

Return:
None

9.3.3.37 RTC_GetClockValue

Get the RTC clock date and time.

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

Parameters:

DateStruct: The structure containing basic Date configuration including leap year state, year, month, date and day.
TimeStruct: The structure containing basic Time configuration including hour mode, hour, AM/PM mode in 12H mode, minute and second. (Refer to "9.3.4 Data structure Description" for details)

Description:
This function will get RTC clock date and time, including leap year, year, month, date, day, hour mode, hour, AM/PM mode in 12H mode, minute and second.
RTC_GetLeapYear(), **RTC_GetYear()**, **RTC_GetMonth()**, **RTC_GetDate()**,

RTC_GetDay(), **RTC_GetHourMode()**, **RTC_GetHour()**,**RTC_GetAMPM()**, **RTC_GetMin()** and **RTC_GetSec()** will be called by it.

Return:
None

9.3.3.38 RTC_SetAlarmValue

Set the RTC alarm date and time.

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

Parameters:

AlarmStruct: The structure containing basic alarm configuration including date, day, hour, AM/PM mode in 12H mode and minute. (Refer to "9.3.4 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

9.3.3.39 RTC_GetAlarmValue

Get the RTC alarm date and time.

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

Parameters:

AlarmStruct: The structure containing basic alarm configuration including date, day, hour, AM/PM mode in 12H mode and minute. (Refer to "9.3.4 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

9.3.4 Data Structure Description

9.3.4.1 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.

9.3.4.2 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.

9.3.4.3 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:

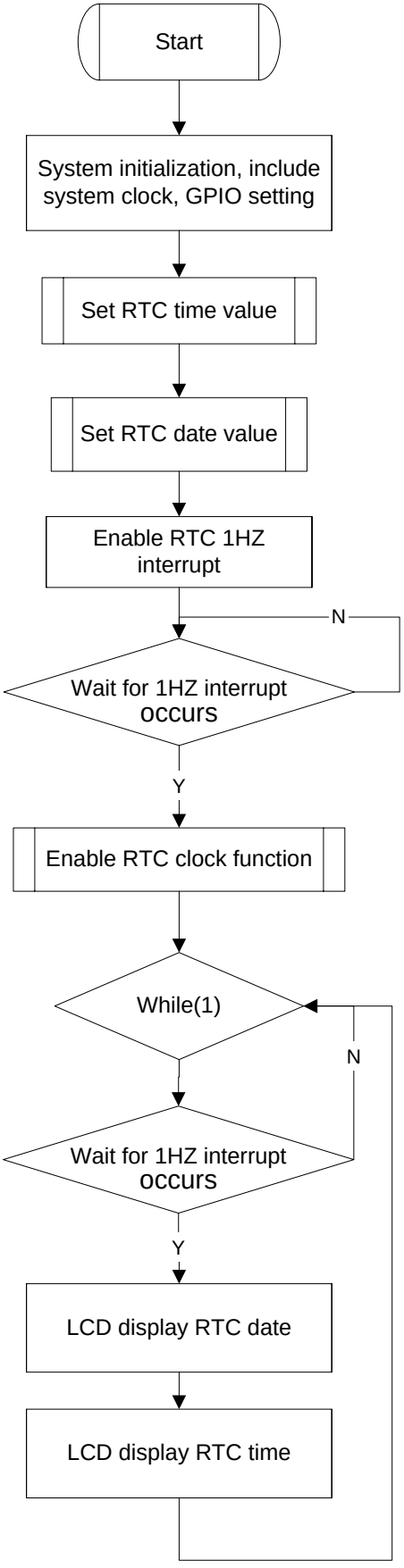
- **RTC_AM_MODE:** select AM mode for 12H mode,
- **RTC_PM_MODE:** select PM mode for 12H mode.
- **RTC_AMPM_INVALID:** when hour mode is 24H mode.

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

9.3.5 Programming Example

This is a simple application based on the TMPM33x Peripheral Driver (RTC), which set RTC date as (10-12-31 FRI) and set time as (PM 11:59:50).

9.3.5.1 Flow Chart


Start
System initialization, include system clock, GPIO setting
Set RTC time value
Set RTC date value
Enable RTC 1HZ interrupt
N
Wait for 1HZ interrupt occurs
Y
Enable RTC clock function
While(1)
N
Wait for 1HZ interrupt occurs
Y
LCD display RTC date
LCD display RTC time

9.3.5.2 Code and Explanation for the Example

At first, create RTC_DateTypeDef struct and RTC_TimeTypeDef struct, and then fill in all the data fields. For example,

```
RTC_DateTypeDef DateStruct;
RTC_TimeTypeDef TimeStruct;
DateStruct.LeapYear = RTC_LEAP_YEAR_2;
DateStruct.Year = (uint8_t)10;
DateStruct.Month = (uint8_t)12;
DateStruct.Date = (uint8_t)31;
DateStruct.Day = RTC_FRI;
TimeStruct.HourMode = RTC_12_HOUR_MODE;
TimeStruct.Hour = (uint8_t)11;
TimeStruct.AmPm = RTC_PM_MODE;
TimeStruct.Min = (uint8_t)59;
TimeStruct.Sec = (uint8_t)50;
```

After the setting above, Set RTC date and time value, and then enable RTC 1HZ interrupt.

```
RTC_SetTimeValue(&TimeStruct);
RTC_SetDateValue(&DateStruct);
__disable_irq();
/* enable RTC interrupt */
NVIC_ClearPendingIRQ(INTRTC_IRQn);
TSB_CG->IMCGC = 0x00000020;
TSB_CG->IMCGC = 0x00000021;
NVIC_EnableIRQ(INTRTC_IRQn);
/* Enable 1Hz interrupt */
RTC_SetAlarmOutput(RTC_PULSE_1_HZ);
/* Enable RTCINT */
RTC_SetRTCINT(ENABLE);
__enable_irq();
```

Then waiting for 1HZ interrupt occurs.

```
/* waiting for RTC register set finish */
  while(fRTC_1HZ_INT != 1);
  fRTC_1HZ_INT = 0;
```

At last enable RTC clock function.

```
/* Enable RTC Clock function */
RTC_EnableClock();
```

Use RTC_GetYear() / RTC_GetMonth() / RTC_GetDate() / RTC_GetDay() functions to get the RTC date value.

```
uint8_t Year = 0U;
uint8_t Month = 0U;
uint8_t Date = 0U;
uint8_t Day = 0U;
Year = RTC_GetYear();
Month = RTC_GetMonth();
Date = RTC_GetDate(RTC_CLOCK_MODE);
Day = RTC_GetDay(RTC_CLOCK_MODE);
```

Or call one function as below:

```
RTC_DateTypeDef DateStruct;
RTC_GetDateValue(&DateStruct);
```

Use RTC_GetHourMode() / RTC_GetHour() / RTC_GetMin() / RTC_GetSec() to get the RTC time value.

```
uint8_t HourMode = 0U;
uint8_t Hour = 0U;
uint8_t Min = 0U;
uint8_t Sec = 0U;
HourMode = RTC_GetHourMode();
Hour = RTC_GetHour(RTC_CLOCK_MODE);
Min = RTC_GetMin(RTC_CLOCK_MODE);
Sec = RTC_GetSec();
```

Or call one function as below:

```
RTC_TimeTypeDef TimeStruct;
RTC_GetTimeValue(&TimeStruct);
```

10. SBI

10.1 Overview

This device contains some Serial Bus Interface channels (SBI0, SBI1 and SBI2). Each channel can operate in I2C bus mode with multi-master capability.
In I2C bus mode, the SBI is connected to external devices via SCL and SDA.
Data can be transferred in free data format by the SBI channels. In free data format, data is always sent by master-transmitter and received by slave-receiver.

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

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm33**x**_sbi.c(*), with /Libraries/TX03_Periph_Driver/inc/tmpm33**x**_sbi.h(*) containing the macros, data types, structures and API definitions for use by applications.

***Note: x** can be 0,2,3

10.2 Difference among TMPM330, TMPM332 and TMPM333 in SBI

TMPM330/M333.have three Serial Bus Interface channels (SBI0, SBI1, SBI2); TMPM332 only has two Serial Bus Interface channels (SBI0, SBI1). SBI2 is invalid for TMPM332.

10.3 API Functions

10.3.1 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***)

10.3.2 Detailed Description

Functions listed above can be divided into four parts:

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

10.3.3 Function Documentation

Note: in all of the following APIs, parameter "TSB_SBI_TypeDef* ***SBIx***" can be one of the following values:
TSB_SBI0, **TSB_SBI1** or **TSB_SBI2** (only for TMPM330/333).

10.3.3.1 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

10.3.3.2 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

10.3.3.3 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

10.3.3.4 SBI_InitI2C

Initialize the specified SBI channel in I2C mode.

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

Parameters:
SBIx is the specified SBI channel.
InitI2CStruct is the structure containing SBI configuration (refer to 10.2.4 Data Structure Description for details).

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

Return:
None

10.3.3.5 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

10.3.3.6 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

10.3.3.7 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

10.3.3.8 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

10.3.3.9 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

10.3.3.10 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.

10.3.3.11 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, or
- **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

10.3.3.12 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

10.3.3.13 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

10.3.3.14 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, or
- **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

10.3.4 Data Structure Description

10.3.4.1 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,** enable the generation of ACK clock;
- **DISABLE,** disable the generation of ACK clock.

10.3.4.2 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.

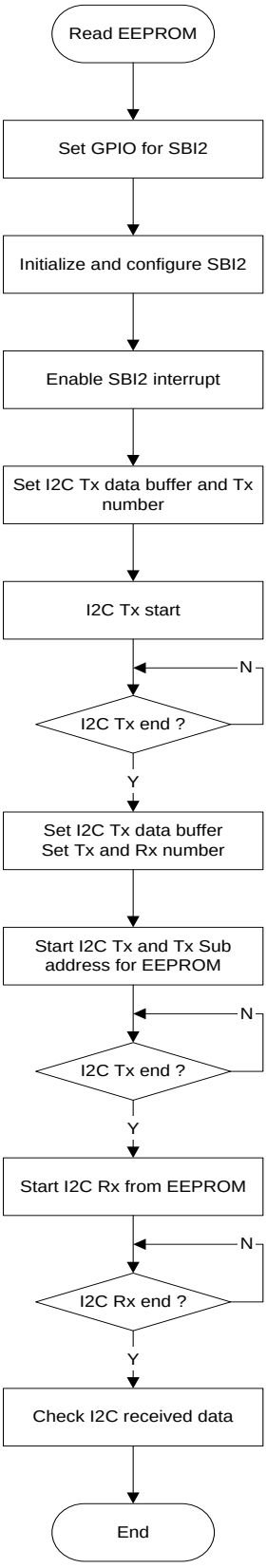
uint32_t
TRx specifies transfer or Receive selection monitor.

uint32_t
MasterSlave specifies master or slave selection monitor.

10.3.5 Programming Example

This is a simple application based on the TMPM33x Peripheral Driver(SBI), which sends 2 bytes from SBI2 to a EEPROM (address is 0xA0), and then read the 2-byte data from the EEPROM to SBI2.

10.3.5.1 Flow Chart




INT_I2C
Read the I2C status
MST mode ?
Y
Tx mode ?
N
Rx next data ?
N
Stop I2C
Y
gI2C_RCnt to Max ?
N
gI2C_RCnt to Max-1 ?
Y
Y
Set next I2C data bit to 1 bit
Set I2C to No ACK mode
N
Y
LRB = 0 ?
N
Read I2C data from data buffer
Y
Tx next data ?
N
The 1st Rx data ?
N
Y
Stop I2C
Y
Not save data to I2C Rx buffer, only dummy read
Save data to I2C Rx buffer,
Set the next data to I2C data buffer and I2C Tx counter ++
gI2C_RCnt++
N
End

10.3.5.2 Code and Explanation for the Example

At first, create a SBI_InitI2CTypeDef structure and fill all the data fields. For example,

```
myI2C.I2CSelfAddr = SELFADDR;
myI2C.I2CDataLen = SBI_I2C_DATA_LEN_8;
myI2C.I2CACKState = ENABLE;
myI2C.I2CClkDiv = SBI_I2C_CLK_DIV_328;
```

Then enable, initialize and configure SBI2.

```
SBI_Enable(TSB_SBI2);
SBI_SWReset(TSB_SBI2);
SBI_InitI2C(TSB_SBI2, &myI2C);
NVIC_EnableIRQ(INTSBI2_IRQn);
```

Following is the procedure of write operation.

Initial I2C Tx data buffer and counter, after checked the I2C bus is free, set slave address and direction of transfer. Then send start condition.

```
do{
    i2c_state = SBI_GetI2CState(TSB_SBI2);
} while (i2c_state.Bit.BusState);;
SBI_SetSendData(TSB_SBI2, SLAVEADDR | SBI_I2C_SEND);
SBI_GenerateI2CStart(TSB_SBI2);
```

Then in i2C interrupt: read the I2C bus state; send the next data, and check I2C data transfer end, stop I2C.

```
SBIx = TSB_SBI2;
sbi_sr = SBI_GetI2CState(SBIx)
……
SBI_SetSendData(SBIx, gI2C_TxData[gI2C_WCnt]);
……
SBI_GenerateI2CStop(SBIx);
```

Now data in I2C Tx buffer has been written into the EEPROM.

Following is the procedure of read operation.

Initial I2C Tx data buffer, Tx counter and Rx counter. After checked the I2C bus is free, set slave address and direction of transfer. Then send start condition.

```
do{
    i2c_state = SBI_GetI2CState(TSB_SBI2);
} while (i2c_state.Bit.BusState);
SBI_SetSendData(TSB_SBI2, SLAVEADDR | SBI_I2C_SEND);
SBI_GenerateI2CStart(TSB_SBI2);
```

Then in i2C interrupt: Read the I2C bus state, send the next data, and check I2C data transfer end, stop I2C (The sub address for EEPROM is sent in this process).

```
SBIx = TSB_SBI2;
sbi_sr = SBI_GetI2CState(SBIx)
……
```

```
SBI_SetSendData(SBIx, gI2C_TxData[gI2C_WCnt]);
……
SBI_GenerateI2CStop(SBIx);
```

After The sub address for EEPROM is sent, read the data of EEPROM. The first interrupt after I2C start for send slave address and receive direction, dummy read the I2C data buffer register to release PIN.

```
    SBI_SetSendData(TSB_SBI2, SLAVEADDR | SBI_I2C_RECEIVE);
    SBI_GenerateI2CStart(TSB_SBI2);
```

(The first interrupt after I2C start for send slave address and receive direction, dummy read the I2C data buffer register to release PIN.)

Then in i2C interrupt: Read the I2C bus state and data buffer, setting for receive the next I2C data(After Rx the data second to last, setting Not generate ACK for next data Rx end; after Rx the last data, setting I2C bit number to 1 for only generate 1 clock for stop condition).

```
SBIx = TSB_SBI2;
sbi_sr = SBI_GetI2CState(SBIx)
……
tmp = SBI_GetReceiveData(SBIx);
……
SBI_SetI2CBitNum(SBIx, SBI_I2C_DATA_LEN_1);
……
SBI_SetI2CACK(TSB_SBI2,DISABLE);
```

Now data from EEPROM has been read and save to I2C Rx data buffer.

*Note: For I2C data ACK mode is be set to “No-ACK” after receiving data finishs.So please re-set the ACK mode and other parameter for next I2C process.

Since the write address and the read address in EEPROM are the same. So the result of transfer can be check by comparing the data received with the data sent.

11. TMRB

11.1 Overview

This device contains ten channels of multi-functional 16-bit timer/event counter (TB0 through TB9). 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 TMRB to perform the following three measurements:

- Frequency measurement
- Pulse width measurement
- Time difference measurement

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

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm33**x**_tmrb.c(*), with /Libraries/TX03_Periph_Driver/inc/tmpm33**x**_tmrb.h(*) containing the macros, data types, structures and API definitions for use by applications.

***Note**: "**x**" can be 0, 2, 3.

11.2 Difference among TMPM330, TMPM332 and TMPM333 in TMRB

None

11.3 API Functions

11.3.1 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***)

11.3.2 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() and TMRB_SetDoubleBuf() handle other specified functions.

11.3.3 Function Documentation

Note: in all of the following APIs, unless specially specified, parameter "TSB_TB_TypeDef* ***TBx***" can be one of the following values:
TSB_TB0, TSB_TB1, TSB_TB2, TSB_TB3, TSB_TB4, TSB_TB5, TSB_TB6, TSB_TB7, TSB_TB8 or TSB_TB9,

11.3.3.1 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

11.3.3.2 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***.
~~TBxEN is modified by this function.~~

Return:
None

11.3.3.3 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 TB 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

11.3.3.4 TMRB_Init

Initialize the specified TB channel.

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

Parameters:
TBx is the specified TMRB channel.
InitStruct is the structure containing basic TMRB configuration including count mode, source clock division, leadingtiming value, trailingtiming value and up-counter work mode (refer to "11.3.4 Data Structure Description" for details).

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

Return:
None

11.3.3.5 TMRB_SetCaptureTiming

Configure the capture timing.

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

Parameters:

TBx is the specified TMRB channel, and the capture function is only available to **TSB_TB0, TSB_TB1, TSB_TB2, TSB_TB3, TSB_TB4, TSB_TB5 or TSB_TB6**.

CaptureTiming specifies TMRB capture timing, which can be

- **TMRB_CAPTURE_IN_RISING**: Capture the up-counter at the time of the rising edge of the pulse which is input to either of two input ports of the specified TMRB channel.
- **TMRB_CAPTURE_IN0_RISING_IN1_FALLING**: Capture the up-counter at the time of the rising edge of the pulse which is input to No.0 input port of the specified TMRB channel (TBxIN0) or at the time of the falling edge of the pulse which is input to No.1 input port of the specified TMRB channel (TBxIN1) (*).
- **TMRB_CAPTURE_OUTPUT_EDGE**: Capture the up-counter at the time of both edges of the pulse which is output from TB7OUT, TB8OUT or TB9OUT (**).
- **TMRB_DISABLE_CAPTURE**: Disable the capture function of the specified TMRB channel.

Description:

If ***CaptureTiming*** is set as **TMRB_CAPTURE_IN_RISING**, then at the time of the rising edge of the pulse input to No.0 input port of the specified TMRB channel (TBxIN0), the value in up-counter will be captured and saved into capture register0 of the TMRB channel. And at the time of the rising edge of the pulse input to No.1 input port of the specified TMRB channel (TBxIN1), the value in up-counter will be captured and saved into capture register1 of the TMRB channel.

If ***CaptureTiming*** is set as **TMRB_CAPTURE_IN0_RISING_IN1_FALLING**, then at the time of the rising edge of the pulse input to No.0 input port of the specified TMRB channel (TBxIN0), the value in up-counter will be captured and saved into capture register0 of the TMRB channel. And at the time of the falling edge of the pulse input to No.1 input port of the specified TMRB channel (TBxIN1), the value in up-counter will be captured and saved into capture register1 of the TMRB channel (*).

If ***CaptureTiming*** is set as **TMRB_CAPTURE_OUTPUT_EDGE**, then at the time of the rising edge of the pulse output from TB7OUT, TB8OUT or TB9OUT (**), the value in up-counter will be captured and saved into capture register0 of the TMRB channel. And at the time of the falling edge of the pulse output from TB7OUT, TB8OUT or TB9OUT (**), the value in up-counter will be captured and saved into capture register1 of the TMRB channel.

Return:

None

Notes:

(*) TMRB7, TMRB8 and TMRB9 do not contain input port.

(**)TB7OUT can trigger the capture function of TMRB0 and TMRB1.
TB8OUT can trigger the capture function of TMRB2, TMRB3 and TMRB4.
TB9OUT can trigger the capture function of TMRB5 and TMRB6.

11.3.3.6 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 "11.3.4 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

11.3.3.7 TMRB_GetINTFactor

Indicate what causes the interrupt.

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

Parameters:
TBx is the specified TMRB channel.

Description:
This function should be used in ISR to indicate the factor of interrupt. Bit of **MatchLeadingTiming** indicates if the up-counter matches with leadingtiming value,

Bit of **MatchTrailingTiming** Indicates if the up-counter matches with trailingtiming value, and bit of **Overflow** indicates if overflow had occurred before the interrupt.

Return:

TMRB Interrupt factor. Each bit has the following meaning:

MatchLeadingTiming(Bit0): a match with the leadingtiming value is detected

MatchTrailingTiming(Bit1): a match with the trailingtiming value is detected

OverFlow(Bit2): an up-counter is overflow

Notes:

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
}
```

11.3.3.8 TMRB_SetINTMask

Mask the specified TMRB interrupt.

Prototype:

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

Parameters:

TBx is the specified TMRB channel.

INTMask specifies the interrupt to be masked, which can be

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

- **TMRB_NO_INT_MASK**: Unmask the interrupt.

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

Return:
None

11.3.3.9 TMRB_ChangeLeadingTiming

Change the value of leadingtiming for the specified channel.

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

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

Description:
This function will specify the absolute value of leadingtiming for the specified TMRB. The actual interval of leadingtiming depends on the configuration of CG and the value of ***ClkDiv*** (refer to “11.3.4 Data Structure Description” for details).

Return:
None

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

11.3.3.10 TMRB_ChangeTrailingTiming

Change the value of trailingtiming for the specified channel.

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

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

Description:
This function will specify the absolute value of trailingtiming for the specified TMRB. The actual interval of trailingtiming depends on the configuration of CG and the value of ***ClkDiv*** (refer to "11.3.4 Data Structure Description" for details).

Return:
None

Notes:
TrailingTiming must be not smaller than ***LeadingTiming***.

11.3.3.11 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

11.3.3.12 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

11.3.3.13 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 and the value captured will be saved into the capture register0.

Return:

None

11.3.3.14 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

11.3.3.15 TMRB_SetSyncMode

Enable or disable the synchronous mode of specified TMRB channel.

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

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

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

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

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

Return:
None

Notes:
TMRB1 through TMRB3 and TMRB5 through TMRB7 must start counting by calling **TMRB_SetRunState()** before TMRB0 and TMRB4 start counting, so that start timing can be synchronized.

11.3.3.16 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:
The register TBxRG0 (***LeadingTiming***) and TBxRG1 (***TrailingTiming***) and their buffers are assigned to the same address. If double buffering is disabled, the same value is written to the registers and their buffers.
If double buffering is enabled, the value is only written to each register buffer. Therefore, to write an initial value to the registers, TBxRG0 (***LeadingTiming***) and TBxRG1 (***TrailingTiming***), the double buffering must be set to **DISABLE**. Then **ENABLE** double buffering and write the following data to the register, which can be loaded when the corresponding interrupt occurs automatically.

Return:
None

11.3.4 Data Structure Description

11.3.4.1 TMRB_InitTypeDef

Data Fields:

uint32_t

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

uint32_t

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

- **TMRB_CLK_DIV_2**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 2;
- **TMRB_CLK_DIV_8**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 8;
- **TMRB_CLK_DIV_32**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 32.

uint32_t

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

uint32_t

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

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

uint32_t

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

11.3.4.2 TMRB_FFOutputTypeDef

Data Fields:

uint32_t

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

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

uint32_t

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

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

11.3.4.3 TMRB_INTFactor

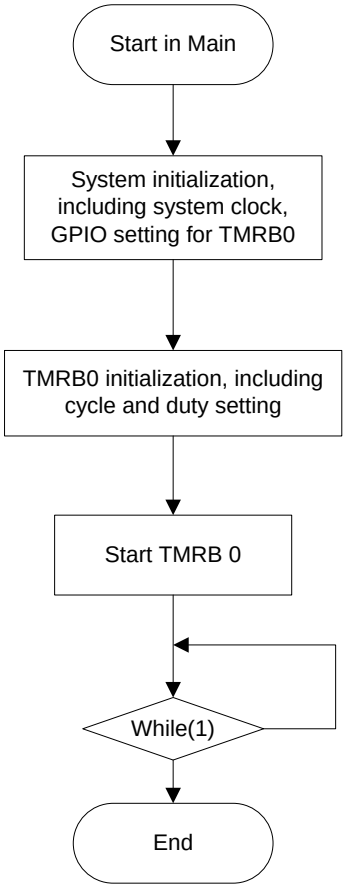
Data Fields:

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

11.3.5 Programming Example

This is a simple application based on the TMPM33x Peripheral Driver(TMRB), which use TMRB channel 0 internal interval timer mode to generate TMRB interrupt every 500ms.

11.3.5.1 Flow Chart



11.3.5.2 Code and Explanation for the Example

At first, in main(), create a TMRB_InitTypeDef structure, then fill all the data fields. For example,

```
TMRB_InitTypeDef myTB;
myTB.Mode = TMRB_INTERVAL_TIMER;
myTB.ClkDiv = TMRB_CLK_DIV_8;
myTB.TrailingTiming = TMRB_1ms;          /* Specific value depends on system clock */
myTB.UpCntCtrl = TMRB_AUTO_CLEAR;
myTB.LeadingTiming = TMRB_1ms / 2;      /* Specific value depends on system clock */
```

Then, enable and initialize the TMRB channel 0.

```
TMRB_Enable(TSB_TB0);
TMRB_Init(TSB_TB0, &myTB);
```

At the end of main(), enable INTTB0, and then start the counter of TMRB channel 0.

```
TMRB_SetRunState(TSB_TB0, TMRB_RUN);
```

12. SIO/UART

12.1 Overview

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

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

All driver APIs are contained in /Libraries/TX03_Periph_Driver/src/tmpm33**x**_uart.c(*), with /Libraries/TX03_Periph_Driver/inc/tmpm33**x**_uart.h(*) containing the macros, data types, structures and API definitions for use by applications.

***Note**: "**x**" can be 0, 2, 3.

12.2 Difference among TMPM330, TMPM332 and TMPM333 in SIO/UART

Either TMPM330 or TMPM333 has 3 serial I/O channels: SC0(**UART0**), SC1(**UART1**) and SC2(**UART2**). And TMPM332 has two channels: SC0(**UART0**) and SC1(**UART1**).

12.3 API Functions

12.3.1 Function List

- void UART_Enable(TSB_SC_TypeDef* ***UARTx***)
- void UART_Disable(TSB_SC_TypeDef* ***UARTx***)
- WorkState UART_GetBufState(TSB_SC_TypeDef* ***UARTx***, uint8_t ***Direction***)
- void UART_SWReset(TSB_SC_TypeDef* ***UARTx***)
- void UART_Init(TSB_SC_TypeDef* ***UARTx***, UART_InitTypeDef* ***InitStruct***)
- uint32_t UART_GetRxData(TSB_SC_TypeDef* ***UARTx***)
- void UART_SetTxData(TSB_SC_TypeDef* ***UARTx***, uint32_t ***Data***)
- void UART_DefaultConfig(TSB_SC_TypeDef* ***UARTx***)
- UART_Err UART_GetErrState(TSB_SC_TypeDef* ***UARTx***)

- void UART_SetWakeUpFunc(TSB_SC_TypeDef* ***UARTx***, FunctionalState ***NewState***)
- void UART_SetIdleMode(TSB_SC_TypeDef* ***UARTx***, FunctionalState ***NewState***)
- void UART_FIFOConfig(TSB_SC_TypeDef * ***UARTx***, FunctionalState ***NewState***)
- void UART_SetFIFOTransferMode(TSB_SC_TypeDef * ***UARTx***, uint32_t ***TransferMode***)
- void UART_TRxAutoDisable(TSB_SC_TypeDef * ***UARTx***, UART_TRxDisable ***TrxAutoDisable***
- void UART_RxFIFOINTCtrl(TSB_SC_TypeDef * ***UARTx***, FunctionalState ***NewState***)
- void UART_TxFIFOINTCtrl(TSB_SC_TypeDef * ***UARTx***, FunctionalState ***NewState***)
- void UART_RxFIFOByteSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***BytesUsed***)
- void UART_RxFIFOFillLevel(TSB_SC_TypeDef * ***UARTx***,uint32_t ***RxFIFOLevel***)
- void UART_RxFIFOINTSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***RxINTCondition***)
- void UART_RxFIFOClear(TSB_SC_TypeDef * ***UARTx***)
- void UART_TxFIFOFillLevel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***TxFIFOLevel***)
- void UART_TxFIFOINTSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***TxINTCondition***)
- void UART_TxFIFOClear(TSB_SC_TypeDef * ***UARTx***)
- uint32_t UART_GetRxFIFOFillLevelStatus(TSB_SC_TypeDef * ***UARTx***)
- uint32_t UART_GetRxFIFOOverRunStatus(TSB_SC_TypeDef * ***UARTx***)
- uint32_t UART_GetTxFIFOFillLevelStatus(TSB_SC_TypeDef * ***UARTx***)
- uint32_t UART_GetTxFIFOUnderRunStatus(TSB_SC_TypeDef * ***UARTx***)
- void SIO_Enable(TSB_SC_TypeDef * ***SIOx***)
- void SIO_Disable(TSB_SC_TypeDef * ***SIOx***)
- uint8_t SIO_GetRxData(TSB_SC_TypeDef * ***SIOx***)
- void SIO_SetTxData(TSB_SC_TypeDef * ***SIOx***, uint8_t ***Data***)
- void SIO_Init(TSB_SC_TypeDef * ***SIOx***, uint32_t ***IOClkSel***, SIO_InitTypeDef * ***InitStruct***)

12.3.2 Detailed Description

Functions listed above can be divided into four parts:

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

UART_TxFIFOClear(),UART_GetRxFIFOFillLevelStatus(),
UART_GetRxFIFOOverRunStatus(),UART_GetTxFIFOFillLevelStatus()and
UART_GetTxFIFOUnderRunStatus(),

12.3.3 Function Documentation

Note: in all of the following APIs, parameter "TSB_SC_TypeDef* ***UARTx***" can be one of Values below:
UART0, **UART1** or **UART2** (only for TMPM330/333)
parameter "TSB_SC_TypeDef* ***SIOx***" can be one of the following values:
SIO0, SIO1 or SIO2.(only for TMPM330/333)

12.3.3.1 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

12.3.3.2 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

12.3.3.3 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.

12.3.3.4 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

12.3.3.5 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 "12.3.4 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

12.3.3.6 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

12.3.3.7 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

12.3.3.8 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

12.3.3.9 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.

12.3.3.10 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

12.3.3.11 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

12.3.3.12 UART_FIFOConfig

Enable or disable the FIFO of specified UART channel.

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

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

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

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

Return:
None

12.3.3.13 UART_SetFIFOTransferMode

Transfer mode setting.

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

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

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

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

Return:
None

12.3.3.14 UART_TRxAutoDisable

Controls automatic disabling of transmission and reception.

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

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

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

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

Return:
None

12.3.3.15 UART_RxFIFOINTCtrl

Enable or disable receive interrupt for receive FIFO.

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

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

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

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

Return:
None

12.3.3.16 UART_TxFIFOINTCtrl

Enable or disable transmit interrupt for transmit FIFO.

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

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

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

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

Return:
None

12.3.3.17 UART_RxFIFOByteSel

Bytes used in receive FIFO.

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

Parameters:

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

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

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

Return:
None

12.3.3.18 UART_RxFIFOFillLevel

Receive FIFO fill level to generate receive interrupts.

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

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

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

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

Return:
None

12.3.3.19 UART_RxFIFOINTSel

Select RX interrupt generation condition.

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

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

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

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

Return:
None

12.3.3.20 UART_RxFIFOClear

Clear Receive FIFO.

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

Parameters:
UARTx is the specified UART channel.

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

Return:
None

12.3.3.21 UART_TxFIFOFillLevel

Transmit FIFO fill level to generate receive interrupts.

Prototype:
void
UART_TxFIFOFillLevel (TSB_SC_TypeDef * ***UARTx,***
uint32_t ***TxFIFOLevel***)

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

This parameter can be one of the following values:
UART_TXFIFO4B_FLEVLE_0_0B, UART_TXFIFO4B_FLEVLE_1_1B, UART_TXFIFO4B_FLEVLE_2_0B, UART_TXFIFO4B_FLEVLE_3_1B

Description:
This function will set Transmit FIFO fill level for generate receive interrupts of specified UART channel selected by ***UARTx***, But the level should be **UART_TXFIFO4B_FLEVLE_0_0B, UART_TXFIFO4B_FLEVLE_1_1B, UART_TXFIFO4B_FLEVLE_2_0B, UART_TXFIFO4B_FLEVLE_3_1B.**

Return:
None

12.3.3.22 UART_TxFIFOINTSel

Select TX interrupt generation condition.

Prototype:
void
UART_TxFIFOINTSel (TSB_SC_TypeDef * ***UARTx,***
uint32_t ***TxINTCondition***)

Parameters:
UARTx is the specified UART channel.
TxINTCondition is the TX interrupt generation condition.

This parameter can be one of the following values:
UART_TFIS_REACH_FLEVEL or **UART_TFIS_REACH_EXCEED_FLEVEL**

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

Return:

None

12.3.3.23 UART_TxFIFOClear

Clear Transmit FIFO.

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

Parameters:
UARTx is the specified UART channel.

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

Return:
None

12.3.3.24 UART_GetRxFIFOFillLevelStatus

Indicate the status of receive FIFO fill level.

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

Parameters:
UARTx is the specified UART channel.

Description:
This function will Indicate the receive FIFO fill level status, which UART channel selected by ***UARTx*.**

Return:
UART_TRXFIFO_EMPTY: TX FIFO fill level is empty.
UART_TRXFIFO_1B: TX FIFO fill level is 1 byte.
UART_TRXFIFO_2B: TX FIFO fill level is 2 bytes.
UART_TRXFIFO_3B: TX FIFO fill level is 3 bytes.
UART_TRXFIFO_3B: TX FIFO fill level is 4 bytes.

12.3.3.25 UART_GetRxFIFOOverRunStatus

Indicate the status of Receive FIFO overrun.

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

Parameters:
UARTx is the specified UART channel.

Description:
This function will Indicate the Receive FIFO overrun status, which UART channel selected by ***UARTx*****.**

Return:
UART_RXFIFO_OVERRUN: Flags for RX FIFO overrun.

12.3.3.26 UART_GetTxFIFOFillLevelStatus

Status of transmit FIFO fill level.

Prototype:
uint32_t
UART_GetTxFIFOFillLevelStatus (TSB_SC_TypeDef * ***UARTx***)

Parameters:
UARTx is the specified UART channel.

Description:
Status of transmit FIFO fill level.

Return:
UART_TRXFIFO_EMPTY: TX FIFO fill level is empty.
UART_TRXFIFO_1B: TX FIFO fill level is 1 byte.
UART_TRXFIFO_2B: TX FIFO fill level is 2 bytes.
UART_TRXFIFO_3B: TX FIFO fill level is 3 bytes.
UART_TRXFIFO_3B: TX FIFO fill level is 4 bytes.

12.3.3.27 UART_GetTxFIFOUnderRunStatus

Transmit FIFO under run.

Prototype:
uint32_t
UART_ GetTxFIFOUnderRunStatus (TSB_SC_TypeDef * ***UARTx***)

Parameters:
UARTx is the specified UART channel.

Description:
Transmit FIFO under run.

Return:
UART_TXFIFO_UNDERRUN: Flags for TX FIFO under-run.

12.3.3.28 SIO_Enable

Enable the specified SIO channel.

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

Parameters:
SIOx is the specified SIOx channel.

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

Return:
None

12.3.3.29 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

12.3.3.30 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.

12.3.3.31 SIO_SetTxData

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

Prototype:
void
SIO_SetTxData(TSB_SC_TypeDef* ***SIOx***,
uint8_t ***Data***)

Parameters:
SIOx is the specified SIO channel.
Data is a frame to be sent,

Description:

This function will set the data to be sent from the specified SIO channel selected by ***SIOx***.

Return:
None

12.3.3.32 SIO_Init

Initialize and configure the specified SIO channel.

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

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

12.3.4 Data Structure Description

12.3.4.1 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.

12.3.4.2 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:

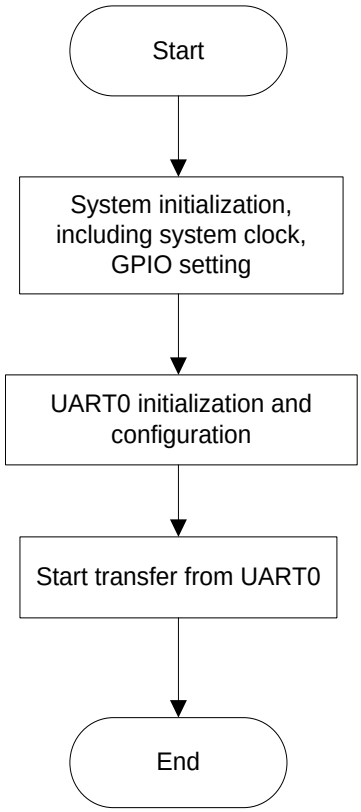
- **SIO_BR_DIVIDER_1**
- **SIO_BR_DIVIDER_2**
- **SIO_BR_DIVIDER_3**
- **SIO_BR_DIVIDER_4**
- **SIO_BR_DIVIDER_5**
- **SIO_BR_DIVIDER_6**
- **SIO_BR_DIVIDER_7**
- **SIO_BR_DIVIDER_8**
- **SIO_BR_DIVIDER_9**
- **SIO_BR_DIVIDER_10**
- **SIO_BR_DIVIDER_11**
- **SIO_BR_DIVIDER_12**

- **SIO_BR_DIVIDER_13**
- **SIO_BR_DIVIDER_14**
- **SIO_BR_DIVIDER_15**
- **SIO_BR_DIVIDER_16**

Programming Example

This is a simple application based on the TMPM33x Peripheral Driver(UART), which configures UART0 (Serial Channel 0) and start transmission.

12.3.4.3 Flow Chart



12.3.4.4 Code and Explanation for the Example

At first, create a UART_InitTypeDef struct and fill all the data fields. For example,

```
UART_InitTypeDef myUART;
myUART.BaudRate = 115200;
myUART.DataBits = UART_DATA_BITS_8;
myUART.StopBits = UART_STOP_BITS_1;
myUART.Parity = UART_NO_PARITY;
myUART.Mode = UART_ENABLE_RX | UART_ENABLE_TX;
myUART.FlowCtrl = UART_NONE_FLOW_CTRL;
```

Then enable, initialize and configure UART channels.

```
UART_Enable (UART0);
UART_Init(UART0, &myUART);
```

After the setting above, enable SC0 TX interrupt. Then start sending data.

```
UART_SetTxData(UART0, TxBuf[0]);
```

Here TxBuf is a character array.

The rest process of data flow can be finished in ISR of SC0 TX interrupt.

13. WDT

13.1 Overview

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

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

This driver is contained in \Libraries\TX03_Periph_Driver\src\tmpm33**x**_wdt.c (*), with \Libraries/TX03_Periph_Driver\inc\tmpm33**x**_wdt.h (*) containing the API definitions for use by applications.

***Note**: "**x**" can be 0, 2, 3.

13.2 Difference among TMPM330, TMPM332 and TMPM333 in WDT

None

13.3 API Functions

13.3.1 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)

13.3.2 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().

13.3.3 Function Documentation

13.3.3.1 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

13.3.3.2 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

13.3.3.3 WDT_SetOverflowOutput

Set WDT to generate NMI interrupt or to reset when the counter overflows.

Prototype:
void
WDT_SetOverflowOutput(uint32_t ***OverflowOutput***)

Parameters:
OverflowOutput: Select function of WDT when counter overflow.
This parameter can be one of the following values:

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

Description:
This function will set WDT to generate NMI interrupt if the counter overflows when ***OverflowOutput*** is **WDT_NMIINT,** and set WDT to generate reset if the counter overflows when ***OverflowOutput*** is **WDT_WDOUT**.

Return:
None

13.3.3.4 WDT_Init

Initialize and configure WDT.

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

Parameters:

InitStruct: The structure containing basic WDT configuration including detect time and WDT output when counter overflow. (Refer to "13.3.4 Data structure Description" for details)

Description:

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

Return:

None

13.3.3.5 WDT_Enable

Enable the WDT function.

Prototype:

void
WDT_Enable(void)

Parameters:

None

Description:

This function will enable WDT.

Return:

None

13.3.3.6 WDT_Disable

Disable the WDT function.

Prototype:

void
WDT_Disable(void)

Parameters:

None

Description:

This function will disable WDT.

Return:
None

13.3.3.7 WDT_WriteClearCode

Write the clear code.

Prototype:
void
WDT_WriteClearCode (void)

Parameters:
None

Description:
This function will clear the WDT counter.

Return:
None

13.3.4 Data Structure Description

13.3.4.1 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.

13.3.5 Programming Example

This is a simple application based on the TMPM33x Peripheral Driver (WDT). The following example shows how to set up the watchdog timer API to generate NMI interrupt.

13.3.5.1 Flow Chart

None

13.3.5.2 Code and Explanation for the Example

At first, in main(), create a WDT_InitTypeDef struct and fill in all the data fields.

```
WDT_InitTypeDef WDT_InitStruct;
WDT_InitStruct.DetectTime = WDT_DETECT_TIME_EXP_25;
WDT_InitStruct.OverflowOutput = WDT_NMIINT;
```

Initialize and configure Watch Dog timer then enable it.

```
WDT_Init(&WDT_InitStruct);
WDT_Enable();
```

Wait for the NMI interrupt to occur.

```
while(1)
{
}
```