TOSHIBA

TX00 ペリフェラルドライバ
ユーザーガイド
(TMPM061)

第一版
2017 年 9 月

東芝デバイス＆ストレージ株式会社

CMDR-M061UG-01J

本製品取り扱い上のお願い

- ソフトウエア使用権許諾契約書の同意無しに使用しないで下さい。

目次

1. はじめに

本ペリフェラルドライバセットは、東芝製マイコンTMPM061用です。

TX00 ペリフェラルドライバでは、ユーザーアプリケーション内で各ペリフェラルを簡単に使用するためのマクロ、データ構造、関数および使用例を用意しています。

TMPM061 ペリフェラルドライバは以下の仕様に基づいています。
- スタートアップルーチンといくつかの関数を除き、C 言語で記述されています。

2. TX00 ペリフェラルドライバの構成

/Libraries
TX00 CMSIS ファイルと TMPM061 ペリフェラルドライバが格納されています。

/Libraries/ TX00_CMSIS
このフォルダには TMPM061 CMSIS ファイルのデバイス・ペリフェラル・アクセス・レイヤーが格納されています。

/Libraries/TX00_Periph_Driver
TMPM061 ペリフェラルドライバの全てのソースコードが格納されています。

/Libraries/TX00_Periph_Driver/inc
TMPM061 ペリフェラルドライバのヘッダファイルが格納されています。

/Libraries/TX00_Periph_Driver/src
TMPM061 ペリフェラルドライバのソースファイルが格納されています。

/Project
TMPM061 ペリフェラルドライバのテンプレートプロジェクトと使用例が格納されています。

/Project/Template
TMPM061 ペリフェラルドライバのテンプレートプロジェクトが格納されています。

/Project/Examples
TMPM061 ペリフェラルドライバの使用例が格納されています。

/Utilities/TMPM061-EVAL
TMPM061 ボードのハードウェアリソース用の設定ファイル、およびドライバファイル (例： led, key) が格納されています。

3. ADC

3.1 概要

本デバイスは、10 ビット逐次変換方式アナログ/デジタルコンバータ(AD コンバータ)を 1 ユニット内蔵します。

TMPM061FWFG では ADC の以下の機能は使用できません。関連するレジスタの設定は行わないでください。

機能	レジスタ
最優先変換	ADMOD2, ADREGSP
AD 監視機能	ADMOD3, ADMOD5, ADCMP0, ADCMP1
AD 変換のハードウエアによる起動	ADMOD4 <ADHTG> <ADHS> <HADHTG> <HADHS>

TMPM061FWFG では ADC の入力チャネルとして 0~3 の 4 チャネルを使用します。各チャネルに入力されるアナログ信号は以下の通りです。

チャネル	入力
チャネル 0	AIN0 端子 (PF0/98pin)
チャネル 1	AIN1 端子 (PF1/99pin)
チャネル 2	温度センサ出力
チャネル 3	ΔΣADC 基準電圧回路(BGR)出力

変換チャネルは ADMOD0<SCAN>、ADMOD1<ADSCN> <ADCH>で指定します。使用可能な設定を下表に示します。

		ADMOD1<ADCH[3:0]>			
		0000	0001	0010	0011 to 1111
ADMOD0<SCAN>=0	チャネル固定	AIN0	AIN1	AIN2	設定不可
ADMOD0<SCAN>=1	ADMOD1<ADSCN>=00 4 チャネルスキャン	AIN0	AIN0 ~ AIN1	AIN0 ~ AIN2	
	ADMOD1<ADSCN>=01 8 チャネルスキャン	AIN0	AIN0 ~ AIN1	AIN0 ~ AIN2	
	ADMOD1<ADSCN>=02 12 チャネルスキャン	AIN0	AIN0 ~ AIN1	AIN0 ~ AIN2	

ADC ドライバ API は、各モジュールの設定機能を持ち、チャネル選択、モード設定、モニタ機能設定、割り込み設定、ステータスリード、AD 変換結果の取得などの機能を提供します。

全ドライバ API は、アプリで使用する API 定義を格納する以下のファイルで構成されています。

/Libraries/TX00_Periph_Driver\src\tmpm061_adc.c
/Libraries/TX00_Periph_Driver\inc\tmpm061_adc.h

3.2 API 関数

3.2.1 関数一覧

- void ADC_SWReset(void)
- void ADC_SetClk(uint32_t ***Conversion_Time***, uint32_t ***Prescaler_Output***)

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

3.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。

1) AD 変換設定:
   ADC_SetClk(), ADC_SetScanMode(), ADC_SetRepeatMode(), ADC_SetINTMode(), ADC_SetInputChannel(), ADC_SetChannelScanMode(), ADC_SetVref()
2) AD 変換の開始:
   ADC_Start()
3) AD 変換ステータス/結果の読み出し:
   ADC_GetConvertState(), ADC_GetConvertResult()
4) その他:
   ADC_SWReset(), ADC_SetIdleMode()

3.2.3 関数仕様

3.2.3.1 ADC_SWReset

ADC のソフトウェアリセット

関数のプロトタイプ宣言:
void
ADC_SWReset(void)

引数:
なし

機能:
ADC をソフトウェアリセットします。

戻り値:
なし

3.2.3.2 ADC_SetClk

AD 変換サンプルホールド時間とプリスケーラ出力(SCLK)の設定

関数のプロトタイプ宣言:
void
ADC_SetClk(uint32_t ***Conversion_Time***,
uint32_t ***Prescaler_Output***)

引数:
Conversion_Time: 以下から ADC 変換時間を選択します。

- **ADC_CONVERSION_35_CLOCK:** 35.5 クロック
- **ADC_CONVERSION_42_CLOCK:** 42 クロック
- **ADC_CONVERSION_68_CLOCK:** 68 クロック
- **ADC_CONVERSION_81_CLOCK:** 81 クロック

Prescaler_Output: 以下から ADC プリスケーラ出力(ADCLK)を選択します。
- **ADC_FC_DIVIDE_LEVEL_1:** fc
- **ADC_FC_DIVIDE_LEVEL_2:** fc / 2
- **ADC_FC_DIVIDE_LEVEL_4:** fc / 4
- **ADC_FC_DIVIDE_LEVEL_8:** fc / 8
- **ADC_FC_DIVIDE_LEVEL_16:** fc / 16

機能:
Conversion_Time で ADC 変換時間を設定し、***Prescaler_Output*** でプリスケーラ出力を設定します。

補足:
AD変換中は、この関数を使わないでください。またAD変換状態を確認するための**ADC_GetConvertState()**が**BUSY**でない場合、この関数をコールすることができます。

戻り値:
なし

3.2.3.3 ADC_Start

AD 変換の開始

関数のプロトタイプ宣言:
void
ADC_Start(void)

引数:
なし

機能:
AD 変換を開始します。

補足:
この関数をコールする前に、以下のいずれかのモードを選択してください:
チャネル固定シングル変換モード
チャネルスキャンシングル変換モード
チャネル固定リピート変換モード
チャネルスキャンリピート変換モード
詳細は、**ADC_SetScanMode(), ADC_SetRepeatMode(), ADC_SetInputChannel(), ADC_SetScanChannel()** を参照してください。

AD 変換をスタートさせる場合、**ADC_SetVref (ENABLE)**をコールして Vref を有効にしてください。なお、Vref 有効後、3 μs の安定時間が必要です。その後、**ADC_Start()**をコールしてください。

戻り値:

なし

3.2.3.4 ADC_SetScanMode

スキャンモードの設定

関数のプロトタイプ宣言:
void
ADC_SetScanMode(FunctionalState ***NewState***)

引数:
NewState: 以下から、スキャンモードを設定します。
- **ENABLE:** チャネルスキャン
- **DISABLE:** チャネル固定

機能:
AD 変換スキャンモードを設定します。

戻り値:
なし

3.2.3.5 ADC_SetRepeatMode

リピートモードの設定

関数のプロトタイプ宣言:
void
ADC_SetRepeatMode(FunctionalState ***NewState***)

引数:
NewState: 以下から、リピートモードを設定します。
- **ENABLE:** リピート変換
- **DISABLE:** シングル変換

機能:
リピートモードを設定します。

戻り値:
なし

3.2.3.6 ADC_SetINTMode

チャネル固定リピート変換モード時の割り込みタイミングの設定

関数のプロトタイプ宣言:
void
ADC_SetINTMode(uint8_t ***INTMode***)

引数:
INTMode: 以下から、割り込みタイミングを選択します。
- **ADC_INT_SINGLE:** 1 回毎、割り込み発生

- **ADC_INT_CONVERSION_4:** 4 回毎、割り込み発生
- **ADC_INT_CONVERSION_8:** 8 回毎、割り込み発生

機能:
チャネル固定リピート変換モード時の割り込みタイミングを設定します。

補足:
この関数は、チャネル固定リピート変換モード時のみ有効です。
以下は、チャネル固定リピート変換モードの例です:
1. **ADC_SetScanMode(DISABLE)**.
2. **ADC_SetRepeatMode(ENABLE)**.

戻り値:
なし

3.2.3.7 ADC_GetConvertState

AD 変換終了フラグの取得

関数のプロトタイプ宣言:
WorkState
ADC_GetConvertState(void)

引数:
なし

機能:
AD 変換終了フラグ (通常)を取得します。この関数は、AD 変換が終了したかどうかを確認するために使います。

戻り値:
AD 変換状態:
DONE: AD 変換終了
BUSY: AD 変換中

3.2.3.8 ADC_SetInputChannel

アナログ入力チャネルの選択

関数のプロトタイプ宣言:
void
ADC_SetInputChannel(uint8_t ***InputChannel***)

引数:
InputChannel: 以下から、いずれか 1 つのアナログ入力チャネルを使用します。
- **ADC_AN_0:** AIN0 端子
- **ADC_AN_1:** AIN1 端子
- **ADC_AN_2:** 温度センサ出力

機能:
アナログ入力チャネルを選択します。

戻り値:
なし

3.2.3.9 ADC_SetChannelScanMode

チャネルスキャンモード時の動作選択

関数のプロトタイプ宣言:
void
ADC_SetChannelScanMode(ADC_ChannelScanMode ***ScanMode***)

引数:
ScanMode: 以下から、チャネルスキャンモード時の動作を選択します。
- **ADC_SCAN_4CH**
- **ADC_SCAN_8CH**
- **ADC_SCAN_12CH**

機能:
チャネルスキャンモードを選択した際の動作を選択します。
アナログ入力チャネルの設定により変換するチャネルが決まります。下表に変換チャネルを示します。

		ADC_SetInputChannel()			
		ADC_AN_0	ADC_AN_1	ADC_AN_2	
ADC_SetScanMode(DISABLE)	チャネル固定スキャンモード	AIN0	AIN1	AIN2	
ADC_SetScanMode(ENABLE)	ADC_SetChannelScanMode(ADC_SCAN_4CH) 4 チャネルスキャン	AIN0	AIN0 ~ AIN1	AIN0 ~ AIN2	
	ADC_SetChannelScanMode(ADC_SCAN_8CH) 8 チャネルスキャン	AIN0	AIN0 ~ AIN1	AIN0 ~ AIN2	
	ADC_SetChannelScanMode(ADC_SCAN_12CH) 12 チャネルスキャン	AIN0	AIN0 ~ AIN1	AIN0 ~ AIN2	

戻り値:
なし

3.2.3.10 ADC_SetIdleMode

IDLE モード時の ADC 動作制御の指定

関数のプロトタイプ宣言:
void
ADC_SetIdleMode(FunctionalState ***NewState***)

引数:
NewState: IDLE モード時の ADC 動作状態を指定します。
- **ENABLE :** 動作
- **DISABLE :** 停止

機能:
IDLE モード時の ADC 動作制御の動作/停止を指定します。
システムが IDLE モードに遷移する前に実行する必要があります。

戻り値:
なし

3.2.3.11 ADC_SetVref

ADC Vref アプリケーションの回路 ON/OFF 制御

関数のプロトタイプ宣言:
void
ADC_SetVref(FunctionalState ***NewState***)

引数:
NewState: ADC Vref アプリケーションの回路 ON/OFF を指定します。
- **ENABLE :** Vref ON
- **DISABLE :** Vref OFF

機能:
ADC Vref アプリケーションの回路 ON/OFF を制御します。

補足:
スタンバイモード遷移前に **ADC_SetVref(DISABLE**)を実行してください。

戻り値:
なし

3.2.3.12 ADC_GetConvertResult

AD 変換レジスタの変換結果格納フラグステート、オーバーランフラグ、変換結果の確認

関数のプロトタイプ宣言:
ADC_ResultTypeDef
ADC_GetConvertResult(uint8_t ***ADREGx***)

引数:
ADREGx: AD 変換結果レジスタを選択します。
- **ADC_REG_0**, **ADC_REG_1**, **ADC_REG_2**

機能:
ADREGx に設定された AD 変換結果格納フラグ、オーバーランフラグ、変換結果を確認します。

戻り値:
ADC_Result 構造体による AD 変換結果

3.2.4 データ構造

3.2.4.1 ADC_ResultTypeDef

構造体のメンバ ure:

uint32_t		
All: AD 変換結果		
Bit		
uint32_t		
Stored:	1	AD 変換結果格納フラグ '1': 変換結果あり '0': 変換結果なし
uint32_t		
OverRun:	1	オーバーランフラグ '1': 発生あり '0': 発生なし
uint32_t		
Reserved1:	4	未使用
uint32_t		
ADResult:	10	AD 変換結果
uint32_t		
Reserved2:	16	未使用

4. CG

4.1 概要

クロック/モード制御ブロックでは、クロックギアやプリスケーラクロックの選択、発振器のウォーミングアップ等を設定することが可能です。また、低消費電力モードがあり、モード遷移を行うことで電力の消費を抑えることが可能です。
クロックに関連する機能としては以下のようなものがあります。

・システムクロックの制御
・プリスケーラクロックの制御
・ウォーミングアップタイマの制御

また、動作モードとして NORMAL モードと各種低消費電力モードがあり、使用方法に応じて消費電力を抑えることができます。

全ドライバ API は、アプリで使用する API 定義、マクロ、データタイプ、構造を格納する以下のファイルで構成されています。
/Libraries/TX00_Periph_Driver\src\tmpM061_cg.c
/Libraries/TX00_Periph_Driver\inc\tmpM061_cg.h

CG のクロックとして、以下のシンボルを使用しています。詳しくは MCU データシートの「クロックシステムブロック図」を参照してください。

fEHOSC : 外部高速発振クロック、X1、X2 端子に接続する発振子より入力されるクロック
fEHCLKIN : 外部高速クロック、X1 から入力する高速クロック
fIHOSC : 内部高速発振クロック、内部高速発振器より入力されるクロック
fs : 外部低速発振クロック、XT1、XT2 端子に接続する発振子より入力されるクロック
fELCLKIN : 外部低速クロック、PJ5(33pin)から入力する低速クロック
fosc : fEHOSC または fIHOSC のどちらか選択されたクロック
fc : CGEHCLKSEL<EHCLKSEL>で選択されたクロック(高速クロック)
fgear : CGSYSCR<GEAR[2:0]>で選択された分周クロック
fsys : CGCKSEL<SYSCK>で選択されたクロック(システムクロック)
fperiph : CGSYSCR<FPSEL[1:0]>で選択されたプリスケーラ用クロック
φT0 : CGSYSCR<PRCK[2:0]>で選択されたクロック (プリスケーラクロック)

高速クロック fc、プリスケーラクロック φT0 は以下のように分周することが可能です。
・高速クロック: fc, fc/2, fc/4, fc/8, fc/16
・プリスケーラクロック: fperiph, fperiph/2, fperiph/4, fperiph/8, fperiph/16, fperiph/32

4.2 API 関数

4.2.1 関数一覧

- void CG_SetFgearLevel(CG_DivideLevel ***DivideFgearFromFc***)
- CG_DivideLevel CG_GetFgearLevel(void)
- void CG_SetPhiT0Src(CG_PhiT0Src ***PhiT0Src***)

- CG_PhiT0Src CG_GetPhiT0Src(void)
- Result CG_SetPhiT0Level(CG_DivideLevel ***DividePhiT0FromFc***)
- CG_DivideLevel CG_GetPhiT0Level(void)
- void CG_SetSCOUTSrc(CG_SCOUTSrc ***Source***)
- CG_SCOUTSrc CG_GetSCOUTSrc(void)
- void CG_SetWarmUpTime(CG_WarmUpSrc ***Source***, uint16_t ***Time***)
- void CG_StartWarmUp(void)
- WorkState CG_GetWarmUpState(void)
- Result CG_SetFosc(CG_FoscSrc ***Source***, FunctionalState ***NewState***)
- void CG_SetFoscSrc(CG_FoscSrc ***Source***)
- CG_FoscSrc CG_GetFoscSrc(void)
- FunctionalState CG_GetFoscState(CG_FoscSrc ***Source***)
- Result CG_SetFs(FunctionalState ***NewState***)
- FunctionalState CG_GetFsState(void)
- void CG_SetPortM(CG_PortMMode ***Mode***)
- void CG_SetLowOscSrc(CG_LoscSrc ***Source***)
- void CG_SetExtHighClk(FunctionalState ***NewState***)
- void CG_SetSTBYMode(CG_STBYMode ***Mode***)
- CG_STBYMode CG_GetSTBYMode(void)
- void CG_SetExitStopModeFosc(FunctionalState ***NewState***)
- FunctionalState CG_GetExitStopModeFoscState(void)
- void CG_SetExitStopModeFs(FunctionalState ***NewState***)
- FunctionalState CG_GetExitStopModeFsState(void)
- void CG_SetPinStateInStopMode(FunctionalState ***NewState***)
- FunctionalState CG_GetPinStateInStopMode(void)
- void CG_SelExtHighClk(CG_EHClkSrc ***Source***)
- Result CG_SetFsysSrc(CG_FsysSrc ***Source***)
- CG_FsysSrc CG_GetFsysSrc(void)
- void CG_SetSTBYReleaseINTSrc(CG_INTSrc ***INTSource***,
  CG_INTActiveState ***ActiveState***,
  FunctionalState ***NewState***)
- CG_INTActiveState CG_GetSTBYReleaseINTState(CG_INTSrc ***INTSource***)
- void CG_ClearINTReq(CG_INTSrc ***INTSource***)
- CG_ResetFlag CG_GetResetFlag(void)

4.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています。

1) クロックの選択:
   CG_SetFgearLevel(), CG_GetFgearLevel(), CG_SetPhiT0Src(), CG_GetPhiT0Src(), CG_SetPhiT0Level(), CG_GetPhiT0Level(), CG_SetSCOUTSrc(), CG_GetSCOUTSrc(), CG_SetWarmUpTime(), CG_StartWarmUp(), CG_GetWarmUpState(), CG_SetFosc(),CG_SetFoscSrc(), CG_GetFoscSrc(), CG_GetFoscState(), CG_SetFs(), CG_GetFsState(), CG_SetFsysSrc(), CG_GetFsysSrc(), CG_SetPortM()
2) スタンバイモードの設定:
   CG_SetSTBYMode(), CG_GetSTBYMode( ), CG_SetExitStopModeFosc(), CG_GetExitStopModeFoscState(), CG_SetExitStopModeFs(), CG_GetExitStopModeFsState(), CG_SetPinStateInStopMode(), CG_GetPinStateInStopMode()
3) 割り込みの設定:
   CG_SetSTBYReleaseINTSrc(), CG_GetSTBYReleaseINTState(), CG_ClearINTReq(),CG_GetResetFlag()

4.2.3 関数仕様

4.2.3.1 CG_SetFgearLevel

fgear,fc 間の分周レベル設定

関数のプロトタイプ宣言:
void
CG_SetFgearLevel(CG_DivideLevel ***DivideFgearFromFc***)

引数:
DivideFgearFromFc: 以下から、fgear,fc 間の分周レベルを選択します。
- **CG_DIVIDE_1**: fgear = fc
- **CG_DIVIDE_2**: fgear = fc/2
- **CG_DIVIDE_4**: fgear = fc/4
- **CG_DIVIDE_8**: fgear = fc/8
- **CG_DIVIDE_16**: fgear = fc/16

機能:
fgear,fc 間の分周レベルを設定します。

戻り値:
なし

4.2.3.2 CG_GetFgearLevel

fgear,fc 間の分周レベルの取得

関数のプロトタイプ宣言:
CG_DivideLevel
CG_GetFgearLevel (void)

引数:
なし

機能:
fgear,fc 間の分周レベルを取得します。
レジスタから読み出した値が“Reserved” の場合、**CG_DIVIDE_UNKNOWN** を返します。

戻り値:
fgear, fc 間の分周レベルで、下記のいずれかの値になります。
CG_DIVIDE_1: fgear = fc
CG_DIVIDE_2: fgear = fc/2
CG_DIVIDE_4: fgear = fc/4
CG_DIVIDE_8: fgear = fc/8
CG_DIVIDE_16: fgear = fc/16
CG_DIVIDE_UNKNOWN: 無効なデータ

4.2.3.3 CG_SetPhiT0Src

PhiT0(ΦT0) ,fc 間の PhiT0(ΦT0) ソースの設定

関数のプロトタイプ宣言:
void
CG_SetPhiT0Src(CG_PhiT0Src ***PhiT0Src***)

引数:
PhiT0Src: 以下から PhiT0 ソースを選択します。
- **CG_PHIT0_SRC_FGEAR** : fgear が PhiT0 ソース
- **CG_PHIT0_SRC_FC**: fc が PhiT0 ソース
- **CG_PHIT0_SRC_FS**: PhiT0 ソースが fs

機能:
PhiT0 (ΦT0) ソースを選択します。

戻り値:
なし

4.2.3.4 CG_GetPhiT0Src

PhiT0 (ΦT0) ソースの取得

関数のプロトタイプ宣言:
CG_PhiT0Src
CG_GetPhiT0Src (void)

引数:
なし

機能:
PhiT0 (ΦT0) ソースを取得します。

戻り値:
CG_PHIT0_SRC_FGEAR : fgear が PhiT0 ソース
CG_PHIT0_SRC_FC: fc が PhiT0 ソース
CG_PHIT0_SRC_FS: PhiT0 ソースが fs

4.2.3.5 CG_SetPhiT0Level

PhiT0 (ΦT0) と fc 間の分周レベルの設定

関数のプロトタイプ宣言:
Result
CG_SetPhiT0Level (CG_DivideLevel ***DividePhiT0FromFc***)

引数:
DividePhiT0FromFc: PhiT0 (ΦT0) と fc 間の分周レベルを下記の値から設定します。
- **CG_DIVIDE_1**: ΦT0 = fc
- **CG_DIVIDE_2**: ΦT0 = fc/2
- **CG_DIVIDE_4**: ΦT0 = fc/4
- **CG_DIVIDE_8**: ΦT0 = fc/8
- **CG_DIVIDE_16**: ΦT0 = fc/16
- **CG_DIVIDE_32**: ΦT0 = fc/32
- **CG_DIVIDE_64**: ΦT0 = fc/64

- **CG_DIVIDE_128**: ΦT0 = fc/128
- **CG_DIVIDE_256**: ΦT0 = fc/256
- **CG_DIVIDE_512**: ΦT0 = fc/512

機能:
プリスケーラークロックの分周レベルを設定します。

戻り値:
SUCCESS: 設定成功
ERROR:エラー

4.2.3.6 CG_GetPhiT0Level

PhiT0(ΦT0) ,fc 間の分周レベルの取得

関数のプロトタイプ宣言:
CG_DivideLevel
CG_GetPhiT0Level(void)

引数:
なし

機能:
PhiT0(ΦT0) ,fc 間の分周レベルを取得します。
レジスタから読み出した値が“Reserved”の場合、**CG_DIVIDE_UNKNOWN** を返します。

戻り値:
PhiT0(ΦT0) ,fc 間の分周レベルを以下から設定します。
CG_DIVIDE_1: ΦT0 = fc
CG_DIVIDE_2: ΦT0 = fc/2
CG_DIVIDE_4: ΦT0 = fc/4
CG_DIVIDE_8: ΦT0 = fc/8
CG_DIVIDE_16: ΦT0 = fc/16
CG_DIVIDE_32: ΦT0 = fc/32
CG_DIVIDE_64: ΦT0 = fc/64
CG_DIVIDE_128: ΦT0 = fc/128
CG_DIVIDE_256: ΦT0 = fc/256
CG_DIVIDE_512: ΦT0 = fc/512
CG_DIVIDE_UNKNOWN : 無効データ

4.2.3.7 CG_SetSCOUTSrc

SCOUT 出力ソースクロック設定

関数のプロトタイプ宣言:
void
CG_SetSCOUTSrc(CG_SCOUTSrc ***Source***)

引数:
Source: 以下から、SCOUT 出力のソースクロックを選択します。
- **CG_SCOUT_SRC_FS**: fs に設定

- **CG_SCOUT_SRC_HALF_FSYS**: fsys/2 に設定
- **CG_SCOUT_SRC_FSYS**: SCOUT fsys に設定
- **CG_SCOUT_SRC_PHIT0**: SCOUTΦT0 に設定

機能:
SCOUT 出力のソースクロックを設定します。

戻り値:
なし

4.2.3.8 CG_GetSCOUTSrc

SCOUT 出力ソースクロック設定の取得

関数のプロトタイプ宣言:
SCOUTSrc
CG_GetSCOUTSrc(void)

引数:
なし

機能:
SCOUT 出力ソースクロック設定を取得します。

戻り値:
SCOUT 出力ソースクロック設定:
CG_SCOUT_SRC_FS: fs に設定
CG_SCOUT_SRC_HALF_FSYS: fsys/2 に設定
CG_SCOUT_SRC_FSYS: fsys に設定
CG_SCOUT_SRC_PHIT0: ΦT0 に設定

4.2.3.9 CG_SetWarmUpTime

ウォームアップ時間の設定

関数のプロトタイプ宣言:
void
CG_SetWarmUpTime (CG_WarmUpSrc ***Source***,
uint16_t ***Time***)

引数:
Source: 以下から、ウォームアップカウンタのソースクロックを選択します。
- **CG_WARM_UP_SRC_OSC1**: fosc1 に設定
- **CG_WARM_UP_SRC_OSC2**: fosc2 に設定
- **CG_WARM_UP_SRC_XT1**: fs に設定

Time: Source が **CG_WARM_UP_SRC_OSC1** または **CG_WARM_UP_SRC_OSC2** の場合、クロックは 0U から 0x1000U になります。 ***Source*** が **CG_WARM_UP_SRC_XT1** の場合、クロックは 0U から 0x4000U になります。

機能:

ウォームアップ時間とウォームアップカウンタを設定します。計算式は下記になります。
Setting_value = ((warm-up time) / (input cycle time by frequency))/16

ウォームアップ時間の計算レジスタ値例:
/* set up warm time 100us, input cycle by frequency is 8M */
value = 100*10E(-6)/(1/(8*10E(6)))/16=0x0320>>4=0x32

戻り値:
なし

4.2.3.10 CG_StartWarmUp

ウォームアップ開始

関数のプロトタイプ宣言:
void
CG_StartWarmUp (void)

引数:
なし

機能:
ウォームアップを開始します。

戻り値:
なし

4.2.3.11 CG_GetWarmUpState

ウォーミングアップ動作状態 (動作中、完了)の確認

関数のプロトタイプ宣言:
WorkState
CG_GetWarmUpState (void)

引数:
なし

機能:
ウォーミングアップ動作状態を確認します。
ウォームアップ時間の使用例:
CG_SetWarmUpTime(CG_WARM_UP_SRC_OSC1, 0x32);
/* start warm up */
CG_StartWarmUp();
/* check warm up is finished or not */
While(CG_GetWarmUpState() == BUSY);

戻り値:
ウォーミングアップ動作状態:
DONE:ウォーミングアップ動作終了
BUSY:ウォーミングアップ動作中

4.2.3.12 CG_SetLowOscSrc

低速クロックの選択

関数のプロトタイプ宣言:
void
CG_SetLowOscSrc(CG_LoscSrc ***Source***)

引数:
Source: 以下から低速クロックを選択します。
- CG_LOSC_OSC: 低速発振器入力 (fELOSC)
- CG_LOSC_CLK : 低速クロック入力 (fELCLKIN)

機能:
低速クロックを選択します。

戻り値:
なし

4.2.3.13 CG_SetExtHighClk

外部高速発振器の動作選択

関数のプロトタイプ宣言:
void
CG_SetExtHighClk (FunctionalState ***NewState***)

引数:
NewState: 以下から外部高速発振器の動作を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
外部高速発振器の動作を選択します。

戻り値:
なし

4.2.3.14 CG_SetFosc

高速発振器の動作選択 (外部、内部)

関数のプロトタイプ宣言:
Result
CG_SetFosc(CG_FoscSrc ***Source***,
FunctionalState ***NewState***)

引数:
Source: 以下から、高速発振器を選択します。
- **CG_FOSC_OSC1**: 外部高速発振器
- **CG_FOSC_OSC2**: 内部高速発振器

NewState: 以下から高速発振器の動作を選択します。
- **ENABLE**: 発振
- **DISABLE**: 停止

機能:
高速発振器の動作を選択します。
システムクロック(fsys)として fgear を選択した場合、高速発振器は選択できず **ERROR** を返します。

戻り値:
SUCCESS: 成功
ERROR: 失敗

4.2.3.15 CG_SetFoscSrc

高速発振器のソース選択

関数のプロトタイプ宣言:
void
CG_SetFoscSrc(CG_FoscSrc ***Source***)

引数:
Source: 以下から高速発振器のソースを選択します。
- **CG_FOSC_OSC1**: 外部発振器
- **CG_FOSC_OSC2**: 内部発振器

機能:
高速発振器のソースを選択します。

戻り値:
なし

4.2.3.16 CG_GetFoscSrc

高速発振器のソース選択状態の取得

関数のプロトタイプ宣言:
CG_FoscSrc
CG_GetFoscSrc(void)

引数:
なし

機能:
高速発振器のソース選択状態を取得します。

戻り値:
高速発振器のソース
CG_FOSC_OSC1: 外部発振器
CG_FOSC_OSC2: 内部発振器

4.2.3.17 CG_GetFoscState

高速発振器の動作選択状態の取得

関数のプロトタイプ宣言:
FunctionalState
CG_GetFoscState(CG_FoscSrc ***Source***)

引数:
Source: 以下から高速発振器を選択します。
- **CG_FOSC_OSC1**: 外部高速発振器
- **CG_FOSC_OSC2**: 内部高速発振器

機能:
高速発振器の動作選択状態を取得します。

戻り値:
高速発振器の動作選択状態
ENABLE: 発振
DISABLE: 停止

4.2.3.18 CG_SetFs

外部低速発振器(XT1)の動作選択

関数のプロトタイプ宣言:
Result
CG_SetFs(FunctionalState ***NewState***)

引数:
NewState: 以下から外部低速発振器の動作を選択します。
- **ENABLE**: 発振
- **DISABLE**: 停止

機能:
外部低速発振器の動作を選択します。
システムクロック(fsys)として fs が選択されている場合、低速発振器(XT1)は停止できません。この場合、本 API は **ERROR** を返却します。

戻り値:
外部低速発振器の動作選択:
SUCCESS: 成功
ERROR: 失敗

4.2.3.19 CG_GetFsState

外部低速発振器(XT1)の動作選択状態の取得

関数のプロトタイプ宣言:
FunctionalState
CG_GetFsState (void)

引数:
なし

機能:
外部低速発振器(XT1)の動作選択状態を取得します。

戻り値:
外部低速発振器(XT1)の動作選択状態:
ENABLE: 発振
DISABLE: 停止

4.2.3.20 CG_SetPortM

外部高速発振器選択

関数のプロトタイプ宣言:
void
CG_SetPortM(CG_PortMMode ***Mode***)

引数:
Mode:以下から外部高速発振器を選択します
- **CG_PORTM_AS_GPIO** : X1/X2(fEHOSC)を使用しない。
- **CG_PORTM_AS_HOSC**: X1/X2(fEHOSC)を使用する。

機能:
外部高速発振器を選択します。

戻り値:
なし

4.2.3.21 CG_SetSTBYMode

低消費電力モード選択

関数のプロトタイプ宣言:
void
CG_SetSTBYMode(CG_STBYMode ***Mode***)

引数:
Mode: 以下から低消費電力モードを選択します。
- **CG_STBY_MODE_STOP**: STOP モード
- **CG_STBY_MODE_SLEEP**: SLEEP モード
- **CG_STBY_MODE_IDLE**: IDLE モード

機能:
低消費電力モードを選択します。

戻り値:
なし

4.2.3.22 CG_GetSTBYMode

低消費電力モード選択状態の取得

関数のプロトタイプ宣言:
CG_STBYMode
CG_GetSTBYMode (void)

引数:
なし

機能:
低消費電力モード選択状態を取得します。
選択状態が“Reserved”の場合、本 API は“**CG_STBY_MODE_UNKNOWN**”を返却します。

戻り値:
低消費電力モードの選択状態:
CG_STBY_MODE_STOP: STOP モード
CG_STBY_MODE_SLEEP: SLEEP モード
CG_STBY_MODE_IDLE: IDLE モード
CG_STBY_MODE_UNKNOWN: 無効データ

4.2.3.23 CG_SetExitStopModeFosc

STOP モード解除後の自動高速発振選択

関数のプロトタイプ宣言:
void
CG_SetExitStopModeFosc(FunctionalState ***NewState***)

引数:
NewState: 以下から STOP モード解除後の自動高速発振を選択します。
- **ENABLE :** 発振
- **DISABLE :** 停止

機能:
STOP モード解除後の自動高速発振を選択します。

戻り値:
なし

4.2.3.24 CG_GetExitStopModeFoscState

STOP モード解除後の自動高速発振選択状態の取得

関数のプロトタイプ宣言:
FunctionalState
CG_GetExitStopModeFoscState (void)

引数:
なし

機能:
STOP モード解除後の自動高速発振選択状態を取得します。

戻り値:
STOP モード解除後の自動高速発振選択状態:
ENABLE: 発振
DISABLE: 停止

4.2.3.25 CG_SetExitStopModeFs

STOP モード解除後の自動低速発振選択

関数のプロトタイプ宣言:
void
CG_SetExitStopModeFs (FunctionalState ***NewState***)

引数:
NewState: STOP モード解除後の自動低速発振を選択します。
- **ENABLE:** 発振
- **DISABLE:** 停止

機能:
STOP モード解除後の自動低速発振を選択します。

戻り値:
なし

4.2.3.26 CG_GetExitStopModeFsState

STOP モード解除後の自動低速発振選択状態の取得

関数のプロトタイプ宣言:
FunctionalState
CG_GetExitStopModeFsState (void)

引数:
なし

機能:
STOP モード解除後の自動低速発振選択状態を取得します。

戻り値:
STOP モード解除後の自動低速発振選択状態
ENABLE: 発振
DISABLE: 停止

4.2.3.27 CG_SetPinStateInStopMode

STOP モード中の端子状態制御

関数のプロトタイプ宣言:
void
CG_SetPinStateInStopMode (FunctionalState ***NewState***)

引数:
NewState: 以下から STOP モード中の端子状態を選択します。
- **DISABLE:** 端子をドライブしない。
- **ENABLE:** 端子をドライブする。

機能:
STOP モード中の端子状態を選択します。

戻り値:
なし

4.2.3.28 CG_GetPinStateInStopMode

STOP モード中の端子状態の選択状態の取得

関数のプロトタイプ宣言:
FunctionalState
CG_GetPinStateInStopMode (void)

引数:
なし

機能:
STOP モード中の端子状態の選択状態を取得します。

戻り値:
STOP モード中の端子状態の選択状態:
DISABLE: 端子をドライブしない。
ENABLE: 端子をドライブする。

4.2.3.29 CG_SetFsysSrc

システムクロックの選択

関数のプロトタイプ宣言:
Result
CG_SetFsysSrc (CG_FsysSrc ***Source***)

引数:
Source: 以下からシステムクロック(fsys)を選択します。
- **CG_FSYS_SRC_FGEAR:** 高速
- **CG_FSYS_SRC_FS**: 低速

機能:
システムクロックを選択します。
CG_FSYS_SRC_FGEAR を選択する場合、事前に高速発振器(X1)を発振状態にしてください。**CG_FSYS_SRC_FS** を選択する場合、事前に低速発振器(TX1)を発振

状態にしてください。上記のようになっていない場合、本 API は **ERROR** を返却します。

戻り値:
SUCCESS: 成功
ERROR: 失敗

4.2.3.30 CG_GetFsysSrc

システムクロックの選択状態の取得

関数のプロトタイプ宣言:
CG_FsysSrc
CG_GetFsysSrc (void)

引数:
なし

機能:
システムクロックの選択状態を取得します。

戻り値:
システムクロックの選択状態:
CG_FSYS_SRC_FGEAR: 高速
CG_FSYS_SRC_FS: 低速

4.2.3.31 CG_SetSTBYReleaseINTSrc

スタンバイモードの解除割り込み要因の設定

関数のプロトタイプ宣言:
void
CG_SetSTBYReleaseINTSrc (CG_INTSrc ***INTSource***,
CG_INTActiveState ***ActiveState***,
FunctionalState ***NewState***)

引数:
INTSource: スタンバイモードの解除割り込み要因を選択します。
- **CG_INT_SRC_INTLVD** : INTLVD
- **CG_INT_SRC_0** : INT0
- **CG_INT_SRC_1** : INT1
- **CG_INT_SRC_2** : INT2
- **CG_INT_SRC_3** : INT3
- **CG_INT_SRC_RTC**: RTC interrupt.

ActiveState: 解除トリガのアクティブ状態を選択します。

割り込み要因	選択できるアクティブレベル	説明
CG_INT_SRC_RTC	**CG_INT_ACTIVE_STATE_FALLING**	↓エッジ
CG_INT_SRC_INTLVD	**CG_INT_ACTIVE_STATE_RISGING**	↑エッジ
上記以外	**CG_INT_ACTIVE_STATE_L**	"Low"レベル
	CG_INT_ACTIVE_STATE_H	"High"レベル

CG_INT_ACTIVE_STATE_FALLING	↓エッジ
CG_INT_ACTIVE_STATE_RISGING	↑エッジ
CG_INT_ACTIVE_STATE_BOTH_EDGES	両エッジ

NewState: 解除トリガの有効/無効を選択します。
- **ENABLE**: 許可
- **DISABLE:** 禁止

機能:
スタンバイモードの解除割り込み要因を設定します。

戻り値:
なし

4.2.3.32 CG_GetSTBYReleaseINTState

スタンバイモードの解除割り込み要因のアクティブ状態の取得

関数のプロトタイプ宣言:
CG_INT_ActiveState
CG_GetSTBYReleaseINTSrc(CG_INTSrc ***INTSource***)

引数:
INTSource: 解除割り込み要因を選択します。
CG_INT_SRC_INTLVD CG_INT_SRC_0, CG_INT_SRC_1, CG_INT_SRC_2, CG_INT_SRC_3, CG_INT_SRC_RTC

機能:
スタンバイモードの解除割り込み要因のアクティブ状態を取得します。

戻り値:
解除割り込みソースのアクティブ状態:
CG_INT_ACTIVE_STATE_FALLING: ↓エッジ
CG_INT_ACTIVE_STATE_RISING: ↑エッジ
CG_INT_ACTIVE_STATE_BOTH_EDGES: 両エッジ
CG_INT_ACTIVE_STATE_INVALID: 無効な値

4.2.3.33 CG_ClearINTReq

スタンバイ解除割り込み要求のクリア

関数のプロトタイプ宣言:
void
CG_ClearINTReq(CG_INTSrc ***INTSource***)

引数:
INTSource: 解除割り込み要因を選択します。
CG_INT_SRC_INTLVD CG_INT_SRC_0, CG_INT_SRC_1, CG_INT_SRC_2, CG_INT_SRC_3, CG_INT_SRC_RTC

機能:

スタンバイ解除割り込み要求をクリアします。

戻り値:
なし

4.2.3.34 CG_SelExtHighClk

外部高速発振器の切り替え

関数のプロトタイプ宣言:
void
CG_SelExtHighClk (CG_EHClkSrc ***Source***)

引数:
Source:

- CG_EHCLK_OSCSEL: CGOSCCR<OSCSEL>で選択された高速発振器
- CG_EHCLK_OSC2: 外部高速発振器

機能:
外部高速発振器の切り替えを行います。

戻り値:
なし

4.2.3.35 CG_GetResetFlag

リセットフラグの取得とクリア

関数のプロトタイプ宣言:
CG_ResetFlag
CG_GetResetFlag(void)

引数:
なし

機能:
リセットフラグの取得とクリアを行います。

戻り値:
リセットフラグ:
ResetPin (Bit 0)：RESET 端子によるリセット
WDTReset (Bit 2)：WDT によるリセット
DebugReset (Bit 4)：<SYSRESETREQ>によるリセット

4.2.4 データ構造

4.2.4.1 CG_ResetFlag

メンバ:
uint32_t
All CG リセット要因を指定します。

ビットフィールド:
uint32_t
ResetPin 1 : RESET 端子によるリセット
uint32_t
Reserved 1 : 未使用
uint32_t
WDTReset 1 : WDT によるリセット
uint32_t
Reserved2 1 : 未使用
uint32_t
DebugReset 1 : <SYSRESETREQ>によるリセット
uint32_t
Reserverd3 27 : 未使用

5. DSADC

5.1 概要

TMPM061FWFG は3 ユニットの24bitΔΣADコンバータ(DSADC)を内蔵しています。
DSADC の同期スタート機能における、マスタユニットとスレーブユニットの割り当ては以下の通りです。

マスタ/スレーブ割り当て

マスタ	スレーブ
ユニットA	ユニットB ユニットC

DSADC の基準電圧回路(BGR)は温度センサと共通に使用しており、使用するためには温度センサの制御レジスタ(TEPEN)の設定も必要です。

特徴

DSADC には、以下のような特徴があります。
·変換スタート
ソフトウエアによる変換スタート
·変換モード
- シングル変換
- リピート変換
·ステータスフラグ
- 変換結果格納フラグ
- オーバーランフラグ
- 変換終了フラグ
- 変換中フラグ
·変換クロックを分周可能
fc/1、fc/2、fc/4、fc/8
·変換終了割り込みを出力
·変換開始補正機能
·ユニット間同時スタート機能

DSADC を使用する場合、下記のとおり端子処理を行ってください。
·VREFINx に基準電源の接続はしない
·AGNDREFx はDVSS に接続
·VREFINx とAGNDREFx の間に1μF のコンデンサを接続

DSADC を使用しない場合、下記のとおり端子処理を行ってください。
·VREFINx はDVDD3 に接続
·AGNDREFx はDVSS に接続

また、温度センサも使用しない場合、基準電圧回路に関し下記のとおり端子処理を行ってください。
·DSRVDD3、SRVDD はDVDD3 に接続
·DSRVSS はDVSS に接続

5.2 API 関数

5.2.1 関数一覧

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

5.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。:

1) AD 変換設定:
   DSADC_SetClk(), DSADC_ChangeMode(), DSADC_Init(), DSADC_SetClkSupply() , DSADC_SetAmplifier()
2) AD 変換の開始:
   DSADC_Start()
3) AD 変換ステータス/結果の読み出し:
   DSADC_GetConvertResult(), DSADC_GetStatus()
4) その他:
   DSADC_SWReset()

5.2.3 関数仕様

5.2.3.1 DSADC_SetClk

AD 変換クロックの選択

関数のプロトタイプ宣言:

void
DSADC_SetClk(TSB_DSAD_TypeDef *DSADCx,uint32_t Clk)

引数:

DSADCx: 以下から DSADC ユニットを選択します。

- **TSB_DSAD0:** ユニット 0
- **TSB_DSAD1:** ユニット 1
- **TSB_DSAD2:** ユニット 2

Clk: 以下から AD 変換クロックを選択します。

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

機能:

AD 変換クロックを選択します。

補足:
AD変換中に本APIをコールしないでください。
本APIをコールする前に**DSADC_GetStatus()**をコールしてDSAD変換状態が**BUSY**でないことを確認してください。

戻り値:
なし

5.2.3.2 DSADC_SWReset

ソフトウェアリセット

関数のプロトタイプ宣言:
void
DSADC_SWReset(TSB_DSAD_TypeDef *DSADCx)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSAD0:** ユニット 0
- **TSB_DSAD1:** ユニット 1
- **TSB_DSAD2:** ユニット 2

機能:
ソフトウェアリセットを行います。

補足:
DSADCLK<ADCLK>以外のレジスタを初期化します。.

戻り値:
なし

5.2.3.3 DSADC_Start

変換開始

関数のプロトタイプ宣言:
void
DSADC_Start(TSB_DSAD_TypeDef *DSADCx)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSAD0:** ユニット 0
- **TSB_DSAD1:** ユニット 1
- **TSB_DSAD2:** ユニット 2

機能:
変換を開始します。

補足:
本 API をコールする前に、以下のいずれかのモードを選択してください。
　シングル変換モード

連続変換モード
詳細は **DSADC_ChangeMode()**を参照してください。

補足:
AD 変換を開始する前に、DSADC の設定手順があります。詳細は、データシートの DSADC“起動および停止手順”を参照してください。

戻り値:
なし

5.2.3.4 DSADC_ChangeMode

同期モードと変換モードの選択

関数のプロトタイプ宣言:
void
DSADC_ChangeMode(TSB_DSAD_TypeDef *DSADCx,uint8_t SyncMode,uint8_t ConvMode)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSAD0:** ユニット 0
- **TSB_DSAD1:** ユニット 1
- **TSB_DSAD2:** ユニット 2

SyncMode: 以下から同期モードを選択します。
- **DSADC_A_SYNC_MODE**: 個別モード
- **DSADC_SYNC_MODE**: 同期モード

ConvMode: 以下から変換モードを選択します。
- **DSADC_SINGLE_MODE**: シングル変換
- **DSADC_REPEAT_MODE**: 連続変換

機能:
同期モードと変換モードを選択します。

戻り値:
なし

5.2.3.5 DSADC_SetAmplifier

ゲインアンプの設定

関数のプロトタイプ宣言:
void
DSADC_SetAmplifier(TSB_DSAD_TypeDef *DSADCx,uint32_t Amplifier)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSAD0:** ユニット 0
- **TSB_DSAD1:** ユニット 1
- **TSB_DSAD2:** ユニット 2

Gain: 以下からゲインアンプを選択します。
- **DSADC_GAIN_1x:** x1
- **DSADC_GAIN_2x:** x2
- **DSADC_GAIN_4x:** x4
- **DSADC_GAIN_8x:** x8
- **DSADC_GAIN_16x:** x16

機能:
ゲインアンプを選択します。

戻り値:
なし

5.2.3.6 DSADC_GetConvertResult

変換結果の取得

関数のプロトタイプ宣言:
uint32_t
DSADC_GetConvertResult(TSB_DSAD_TypeDef *DSADCx)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSAD0:** ユニット 0
- **TSB_DSAD1:** ユニット 1
- **TSB_DSAD2:** ユニット 2

機能:
変換結果を取得します。

戻り値:
変換結果

5.2.3.7 DSADC_Init

DSADC ユニットのコンフィグレーション

関数のプロトタイプ宣言:
void
DSADC_Init(TSB_DSAD_TypeDef *DSADCx,DSADC_InitTypeDef * InitStruct)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSAD0:** ユニット 0
- **TSB_DSAD1:** ユニット 1
- **TSB_DSAD2:** ユニット 2

InitStruct: DSAD 変換のコンフィグレーションを行う構造体を指定します。
- **InitStruct->Clk:** 変換クロックの選択
- **InitStruct->BiasEn:** バイアス制御
- **InitStruct->ModulatorEn:** モジュレータ制御
- **InitStruct->SyncMode:** 同期モードの設定

- **InitStruct->Repeatmode:** 変換モードの設定
- **InitStruct->Amplifier:** ゲインアンプの選択
- **InitStruct->Offset:** 変換開始補正時間の設定
- **InitStruct->CorrectEn:** 変換開始補正の設定

機能:
DSAD 変換のコンフィグレーションを行います。

戻り値:
なし

5.2.3.8 DSADC_GetStatus

DSAD 変換フラグの取得

関数のプロトタイプ宣言:
DSAD_status
DSADC_GetStatus (TSB_DSAD_TypeDef *DSADCx)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSAD0:** ユニット 0
- **TSB_DSAD1:** ユニット 1
- **TSB_DSAD2:** ユニット 2

機能:
DSAD 変換フラグを取得します。

戻り値:
DSAD 変換フラグ:
F_ResultStore (Bit 0): 変換結果格納フラグ(1: 格納された、0:格納されていない)
F_Overrun (Bit 1): オーバーランフラグ(1: 発生した、0:発生していない)
F_Convert (Bit 2): 変換中フラグ(1: 変換中、0:変換していない)
F_ConvertEnd (Bit 3): 変換終了フラグ(1: 変換終了、0:変換終了していない)
ConversionResult(Bit 8~31): 変換結果

5.2.3.9 DSADC_SetClkSupply

DSADC クロックの選択

関数のプロトタイプ宣言:
void
DSADC_SetClkSupply(TSB_DSAD_TypeDef *DSADCx,FunctionalState NewState)

引数:
InputChannel: アナログ入力チャネルを選択してください。
- **DSADC_AN_02:** チャネル 2
- **DSADC_AN_03:** チャネル 3
- **DSADC_AN_04:** チャネル 4
- **DSADC_AN_05:** チャネル 5

NewState: DSADC クロックを選択します。

- **ENABLE :** 許可
- **DISABLE:** 禁止

機能:
DSADC クロックを選択します。

戻り値:
なし

5.2.4 データ構造

5.2.4.1 DSADC_InitTypeDef

構造体のメンバ:

ビットフィールド:

uint8_t ***Clk*** (Bit 0)	変換クロック
uint8_t ***BiasEn*** (Bit 1)	バイアス制御 1: 動作、0: 停止
uint8_t ***ModulatorEn*** (Bit 2)	モジュレータ制御 1: 動作、0: 停止
uint8_t ***SyncMode*** (Bit 3)	同期モード 1: 同期動作、0: 個別動作
uint8_t ***Repeatmode*** (Bit 4)	変換モード (1: 連続変換、0: シングル変換)
uint8_t ***Amplifier*** (Bit 5~Bit 6)	ゲインアンプ
Uint16_t ***Offset*** (Bit 7)	変換開始補正時間
Uint8_t ***CorrectEn*** (Bit 8)	変換開始補正

5.2.4.2 DSAD_status

共用体のメンバ:

uint32_t
All DSAD 変換結果

ビットフィールド:

uint32_t ***F_ResultStore*** (Bit 0)	変換結果格納フラグ(1: 格納された、0: 格納されていない)
uint32_t ***F_Overrun*** (Bit 1)	オーバーランフラグ(1: オーバーラン発生、0: 発生していない)
uint32_t ***F_Convert*** (Bit 2)	変換中フラグ(1: 変換中、0: 変換していない)
uint32_t ***F_ConvertEnd*** (Bit 3)	変換終了フラグ(1: 変換終了、0: 変換終了していない)
uint32_t ***Reserved*** (Bit4~Bit7)	未使用
uint32_t ***ConversionResult*** (Bit8~Bit31)	変換結果

6. FC

6.1 概要

本デバイスは、フラッシュメモリを内蔵しています。フラッシュメモリのサイズは、128Kbyte です。

オンボードプログラミングにおいて、CPU はソフトウェアを実行し、flash メモリへのデータ書き込み / 削除 を行います。データ書き込み / 削除は JEDEC 標準型コマンドに従って行います。また、Flash メモリをモニターするレジスタを提供し、各ブロックのプロテクション状態の表示、セキュリティ機能の設定を行います。

ブロック構成は、デバイスのデータシートを参照してください。

全ドライバ API は、アプリで使用する API 定義、マクロ、データタイプ、構造を格納する以下のファイルで構成されています。
\Libraries\TX00_Periph_Driver\src\tmpm061_fc.c
\Libraries\TX00_Periph_Driver \inc\tmpm061_fc.h

6.2 API 関数

6.2.1 関数一覧

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

6.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。

1) セキュリティ設定(Flash ROM データの読み出し、デバッグ):
   FC_SetSecurityBit(), FC_GetSecurityBit()
2) 自動動作状態およびプロテクト状態の取得:
   FC_GetBusyState(), FC_GetBlockProtectState()
3) プロテクトの設定とプロテクト状態の取得:
   FC_ProgramBlockProtectState(), FC_EraseBlockProtectState()
4) 自動実行コマンド(書き込み、チップ消去、ブロック消去):
   FC_WritePage(), FC_EraseBlock(), FC_EraseChip()

6.2.3 関数仕様

6.2.3.1 FC_SetSecurityBit

セキュリティビットの設定

関数のプロトタイプ宣言:
void
FC_SetSecurityBit (FunctionalState ***NewState***)

引数:
NewState: セキュリティビットを設定します。
- **DISABLE**: セキュリティ機能設定不可
- **ENABLE**: セキュリティビット設定可能

機能:
1) 書き込み/消去プロテクト用のすべてのプロテクトビット (PSRA<BLKn>)を”1” にします。
2) FCSECBIT<SECBIT>を”1”にします。

上記の 2 つの条件が成立すると、セキュリティ機能が有効になります。セキュリティ機能が有効な状態の制限内容は次の通りです。
- ROM 領域のデータの読み出し。
- JTAG/SW、トレースの通信

したがって、この API を使用する場合は、注意して実行してください。

FCSECBIT<SECBIT>はパワーオンリセットで初期化されます。

戻り値:
なし

6.2.3.2 FC_GetSecurityBit

セキュリティビットの設定状態の取得

関数のプロトタイプ宣言:
FunctionalState
FC_GetSecurityBit(void)

引数:
なし

機能:
セキュリティビットの設定状態を取得します。

戻り値:
セキュリティビットの設定状態:
DISABLE: セキュリティ機能設定不可
ENABLE: セキュリティビット設定可能

6.2.3.3 FC_GetBusyState

自動動作状態の取得

関数のプロトタイプ宣言:
WorkState
FC_GetBusyState (void)

引数:
なし

機能:
自動動作状態を取得します。

戻り値:
自動動作状態:
BUSY: 自動動作中
DONE: 自動動作終了

6.2.3.4 FC_GetBlockProtectState

ブロックのプロテクト状態の取得

関数のプロトタイプ宣言:
FunctionalState
FC_GetBlockProtectState(uint8_t ***BlockNum***)

引数:
BlockNum: ブロック番号を選択します。
- **FC_BLOCK_0 ~ FC_BLOCK_3**

機能:
各ブロックのプロテクト状態を示します。プロテクト状態の時には、書き込み、消去ができません。

戻り値:
ブロックプロテクトの状態:
DISABLE: プロテクト状態ではない。
ENABLE: プロテクト状態

6.2.3.5 FC_ProgramBlockProtectState

ブロックのプロテクト設定

関数のプロトタイプ宣言:
FC_Result
FC_ProgramProtectState(uint8_t ***BlockNum***)

引数:
BlockNum: ブロック番号を選択します。
- **FC_BLOCK_0 ~ FC_BLOCK_3**

機能:
ブロックプロテクトを設定します。プロテクト状態の時には、書き込み、消去ができません。

戻り値:
プロテクト設定結果:
FC_SUCCESS: プロテクト設定の成功
FC_ERROR_PROTECTED: プロテクト設定の失敗(すでにプロテクト済の場合は再度プロテクト設定を行いません)

FC_ERROR_OVER_TIME: プロテクト設定の失敗(自動動作のタイムアウト)

6.2.3.6 FC_EraseBlockProtectState

プロテクトの解除

関数のプロトタイプ宣言:
FC_Result
FC_EraseBlockProtectState(uint8_t ***BlockGroup***)

引数:
BlockGroup: ブロックグループを指定してください。
- **FC_BLOCK_GROUP_0**: ブロック 0 ~ 3

機能:
プロテクトビットを"0"にすることでプロテクトを解除します。

戻り値:
プロテクト解除結果:
FC_SUCCESS: プロテクト解除の成功
FC_ERROR_OVER_TIME: プロテクト解除の失敗(自動動作のタイムアウト)

6.2.3.7 FC_WritePage

ページ単位の書き込み

関数のプロトタイプ宣言:
FC_Result
FC_WritePage(uint32_t ***PageAddr***, uint32_t * ***Data***)

引数:
PageAddr: ページの開始アドレスを指定します。
Data: 書き込むデータバッファへのポインタを指定します。サイズは FC_PAGE_SIZE(128Byte)です。

機能:
ページ書き込みを行います。
自動ページ書き込みは、既に消去された 1 ページにつき一回のみ実施されます。データ値が“1” または “0” のいずれかであっても、2 回以上書き込みを実施しないでください。

補足: あらかじめデータを消去せずに書き込みを行うと、デバイスに損傷を与える恐れがあります。

戻り値:
ページ書き込み結果:
FC_SUCCESS: 書き込み成功
FC_ERROR_PROTECTED: 書き込み失敗(ブロックにプロテクトが設定されている)
FC_ERROR_OVER_TIME: 書き込みの失敗(自動動作のタイムアウト)

6.2.3.8 FC_EraseBlock

ブロック単位の消去

関数のプロトタイプ宣言:
FC_Result
FC_EraseBlock(uint32_t ***BlockAddr***)

引数:
BlockAddr: ブロック開始アドレスを指定してください。

機能:
ブロック単位の消去を行います。プロテクトされていないブロックに対してのみ消去を行います。

戻り値:
ブロック消去結果:
FC_SUCCESS: 消去成功
FC_ERROR_PROTECTED: 消去失敗(ブロックにプロテクトが設定されている)
FC_ERROR_OVER_TIME: 消去の失敗(自動動作のタイムアウト)

6.2.3.9 FC_EraseChip

チップ消去

関数のプロトタイプ宣言:
FC_Result
FC_EraseChip(void)

引数:
なし

機能:
チップ消去を行います。ブロックの一部にプロテクトが設定されている場合、そのブロックのデータは消去されません。

戻り値:
チップ消去結果:
FC_SUCCESS: チップ消去成功。ただしブロックの一部にプロテクトが設定されている場合、そのブロックのデータは消去されません。
FC_ERROR_PROTECTED: 消去失敗(すべてのブロックにプロテクトが設定されている)
FC_ERROR_OVER_TIME: 消去の失敗(自動動作のタイムアウト)

7. LCD

7.1 概要

TMPM061FWFG は、液晶表示器(LCD)を直接駆動するドライブおよびその制御回路を内蔵しています。

LCD との接続端子は、次のとおりです。

1. セグメント出力端子:40 本(SEG39 ~ SEG0)
2. コモン出力端子:4 本(COM3 ~ COM0)

ほかに駆動用電源端子として VLC 端子、外部ブリーダ抵抗接続端子として LV1, LV2 端子があります。

注) スタティック、1/3、1/2 デューティで使用する場合、未使用のコモン出力端子はオープンにしてください。(バイアス電圧が出力されます)。

直接駆動が可能な LCD は、次の 5 種類です。

1. 1/4 デューティ(1/3 バイアス) LCD 最大 160 画素 (8 セグメント× 20 桁)
2. 1/3 デューティ(1/3 バイアス) LCD 最大 120 画素 (8 セグメント× 15 桁)
3. 1/3 デューティ(1/2 バイアス) LCD 最大 120 画素 (8 セグメント× 15 桁)
4. 1/2 デューティ(1/2 バイアス) LCD 最大 80 画素 (8 セグメント× 10 桁)
5. スタティック LCD 最大 40 画素(8 セグメント× 5 桁)

LCD ドライバ API は、本デバイスの LCD モジュール用関数セットで、LCD バイアス設定、デューティ設定、表示バッファへのデータ書き込み、LCD 表示に関連する設定などを提供します。

全ドライバ API は、アプリで使用する API 定義、マクロ、データタイプ、構造を格納する以下のファイルで構成されています。

Libraries\TX00_Periph_Driver\src\tmpm061_lcd.c
Libraries\TX00_Periph_Driver\inc\tmpm061_lcd.h

7.2 API 関数

7.2.1 関数一覧

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

7.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています。:

1) LCD 設定:
   LCD_SetDisplay(), LCD_SetDutyBias(), LCD_SetBaseFreq(),
   LCD_SetLowBleederTime(), LCD_SetInternalBleeder(), LCD_SetBleederSource()
2) LCD 回路制御:
   LCD_Enable(), LCD_Disable()
3) LCD バッファへのデータ書き込み:
   LCD_WriteBuf()

7.2.3 関数仕様

7.2.3.1 LCD_Enable

LCD ドライバ動作の有効化

関数のプロトタイプ宣言:
void
LCD_Enable(void)

引数:
なし

機能:
LCD ドライバ動作を有効にします。

戻り値:
なし

7.2.3.2 LCD_Disable

LCD ドライバ動作の無効化

関数のプロトタイプ宣言:
void
LCD_Disable(void)

引数:
なし

機能:
LCD ドライバ動作を無効にします。

戻り値:
なし

7.2.3.3 LCD_SetDisplay

LCD 表示制御

関数のプロトタイプ宣言:
void
LCD_SetDisplay(FunctionalState ***NewState***)

引数:
NewState: 以下から LCD 表示制御を行います。
- **ENABLE**: 表示 Blanking
- **DISABLE**: 表示 Enable(Blanking 解除)

機能:
LCD 表示制御を行います。

戻り値:
なし

7.2.3.4 LCD_SetDutyBias

LCD 駆動方式の設定

関数のプロトタイプ宣言:
void
LCD_SetDutyBias(LCD_DutyBias ***DutyBias***)

引数:
DutyBias: 以下から LCD 駆動方式を設定します。
- **LCD_DUTY4_BIAS3 :** 1/4 デューティ, 1/3 バイアス
- **LCD_DUTY3_BIAS3 :** 1/3 デューティ, 1/3 バイアス
- **LCD_DUTY3_BIAS2 :** 1/3 デューティ, 1/2 バイアス
- **LCD_DUTY2_BIAS2 :** 1/2 デューティ, 1/2 バイアス
- **LCD_STATIC :** スタティック

機能:
LCD 駆動用デューティとバイアスを設定します。

戻り値:
なし

7.2.3.5 LCD_SetBaseFreq

ベース周波数の選択

関数のプロトタイプ宣言:
void
LCD_SetBaseFreq(LCD_BaseFreq ***Freq***)

引数:
Freq: 以下からベース周波数を選択します。
- **LCD_FSYS_DIVIDE_2_POWER_18 :** F_base = fsys / (2^18)
- **LCD_FSYS_DIVIDE_2_POWER_17 :** F_base = fsys / (2^17)
- **LCD_FSYS_DIVIDE_2_POWER_16 :** F_base = fsys / (2^16)
- **LCD_FSYS_DIVIDE_2_POWER_15 :** F_base = fsys / (2^15)
- **LCD_FSYS_DIVIDE_2_POWER_14 :** F_base = fsys / (2^14)
- **LCD_FS_DIVIDE_2_POWER_9 :** F_base = fs / (2^9)
- **LCD_FS_DIVIDE_2_POWER_8 :** F_base = fs / (2^8)

機能:

ベース周波数を選択します。このベース周波数を選択すると、フレーム周波数が決まります。

ベース周波数とフレーム周波数の関係:

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

戻り値:
なし

7.2.3.6 LCD_SetLowBleederTime

内部ブリーダ低抵抗の接続時間の選択

関数のプロトタイプ宣言:
void
LCD_SetLowBleederTime(LCD_LowResistorConnectionTime ***Time***)

引数:
Time: 以下から内部ブリーダ低抵抗の接続時間を選択します。

- **LCD_NOT_CONNECT**: 接続なし
- **LCD_2_POWER_7_DEVIDE_BASE_FREQ** : (2^7)/F_base
- **LCD_2_POWER_6_DEVIDE_BASE_FREQ** : (2^6)/F_base
- **LCD_2_POWER_5_DEVIDE_BASE_FREQ** : (2^5)/F_base
- **LCD_2_POWER_4_DEVIDE_BASE_FREQ** : (2^4)/F_base
- **LCD_2_POWER_3_DEVIDE_BASE_FREQ** : (2^3)/F_base
- **LCD_ALWAYS_CONNECT** : 常時接続

機能:
内部ブリーダ低抵抗の接続時間を選択します。

内部ブリーダ抵抗は、高抵抗および低抵抗の 2 系統によって構成されます。高抵抗と低抵抗はバイアス毎に並列接続されており、このうち低抵抗はアナログスイッチが併設されていますので LCDCR2<LRSE>によってブリーダ低抵抗の接続時間を調整することができます。

アナログスイッチが ON の期間は高抵抗に対し低抵抗が並列接続され見かけ上の抵抗値が低くなることで LCD ドライバの駆動能力を上げることができます。基本的に低抵抗の接続時間を長くすると LCD パネルの駆動能力は高くなりますが、その分消費電力が大きくなります。

逆に接続時間を短くすると駆動能力は低くなりますが、消費電力は、少なくなります。

駆動能力が不足すると LCD 表示が滲むなどの影響が現れますので、使用する LCD パネルに合わせて最適な設定値に調整してください。

戻り値:
なし

7.2.3.7 LCD_SetInternalBleeder

内部ブリーダ高抵抗の選択

関数のプロトタイプ宣言:
void
LCD_SetInternalBleeder(LCD_BleederResistorValue ***Value***)

引数:
Value: 以下から内部ブリーダ高抵抗を選択します。
- **LCD_BLEEDER_RESISTOR_200K** : 200KΩ(Typ.)
- **LCD_BLEEDER_RESISTOR_500K** : 500KΩ(Typ.)

機能:
内部ブリーダ高抵抗を選択します。

戻り値:
なし

7.2.3.8 LCD_SetBleederSource

ブリーダ抵抗の内部/外部切り替え

関数のプロトタイプ宣言:
void
LCD_SetBleederSource(LCD_BleederResistorSource ***Source***)

引数:
Source: ブリーダ抵抗の内部/外部を切り替えます。
- **LCD_BLEEDER_RESISTOR_EXTERNAL:** 外部
- **LCD_BLEEDER_RESISTOR_INTERNAL :** 内部

機能:

ブリーダ抵抗の内部/外部を切り替えます。

戻り値:
なし

7.2.3.9 LCD_WriteBuf

LCD バッファへのデータ書き込み

関数のプロトタイプ宣言:
void
LCD_WriteBuf(LCD_BufIndex ***TargetBuf***, uint8_t ***Data***)

引数:
TargetBuf: LCD バッファへのインデックスを設定します。LCD_BufIndex については後述のデータ構造を参照してください。
1 セグメントの場合:
- **LCD_BUF_SEG00, LCD_BUF_SEG01, LCD_BUF_SEG02**
- **.....**
- **LCD_BUF_SEG37, LCD_BUF_SEG38, LCD_BUF_SEG39**

2 セグメントの場合:
- **LCD_BUF_SEG0100, LCD_BUF_SEG0302, LCD_BUF_SEG0504**
- **.....**
- **LCD_BUF_SEG3534, LCD_BUF_SEG3736, LCD_BUF_SEG3938**

Data: LCD バッファへ書き込む値を指定します。

機能:
LCD バッファへのデータを書き込みます。
例えば、
LCD_WriteBuf(**LCD_BUF_SEG35, 0x04**)をコールすると、セグメント 35 のデータバッファに 0x04 をライトします。
LCD_WriteBuf(**LCD_BUF_SEG3534, 0x56**)をコールすると、セグメント 35 のデータバッファに 0x05 を書き込み、またセグメント 34 のデータバッファに 0x06 を書き込みます。

戻り値:
なし

7.2.4 データ構造

なし

8. LVD

8.1 概要

本製品は、電圧検出回路 (LVD)を内蔵しています。電圧検出回路は、DVDD3 を供給するために電圧の低下/上昇を検出することにより、NMI 信号または INTLVD を発生させます。

LVD ドライバの API では、LVD 機能の有効/無効、検出電圧の設定、電圧検出状態の取得などの機能セットが提供されています。

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX00_Periph_Driver/src/tmpm061_lvd.c
/Libraries/TX00_Periph_Driver/inc/tmpm061_lvd.h

8.2 API 関数

8.2.1 関数一覧

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

8.2.2 関数の種類

関数は、主に以下の 2 種類に分かれています。:

1) LVD 機能の設定:
   LVD_Enable(), LVD_Disable(), LVD_SetVoltage(), LVD_SetINTOutput(), LVD_SetINTCondition()
2) 電圧検出状態の確認:
   LVD_GetStatus()

8.2.3 関数仕様

8.2.3.1 LVD_Enable

電源検出動作の許可

関数のプロトタイプ宣言:
void
LVD_Enable(void)

引数:
なし

機能:
電源検出動作を許可します。

戻り値:
なし

8.2.3.2 LVD_Disable

電源検出動作の禁止

関数のプロトタイプ宣言:
void
LVD_Disable(void)

引数:
なし

機能:
電源検出動作を禁止します。

戻り値:
なし

8.2.3.3 LVD_SetVoltage

検出電圧の設定

関数のプロトタイプ宣言:
void
LVD_SetVoltage(uint32_t ***Voltage***)

引数:
Voltage: 以下から検出電圧を選択します。

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

機能:
検出電圧を選択します。

戻り値:
なし

8.2.3.4 LVD_GetStatus

LVDLVL2 電圧検出ステータスの取得

関数のプロトタイプ宣言:
LVD_SupplyStatus
LVD_GetStatus(void)

引数:
なし

機能:
LVDLVL2 電圧検出ステータスを取得します。

戻り値:
LVDLVL2 電圧検出ステータス:
LVD_SUPPLY_HIGH: 電源電圧は LVD_SetVoltage()により設定された検出電圧以上
LVD_SUPPLY_LOW: 電源電圧は LVD_SetVoltage()により設定された検出電圧以下

8.2.3.5 LVD_SetINTOutput

INTLVD 信号の出力

関数のプロトタイプ宣言:
void
LVD_SetINTOutput(FunctionalState ***NewState***)

引数:
NewState: 以下から INTLVD 信号の出力を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
INTLVD 信号の出力を選択します。

戻り値:
なし

8.2.3.6 LVD_SetINTCondition

INT 発生条件の設定

関数のプロトタイプ宣言:
void
LVD_SetINTCondition(uint32_t ***Condition***)

引数:
Condition: 以下から INT 発生条件を選択します。
- **LVD_INTSEL_LOWER :** 電源電圧低下時に設定電圧よりも下がった場合のみ
- **LVD_INTSEL_LOWER_UPPER**: 電源電圧低下時に設定電圧よりも下がった場合および上昇時に設定電圧よりも上がった場合

機能:

INT 発生条件を選択します。

戻り値:
なし

8.2.4 データ構造

なし

9. RTC

9.1 概要

RTC の機能概略は以下です。

- 時計機能(時間, 分、秒)
- カレンダー機能(日月, 週, うるう年)
- 24 時間計と 12 時間計 (am/ pm)のいずれかを選択可能
- +/- 30 秒補正機能 (ソフトウェアによる補正)
- アラーム機能 (アラーム出力)
- アラーム割り込み発生
- クロック補正機能
- 1MHz クロック出力機能

本 RTC ドライバは、年、うるう年、月、日、曜日、時間、分、秒、時間モードなどを格納する RTC クロック、アラームの設定を行う関数セットです。

本ドライバ は、アプリで使用する API 定義を格納する以下のファイルで構成されています。
/Libraries/TX00_Periph_Driver/src/tmpm061_rtc.c
/Libraries/ TX00_Periph_Driver/inc/tmpm061_rtc.h

9.2 API 関数

9.2.1 関数一覧

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

- void RTC_DisableAlarm(void)
- void RTC_SetRTCINT(FunctionalState ***NewState***)
- void RTC_SetAlarmOutput(uint8_t ***Output***)
- void RTC_ResetClockSec(void)
- RTC_ReqState RTC_GetResetClockSecReq(void)
- void RTC_ResetAlarm(void)
- void RTC_SetDateValue(RTC_DateTypeDef * ***DateStruct***)
- void RTC_GetDateValue(RTC_DateTypeDef * ***DateStruct***)
- void RTC_SetTimeValue(RTC_TimeTypeDef * ***TimeStruct***)
- void RTC_GetTimeValue(RTC_TimeTypeDef * ***TimeStruct***)
- void RTC_SetAlarmValue(RTC_AlarmTypeDef * ***AlarmStruct***)
- void RTC_GetAlarmValue(RTC_AlarmTypeDef * ***AlarmStruct***)
- void RTC_SetCorrectionProtect(FunctionalState ***WriteEnableState***)
- void RTC_EnableErrCorrection(void)
- void RTC_DisableErrCorrection(void)
- void RTC_ConfigErrCorrection(RTC_AdjustInterval ***Interval***, int8_t ***Value***)

9.2.2 関数の種類

関数は、主に以下の 5 種類に分かれています。:

1) RTC 機能の年月日の設定:
   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) RTC 機能の時間の設定:
   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(clock)の設定:
   RTC_EnableClock(), RTC_DisableClock(), RTC_SetTimeAdjustReq(), RTC_GetTimeAdjustReq(), RTC_ResetClockSec(), RTC_GetResetClockSec()
4) RTC(alarm)の設定:
   RTC_EnableAlarm(), RTC_DisableAlarm(), RTC_ResetAlarm(), RTC_SetAlarmValue(), RTC_GetAlarmValue()
5) その他:
   RTC_SetAlarmOutput(), RTC_SetRTCINT(), RTC_UpdateData(), RTC_GetAccessStatus(), RTC_ClearInitReq(), RTC_GetInitStatus(), RTC_EnableErrCorrection(), RTC_DisableErrCorrection(), RTC_ConfigErrCorrection(), RTC_SetCorrectionProtect()

9.2.3 関数仕様

9.2.3.1 RTC_SetSec

時計の秒桁設定

関数のプロトタイプ宣言:
void
RTC_SetSec(uint8_t ***Sec***);

引数:
Sec:最大 59 までの秒桁設定の値。

機能:
時計の秒桁値を設定します。RTC レジスタは、INTRTC のタイミングに同期して書換えられます。この関数の呼び出し後、RTC1Hz 割り込みを待つ必要があります。

戻り値:
なし

9.2.3.2 RTC_GetSec

時計の秒桁設定

関数のプロトタイプ宣言:
uint8_t
RTC_GetSec(void);

引数:
なし。

機能:
時計の秒桁の値を返します。

戻り値:
時計の秒桁:
0 ~ 59

9.2.3.3 RTC_SetMin

時計/アラームの分桁設定

関数のプロトタイプ宣言:
void
RTC_SetMin(RTC_FuncMode ***NewMode***,
uint8_t ***Min***);

引数:
NewMode: RTC モードを選択します。
- **RTC_CLOCK_MODE**: 時計機能
- **RTC_ALARM_MODE**: アラーム機能

Min: 最大 59 までの分析を設定します。

機能:
NewMode が **RTC_CLOCK_MODE** の場合、時計の分析を設定します。
NewMode が **RTC_ALARM_MODE** の場合、アラームの分析を設定します。
RTC レジスタは、INTRTC のタイミングに同期して書き換えられます。この関数を呼び出した後に、1HZ 割り込みが発生するのを待つ必要があります。

戻り値:
なし

9.2.3.4 RTC_GetMin

時計/アラームの分析読み込み

関数のプロトタイプ宣言:
uint8_t
RTC_GetMin(RTC_FuncMode ***NewMode***);

引数:
NewMode: RTC モードを選択します。
- **RTC_CLOCK_MODE**: 時計機能
- **RTC_ALARM_MODE**: アラーム機能

機能:
NewMode が **RTC_CLOCK_MODE** の場合、時計の分析の値を返します。
NewMode が **RTC_ALARM_MODE** の場合、アラームの分析の値を返します。

戻り値:
分析:
0 ~ 59

9.2.3.5 RTC_GetAMPM

12 時間モードの AM/PM 読み込み

関数のプロトタイプ宣言:
uint8_t
RTC_GetAMPM(RTC_FuncMode ***NewMode***);

引数:
NewMode: RTC モードを選択します。
- **RTC_CLOCK_MODE**: 時計機能
- **RTC_ALARM_MODE**: アラーム機能

機能:
時計/アラームの AM/PM を返します。
NewMode が **RTC_CLOCK_MODE** の場合、時計の AM/PM を返します。
NewMode が **RTC_ALARM_MODE** の場合、アラームの AM/PM を返します。

戻り値:

時計モード:
RTC_AM_MODE: AM
RTC_PM_MODE: PM

9.2.3.6 RTC_SetHour24

24 時間モードの時計/アラーム時桁設定

関数のプロトタイプ宣言:
void
RTC_SetHour24(RTC_FuncMode ***NewMode***,
uint8_t ***Hour***);

引数:
NewMode: RTC モードを選択します。
- **RTC_CLOCK_MODE**: 時計機能
- **RTC_ALARM_MODE**: アラーム機能

Hour: 最大 23 までの時桁を設定します。

機能:
24 時間モードの時計/アラームの時桁を設定します。
NewMode が **RTC_CLOCK_MODE** の場合、時計機能の時桁を設定し、
NewMode が **RTC_ALARM_MODE** の場合、アラームの時桁を設定します。
RTC レジスタは、INTRTC のタイミングに同期して書換えられます。この関数の実行後、1Hz 割り込みが発生するのを待つ必要があります。

*12 時間モードから 24 時間モードに変更する場合、本関数 **RTC_SetHour24()** によって HOURR レジスタを再設定してください。

戻り値:
なし

9.2.3.7 RTC_SetHour12

12 時間モードの時計/アラーム時桁設定

関数のプロトタイプ宣言:
void
RTC_SetHour12(RTC_FuncMode ***NewMode***,
uint8_t ***Hour***,
uint8_t ***AmPm***);

引数:
NewMode: RTC モードを選択します。
- **RTC_CLOCK_MODE**: 時計機能
- **RTC_ALARM_MODE**: アラーム機能

Hour: 最大 11 までの時桁を設定します。

AmPm: 以下から時間モードを選択します。
- **RTC_AM_MODE**: 12H モードの AM モード
- **RTC_PM_MODE**: 12H モードの PM モード

機能:
12 時間モードの時計/アラームの時桁を設定します。
NewMode が **RTC_CLOCK_MODE** の場合、時計機能の時桁を設定し、
NewMode が **RTC_ALARM_MODE** の場合、アラーム機能の時桁を設定します。
RTC レジスタは、INTRTC のタイミングに同期して書換えられます。この関数の実行後、1Hz 割り込みが発生するのを待つ必要があります。

*24 時間モードから 12 時間モードに変更する場合、本関数 **RTC_SetHour12()** によって HOURR レジスタを再度設定してください。

戻り値:
なし

9.2.3.8 RTC_GetHour

時計/アラームの時桁読み込み

関数のプロトタイプ宣言:
uint8_t
RTC_GetHour(RTC_FuncMode ***NewMode***);

引数:
NewMode: RTC モードを選択します。
- **RTC_CLOCK_MODE**: 時計機能
- **RTC_ALARM_MODE**: アラーム機能

機能:
時計/アラームの時桁を返します。
NewMode が **RTC_CLOCK_MODE** の場合、時計機能の時桁の値を返し、
NewMode が **RTC_ALARM_MODE** の場合、アラーム機能の時桁の値を返します。

戻り値:
24 時間モードでの時桁:
0 ~ 23
12H 時間モードでの時桁:
0 ~ 11

9.2.3.9 RTC_SetDay

時計/アラームの曜日設定

関数のプロトタイプ宣言:
void
RTC_SetDay(RTC_FuncMode ***NewMode***,
uint8_t ***Day***);

引数:
NewMode: RTC モードを選択します。
- **RTC_CLOCK_MODE**: 時計機能
- **RTC_ALARM_MODE**: アラーム機能

Day:曜日を選択します。
- **RTC_SUN:** 日曜日
- **RTC_MON:** 月曜日
- **RTC_TUE:** 火曜日
- **RTC_WED:** 水曜日
- **RTC_THU:** 木曜日
- **RTC_FRI:** 金曜日
- **RTC_SAT:** 土曜日

機能:
時計/アラームの曜日を設定します。
NewMode が **RTC_CLOCK_MODE** の場合、時計機能の曜日を設定します。
NewMode が **RTC_ALARM_MODE** の場合、アラーム機能の曜日を設定します。
RTC レジスタは、INTRTC のタイミングに同期して書換えられます。この関数の実行後、1Hz 割り込みが発生するのを待つ必要があります。

戻り値:
なし

9.2.3.10 RTC_GetDay

時計/アラームの曜日の読み込み

関数のプロトタイプ宣言:
uint8_t
RTC_GetDay(RTC_FuncMode ***NewMode***);

引数:
NewMode: RTC モードを選択します。
- **RTC_CLOCK_MODE**: 時計機能
- **RTC_ALARM_MODE**: アラーム機能

機能:
時計/アラームの曜日を返します。
NewMode が **RTC_CLOCK_MODE** の場合、時計機能の曜日を返し、
NewMode が **RTC_ALARM_MODE** の場合、アラーム機能の曜日を返します。

戻り値:
曜日の値:
0 ~ 6

9.2.3.11 RTC_SetDate

時計/アラームの日桁設定

関数のプロトタイプ宣言:
void
RTC_SetDate(RTC_FuncMode ***NewMode***,
uint8_t ***Date***);

引数:

NewMode: RTC モードを選択します。
- **RTC_CLOCK_MODE**: 時計機能
- **RTC_ALARM_MODE**: アラーム機能

Date: 1 から 31 の日桁を設定します。

機能:
時計/アラームの日桁を設定します。
NewMode が **RTC_CLOCK_MODE** の場合は、時計機能の日桁を設定し、
NewMode が **RTC_ALARM_MODE** の場合は、アラーム機能の日桁を設定します。
RTC レジスタは、INTRTC のタイミングに同期して書換えられます。この関数を呼び出した後に、1Hz 割り込みが発生するのを待つ必要があります。

戻り値:
なし

9.2.3.12 RTC_GetDate

時計/アラームの日桁読み込み

関数のプロトタイプ宣言:
uint8_t
RTC_GetDate(RTC_FuncMode ***NewMode***);

引数:
NewMode: RTC モードを選択します。
- **RTC_CLOCK_MODE**: 時計機能
- **RTC_ALARM_MODE**: アラーム機能

機能:
時計/アラームの日桁を返します。
NewMode が **RTC_CLOCK_MODE** の場合、時計機能の日桁の値を返し、
NewMode が **RTC_ALARM_MODE** の場合、アラーム機能の日桁の値を返します。

戻り値:
日桁:
1 ~ 31

9.2.3.13 RTC_SetMonth

時計の月桁設定

関数のプロトタイプ宣言:
void
RTC_SetMonth(uint8_t ***Month***);

引数:
Month: 1 から 12 の月桁を設定します。

機能:
時計の月桁を設定します。

RTC レジスタは、INTRTC のタイミングに同期して書換えられます。この関数の実行後、1Hz 割り込みが発生するのを待つ必要があります。

戻り値:
なし

9.2.3.14 RTC_GetMonth

時計の月桁読み込み

関数のプロトタイプ宣言:
uint8_t
RTC_GetMonth(void);

引数:
なし。

機能:
時計の月桁の値を返します。

戻り値:
月桁:
1 ~ 12

9.2.3.15 RTC_SetYear

時計の年桁設定

関数のプロトタイプ宣言:
void
RTC_SetYear(uint8_t ***Year***);

引数:
Year: 最大 99 までの年の値

機能:
時計の年桁を設定します。
RTC レジスタは、INTRTC のタイミングに同期して書換えられます。この関数の実行後、1Hz 割り込みが発生するのを待つ必要があります。

戻り値:
なし

9.2.3.16 RTC_GetYear

時計の年桁の読み込み

関数のプロトタイプ宣言:
uint8_t
RTC_GetYear(void);

引数:
なし。

機能:
時計の年桁の値を返します。

戻り値:
年桁:
0 ~ 99

9.2.3.17 RTC_SetHourMode

24 時間時計/12 時間時計の選択

関数のプロトタイプ宣言:
void
RTC_SetHourMode(uint8_t ***HourMode***);

引数:
HourMode: 時間モードを選択します。
- **RTC_12_HOUR_MODE**: 12 時間時計
- **RTC_24_HOUR_MODE**.: 24 時間時計

機能:
24 時間時計/12 時間時計を選択します。
HourMode が **RTC_24_HOUR_MODE** の時、12 時間時計を選択し、
HourMode が **RTC_12_HOUR_MODE** の時、24 時間時計を選択します。

* 本関数を実行する前に **RTC_DisableClock()** を実行し、時計を停止してください。
(詳細は “RTC_DisableClock” を参照)

戻り値:
なし

9.2.3.18 RTC_GetHourMode

時計モードの読み込み

関数のプロトタイプ宣言:
uint8_t
RTC_GetHourMode(void);

引数:
なし。

機能:
時計モードを読み込みます。

戻り値:
時計モード
RTC_24_HOUR_MODE: 24 時間時計

RTC_12_HOUR_MODE: 12 時間時計

9.2.3.19 RTC_SetLeapYear

うるう年の設定

関数のプロトタイプ宣言:
void
RTC_SetLeapYear(uint8_t ***LeapYear***);

引数:
LeapYear: 以下からうるう年を選択します。
- **RTC_LEAP_YEAR_0**: 現在の年（今年）がうるう年
- **RTC_LEAP_YEAR_1**: 現在がうるう年から 1 年目
- **RTC_LEAP_YEAR_2**: 現在がうるう年から 2 年目
- **RTC_LEAP_YEAR_3**: 現在がうるう年から 3 年目

機能:
うるう年を設定します。
LeapYear が **RTC_LEAP_YEAR_0** の場合、現在の年（今年）がうるう年で、
LeapYear が **RTC_LEAP_YEAR_1** の場合、現在がうるう年から 1 年目で、
LeapYear が **RTC_LEAP_YEAR_2** の場合、現在がうるう年から 2 年目で、
LeapYear が **RTC_LEAP_YEAR_3** の場合、現在がうるう年から 3 年目になります。

戻り値:
なし

9.2.3.20 RTC_GetLeapYear

うるう年の読み込み

関数のプロトタイプ宣言:
uint8_t
RTC_GetLeapYear(void);

引数:
なし。

機能:
うるう年の状態を返します。

戻り値:
うるう年の状態を表す値

9.2.3.21 RTC_SetTimeAdjustReq

+/– 30 秒の補正

関数のプロトタイプ宣言:
void
RTC_SetTimeAdjustReq(void);

引数:
なし。

機能:
秒の補正をします。要求は秒カウンタのカウントアップ時にサンプリングされ、、秒が 0～29 秒の場合、秒桁のみ "0" になります。また、30～59 秒のときは分を桁上げして秒を"0"にします。

戻り値:
なし

9.2.3.22 RTC_GetTimeAdjustReq

ADJUST 要求状態の読み込み

関数のプロトタイプ宣言:
RTC_ReqState
RTC_GetTimeAdjustReq(void);

引数:
なし。

機能:
ADJUST 要求状態を読み込みます。**RTC_SetTimeAdjustReq()** の実行後に、この関数を実行し、繰り返して要求をしないようにします。

戻り値:
ADJUST 要求状態を読み込みます。
RTC_NO_REQ : ADJUST 要求なし
RTC_REQ: ADJUST 要求あり

9.2.3.23 RTC_EnableClock

時計機能の起動

関数のプロトタイプ宣言:
void
RTC_EnableClock(void);

引数:
なし。

機能:
時計機能を有効にします。

戻り値:
なし

9.2.3.24 RTC_DisableClock

時計機能の終了

関数のプロトタイプ宣言:
void
RTC_DisableClock(void);

引数:
なし。

機能:
時計機能を無効にします。

戻り値:
なし

9.2.3.25 RTC_EnableAlarm

アラーム機能の起動

関数のプロトタイプ宣言:
void
RTC_EnableAlarm(void);

引数:
なし。

機能:
アラーム機能を有効にします。

戻り値:
なし

9.2.3.26 RTC_DisableAlarm

アラーム機能の終了

関数のプロトタイプ宣言:
void
RTC_DisableAlarm(void);

引数:
なし。

機能:
アラーム機能を無効にします。

戻り値:
なし

9.2.3.27 RTC_SetRTCINT

INTRTC 割り込みの有効/無効設定

関数のプロトタイプ宣言:
void
RTC_SetRTCINT(FunctionalState ***NewState***);

引数:
NewState*:* 以下から *INT RTC* の有効/無効を選択します。
- ***ENABLE****:* INTRTC 割り込み有効
- ***DISABLE****:* INTRTC 割り込み無効

機能:
NewState が **ENABLE** の場合、RTCINT を有効にし、***NewState*** が **DISABLE** の場合、RTCINT を無効にします。

戻り値:
なし

9.2.3.28 RTC_SetAlarmOutput

ALARM 端子の出力設定

関数のプロトタイプ宣言:
void
RTC_SetAlarmOutput(uint8_t ***Output***);

引数:
Output: 以下から、アラーム端子の出力を選択します。
- **RTC_LOW_LEVEL**: “0” パルス
- **RTC_PULSE_1_HZ**: 1Hz 周期の “0” パルス
- **RTC_PULSE_16_HZ**: 16Hz 周期の “0” パルス
- **RTC_PULSE_2_HZ**: 2Hz 周期の “0” パルス
- **RTC_PULSE_4_HZ**: 4Hz 周期の “0” パルス
- **RTC_PULSE_8_HZ**: 8Hz 周期の “0” パルス

機能:
アラーム端子の出力を設定します。
Output が **RTC_LOW_LEVEL** の場合、時計に同期してアラーム端子の出力は “0” になり、***Output*** が **RTC_PULSE_ n*_HZ** の場合、アラーム端子の出力は n*Hz 周期の “0” パルスになります。(n* は次のいずれかの値:1,2,4,8,16)

戻り値:
なし

9.2.3.29 RTC_ResetClockSec

時計秒カウンタのリセット

関数のプロトタイプ宣言:

void
RTC_ResetClockSec(void);

引数:
なし。

機能:
時計秒カウンタをリセットします。

戻り値:
なし

9.2.3.30 RTC_GetResetClockSecReq

時計秒カウンタのリセット要求状態の読み込み

関数のプロトタイプ宣言:
RTC_ReqState
RTC_GetResetClockSecReq(void);

引数:
なし。

機能:
時計秒カウンタのリセット要求状態を読み込みます。リセット要求は、低速クロックを使用してサンプリングします。クロックが安定するために、**RTC_ResetClockSec()** の実行後に本関数を実行してください。

戻り値:
リセット要求状態
RTC_NO_REQ: リセット要求なし
RTC_REQ: リセット要求あり

9.2.3.31 RTC_ResetAlarm

アラームリセット

関数のプロトタイプ宣言:
void
RTC_ResetAlarm(void)

引数:
なし

機能:
アラームレジスタ(分, 時, 日, 週桁レジスタ)を初期化します。
初期化後は、00 分, 00 時, 01 日, 日曜日になります。

戻り値:
なし

9.2.3.32 RTC_SetDateValue

時計の日付設定

関数のプロトタイプ宣言:
void
RTC_SetDateValue(RTC_DateTypeDef * ***DateStruct***);

引数:
DateStruct: うるう年、年、月、曜日、日を格納する構造体 (詳細は「データ構造」を参照)

機能:
時計の日付(うるう年、年、月、曜日、日)を読み込みます。
RTC_SetLeapYear(), **RTC_SetYear()**, **RTC_SetMonth()**, **RTC_SetDate()**, **RTC_Setday()**を実行します。

戻り値:
なし

9.2.3.33 RTC_GetDateValue

時計の日付の読み込み

関数のプロトタイプ宣言:
void
RTC_GetDateValue(RTC_DateTypeDef * ***DateStruct***);

引数:
DateStruct: うるう年、年、月、曜日、日を格納する含む構造体。(詳細は「データ構造」を参照)

機能:
時計のうるう年、年、月、曜日、日を読み込みます。
RTC_GetLeapYear(), **RTC_GetYear()**, **RTC_GetMonth()**, **RTC_GetDate(), RTC_Getday()**を実行します。

戻り値:
なし

9.2.3.34 RTC_SetTimeValue

時計の時刻設定

関数のプロトタイプ宣言:
void
RTC_SetTimeValue(RTC_TimeTypeDef * ***TimeStruct***);

引数:
TimeStruct:時間モード、時間、12 時間モードの AM/PM モード、分、秒を格納する構造体。(詳細は「データ構造」を参照)

機能:
時間モード、時間、12 時間モードの AM/PM モード、分、秒を設定します。
RTC_SetHourMode(), **RTC_SetHour12()**, **RTC_SetHour24()**, **RTC_SetMin()**, **RTC_SetSec()** を実行します。

戻り値:
なし

9.2.3.35 RTC_GetTimeValue

時計の時刻の読み込み

関数のプロトタイプ宣言:
void
RTC_GetTimeValue(RTC_TimeTypeDef * ***TimeStruct***);

引数:
TimeStruct: 時間モード、時間、12 時間モードの AM/PM モード、分、秒を格納する構造体。(詳細は「データ構造」を参照)

機能:
時刻(時間モード、時間、12 時間モードの AM/PM モード、分、秒)を読み込みます。
RTC_GetHourMode(), **RTC_GetHour()**, **RTC_GetAMPM()**, **RTC_GetMin(),** **RTC_GetSec()** が実行されます。

戻り値:
なし

9.2.3.36 RTC_SetAlarmValue

アラームの日時設定

関数のプロトタイプ宣言:
void
RTC_SetAlarmValue(RTC_AlarmTypeDef * ***AlarmStruct***)

引数:

AlarmStruct: 日、曜日、時間、12 時間モードの AM/PM、秒を格納する構造体。(詳細は「データ構造」を参照)

機能:
アラームの日時(日、曜日、時間、12 時間モードの AM/PM モード、秒を含む) を設定します。**RTC_SetDate()**, **RTC_SetDay()**, **RTC_SetHour12()**, **RTC_SetHour24()** , **RTC_SetMin()** をコールします。

戻り値:
なし

9.2.3.37 RTC_GetAlarmValue

アラームの日時の取得

関数のプロトタイプ宣言:
void
RTC_GetAlarmValue(RTC_AlarmTypeDef * ***AlarmStruct***)

引数:

AlarmStruct: 日、曜日、時間、12 時間モードの AM/PM、秒を格納する構造体。
(詳細は「データ構造」を参照)

機能:
アラームの日時(日、曜日、時間、12 時間モードの AM/PM モード、秒を含む) を読み込みます。
RTC_GetDate(), **RTC_GetDay()**, **RTC_GetHour()** , **RTC_GetAMPM(),** **RTC_GetMin()** をコールします。

戻り値:
なし

9.2.3.38 RTC_SetCorrectionProtect

補正機能レジスタ書き込み制御

関数のプロトタイプ宣言:
void
RTC_SetCorrectionProtect (FunctionalState WriteEnableState)

引数:

WriteEnableState: 以下から書き込みの許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
補正機能レジスタ書き込み制御を行います。

補足:
初期状態は書き込み許可です。
禁止を選択すると、RTCADJCTL と RTCADJDAT レジスタへの書き込みができなくなります。

戻り値:
なし

9.2.3.39 RTC_EnableErrCorrection

補正機能制御の有効化

関数のプロトタイプ宣言:

void
RTC_EnableErrCorrection(void)

引数:

なし

機能:
補正機能制御を有効にします。
本 API をコールした場合、**RTC_UpdateData()** をコールし、その後 **RTC_GetAccessStatus()** から DONE を取得できるまでポーリングしてください。

戻り値:
なし

9.2.3.40 RTC_DisableErrCorrection

補正機能制御の無効化

関数のプロトタイプ宣言:
void
RTC_DisableErrCorrection(void)

引数:

なし

機能:
補正機能制御を無効にします。
本 API をコールした場合、**RTC_UpdateData()** をコールし、その後 **RTC_GetAccessStatus()** から DONE を取得できるまでポーリングしてください。

戻り値:
なし

9.2.3.41 RTC_ConfigErrCorrection

補正基準時間の設定

関数のプロトタイプ宣言:
void
RTC_ConfigErrCorrection(RTC_AdjustInterval ***Interval***,
int8_t ***Value***)

引数:

Interval:以下から補正期間を選択します。
- **RTC_ADJUST_20_SEC**: 20 秒毎に補正
- **RTC_ADJUST_30_SEC**: 30 秒毎に補正

Value: 補正時間を-128 ~ +127 から指定します。

機能:
補正基準時間を設定します。

補足:
Value は符号付 8 ビットの値です。***Value*** の最下位ビットは無効です。
例えば、***Value*** = +101 は ***Value*** = +100 と同じです。

Interval = RTC_ADJUST_20_SEC の場合、以下の計算式を使用してください。
Shifting amount = 24 * 3600 / 20 * (Value / 32768) ≈ 0.132 * Value (sec / day)

Interval = RTC_ADJUST_30_SEC の場合、以下の計算式を使用してください。
Shifting amount = 24 * 3600 / 30 * (Value / 32768) ≈ 0.088 * Value (sec / day)

本 API をコールした場合、**RTC_UpdateData()** をコールし、その後 **RTC_GetAccessStatus()** から DONE を取得できるまでポーリングしてください。

戻り値:
なし

9.2.4 データ構造

9.2.4.1 RTC_DateTypeDef

メンバ:
uint8_t
LeapYear **:** うるう年を設定します:

- **RTC_LEAP_YEAR_0:** 現在の年(今年)がうるう年
- **RTC_LEAP_YEAR_1:** 現在がうるう年から 1 年目
- **RTC_LEAP_YEAR_2:** 現在がうるう年から 2 年目
- **RTC_LEAP_YEAR_3:** 現在がうるう年から 3 年目

uint8_t
Year 年桁の値(0～99)。

uint8_t
Month 月桁の値(1～12)。

uint8_t
Date の日桁の値(1～31)。

uint8_t
Day 週桁の値を以下。

- **RTC_SUN:** 日曜日
- **RTC_MON:** 月曜日
- **RTC_TUE:** 火曜日
- **RTC_WED:**水曜日
- **RTC_THU:**木曜日
- **RTC_FRI:** 金曜日
- **RTC_SAT:** 土曜日

9.2.4.2 RTC_TimeTypeDef

メンバ:
uint8_t

HourMode 24 時間時計、12 時間時計のモード選択の値:

- **RTC_12_HOUR_MODE:** 12 時間モード
- **RTC_24_HOUR_MODE:** 24 時間モード

uint8_t
Hour 時間桁の値。(24 時間モード:0～23、12 時間モード:0～11)

uint8_t
AmPm 12 時間モード時の AM/PM の値:

- **RTC_AM_MODE:** AM モード
- **RTC_PM_MODE:** PM モード
- **RTC_AMPM_INVALID:** 24 時間モード

uint8_t
Min 0～59 までの分析の値。

uint8_t
Sec 0～59 までの秒桁の値。

9.2.4.3 RTC_AlarmTypeDef

メンバ:

uint8_t
Date アラーム機能有効時の日桁の値(1～31)。

uint8_t
Day アラーム機能有効時の週桁の値。

- **RTC_SUN:** 日曜日
- **RTC_MON:**月曜日
- **RTC_TUE:**火曜日
- **RTC_WED:**水曜日
- **RTC_THU:**木曜日
- **RTC_FRI:**金曜日
- **RTC_SAT:** 土曜日

uint8_t
Hour アラーム機能有効時の時間桁の値。

uint8_t
AmPm アラーム機能有効時の AM/PM 選択の値:

- **RTC_AM_MODE: A**M モード
- **RTC_PM_MODE:** PM モード
- **RTC_AMPM_INVALID:** 24 時間モード

uint8_t
Min アラーム機能有効時の分析の値(0～59)。

10. SBI

10.1 概要

本デバイスはシリアルバスインターフェースチャンネルを有し、各チャンネルはマルチマスタが可能ン I2C バスで動作可能です。
I2C バスモードでは、SCL および SDA を通して外部デバイスと接続されます。
SBI チャンネルによりデータをフリーデータフォーマットで転送できます。フリーデータフォーマットでは、マスタモード時は送信、スレーブモード時は受信になります。

SBI ドライバ API 関数は、SBI チャネルの自己アドレス、クロック分周、ACK クロック生成等の設定、I2C の開始・終了条件のデータ転送、データ受信・送信の制御、状態復帰、SBI チャンネルモードの表示などの機能の設定を行う関数セットです。

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX00_Periph_Driver/src/tmpm061_sbi.c
/Libraries/TX00_Periph_Driver/inc/tmpm061_sbi.h

10.2 API 関数

10.2.1 関数一覧

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

10.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。:

1) 共通機能の設定:
   SBI_Enable(), SBI_Disable(), SBI_SetI2CACK(), SBI_SetI2CBitNum(), SBI_InitI2C()
2) 転送制御:
   SBI_ClearI2CINTReq(), SBI_GenerateI2Cstart(),
   SBI_GenerateI2Cstop(),SBI_SetSendData(), SBI_GetReceiveData()

3) ステータス確認:
SBI_GetI2CState()
4) その他:
SBI_SWReset(), SBI_SetIdleMode(), SBI_EnableI2CfreeDataMode()

10.2.3 関数仕様

補足: 以下の関数の中で、引数 “TSB_SBI_TypeDef* ***SBIx***” は **TSB_SBI0** を指定してください。

10.2.3.1 SBI_Enable

シリアルバスインタフェース動作の許可

関数のプロトタイプ宣言:
void
SBI_Enable(TSB_SBI_TypeDef* ***SBIx***)

引数:
SBIx: SBI チャネルを指定します。

機能:
SBI 動作を有効にします。

戻り値:
なし

10.2.3.2 SBI_Disable

シリアルバスインタフェース動作の禁止

関数のプロトタイプ宣言:
void
SBI_Disable(TSB_SBI_TypeDef* ***SBIx***)

引数:
SBIx: SBI チャネルを指定します。

機能:
SBI 動作を無効にします。

戻り値:
なし

10.2.3.3 SBI_SetI2CACK

I2C バスモードにおける ACK 選択

関数のプロトタイプ宣言:
void
SBI_SetI2CACK(TSB_SBI_TypeDef* ***SBIx***,
FunctionalState ***NewState***)

引数:
SBIx: SBI チャネルを指定します。
NewState: ACK の発生有無を選択します。
- **ENABLE**: 発生する。
- **DISABLE**: 発生しない。

機能:
I2C 通信のアクノリッジメントクロック(ACK)のためのクロックを発生する/発生しないを選択します。***NewState*** を **ENABLE** にすると ACK クロックを発生し、**DISABLE** にすると ACK クロックを発生しません。

戻り値:
なし

10.2.3.4 SBI_InitI2C

I2C バスモードにおける通信の初期化

関数のプロトタイプ宣言:
void
SBI_InitI2C(TSB_SBI_TypeDef* ***SBIx***,
SBI_InitI2CTypeDef* ***InitI2CStruct***)

引数:
SBIx: SBI チャネルを指定します。

InitI2CStruct: SBI に関する構造体です。(詳細は"データ構造"を参照)

機能:
I2C バスアドレス、転送ビット数、出力クロックの周波数選択、ACK クロック生成、I2C 転送モードの初期化を行います。

戻り値:
なし

10.2.3.5 SBI_SetI2CBitNum

I2C バスモードにおける転送ビット数の選択

関数のプロトタイプ宣言:
void
SBI_SetI2CBitNum(TSB_SBI_TypeDef* ***SBIx***,
uint32_t ***I2CBitNum***)

引数:
SBIx: SBI チャネルを指定します。

I2CBitNum: 転送ビット数(1～8)を選択します。
- **SBI_I2C_DATA_LEN_8:** データ長 8
- **SBI_I2C_DATA_LEN_1:** データ長 1

- **SBI_I2C_DATA_LEN_2:** データ長 2
- **SBI_I2C_DATA_LEN_3:** データ長 3
- **SBI_I2C_DATA_LEN_4:** データ長 4
- **SBI_I2C_DATA_LEN_5:** データ長 5
- **SBI_I2C_DATA_LEN_6:** データ長 6
- **SBI_I2C_DATA_LEN_7:** データ長 7

機能:
転送ビット数を選択します。

戻り値:
なし

10.2.3.6 SBI_SWReset

ソフトウェアリセットの発生

関数のプロトタイプ宣言:
void
SBI_SWReset(TSB_SBI_TypeDef* ***SBIx***)

引数:
SBIx: SBI チャネルを指定します。

機能:
シリアルバスインターフェース回路を初期化するリセット信号を発生します。リセット後、すべての制御レジスタやステータスフラグはリセット後の値に初期化されます。

戻り値:
なし

10.2.3.7 SBI_ClearI2CINTReq

I2C バスモードにおける INTSBIx 割り込み要求解除

関数のプロトタイプ宣言:
void
SBI_ClearI2CINTReq(TSB_SBI_TypeDef* ***SBIx***)

引数:
SBIx: SBI チャネルを指定します。

機能:
SBI 割り込み要求を解除します。

戻り値:
なし

10.2.3.8 SBI_GenerateI2CStart

I2C バスモードにおけるスタート状態の発生

関数のプロトタイプ宣言:
void
SBI_GenerateI2CStart(TSB_SBI_TypeDef* ***SBIx***)

引数:
SBIx: SBI チャネルを指定します。

機能:
I2C バスモードをマスタにし、I2c バスにスタートコンディションを出力します。

戻り値:
なし

10.2.3.9 SBI_GenerateI2CStop

I2C バスモードにおけるストップ状態の発生

関数のプロトタイプ宣言:
void
SBI_GenerateI2CStop(TSB_SBI_TypeDef* ***SBIx***)

引数:
SBIx: SBI チャネルを指定します。

機能:
I2C バスモードをマスタにし、I2c バスにストップコンディションを出力します。

戻り値:
なし

10.2.3.10 SBI_GetI2CState

I2C バスモードにおける SBI チャネルの状態の読み込み

関数のプロトタイプ宣言:
SBI_I2CState
SBI_GetI2CState(TSB_SBI_TypeDef* ***SBIx***)

引数:
SBIx: SBI チャネルを指定します。

機能:
I2C バスモード中の SBI チャネルの状態を読み込みます。SBI 割り込みの ISR で本関数をコールし、SBI チャネルの状態によってプロセスを変更します。

戻り値:
I2C モードでの SBI チャネルの状態

10.2.3.11 SBI_SetIdleMode

IDLE モード時の動作の許可/禁止

関数のプロトタイプ宣言:
void
SBI_SetIdleMode(TSB_SBI_TypeDef* ***SBIx***,
FunctionalState ***NewState***)

引数:
SBIx: SBI チャネルを指定します。

NewState: システムが idle モードの時の動作を指定します。
- **ENABLE:** 許可。
- **DISABLE:** 禁止。

機能:
NewState が **ENABLE** の場合 IDLE モードに遷移しても SBI チャネルは動作します。**DISABLE** を選択すると IDLE モード時に禁止されます。

戻り値:
なし

10.2.3.12 SBI_SetSendData

データ送信

関数のプロトタイプ宣言:
void
SBI_SetSendData(TSB_SBI_TypeDef* ***SBIx***,
uint32_t ***Data***)

引数:
SBIx: SBI チャネルを指定します。

Data: 送信データ。(最大値は 0xFF です)

機能:
設定データを送信します。**SBI_GenerateI2Cstart()**の実行によりスタートコンディションを出力後、または ACK (通常は SBI 割り込みにより発生)受信後、データを送信します。

戻り値:
なし

10.2.3.13 SBI_GetReceiveData

データ受信

関数のプロトタイプ宣言:
uint32_t
SBI_GetReceiveData(TSB_SBI_TypeDef* ***SBIx***)

引数:
SBIx: SBI チャネルを指定します。

機能:
データを受信します。**SBI_GenerateI2Cstart()**の実行によりスタートコンディションを出力後、または ACK (通常は SBI 割り込みにより発生)受信後、データを受信します。

戻り値:
受信データ

10.2.3.14 SBI_SetI2CFreeDataMode

アドレス認識モードの指定

関数のプロトタイプ宣言:
void
SBI_SetI2CFreeDataMode(TSB_SBI_TypeDef* ***SBIx***,
FunctionalState ***NewState***)

引数:
SBIx: SBI チャネルを指定します。

NewState: アドレス認識モードを指定します。
- **ENABLE:** スレーブアドレスを認識しない。(フリーデータフォーマット)
- **DISABLE:** スレーブアドレスを認識する。

機能:
I2C モードにおけるデータフォーマットをフリーデータフォーマットにします。フリーデータフォーマットの場合、スレーブデバイスがデータ受信中にマスターデバイスは常にデータ送信を行います。転送データをノーマル I2C フォーマットにする場合は **SBI_InitI2C()**をコールしてください。

戻り値:
なし

10.2.4 データ構造

10.2.4.1 SBI_InitI2CTypeDef

メンバ:

uint32_t
***I2CSelfAddr*:** I2C モードにおけるスレーブアドレスを指定します。(0x01～0xFE)

uint32_t
***I2CDataLen*:** I2C モードにおける SBI チャネルの転送ビット数を指定します。
- **SBI_I2C_DATA_LEN_8:** データ長 8
- **SBI_I2C_DATA_LEN_1:** データ長 1
- **SBI_I2C_DATA_LEN_2:** データ長 2
- **SBI_I2C_DATA_LEN_3:** データ長 3
- **SBI_I2C_DATA_LEN_4:** データ長 4
- **SBI_I2C_DATA_LEN_5:** データ長 5

- **SBI_I2C_DATA_LEN_6:** データ長 6
- **SBI_I2C_DATA_LEN_7:** データ長 7

uint32_t
***I2CClkDiv*:** I2C 転送のソースクロックを選択します。
- **SBI_I2C_CLK_DIV_104:** fsys/104
- **SBI_I2C_CLK_DIV_136:** fsys/136
- **SBI_I2C_CLK_DIV_200:** fsys/200
- **SBI_I2C_CLK_DIV_328:** fsys/328
- **SBI_I2C_CLK_DIV_584:** fsys/584
- **SBI_I2C_CLK_DIV_1096:** fsys/1096
- **SBI_I2C_CLK_DIV_2120:** fsys/2120

FunctionalState
***I2CACKState*:** ACK の有効/無効を選択します。
- **ENABLE:** 有効。
- **DISABLE:** 無効。

10.2.4.2 SBI_I2CState

メンバ:

uint32_t
***All*:** I2C モードの全ての状態

Bit Fields:

uint32_t
***LastRxBit*:** 最終受信ビットモニタ

uint32_t
***GeneralCall*:** ゼネラルコール検出モニタ

uint32_t
***SlaveAddrMatch*:** スレーブアドレス一致モニタ

uint32_t
***ArbitrationLost*:** アービトレーションロスト検出モニタ

uint32_t
***INTReq*:** 割り込み要求状態モニタ

uint32_t
***BusState*:** バス状態モニタ

uint32_t
***TRx*:** 送信/受信選択状態モニタ

uint32_t
***MasterSlave*:** マスタ/スレーブ選択状態モニタ

11. TMR16A

11.1 概要

TMR16A には以下の機能があります。
- ・一致割り込み
- ・矩形波出力
- ・リードキャプチャ

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。

/Libraries/TX00_Periph_Driver/src/tmpm061_tmr16a.c
/Libraries/TX00_Periph_Driver/inc/tmpm061_tmr16a.h

11.2 API 関数

11.2.1 関数一覧

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

11.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています:
1) 各タイマの設定:
   TMR16A_SetSrcClk(), TMR16A_SetRunState(), TMR16A_ChangeCycle()
2) ステータスの確認:
   TMR16A_GetCaptureValue()
3) その他:
   TMR16A_SetFlipFlop(), TMR16A_SetClkInCoreHalt(), TMR16A_SetIdleMode()

11.2.3 関数仕様

補足: 引数に記述されている “TSB_T16A_TypeDef* ***T16Ax***” は下記から選択してください。
TSB_T16A0, TSB_T16A1, TSB_T16A2, TSB_T16A3, TSB_T16A4, TSB_T16A5 TSB_T16A6

11.2.3.1 TMR16A_SetIdleMode

IDLE 時の動作設定

関数のプロトタイプ宣言:
void

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

引数:
T16Ax: TMR16A チャネルを指定します。

NewState: IDLE 時の動作を指定します。
- **ENABLE:** 動作
- **DISABLE:** 停止

機能:
NewState が **ENABLE** の場合、IDLE 時でも TMR16A チャネルは動作します。
DISABLE の場合、IDLE 時は動作を停止します。

戻り値:
なし

11.2.3.2 TMR16A_SetClkInCoreHalt

デバッグ HALT 中のクロック動作

関数のプロトタイプ宣言:
void
TMR16A_SetClkInCoreHalt (TSB_T16A_TypeDef* ***T16Ax***,
uint8_t ***ClkState***)

引数:
T16Ax: TMR16A チャネルを指定します。

ClkState**:** デバッグ HALT 中のクロック動作を選択します。
- **TMR16A_RUNNING_IN_CORE_HALT:** 動作
- **TMR16A_STOP_IN_CORE_HALT:** 停止

機能:
デバッグツール使用時に HALT モードに遷移した場合、TMR16A クロック動作/停止の設定を行ないます。

戻り値:
なし

11.2.3.3 TMR16A_SetRunState

カウンタ動作の設定

関数のプロトタイプ宣言:
void
TMR16A_SetRunState(TSB_T16A_TypeDef* ***T16Ax***,
uint32_t ***Cmd***)

引数:
T16Ax: TMR16A チャネルを指定します。

Cmd: カウンタ動作を選択します。
- **TMR16A_RUN:** カウント
- **TMR16A_STOP:** 停止&クリア

機能:
Cmd が **TMR16A_RUN** の場合、アップカウンタがカウントを開始します。
Cmd が **TMR16A_STOP** の場合、アップカウンタはカウントを停止し、同時にカウンタをクリアします。

戻り値:
なし

11.2.3.4 TMR16A_SetSrcClk

ソースクロックの選択

関数のプロトタイプ宣言:
void
TMR16A_SetSrcClk(TSB_T16A_TypeDef* ***T16Ax***,
uint32_t ***SrcClk***)

引数:
T16Ax: TMR16A チャネルを指定します。
SrcClk: 以下からソースクロックを選択します。
- **TMR16A_SYSCK:** fsys
- **TMR16A_PRCK:** φT0

機能:
ソースクロックを選択します。

戻り値:
なし

11.2.3.5 TMR16A_SetFlipFlop

フリップフロップ機能の設定

関数のプロトタイプ宣言:
void
TMR16A_SetFlipFlop(TSB_T16A_TypeDef* ***T16Ax***,
TMR16A_FFOutputTypeDef* ***FFStruct***)

引数:
T16Ax: TMR16A チャネルを指定します。

***FFStruct*:** TMRB のフリップフロップ機能に関する構造体です。(詳細は"データ構造"を参照)

機能:
フリップフロップ出力変更のタイミングを設定します。また出力レベルも設定できます。

戻り値:
なし

11.2.3.6 TMR16A_ChangeCycle

周期の設定

関数のプロトタイプ宣言:
void
TMR16A_ChangeCycle(TSB_T16A_TypeDef* ***T16Ax***,
uint32_t ***Cycle***)

引数:
T16Ax: TMR16A チャネルを指定します。

Cycle: 周期を設定します。最大は 0xFFFF です。

機能:
周期を設定します。実際の周期は、CG の設定と ***ClkDiv***(詳細は"データ構造"を参照)の値によります。

戻り値:
なし

11.2.3.7 TMR16A_GetCaptureValue

キャプチャレジスタの読み込み

関数のプロトタイプ宣言:
uint16_t
TMR16A_GetCaptureValue(TSB_T16A_TypeDef* ***T16Ax***)

引数:
T16Ax: TMR16A チャネルを指定します。

機能:
キャプチャレジスタの値を読み込みます。

戻り値:
キャプチャレジスタの値

11.2.4 データ構造

11.2.4.1 TMR16A_FFOutputTypeDef

メンバ:

uint32_t
TMR16AFlipflopCtrl: フリップフロップのレベルを選択します。
- **TMR16A_FLIPFLOP_INVERT**: 出力を反転(ソフト反転)します。
- **TMR16A_FLIPFLOP_SET**: 出力を"1"にセットします。

- **TMR16A_FLIPFLOP_CLEAR**: 出力を"0"にセットします。

uint32_t
TMR16AFlipflopReverseTrg: フリップフロップの反転トリガを選択します。

- **TMR16A_DISALBE_FLIPFLOP**: 反転トリガを無効にします。
- **TMR16A_FLIPFLOP_MATCH_CYCLE**: アップカウンタと周期との一致時にタイマフリップフロップを反転します。

12. TMRB

12.1 概要

本デバイスは、2 チャンネルの 16 ビットタイマ/ イベントカウンタ (TMRB0, TMRB1)を内蔵しています。各チャネルは下記モードで動作します。

- 16 ビットインタバルタイマモード
- 16 ビットイベントカウンタモード
- 16 ビットプログラマブル矩形波出力 (PPG) モード
- タイマ同期モード(各 4 チャネルの出力設定可能)

また、キャプチャ機能を利用することで、次のような用途に使用することができます。

- 外部トリガパルスからのワンショットパルス出力
- 周波数測定
- パルス幅測定

本ドライバは、クロック分割、サイクル、デューティー期間、キャプチャタイミング、フリップフロップの設定など各チャネルの設定を行う関数セットです。また、アップカウンタ、フリップフロップ出力の制御など動作状態の制御、割り込み要因、キャプチャレジスタ値の取得など、ステータスの表示も行います。

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX00_Periph_Driver/src/tmpm061_tmrb.c
/Libraries/TX00_Periph_Driver/inc/tmpm061_tmrb.h

12.2 API 関数

12.2.1 関数一覧

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

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

12.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。:

1) 各タイマの設定:
   TMRB_Enable(), TMRB_Disable(), TMRB_Init(), TMRB_SetRunState(), TMRB_ChangeLeadingTiming(), TMRB_ChangeTrailingTiming()
2) キャプチャ機能の設定:
   TMRB_SetCaptureTiming(), TMRB_ExecuteSWCapture()
3) ステータスの確認:
   TMRB_GetINTFactor(), TMRB_GetUpCntValue(), TMRB_GetCaptureValue()
4) その他:
   TMRB_SetFlipFlop(), TMRB_SetINTMask(), TMRB_SetIdleMode(), TMRB_SetSyncMode(), TMRB_SetDoubleBuf(), TMRB_SetExtStartTrg(), TMRB_SetClkInCoreHalt (),TMRB_SetExtInput()

12.2.3 関数仕様

補足: 引数に記述されている “TSB_TB_TypeDef* ***TBx***” は下記から選択してください。
TSB_TB0, TSB_TB1.

12.2.3.1 TMRB_Enable

TMRB 動作の許可

関数のプロトタイプ宣言:
void
TMRB_Enable(TSB_TB_TypeDef* ***TBx***)

引数:
***TBx*:** TMRB チャネルを指定します。

機能:
TMRB 動作を有効にします。
チャネルが MPT の場合、本関数は、タイマモードとして MPT チャネルも選択します。

戻り値:
なし

12.2.3.2 TMRB_Disable

TMRB 動作の禁止

関数のプロトタイプ宣言:
void
TMRB_Disable(TSB_TB_TypeDef* ***TBx***)

引数:

***TBx*:** TMRB チャネルを指定します。

機能:
TMRB 動作を無効にします。

戻り値:
なし

12.2.3.3 TMRB_SetRunState

カウンタ動作の設定

関数のプロトタイプ宣言:
void
TMRB_SetRunState(TSB_TB_TypeDef* ***TBx***,
uint32_t ***Cmd***)

引数:
***TBx*:** TMRB チャネルを指定します。

Cmd: カウンタ動作を選択します。
- **TMRB_RUN:** カウント
- **TMRB_STOP:** 停止&クリア

機能:
Cmd が **TMRB_RUN** の場合、アップカウンタがカウントを開始します。
Cmd が **TMRB_STOP** の場合、アップカウンタはカウントを停止し、同時にカウンタをクリアします。

戻り値:
なし

12.2.3.4 TMRB_Init

TMRB チャネルの初期化

関数のプロトタイプ宣言:
void
TMRB_Init(TSB_TB_TypeDef* ***TBx***,
TMRB_InitTypeDef* ***InitStruct***)

引数:
***TBx*:** TMRB チャネルを指定します。

***InitStruct*:** TMRB に関する構造体です。(詳細は"データ構造"を参照)

機能:
カウンティングモード、クロック分周、アップカウンタ設定、サイクル、デューティー期間の初期設定を行います。

戻り値:

なし

12.2.3.5 TMRB_SetCaptureTiming

キャプチャタイミングの設定

関数のプロトタイプ宣言:
void
TMRB_SetCaptureTiming(TSB_TB_TypeDef* ***TBx***,
uint32_t ***CaptureTiming***)

引数:
***TBx*:** TMRB チャネルを指定します。

CaptureTiming: キャプチャタイミングを選択します。
- **TMRB_DISABLE_CAPTURE**: キャプチャ機能を無効にします。
- **TMRB_CAPTURE_IN_RISING**: TBxIN↑
- **TMRB_CAPTURE_IN_RISING_FALLING**: TBxIN↑ TBxIN↓
- **TMRB_CAPTURE_OUTPUT_EDGE**: TBxOUT↑ TBxOUT↓

機能:
CaptureTiming が **TMRB_CAPTURE_IN_RISING** の場合、TBxIN 端子入力の立ち上がりでキャプチャレジスタ 0 (TBxCP0)にカウント値を取り込みます。
CaptureTiming が **TMRB_CAPTURE_IN_RISING_FALLING** の場合、TBxIN 端子入力の立ち上がりでキャプチャレジスタ 0 (TBxCP0)にカウント値を取り込み、TBxIN 端子入力の立ち下がりでキャプチャレジスタ 1 (TBxCP1)にカウント値を取り込みます。
CaptureTiming が **TMRB_CAPTURE_OUTPUT_EDGE** の場合、16 ビットタイマー致出力(TBxOUT)の立ち上がりでキャプチャレジスタ 0 (TBxCP0)にカウント値を取り込み、TBxOUT の立ち下がりでキャプチャレジスタ 1 (TBxCP1)にカウント値を取り込みます。

戻り値:
なし

12.2.3.6 TMRB_SetFlipFlop

フリップフロップ機能の設定

関数のプロトタイプ宣言:
void
TMRB_SetFlipFlop(TSB_TB_TypeDef* ***TBx***,
TMRB_FFOutputTypeDef* ***FFStruct***)

引数:
***TBx*:** TMRB チャネルを指定します。

***FFStruct*:** TMRB のフリップフロップ機能に関する構造体です。(詳細は"データ構造"を参照)

機能:
フリップフロップ出力変更のタイミングを設定します。また出力レベルも設定できます。

戻り値:
なし

12.2.3.7 TMRB_GetINTFactor

割り込み要因の取得

関数のプロトタイプ宣言:
TMRB_INTFactor
TMRB_GetINTFactor(TSB_TB_TypeDef* ***TBx***)

引数:
***TBx*:** TMRB チャネルを指定します。

機能:
割り込み要因を取得します。

戻り値:
TMRB の割り込み要因:
MatchLeadingTiming (Bit0): 一致フラグ(TBxRG0)
MatchTrailingTiming (Bit1): 一致フラグ(TBxRG1)
OverFlow (Bit2): オーバーフローフラグ

補足:
異なる割り込み要因を処理する場合は、以下のように記述してください。

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

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

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

12.2.3.8 TMRB_SetINTMask

割り込みマスクの設定

関数のプロトタイプ宣言:
void
TMRB_SetINTMask(TSB_TB_TypeDef* ***TBx***,
uint32_t ***INTMask***)

引数:
***TBx*:** TMRB チャネルを指定します。

***INTMask*:** マスクする割り込みを選択します。

- **TMRB_MASK_MATCH_TRAILING_INT**: 一致フラグ(TBxRG0)
- **TMRB_MASK_MATCH_LEADING_INT**: 一致フラグ(TBxRG1)
- **TMRB_MASK_OVERFLOW_INT**: オーバーフロー割り込み。
- **TMRB_NO_INT_MASK**: マスクしない。

機能:
TMRB_MASK_MATCH_TRAILING_INT 選択時、アップカウンタ値と TBxRG1 が一致した場合、割り込みは発生しません。
TMRB_MASK_MATCH_LEADING_INT 選択時、アップカウンタ値と TBxRG0 が一致した場合、割り込みは発生しません。
TMRB_MASK_OVERFLOW_INT 選択時、オーバフロー発生時の割り込みは発生しません。
TMRB_NO_INT_MASK 選択時、割り込みマスクはすべてクリアされます。

戻り値:
なし

12.2.3.9 TMRB_ChangeLeadingTiming

デューティの設定

関数のプロトタイプ宣言:
void
TMRB_ ChangeLeadingTiming(TSB_TB_TypeDef* ***TBx***,
uint32_t ***LeadingTiming***)

引数:
***TBx*:** TMRB チャネルを指定します。

***LeadingTiming*:** デューティ値を設定します。最大値は 0xFFFF です。

機能:
デューティを設定します。実際のデューティーのインターバルは、CGの校正と ***ClkDiv***(詳細は"データ構造"を参照) の値によります。

戻り値:
なし。

補足:
LeadingTiming は ***TrailingTiming*** を超えることはできません。

12.2.3.10 TMRB_ChangeTrailingTiming

周期の設定

関数のプロトタイプ宣言:
void
TMRB_ChangeTrailingTiming(TSB_TB_TypeDef* ***TBx***,
uint32_t ***TrailingTiming***)

引数:
***TBx*:** TMRB チャネルを指定します。

TrailingTiming: 周期を設定します。最大は 0xFFFF です。

機能:
周期を設定します。実際の周期は、CG の設定と ***ClkDiv***(詳細は"データ構造"を参照)の値によります。

戻り値:
なし

Note:
TrailingTiming は ***LeadingTiming*** より小さくすることはできません。また PPG モードでは、TBxRG0/1 は TBxRG0 < TBxRG1 となるように設定してください。

12.2.3.11 TMRB_GetUpCntValue

アップカウンタ値の読み込み

関数のプロトタイプ宣言:
uint16_t
TMRB_GetUpCntValue(TSB_TB_TypeDef* ***TBx***)

引数:
***TBx*:** TMRB チャネルを指定します。

機能:
アップカウンタ値の読み込みを行います。

戻り値:
アップカウンタ値

12.2.3.12 TMRB_GetCaptureValue

キャプチャレジスタの読み込み

関数のプロトタイプ宣言:
uint16_t
TMRB_GetCaptureValue(TSB_TB_TypeDef* ***TBx***,
uint8_t ***CapReg***)

引数:
***TBx*:** TMRB チャネルを指定します。

CapReg: キャプチャレジスタを選択します。
- **TMRB_CAPTURE_0:** キャプチャレジスタ 0
- **TMRB_CAPTURE_1:** キャプチャレジスタ 1

機能:
CapReg が **TMRB_CAPTURE_0** の場合、キャプチャレジスタ 0 の値を読み込み、***CapReg*** が **TMRB_CAPTURE_1** の場合、キャプチャレジスタ 1 の値を読み込みます。

戻り値:

キャプチャレジスタの値

12.2.3.13 TMRB_ExecuteSWCapture

ソフトウェアキャプチャの実行

関数のプロトタイプ宣言:
void
TMRB_ExecuteSWCapture(TSB_TB_TypeDef* ***TBx***)

引数:
***TBx*:** TMRB チャネルを指定します。

機能:
キャプチャレジスタ 0 (TBxCP0)にカウント値を取り込みます。

戻り値:
なし

12.2.3.14 TMRB_SetIdleMode

IDLE 時の動作設定

関数のプロトタイプ宣言:
void
TMRB_SetIdleMode(TSB_TB_TypeDef* ***TBx***,
FunctionalState ***NewState***)

引数:
***TBx*:** TMRB チャネルを指定します。

NewState: IDLE 時の動作を指定します。
- **ENABLE:** 動作
- **DISABLE:** 停止

機能:
NewState が **ENABLE** の場合、IDLE 時でも TMRB チャネルは動作します。**DISABLE** の場合、IDLE 時は動作を停止します。

戻り値:
なし

12.2.3.15 TMRB_SetSyncMode

同期モードの切り替え

関数のプロトタイプ宣言:
void
TMRB_SetSyncMode(TSB_TB_TypeDef* ***TBx***,
FunctionalState ***NewState***)

引数:
***TBx*:** TMRB チャネルを以下から選択します。
TSB_TB0, TSB_TB1.

***NewState*:** 同期モードを切り替えます。
- **ENABLE:** 同期動作
- **DISABLE:** 個別動作(チャネル毎)

機能:
TMRB0~TMRB1 を同期モードに設定すると、TMRB0 のスタートに同期して動作がスタートします。

戻り値:
なし

補足:
同期モードを使用するために、TMRB0 のカウントを開始する前に、**TMRB_SetRunState()** によって TMRB0~MRB1 をスタートしてください。

12.2.3.16 TMRB_SetDoubleBuf

ダブルバッファ動作の制御

関数のプロトタイプ宣言:
```
void
TMRB_SetDoubleBuf(TSB_TB_TypeDef* TBx,
                  FunctionalState NewState)
```

引数:
***TBx*:** TMRB チャネルを指定します。

***NewState*:** ダブルバッファの有効/無効を選択します。
- **ENABLE:** 有効。
- **DISABLE:** 無効。

機能:
ダブルバッファ動作の許可/禁止を設定します。

戻り値:
なし

12.2.3.17 TMRB_SetExtStartTrg

外部トリガの設定

関数のプロトタイプ宣言:
```
void
TMRB_SetExtStartTrg (TSB_TB_TypeDef* TBx,
                     FunctionalState NewState,
                     uint8_t TrgMode)
```

引数:

***TBx*:** TMRB チャネルを指定します。

***NewState*:** カウントスタート方法を選択します。
- **ENABLE:** 外部トリガ
- **DISABLE:** ソフトスタート

***TrgMode*:** 外部トリガのアクティブエッジを選択します。
- **TMRB_TRG_EDGE_RISING:** 立ち上がりエッジ
- **TMRB_TRG_EDGE_FALLING:** 立ち下りエッジ

機能:
外部トリガによる変換開始の有無とアクティブエッジの設定を行います。

戻り値:
なし

12.2.3.18 TMRB_SetClkInCoreHalt

デバッグ HALT 中のクロック動作

関数のプロトタイプ宣言:
void
TMRB_SetClkInCoreHalt (TSB_TB_TypeDef* ***TBx***, uint8_t ***ClkState***)

引数:
***TBx*:** TMRB チャネルを指定します。

***ClkState*:** デバッグ HALT 中のクロック動作を選択します。
- **TMRB_RUNNING_IN_CORE_HALT:** 動作
- **TMRB_STOP_IN_CORE_HALT:** 停止

機能:
デバッグツール使用時に HALT モードに遷移した場合、TMRB クロック動作/停止の設定を行ないます。

戻り値:
なし

12.2.3.19 TMRB_SetExtInput

外部入力の設定

関数のプロトタイプ宣言:
void
TMRB_SetExtInput (TSB_TB_TypeDef* ***TBx***)

引数:
***TBx*:** TMRB チャネルを選択します。

機能:
外部入力として TBxIN0/1 を設定します。

戻り値:
なし

12.2.4 データ構造

12.2.4.1 TMRB_InitTypeDef

メンバ:

uint32_t
Mode: タイマモードを選択します。
- **TMRB_INTERVAL_TIMER**: インタバルタイマ
- **TMRB_EVENT_CNT**: イベントカウンタモード

uint32_t
ClkDiv: インターバルタイマのソースクロックの分周を選択します。
- **TMRB_CLK_DIV_2**: fperiph / 2
- **TMRB_CLK_DIV_8**: fperiph / 8
- **TMRB_CLK_DIV_32**: fperiph / 32
- **TMRB_CLK_DIV_64**: fperiph / 64
- **TMRB_CLK_DIV_128**: fperiph / 128
- **TMRB_CLK_DIV_256**: fperiph / 256
- **TMRB_CLK_DIV_512**: fperiph / 512

uint32_t
TrailingTiming: TBnRG1 へ書き込む周期 (最大 0xFFFF)

uint32_t
UpCntCtrl: アップカウンタの動作を選択します。
- **TMRB_FREE_RUN**: 周期が一致した後も、0xFFFF になるまでアップカウンタは停止しません。その後、カウンタがクリアされ、0 からカウントを開始します。
- **TMRB_AUTO_CLEAR**: **TrailingTiming** と一致したときに、0 クリアされ、再スタートします。

uint32_t
LeadingTiming: TBnRG0 に書き込むデューティ (最大 0xFFFF)。**TrailingTiming** 以上の値を設定できません。

12.2.4.2 TMRB_FFOutputTypeDef

メンバ:

uint32_t
FlipflopCtrl: フリップフロップのレベルを選択します。
- **TMRB_FLIPFLOP_INVERT:** TBxFF0 の値を反転(ソフト反転)します。
- **TMRB_FLIPFLOP_SET:** TBxFF0 を"1"にセットします。
- **TMRB_FLIPFLOP_CLEAR:** TBxFF0 を"0"にクリアします。

uint32_t
FlipflopReverseTrg: 以下から、フリップフロップの反転トリガを選択します。
- **TMRB_DISALBE_FLIPFLOP**: 反転トリガを無効にします。
- **TMRB_FLIPFLOP_TAKE_CATPURE_0**: アップカウンタの値がキャプチャレジスタ 0 に取り込まれた時にタイマフリップフロップを反転します。
- **TMRB_FLIPFLOP_TAKE_CATPURE_1**: アップカウンタの値がキャプチャレジスタ 1 に取り込まれた時にタイマフリップフロップを反転します。

- **TMRB_FLIPFLOP_MATCH_TRAILINGTIMING**: アップカウンタと周期との一致時にタイマフリップフロップを反転します。
- **TMRB_FLIPFLOP_MATCH_LEADINGTIMING**: アップカウンタとデューティとの一致時にタイマフリップフロップを反転します。

12.2.4.3 TMRB_INTFactor

メンバ:

uint32_t
All: TMRB 割り込み要因

Bit

uint32_t
MatchLeadingTiming: 1 デューティとの一致検出

uint32_t
MatchTrailingTiming : 1周期との一致検出

uint32_t
OverFlow : 1　　オーバーフロー

uint32_t
Reserverd : 29　　-

13. SIO/UART

13.1 概要

本デバイスのシリアル I/O チャネルは、I/O インタフェースモード(同期通信モード)と 7, 8, 9 ビット長の UART モード(非同期通信)を実装しています。9 ビット UART モードでは、シリアルリンク(マルチコントローラ・システム) でマスタコントローラがスレーブコントローラを起動するときにウェイクアップ機能が使用されます。

本ドライバは 、ボーレート、ビット長、パリティチェック、ストップビット、フローコントロールなどの各チャネルの設定に関する関数、およびデータ送受信、エラーチェックなどの動作に関する機能を備えています。

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX00_Periph_Driver/src/tmpm061_uart.c, with
/Libraries/TX00_Periph_Driver/inc/tmpm061_uart.h

13.2 API 関数

13.2.1 関数一覧

- void UART_Enable(TSB_SC_TypeDef* ***UARTx***)
- void UART_Disable(TSB_SC_TypeDef* ***UARTx***)
- WorkState UART_GetBufState(TSB_SC_TypeDef* ***UARTx***, uint8_t ***Direction***)
- void UART_SWReset(TSB_SC_TypeDef* ***UARTx***)
- void UART_Init(TSB_SC_TypeDef* ***UARTx***, UART_InitTypeDef* ***InitStruct***)
- uint32_t UART_GetRxData(TSB_SC_TypeDef* ***UARTx***)
- void UART_SetTxData(TSB_SC_TypeDef* ***UARTx***, uint32_t ***Data***)
- void UART_DefaultConfig(TSB_SC_TypeDef* ***UARTx***)
- UART_Err UART_GetErrState(TSB_SC_TypeDef* ***UARTx***)
- void UART_SetWakeUpFunc(TSB_SC_TypeDef* ***UARTx***,
  FunctionalState ***NewState***)
- void UART_SetIdleMode(TSB_SC_TypeDef* ***UARTx***, FunctionalState ***NewState***)
- void SIO_Enable(TSB_SC_TypeDef * SIOx)
- void SIO_Disable(TSB_SC_TypeDef * SIOx)
- uint8_t SIO_GetRxData(TSB_SC_TypeDef * SIOx)
- void SIO_SetTxData(TSB_SC_TypeDef * SIOx, uint8_t Data)
- void SIO_Init(TSB_SC_TypeDef * SIOx, uint32_t IOClkSel, SIO_InitTypeDef * InitStruct)

13.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています。:

1) 初期化と設定:
   UART_Enable(), UART_Disable(), UART_Init(), UART_DefaultConfig(),
   SIO_Enable(), SIO_Disable(), SIO_Init()
2) 送受信設定とエラー確認:
   UART_GetBufState(), UART_GetRxData(), UART_SetTxData(),
   UART_GetErrState(),SIO_GetRxData(), SIO_SetTxData()

3) その他:
UART_SWReset(), UART_SetWakeUpFunc(), UART_SetIdleMode()

13.2.3 関数仕様

補足: 引数に記述している“TSB_SC_TypeDef* ***UARTx***” は、以下から選択してください。
UART0, **UART1**, **UART2**, **UART3**.
引数に記述している“TSB_SC_TypeDef* ***SIOx***” は、以下から選択してください。
SIO0, **SIO1**, **SIO2**, **SIO3**.

13.2.3.1 UART_Enable

UART 動作の許可

関数のプロトタイプ宣言:
void
UART_Enable(TSB_SC_TypeDef* ***UARTx***)

引数:
UARTx: UART チャネルを指定します。

機能:
UART 動作を許可します。

戻り値:
なし

13.2.3.2 UART_Disable

UART 動作の禁止

関数のプロトタイプ宣言:
void
UART_Disable(TSB_SC_TypeDef* ***UARTx***)

引数:
UARTx: UART チャネルを指定します。

機能:
UART 動作を禁止します。

戻り値:
なし

13.2.3.3 UART_GetBufState

送受信バッファ状態の読み込み

関数のプロトタイプ宣言:
WorkState
UART_GetBufState(TSB_SC_TypeDef* ***UARTx***,
uint8_t ***Direction***)

引数:
UARTx: UART チャネルを指定します。
Direction: 送信/受信を選択します。
- **UART_RX**: 受信
- **UART_TX**: 送信

機能:
Direction が **UART_RX** の場合、以下の受信バッファの状態を返します。
DONE:受信データはバッファに保存済み
BUSY:データ受信中
Direction が **UART_TX** の場合、以下の送信バッファの状態を返します。
DONE:バッファ中のデータは送信済み
BUSY:データ送信中

戻り値:
DONE: バッファリード/ライト可能状態
BUSY: 送受信中

13.2.3.4 UART_SWReset

ソフトウェアリセット

関数のプロトタイプ宣言:
void
UART_SWReset(TSB_SC_TypeDef* ***UARTx***)

引数:
UARTx: UART チャネルを指定します。

機能:
ソフトウェアリセットが発生します。

戻り値:
なし

13.2.3.5 UART_Init

UART チャネルの初期化

関数のプロトタイプ宣言:
void
UART_Init(TSB_SC_TypeDef* ***UARTx***,
UART_InitTypeDef* ***InitStruct***)

引数:
UARTx: UART チャネルを指定します。
InitStruct: UART に関する構造体です。(詳細は“データ構造”を参照)

機能:

ボーレート、ビット単位の転送、ストップビット、パリティ、転送モード、フローコントロールなどの初期設定を行います。

戻り値:
なし

13.2.3.6 UART_GetRxData

受信データの読み込み

関数のプロトタイプ宣言:
uint32_t
UART_GetRxData(TSB_SC_TypeDef* ***UARTx***)

引数:
UARTx: UART チャネルを指定します。

機能:
受信データを読み込みます。**UART_GetBufState(*UARTx,* UART_RX)**にて **DONE** を読み出した後、もしくは UART (シリアルチャネル) 割り込み関数の中で実行してください。

戻り値:
受信データです。データ範囲は 0x00～0x1FF です

13.2.3.7 UART_SetTxData

送信データの設定

関数のプロトタイプ宣言:
void
UART_SetTxData(TSB_SC_TypeDef* ***UARTx***,
uint32_t ***Data***)

引数:
UARTx: UART チャネルを指定します。
Data: 送信データ(7 ビット、8 ビット、9 ビット)

機能:
送信データを設定します。**UART_GetBufState(*UARTx,* UART_TX)**にて **DONE** を読み出した後、もしくは UART (シリアルチャネル) 割り込み関数の中で実行してください。

戻り値:
なし

13.2.3.8 UART_DefaultConfig

デフォルト構成での初期化

関数のプロトタイプ宣言:

void
UART_DefaultConfig(TSB_SC_TypeDef* ***UARTx***)

引数:
UARTx: UART チャネルを指定します。

機能:
以下の構成で初期化します:
ボーレート: 115200 bps
データ長: 8 ビット
ストップビット: 1 ビット
パリティ: なし
フローコントロール: なし
送受信有効。ボーレートジェネレータはソースクロックとして使用。

戻り値:
なし

13.2.3.9 UART_GetErrState

転送エラーフラグの読み出し

関数のプロトタイプ宣言:
UART_Err
UART_GetErrState(TSB_SC_TypeDef* ***UARTx***)

引数:
UARTx: UART チャネルを指定します。

機能:
転送エラーフラグを読み出します。

戻り値:
UART_NO_ERR: エラーなし
UART_OVERRUN: オーバーランエラー
UART_PARITY_ERR: パリティエラー
UART_FRAMING_ERR: フレーミングエラー
UART_ERRS: 上記の 2 つ以上のエラーが発生している

13.2.3.10 UART_SetWakeUpFunc

9 ビットモード時のウェイクアップ機能の設定

関数のプロトタイプ宣言:
void
UART_SetWakeUpFunc(TSB_SC_TypeDef* ***UARTx***,
FunctionalState ***NewState***)

引数:
UARTx: UART チャネルを指定します。
NewState: ウェイクアップ機能の有効/無効を選択します。

- **ENABLE**: 有効
- **DISABLE**: 無効

機能:
9 ビットモード時のウェイクアップ機能を設定します。
NewState が **ENABLE** の場合、ウェイクアップ機能を有効に、
NewState が **DISABLE** の場合、ウェイクアップ機能を無効に設定します。
ウェイクアップ機能は、9 ビットモード時のみ機能します。

戻り値:
なし

13.2.3.11 UART_SetIdleMode

IDLE 時の動作

関数のプロトタイプ宣言:
void
UART_SetIdleMode(TSB_SC_TypeDef* ***UARTx***,
FunctionalState ***NewState***)

引数:
UARTx: UART チャネルを指定します。
NewState: IDLE 時の動作を選択します。
- **ENABLE**: 動作
- **DISABLE**: 停止

機能:
IDLE 時の動作を選択します。
NewState が **ENABLE** の場合、IDLE 時でも UART チャネルは動作します。**DISABLE** の場合、IDLE 時は動作を停止します。

戻り値:
なし

13.2.3.12 SIO_Enable

SIO 動作の許可

関数のプロトタイプ宣言:
void
SIO_Enable (TSB_SC_TypeDef* ***SIOx***)

引数:
SIOx: SIO チャネルを指定します。

機能:
SIO 動作を許可します。

戻り値:
なし

13.2.3.13 SIO_Disable

SIO 動作の禁止

関数のプロトタイプ宣言:
void
SIO _Disable(TSB_SC_TypeDef* ***SIOx***)

引数:
SIOx: SIO チャネルを指定します。

機能:
SIO 動作を禁止します。

戻り値:
なし

13.2.3.14 SIO_GetRxData

受信用バッファ

関数のプロトタイプ宣言:
uint32_t
SIO_GetRxData(TSB_SC_TypeDef* ***SIOx***)

引数:
SIOx: SIO チャネルを指定します。

機能:
受信用バッファを取得します。

戻り値:
受信用バッファ(値の範囲は 0x00 ~ 0xFF です)

13.2.3.15 SIO_SetTxData

送信用バッファ

関数のプロトタイプ宣言:
void
SIO_SetTxData(TSB_SC_TypeDef* ***SIOx***,
uint8_t ***Data***)

引数:
SIOx: SIO チャネルを指定します。
Data: 送信用バッファ

機能:
送信用バッファを指定します。

戻り値:
なし

13.2.3.16 SIO_Init

SIO チャネルの初期化

関数のプロトタイプ宣言:
void
SIO_Init(TSB_SC_TypeDef* ***SIOx***,
uint32_t ***IOClkSel,***
SIO_InitTypeDef* ***InitStruct***)

引数:
SIOx: SIO チャネルを指定します。
IOClkSel: クロックを選択します。
- **SIO_CLK_BAUDRATE**: ボーレートジェネレータ
- **SIO_CLK_SCLKINPUT**: SCLKx 端子入力

InitStruct: SIO に関する構造体です。(詳細は“データ構造”を参照)

機能:
ボーレート、転送方向、転送モードなどの初期設定を行います。

戻り値:
なし

13.2.4 データ構造

13.2.4.1 UART_InitTypeDef

メンバ:

メンバ:

uint32_t
BaudRate:UART 通信ボーレートを 2400(bps) から 115200(bps) に設定。(*)

uint32_t
DataBits: 転送ビット数を選択します。
- **UART_DATA_BITS_7**: 7 ビットモード
- **UART_DATA_BITS_8**: 8 ビットモード
- **UART_DATA_BITS_9**:9 ビットモード

uint32_t
StopBits:ストップビット長を選択します。
- **UART_STOP_BITS_1:** 1 ビット
- **UART_STOP_BITS_2:** 2 ビット

uint32_t
Parity: パリティを選択します。
- **UART_NO_PARITY:** パリティなし
- **UART_EVEN_PARITY**: 偶数(Even) パリティ
- **UART_ODD_PARITY**: 偶数(Even) パリティ

uint32_t
Mode: 転送モードを選択します。送受信の場合は、送信と受信を OR 演算子によって接続して指定してください。
UART_ENABLE_TX: 送信許可
- **UART_ENABLE_RX:** 受信許可

uint32_t
FlowCtrl: フローコントロールモードを選択します(**)。
- **UART_NONE_FLOW_CTRL**:CTS 無効

*: fperiph の周波数が高すぎる、または、低すぎると、ボーレートが正しく設定できない場合があります。
**: 本バージョンでは UART_NONE_FLOW_CTRL のみ選択可能です。

13.2.4.2 SIO_InitTypeDef

uint32_t
InputClkEdge: 入力クロックエッジを選択します。"0"(SIO_SCLKS_TXDF_RXDR)のみ指定可能です。

uint32_t
IntervalTime: 連続転送時のインターバル時間を選択します。
- **SIO_SINT_TIME_NONE**: なし
- **SIO_SINT_TIME_SCLK_1**: 1*SCLK
- **SIO_SINT_TIME_SCLK_2**: 2*SCLK
- **SIO_SINT_TIME_SCLK_4**: 4*SCLK
- **SIO_SINT_TIME_SCLK_8**: 8*SCLK
- **SIO_SINT_TIME_SCLK_16**: 16*SCLK
- **SIO_SINT_TIME_SCLK_32**: 32*SCLK
- **SIO_SINT_TIME_SCLK_64**: 64*SCLK

uint32_t
TransferMode: 転送モードを選択します。
- **SIO_TRANSFER_PROHIBIT**: 転送禁止
- **SIO_TRANSFER_HALFDPX_RX**: 半二重(受信)
- **SIO_TRANSFER_HALFDPX_TX**: 半二重(送信)
- **SIO_TRANSFER_FULLDPX**: 全二重

uint32_t
TransferDir: 転送方向を選択します。
- **SIO_LSB_FRIST**: LSB FRIST
- **SIO_MSB_FRIST**: MSB FRIST

uint32_t
Mode: 送受信を制御します。有効ビットの組み合わせが可能です。
- **SIO_ENABLE_TX**: 送信許可
- **SIO_ENABLE_RX**: 受信許可

uint32_t
DoubleBuffer: ダブルバッファの許可/禁止を選択します。
- **SIO_WBUF_ENABLE**: 許可
- **SIO_WBUF_DISABLE**: 禁止

uint32_t
BaudRateClock: ボーレートジェネレータ入力クロックを選択します。
- **SIO_BR_CLOCK_T1**: φT1
- **SIO_BR_CLOCK_T4**: φT4
- **SIO_BR_CLOCK_T16**: φT16
- **SIO_BR_CLOCK_T64**: φT64

uint32_t
Divider: 分周値"N"を選択します。

- **SIO_BR_DIVIDER_1**: 1 分周
- **SIO_BR_DIVIDER_2**: 2 分周
- **SIO_BR_DIVIDER_3**: 3 分周
- **SIO_BR_DIVIDER_4**: 4 分周
- **SIO_BR_DIVIDER_5**: 5 分周
- **SIO_BR_DIVIDER_6**: 6 分周
- **SIO_BR_DIVIDER_7**: 7 分周
- **SIO_BR_DIVIDER_8**: 8 分周
- **SIO_BR_DIVIDER_9**: 9 分周
- **SIO_BR_DIVIDER_10**: 10 分周
- **SIO_BR_DIVIDER_11**: 11 分周
- **SIO_BR_DIVIDER_12**: 12 分周
- **SIO_BR_DIVIDER_13**: 13 分周
- **SIO_BR_DIVIDER_14**: 14 分周
- **SIO_BR_DIVIDER_15**: 15 分周
- **SIO_BR_DIVIDER_16**: 16 分周

*: fperiph の周波数が高すぎる、または、低すぎると、ボーレートが正しく設定できない場合があります。

14. WDT

14.1 概要

ウォッチドッグタイマは、ノイズなどの原因によりCPU が誤動作(暴走)を始めた場合、これを検出し正常な状態に戻すことを目的としています。

検出時間、カウンターのオーバーフロー時の出力、アイドルモードでの動作設定などの引数等、ウォッチドッグタイマの設定を行う関数を提供します。

本ドライバは、以下のファイルで構成されています。
\Libraries\TX00_Periph_Driver\src\tmpm061_wdt.c
\Libraries/TX00_Periph_Driver\inc\tmpm061_wdt.h

14.2 API 関数

14.2.1 関数一覧

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

14.2.2 関数の種類

関数は、主に以下の 2 種類に分かれています。:
1) ウォッチドッグタイマ設定:
WDT_SetDetectTime(), WDT_SetOverflowOutput(), WDT_Init(), WDT_Enable(), WDT_Disable(), WDT_WriteClearCode()
2) IDLE モード時の開始・停止など:
WDT_SetIdleMode()

14.2.3 関数仕様

14.2.3.1 WDT_SetDetectTime

検出時間の設定

関数のプロトタイプ宣言:
void
WDT_SetDetectTime(uint32_t ***DetectTime***)

引数:
***DetectTime*:** 検出時間を選択します。
- **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

機能:
WDT の検出時間を設定します。

戻り値:
なし

14.2.3.2 WDT_SetIdleMode

IDLE モード時の動作

関数のプロトタイプ宣言:
void
WDT_SetIdleMode(FunctionalState ***NewState***)

引数:
NewState: IDLE 時の動作の有効/無効を選択します。
- **ENABLE**: 動作
- **DISABLE**. 停止

機能:
本関数は、IDLE モード時の WDT カウンタの動作を設定します。
NewState が **ENABLE** の時は WDT カウンタ動作
NewState が **DISABLE** の時は WDT カウンタ停止

補足:
CPU が IDLE モードに入る前に、設定してください。

戻り値:
なし

14.2.3.3 WDT_SetOverflowOutput

カウンタオーバーフロー時の WDT 動作(NMI 割り込みを発生、またはリセット)の設定。

関数のプロトタイプ宣言:
void
WDT_SetOverflowOutput(uint32_t ***OverflowOutput***)

引数:
OverflowOutput: カウンタオーバーフロー時の設定を選択します。
- **WDT_NMIINT**: NMI 割り込み発生
- **WDT_WDOUT**: リセット

機能:
カウンタオーバーフロー時の NMI 割り込み/リセットの設定を行います。
OverflowOutputi が **WDT_NMIINT** の時、カウンタオーバーフローが発生すると NMI 割り込みが発生します。

戻り値:
なし

14.2.3.4 WDT_Init

WDT の初期化

関数のプロトタイプ宣言:
void
WDT_Init (WDT_InitTypeDef* ***InitStruct***)

引数:
InitStruct:カウンタオーバーフロー発生時の WDT 検出時間、WDT 出力の設定を含む WDT 設定。(詳細は“データ構造” を参照してください)

機能:
カウンタオーバーフロー発生時の WDT 検出時間、WDT 出力の設定を含む WDT 初期設定を行います。WDT_SetDetectTime(), WDT_SetOverflowOutput() が呼び出されます。

戻り値:
なし

14.2.3.5 WDT_Enable

WDT 動作の許可

関数のプロトタイプ宣言:
void
WDT_Enable(void)

引数:
なし。

機能:
WDT 動作を許可します。

戻り値:
なし

14.2.3.6 WDT_Disable

WDT 動作の禁止

関数のプロトタイプ宣言:
void
WDT_Disable(void)

引数:
なし。

機能:
WDT 動作を禁止します。

戻り値:
なし

14.2.3.7 WDT_WriteClearCode

クリアコードの書き込み

関数のプロトタイプ宣言:
void
WDT_WriteClearCode (void)

引数:
なし。

機能:
WDT カウンタにクリアコードを書き込みます。

戻り値:
なし

14.2.4 データ構造

14.2.4.1 WDT_InitTypeDef

メンバ:

uint32_t
DetectTime 検出時間を選択します。

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

uint32_t
***OverflowOutput*:** カウンタオーバーフロー時の設定を選択します。

- **WDT_WDOUT:** リセット
- **WDT_NMIINT:** NMI 割り込み