TOSHIBA

TX00 ペリフェラルドライバ
ユーザーガイド
(TMPM037)

第一版

2017 年 9 月

東芝デバイス＆ストレージ株式会社

CMDR-M037UG-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、TMPM036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。

TMPM036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。
- すべての周辺機能をカバーしています。

2. TX00 ペリフェラルドライバの構成

/Libraries
TX00 CMSIS ファイルと TMPM037 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 TMPM037CMSIS 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
TMPM037 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
TMPM037 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
TMPM037 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
TMPM037 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
TMPM037 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
TMPM037 ペリフェラルドライバの使用例が格納されています。

/Utilities/TMPM037-EVAL
TMPM037 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、入力チャネルとして 0~7 の 8 チャネルを使用します。
以下の特徴があります。
• 割り込み、または外部トリガ入力による機動
• 固定チャネル/スキャンモード
• シングル/リピートモード
• AD監視（2本）

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\tmpm037_adc.c
/Libraries/TX00_Periph_Driver\inc\tmpm037_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(uint32_t ***INTMode***)
- ADC_State ADC_GetConvertState(void)
- void ADC_SetInputChannel(uint32_t ***InputChannel***)
- void ADC_SetChannelScanMode(ADC_ChannelScanMode ***ScanMode***)
- void ADC_SetIdleMode(FunctionalState ***NewState***)
- void ADC_SetVref(FunctionalState ***NewState***)
- void ADC_SetInputChannelTop(uint32_t ***TopInputChannel***)
- void ADC_StartTopConvert(void)
- void ADC_SetMonitor(uint8_t ***ADCMPx***, FunctionalState ***NewState***)
- void ADC_SetResultCmpReg(uint8_t ***ADCMPx***, uint32_t ***ResultComparison***)
- void ADC_SetMonitorINT(uint8_t ***ADCMPx***, ADC_ComparisonState ***NewState***)
- void ADC_SetHWTrg(uint32_t ***HwSource***, FunctionalState ***NewState***)
- void ADC_SetHWTrgTop(uint32_t ***HwSource***, FunctionalState ***NewState***)
- ADC_ResultTypeDef ADC_GetConvertResult (uint32_t ***ADREGx***)
- void ADC_SetCmpValue(uint8_t ***ADCMPx***, uint16_t ***value***)
- void ADC_SetDMAReq(uint8_t ***DMAReq***, FunctionalState ***NewState***)

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(), ADC_SetInputChannelTop(), ADC_SetMonitor(), ADC_SetResultCmpReg(), ADC_SetMonitorINT(), ADC_SetHWTrg(), ADC_SetHWTrgTop(), ADC_SetCmpValue()
2) AD 変換の許可/禁止と開始:
   ADC_Start(), ADC_StartTopConvert()
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_SetDMAReq()

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。

補足:
ADxCLK<ADCLK>を除くレジスタは、すべて初期化されます。
ソフトウェアリセットを行う場合、初期化に 3μs の時間が必要となります。

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 AD 変換時間を選択します。

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

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_6:** fc / 6
- **ADC_FC_DIVIDE_LEVEL_8:** fc / 8
- **ADC_FC_DIVIDE_LEVEL_12:** fc / 12
- **ADC_FC_DIVIDE_LEVEL_16:** fc / 16
- **ADC_FC_DIVIDE_LEVEL_24:** fc / 24
- **ADC_FC_DIVIDE_LEVEL_48:** fc / 48
- **ADC_FC_DIVIDE_LEVEL_96:** fc / 96

機能:
Conversion_Time で AD 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。
変換クロック数は以下の条件を満たすように設定してください。

VREFH AVDD	変換時間
2.7 ~ 3.6V	16.2us 以上
2.3 ~ 3.6V	32.4us 以上

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:
ADC_State
ADC_GetConvertState(void)

引数:
なし

機能:
AD 変換終了フラグ (通常と最優先の両方)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:
NormalBusy(Bit 0) : 通常 AD 変換中
NormalComplete (Bit 1): 通常 AD 変換終了
TopBusy(Bit 2) : 最優先 AD 変換中
TopComplete (Bit 3): 最優先 AD 変換終了

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, ADC_AN_1, ADC_AN_2, ADC_AN_3,
ADC_AN_4, ADC_AN_5, ADC_AN_6, ADC_AN_7

機能:
アナログ入力チャネルを選択します。またアナログ入力チャネルは入力モードに依存します。
チャネル固定モード(**ADC_SetScanMode(DISABLE)**)では、8 チャネルの中から 1 チャネルのみ選択可能です。
チャネルスキャンモード(**ADC_SetScanMode(ENABLE)**)では、アナログ入力チャネルは 4 チャネルスキャンモード(**ADC_SetChannelScanMode(ADC_SCAN_4CH)**)、8 チャネルスキャンモード(**ADC_SetChannelScanMode(ADC_SCAN_8CH)**)により異なります。

補足:
1 チャネルスキャンモード: **ADC_SetScanMode(ENABLE)**
4 チャネルスキャンモード: **ADC_SetChannelScanMode(ADC_SCAN_4CH)**
8 チャネルスキャンモード: **ADC_SetChannelScanMode(ADC_SCAN_8CH)**

戻り値:
なし

3.2.3.9 ADC_SetChannelScanMode

チャネルスキャン時の動作モード設定

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時の動作モードを選択します。
- **ADC_SCAN_4CH:** 4ch スキャン
- **ADC_SCAN_8CH:** 8ch スキャン

機能:
チャネルスキャン時の動作モードを設定します。

補足:
本 API は **ADC_SetInputChannel()**によるアナログ入力チャネルを変更します。この API の詳細は **ADC_SetInputChannel()** の説明を参照してください。

戻り値:
なし

3.2.3.10 ADC_SetIdleMode

IDLE モード時の ADC 動作制御

関数のプロトタイプ宣言:
void

ADC_SetIdleMode(FunctionalState ***NewState***)

引数:
NewState: 以下から、IDLE モード時の ADC 動作を選択します。
- **ENABLE**: 動作
- **DISABLE**: 停止

機能:
IDLE モード時の ADC 動作を制御します。
IDLE モードに移行する前にこの関数をコールする必要があります。

戻り値:
なし

3.2.3.11 ADC_SetVref

Vref 回路の on/off 制御

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

引数:
NewState: 以下から、Vref 回路の状態を選択します。
- **ENABLE**: ON
- **DISABLE**: OFF

機能:
Vref 回路の on/off を制御します。

補足:
低消費電力モードに移行する前に、**ADC_SetVref(DISABLE**) をコールしてください。

戻り値:
なし

3.2.3.12 ADC_SetInputChannelTop

最優先 AD 変換入力チャネルの設定

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

引数:
TopInputChannel: 以下から、最優先 AD 変換入力チャネルを選択します。
ADC_AN_0, ADC_AN_1, ADC_AN_2, ADC_AN_3,
ADC_AN_4, ADC_AN_5, ADC_AN_6, ADC_AN_7

機能:
最優先 AD 変換入力チャネルを設定します。

補足:
最優先 AD 変換入力を 1 チャネルのみ選択できます。

戻り値:
なし

3.2.3.13 ADC_StartTopConvert

最優先 AD 変換の開始

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

引数:
なし

機能:
最優先 AD 変換を開始します。

補足:
この関数をコールする前 **ADC_SetInputChannelTop()**をコールしてください。

戻り値:
なし

3.2.3.14 ADC_SetMonitor

AD 監視機能の許可/禁止

関数のプロトタイプ宣言:
void
ADC_SetMonitor(ADC_CMPCRx ***ADCMPx***,
FunctionalState ***NewState***)

引数:
ADCMPx: 以下から、監視機能設定レジスタを選択します。
- **ADC_CMP_0:** ADCMPCR0
- **ADC_CMP_1:** ADCMPCR1

NewState: 以下から、監視機能を設定します。
- **ENABLE**: 許可(条件成立で AD 監視割り込みを発生します)
- **DISABLE**: 禁止(大小判定カウント数はクリア)

機能:
本デバイスは、2 つの AD 監視機能を持ち、それぞれ設定レジスタで制御します。
ADCMPx 設定で AD 監視レジスタを選択し、NewState で許可/禁止を設定します。

戻り値:
なし

3.2.3.15 ADC_SetResultCmpReg

AD 変換結果と比較するレジスタの設定

関数のプロトタイプ宣言:
void
ADC_SetResultCmpReg(uint8_t ***ADCMPx***,
uint32_t ***ResultComparison***)

引数:
ADCMPx:以下から AD 変換比較レジスタを選択します。
- **ADC_CMP_0: ADCMP0**
- **ADC_CMP_1: ADCMP1**

ResultComparison: AD 変換結果を格納するレジスタを設定します。
ADC_REG_0, **ADC_REG_1**, **ADC_REG_2**, **ADC_REG_3**, **ADC_REG_4**, **ADC_REG_5**, **ADC_REG_6**, **ADC_REG_7**, **ADC_REG_SP**

機能:
AD 変換結果と比較するレジスタを設定します。

戻り値:
なし

3.2.3.16 ADC_SetMonitorINT

AD 監視割り込みの設定

関数のプロトタイプ宣言:
void
ADC_SetMonitorINT(uint8_t ***ADCMPx***,
ADC_ComparisonState ***NewState***)

引数:
ADCMPx:以下から AD 変換比較レジスタを選択します。
- **ADC_CMP_0: ADCMP0**
- **ADC_CMP_1: ADCMP1**

NewState: AD 監視割り込み発生条件を選択します。
ADC_COMPARISON_SMALLER、または **ADC_COMPARISON_LARGER**.

機能:
本デバイスは 2 つの AD 監視機能を持ち、それぞれ設定レジスタで制御します。
ADCMPx で AD 変換レジスタを選択し、***NewState*** で AD 監視割り込みを設定します。

戻り値:
なし

3.2.3.17 ADC_SetHWTrg

通常 AD 変換を開始するためのハードウェア起動要因の選択

関数のプロトタイプ宣言:
void
ADC_SetHWTrg(uint32_t ***HwSource***,
FunctionalState ***NewState***)

引数:
HwSource: 以下から、通常 AD 変換のハードウェア起動要因を選択します。
- **ADC_EXT_TRG:** ADTRG 端子で起動することが可能です。
- **ADC_MATCH_TB:** 16 ビットタイマ/イベントカウンタのコンペアレジスタ 0 一致割り込みで起動することが可能です。

NewState: 以下から、ハードウェア起因による通常 AD 変換開始の許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
HwSource の設定により通常 AD 変換のハードウェア起動要因を設定し、***NewState*** により通常 AD 変換のハードウェア起動の許可/禁止を選択します。
この関数は TB6 の設定にも関連しています。

補足:
本デバイスには HW 起動ソースとしての外部トリガ入力はありませんので、
ADC_SetHWTrg(ADC_EXT_TRG, NewState)という設定はできません。
16 ビットタイマの一致トリガ<ADHTG>、<HADHTG>に"1"を設定して HW 起動リソースによる AD 変換を行う場合、以下の順に設定することにより、一定間隔での AD 変換が可能となります。
- HW のソースを選択
- AD 変換の HW 起動をイネーブル
- タイマ動作

最優先 AD 変換、通常 AD 変換設定は同時に行わないでください。
ADC_SetHWTrg(ADC_MATCH_TB0, ENABLE) と
ADC_SetHWTrgTop(ADC_MATCH_TB1, ENABLE) は同時に設定しないでください。

戻り値:
なし

3.2.3.18 ADC_SetHWTrgTop

最優先 AD 変換を開始するためのハードウェア起動要因の選択

関数のプロトタイプ宣言:
void
ADC_SetHWTrgTop(uint32_t ***HwSource***,
FunctionalState ***NewState***)

引数:
HwSource: 以下から、最優先 AD 変換のハードウェア起動要因を選択します。

- **ADC_EXT_TRG:** ADTRG 端子で起動することが可能です。
- **ADC_MATCH_TB1:** 16 ビットタイマ/イベントカウンタのコンペアレジスタ 0 一致割り込みで起動することが可能です。

NewState: 以下から、ハードウェア起因による通常 AD 変換開始の許可/禁止を選択します。

- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
HwSource の設定により最優先 AD 変換のハードウェア起動要因を設定し、***NewState*** により最優先 AD 変換のハードウェア起動の許可/禁止を選択します。
この関数は TB1 の設定にも関連しています。

補足:
本製品は外部トリガ端子がないため、外部トリガは無効です。このため、第 1 引数に "**ADC_EXT_TRG**"を指定することはできません。
最優先 AD 変換と通常 AD 変換を同時に使うことはできません。
ADC_SetHWTrg(ADC_MATCH_TB0, ENABLE) と
ADC_SetHWTrgTop(ADC_MATCH_TB1, ENABLE) を同時に使うことはできません。

戻り値:
なし

3.2.3.19 ADC_GetConvertResult

AD 変換結果の取得

関数のプロトタイプ宣言:
ADC_ResultTypeDef
ADC_GetConvertResult(uint32_t ***ADREGx***)

引数:
ADREGx: 以下から AD 変換結果レジスタを選択します。
ADC_REG_0, **ADC_REG_1**, **ADC_REG_2**, **ADC_REG_3**, **ADC_REG_4**, **ADC_REG_5**, **ADC_REG_6**, **ADC_REG_7**, **ADC_REG_SP**

機能:
AD 変換終了フラグ、オーバーランフラグ、変換結果を取得します。

補足:
変換結果が格納されると AD 変換格納フラグ ***ADREGx*** が **DONE** になります。本関数によって変換結果が読み出されると、AD 変換結果格納フラグ ***ADREGx*** がクリアされます。

変換結果格納レジスタ(ADREGx)の値が読み出される前に変換結果が上書きされた場合、AD 変換結果格納フラグ ***ADREGx*** に **ADC_OVERRUN** がセットされます。本関数によってオーバーランフラグが読み出されるとオーバーランフラグがクリアされます。

アナログチャネル入力と AD 変換結果レジスタの関係を下表に示します。

表: アナログ入力チャネルと AD 変換結果レジスタの関係

アナログ入力チャネル (ポート A)	A/D 変換結果レジスタ			
	右記以外の変換モード	チャネル固定リピート変換モード (毎回変換)	チャネル固定リピート変換モード(4 回ごと変換)	チャネル固定リピート変換モード(8 回ごと変換)
ADC_AN_0	ADC_REG_0	ADC_REG_0 fixed	ADC_REG_0 -> ADC_REG_3	ADC_REG_0 --> ADC_REG_7
ADC_AN_1	ADC_REG_1			
ADC_AN_2	ADC_REG_2			
ADC_AN_3	ADC_REG_3			
ADC_AN_4	ADC_REG_4			
ADC_AN_5	ADC_REG_5			
ADC_AN_6	ADC_REG_6			
ADC_AN_7	ADC_REG_7			

The AD 変換モードの詳細は、関連 API を参照ください。
最優先 AD 変換結果は、ADC_REG_SP に格納されます。

戻り値:

AD 変換結果構造:

1. AD 変換結果の格納状態:
   - **DONE**: AD 変換終了、AD 変換結果格納完了
   - **BUSY**: 変換中
2. 通常 AD 変換完了状態:
   - **ADC_NO_OVERRUN**: オーバーラン発生なし
   - **ADC_OVERRUN**: オーバーラン発生あり
3. AD 変換結果

3.2.3.20 ADC_SetCmpValue

変換結果レジスタの値との比較値の設定

関数のプロトタイプ宣言:

```
void
ADC_SetCmpValue(uint8_t ADCMPx,
                uint16_t value)
```

引数:

ADCMPx: AD 監視機能の比較レジスタを選択します。

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

value: 変換結果レジスタの値と比較する値を設定します。

機能:

AD 監視チャネルは、チャネル 0 とチャネル 1 の 2 種類です。
ADCMPx で AD 監視チャネルを設定し、***value*** で変換結果レジスタの値と比較する値を設定します。

補足:

AD 監視機能設定手順:
1. **ADC_SetResultCmpReg(ADCMPx, ResultComparison)**
2. **ADC_SetCmpValue(ADCMPx, value)**
3. **ADC_SetMonitorINT(ADCMPx, ResultComparison)**
4. **ADC_SetMonitor(ADCMPx, ENABLE)**

AD 変換終了後、**ADC_SetMonitorINT()**の設定状態と一致すると AD 監視機能割り込みが発生します。

戻り値:
なし

3.2.3.21 ADC_SetDMAReq

通常 AD 変換、最優先 AD 変換の各 DMA 起動要因の設定

関数のプロトタイプ宣言:
void
ADC_SetDMAReq (uint8_t ***DMAReq***,
FunctionalState ***NewState***)

引数:
DMAReq: 以下から AD 変換の種類を選択します。
- **ADC_DMA_REQ_NORMAL**: 通常 AD 変換
- **ADC_DMA_REQ_TOP**: 最優先 AD 変換
- **ADC_DMA_REQ_MONITOR1:** AD 変換監視 0 DMA 起動要因設定 (INTADM0 をトリガに DMAC を起動)
- **ADC_DMA_REQ_MONITOR2:** AD 変換監視 1 DMA 起動要因設定 (INTADM1 をトリガに DMAC を起動)

NewState: 以下から DMA 起動の許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
通常 AD 変換、最優先 AD 変換の各 DMA 起動要因を選択します。

戻り値:
なし

3.2.4 データ構造

3.2.4.1 ADC_ResultTypeDef

メンバ:
WorkState
ADCResultStored: AD 変換結果格納フラグ
- **BUSY**: 変換結果なし
- **DONE**: 変換結果あり

ADC_OverrunState
ADCOverrunState: オーバーランフラグ
- **ADC_NO_OVERRUN**: 発生なし

- **ADC_OVERRUN**: 発生あり

uint16_t
ADCResultValue: AD 変換結果値

3.2.4.2 ADC_State

メンバ:
uint32_t
All : すべての AD 変換状態

ビットフィールド:

uint32_t ***NormalBusy***(Bit 0)	通常 AD 変換中フラグ
uint32_t ***NormalComplete*** (Bit 1)	通常 AD 変換終了フラグ
uint32_t ***TopBusy***(Bit 2)	最優先 AD 変換中フラグ
uint32_t ***TopComplete*** (Bit 3)	最優先 AD 変換終了フラグ
uint32_t ***Reserved*** (Bit 4 ~Bit 31)	未使用

4. CG

4.1 概要

本 CG API は以下の機能を提供します。
- 高速/低速発振器、PLL(逓倍回路)の設定
- クロックギア、プリスケーラクロック、PLL、発振器の設定
- ウォームアップタイマの設定と結果の読み出し
- 低消費電力モードの設定
- 動作モードの変更 (ノーマルモード、低速モード、低消費電力モード)
- スタンバイモードに関する割り込みの設定

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\tmpm037_cg.c
/Libraries/TX00_Periph_Driver\inc\tmpm037_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。

fosc : X1、X2端子からの入力クロック
fpll: PLL により逓倍されたクロック(2 逓倍)
fc : CGPLLSEL<PLLSEL> により選択されたクロック (高速クロック)
fgear : CGSYSCR<GEAR[2:0]>により選択されたクロック
fsys : CGSYSCR<GEAR[2:0]>により選択されたクロック (システムクロック)
fperiph : CGSYSCR<FPSEL[2:0]>により選択されたクロック
ΦT0 : CGSYSCR<PRCK[2:0]>により選択されたクロック (プリスケーラクロック)

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_SetWarmUpTime(CG_WarmUpSrc ***Source***, uint16_t ***Time***)
- void CG_StartWarmUp(void)
- WorkState CG_GetWarmUpState(void)
- Result CG_SetFPLLValue(CG_FpllValue ***NewValue***)
- CG_FpllValue CG_GetFPLLValue(void)
- Result CG_SetPLL(FunctionalState ***NewState***)
- FunctionalState CG_GetPLLState(void)
- Result CG_SetFosc(CG_FoscSrc Source, FunctionalState ***NewState***)
- void CG_SetFoscSrc(CG_FoscSrc ***Source***)
- CG_FoscSrc CG_GetFoscSrc(void)

- FunctionalState CG_GetFoscState(CG_FoscSrc ***Source***)
- void CG_SetSTBYMode(CG_STBYMode ***Mode***)
- CG_STBYMode CG_GetSTBYMode(void)
- Result CG_SetFcSrc(CG_FcSrc ***Source***)
- CG_FcSrc CG_GetFcSrc(void)
- void CG_SetProtectCtrl(FunctionalState ***NewState***)
- void CG_SetSTBYReleaseINTSrc(CG_INTSrc ***INTSource***,
  CG_INTActiveState ***ActiveState***,
  FunctionalState ***NewState***)
- CG_INTActiveState CG_GetSTBYReleaseINTState(CG_INTSrc ***INTSource***)
- void CG_ClearINTReq(CG_INTSrc ***INTSource***)
- CG_NMIFactor CG_GetNMIFlag(void)
- CG_ResetFlag CG_GetResetFlag(void)
- void CG_SetADCClkSupply(FunctionalState NewState)

4.2.2 関数の種類

上記関数は以下の 4 種類に分けられます。

1) クロックの選択:
   CG_SetFgearLevel(), CG_GetFgearLevel(), CG_SetPhiT0Src(), CG_GetPhiT0Src(), CG_SetPhiT0Level(), CG_GetPhiT0Level(), CG_SetWarmUpTime(), CG_StartWarmUp(), CG_GetWarmUpState(), CG_SetFPLLValue(), CG_GetFPLLValue(), CG_SetPLL(), CG_GetPLLState(), CG_SetFosc(), CG_SetFoscSrc(), CG_GetFoscSrc(), CG_GetFoscState(), CG_SetFcSrc(), CG_GetFcSrc(), CG_SetProtectCtrl()
2) スタンバイモードの設定:
   CG_SetSTBYMode(), CG_GetSTBYMode( )
3) 割り込みの設定:
   CG_SetSTBYReleaseINTSrc(), CG_GetSTBYReleaseINTState(), CG_ClearINTReq(), CG_GetNMIFlag(), CG_GetResetFlag()
4) 周辺回路へのクロック供給:
   CG_SetADCClkSupply()

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**: ΦT0 ソースは fgear
- **CG_PHIT0_SRC_FC**: ΦT0 ソースは fc

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**: ΦT0 ソースは fgear
- **CG_PHIT0_SRC_FC**: ΦT0 ソースは fc

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:
プリスケーラクロックの分周レベルを設定します。

戻り値:
- **SUCCESS**: 成功
- **ERROR**: 失敗

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:
- **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_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_OSC_INT_HIGH**: 内部高速発振器を選択
- **CG_WARM_UP_SRC_OSC_EXT_HIGH**: 外部高速発振器を選択

Time: ウォーミングアップカウンタ値を設定します。設定可能な値は 0x0000～0xFFFFU です。

機能:
ウォームアップサイクル数の計算式は下記になります。
ウォーミングアップサイクル数 = (ウォーミングアップ時間) / (ウォーミングアップクロック周期)

高速発振子 10MHz 使用時、ウォーミングアップ時間 5ms を設定する場合の計算例:
(ウォーミングアップ時間)/(ウォーミングアップクロック) = 5ms/(1/10MHz) = 5000cycle = 0xC350
下位 4 ビット

戻り値:
なし

4.2.3.8 CG_StartWarmUp

ウォーミングアップ開始

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

引数:
なし

機能:
ウォーミングアップを開始します。

戻り値:
なし

4.2.3.9 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。

```
Example of using warm-up timer:
CG_SetWarmUpTime(CG_WARM_UP_SRC_OSC_EXT, 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

4.2.3.10 CG_SetFPLLValue

PLL (fsys 用)の逓倍値を設定

関数のプロトタイプ宣言:
Result
CG_SetFPLLValue(uint32_t ***NewValue***)

引数:
NewValue:

- **CG_MUL_2_FPLL:** 2 逓倍

機能:
PLL (fsys 用)の逓倍値を設定します。

戻り値:
- **SUCCESS**: 成功
- **ERROR**: 失敗

補足:
内部高速発振器(IHOSC) をシステムクロックとして使用する場合、PLL 逓倍の使用は禁止です。

4.2.3.11 CG_GetFPLLValue

PLL 逓倍値の取得

関数のプロトタイプ宣言:
uint32_t
CG_GetFPLLValue(void)

引数:
なし

機能:
PLL 逓倍値を取得します。
2 逓倍値以外は使用できません。

戻り値:
PLL 逓倍値:
- **CG_MUL_2_FPLL:** 2 逓倍値

4.2.3.12 CG_SetPLL

PLL 回路の設定

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

引数:
NewState:
- **ENABLE**: PLL 回路を使用する
- **DISABLE**: PLL 回路を使用しない

機能:
PLL 回路の有効/無効を設定します。

戻り値:
- **SUCCESS**: 成功
- **ERROR**: 失敗

4.2.3.13 CG_GetPLLState

PLL 回路の状態の取得

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

引数:
なし。

機能:
PLL 回路の状態を取得します。

戻り値:
PLL 回路の状態
- **ENABLE:** PLL 有効
- **DISABLE:** PLL 無効

4.2.3.14 CG_SetFosc

高速発振器(fosc)の有効/無効設定

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、fosc のソースクロックを選択します。
- **CG_FOSC_OSC_EXT**: 外部高速発信
- **CG_FOSC_OSC_INT**: 内部高速発信

NewState: 以下から、高速発振器の有効/無効を設定します。
- **ENABLE**: 有効
- **DISABLE**: 無効

機能:
高速発信器の有効/無効を設定します。

戻り値:
- **SUCCESS**: 成功
- **ERROR**: 失敗

4.2.3.15 CG_SetFoscSrc

高速発振器(fosc)のソース設定

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

引数:
Source: fosc のソースを選択します。
- **CG_FOSC_OSC_EXT**: 外部高速発信子
- **CG_FOSC_CLKIN_EXT**: 外部クロック入力
- **CG_FOSC_OSC_INT**: 内部高速発信器

機能:
高速発振器(fosc)のソースを設定します。

戻り値:
なし

4.2.3.16 CG_GetFoscSrc

高速発振器のソース取得

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を取得します。

戻り値:
高速発振器のソース
- **CG_FOSC_OSC_EXT**: 外部高速発信子
- **CG_FOSC_CLKIN_EXT**: 外部クロック入力
- **CG_FOSC_OSC_INT**: 内部高速発信器

4.2.3.17 CG_GetFoscState

高速発信器の状態

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

引数:
Source: 以下から、fosc のソースを指定します。
- **CG_FOSC_OSC_EXT**: 外部高速発信
- **CG_FOSC_OSC_INT**: 内部高速発信

機能:
高速発信器の状態を取得します。

戻り値:
fosc の状態
- **ENABLE**: fosc が有効

- **DISABLE**: fosc が無効

4.2.3.18 CG_SetSTBYMode

スタンバイモードの選択

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

引数:
Mode: 以下から、スタンバイモードを選択します。
- **CG_STBY_MODE_STOP1**: STOP1 モード (内部発振器も含めてすべての内部回路が停止)
- **CG_STBY_MODE_IDLE**: IDLE モード(CPU が停止)

機能:
スタンバイモードを選択します。

戻り値:
なし

4.2.3.19 CG_GetSTBYMode

スタンバイモードの取得

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

引数:
なし

機能:
スタンバイモードの設定状態を取得します。
“Reserved”の場合、“**CG_STBY_MODE_UNKNOWN**” を返却します。

戻り値:
- **CG_STBY_MODE_STOP1**: STOP1 モード
- **CG_STBY_MODE_IDLE**: IDLE モード
- **CG_STBY_MODE_UNKNOWN**: 無効なモード

4.2.3.20 CG_SetFcSrc

fc のソース選択

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

引数:

Source: fc のソースを選択します。
- **CG_FC_SRC_FOSC** : fosc を使用
- **CG_FC_SRC_FPLL**: fpll を使用

機能:
fc のソースクロックを選択します。

戻り値:
- **SUCCESS**: 成功
- **ERROR**: 失敗

4.2.3.21 CG_GetFcSrc

fc ソースの取得

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

引数:
なし

機能:
fc ソースを取得します。

戻り値:
fc のソース:
- **CG_FC_SRC_FOSC** : fosc
- **CG_FC_SRC_ FPLL**: fpll

4.2.3.22 CG_SetProtectCtrl

CG レジスタの書き込み制御

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

引数:
NewState
- **DISABLE:** 書き込み禁止
- **ENABLE:** 書き込み許可

機能:
CG レジスタの書き込み許可/禁止を設定します。

戻り値:
なし

4.2.3.23 CG_SetSTBYReleaseINTSrc

スタンバイモードの解除割り込みソースの設定

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: 以下から、スタンバイモードの解除割り込みソースを選択します。
- **CG_INT_SRC_0** : INT0
- **CG_INT_SRC_1** : INT1
- **CG_INT_SRC_2** : INT2
- **CG_INT_SRC_3** : INT3
- **CG_INT_SRC_4** : INT4
- **CG_INT_SRC_5** : INT5

ActiveState: 以下から、解除トリガのアクティブ状態を選択します。
- **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: 以下から、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ソースを設定します。

戻り値:
なし

4.2.3.24 CG_GetSTBYReleaseINTState

スタンバイモードの解除割り込みソースのアクティブ状態の取得

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

引数:
*INTSource***: 以下から、解除割り込みソースを選択します。**
- **CG_INT_SRC_0**: INT0
- **CG_INT_SRC_1**: INT1
- **CG_INT_SRC_2**: INT2
- **CG_INT_SRC_3**: INT3
- **CG_INT_SRC_4**: INT4
- **CG_INT_SRC_5**: INT5

機能:
スタンバイモードの解除割り込みソース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_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
- **CG_INT_ACTIVE_STATE_INVALID**: 無効な値

4.2.3.25 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: 以下から、解除割り込みソースを選択します。
- **CG_INT_SRC_0**: INT0
- **CG_INT_SRC_1**: INT1
- **CG_INT_SRC_2**: INT2
- **CG_INT_SRC_3**: INT3
- **CG_INT_SRC_4**: INT4
- **CG_INT_SRC_5**: INT5

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

4.2.3.26 CG_GetNMIFlag

NMI 起動要因フラグの取得

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

引数:
なし

機能:
NMI 起動要因フラグを取得します。

戻り値:
NMI 起動要因:

- **WDT** (Bit 0): WDT による NMI 発生
- **DetectLowVoltage1** (Bit 2): LVD で電源電圧が設定電圧より下がったことによる NMI 発生
- **DetectLowVoltage2** (Bit 3): LVD で電源電圧が設定電圧より上がったことによる NMI 発生

4.2.3.27 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:
- **PowerOnReset** (Bit0) パワーオンによるリセット
- **PinReset** (Bit1) リセット端子によるリセット
- **WDTReset** (Bit 2) WDT リセット
- **Reserved1** (Bit3): 未使用
- **DebugReset** (Bit 4) SYSRESETREQ リセット
- **Reserved2** (Bit5): 未使用
- **LVDReset** (Bit6): LVD リセット
- **Reserved3** (Bit7~Bit31): 未使用

4.2.3.28 CG_SetADCClkSupply

ADC 用クロック供給の許可/禁止

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

引数:
NewState: 以下から ADC 用クロック供給の許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
ADC 用クロック供給の許可/禁止を設定します。

戻り値:
なし

4.2.4 データ構造

4.2.4.1 CG_NMIFactor

メンバ:
uint32_t
All: CGNMI ソース起動状態を指定します。

ビットフィールド:

uint32_t
WDT(Bit 0): WDT による NMI 発生
uint32_t
Reserved1 (bit1): 未使用
uint32_t
DetectLowVoltage1 (Bit 2): LVD で電源電圧が低電圧を検知した場合に NMI が発生
uint32_t
DetectLowVoltage2 (Bit3): LVD で低電圧からの復帰検知の場合に NMI が発生

Reserved2 (Bit4~Bit31): 未使用

4.2.4.2 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
PowerOnReset (Bit0) パワーオンによるリセットフラグ
uint32_t
PinReset (Bit1) RESET 端子によるリセットフラグ
uint32_t
WDTReset (Bit2) WDT リセットによるリセットフラグ
uint32_t
Reververd1 (Bit3) 未使用
uint32_t
DebugReset (Bit4) SysresetReq によるリセットフラグ
uint32_t
Reververd2 (Bit5) 未使用
uint32_t
LVDReset (Bit6) LVD によるリセットによるリセットフラグ
uint32_t
Reserved3 (Bit7~bit31) 未使用

5. DMAC

5.1 概要

本デバイスは、DMA 要求選択レジスタにより制御される 1 ユニットの DMA コントローラを内蔵しています。このユニットは、 4 つの転送タイプのどれかで動作します。4 つの転送タイプは、メモリ-メモリ、メモリ-周辺回路、周辺回路–メモリ、周辺回路-周辺回路です。各ユニットは 2 チャネルの DMAC を内蔵し、DMA チャネル 0 は DMA チャネル 1 より優先度が高くなります。

DMA ドライバ API は DMAC 設定機能を持ち、引数には、ソースアドレス、ソースアドレスのインクリメント状態、転送ソースのビット幅、転送ソースのバースト幅、宛先アドレス、宛先アドレスのインクリメント状態、転送先ビット幅、転送先バーストサイズ、転送サイズ、転送方向、データ転送ペリフェラル、転送割り込みステータスなどがあります。

全ドライバ API は、アプリ使用の 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\tmpm037_dmac.c
\Libraries/TX00_Periph_Driver\inc\tmpm037_dmac.h

5.2 API 関数

5.2.1 関数一覧

- void DMAC_Enable(void);
- void DMAC_Disable(void);
- DMAC_INTReq DMAC_GetINTReq(void);
- DMAC_TxINTReq DMAC_GetTxINTReq(DMAC_Channel ***Chx***);
- void DMAC_ClearTxINTReq(DMAC_Channel ***Chx***, DMAC_INTSrc ***INTSource***);
- DMAC_TxINTReq DMAC_GetRawTxINTReq(DMAC_Channel ***Chx***);
- WorkState DMAC_GetChannelTxState(DMAC_Channel ***Chx***);
- void DMAC_SetSWBurstReq(DMAC_ReqNum ***BurstReq***);
- DMAC_BurstReqState DMAC_GetSWBurstReqState(void);
- void DMAC_SetLinkedList(DMAC_Channel ***Chx***, uint32_t ***LinkedAddr***);
- WorkState DMAC_GetFIFOState(DMAC_Channel ***Chx***);
- void DMAC_SetDMAHalt(DMAC_Channel ***Chx***, FunctionalState ***NewState***);
- void DMAC_SetLockedTx(DMAC_Channel ***Chx***, FunctionalState ***NewState***)
- void DMAC_SetTxINTConfig(DMAC_Channel ***Chx***, DMAC_INTSrc ***INTSource***, FunctionalState ***NewState***);
- void DMAC_SetDMAChannel(DMAC_Channel ***Chx***, FunctionalState ***NewState***);
- void DMAC_Init(DMAC_Channel ***Chx***, DMAC_InitTypeDef * ***InitStruct***);

5.2.2 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) DMAC 基本設定:
   DMAC_Enable(), DMAC_Disable(), DMAC_SetDMAChannel(), DMAC_Init()
2) DMA 転送割り込みステータス、FIFO または DMA チャネル状態:
   DMAC_GetINTReq(), DMAC_GetTxINTReq(), DMAC_GetRawTxINTReq(), DMAC_GetChannelTxState(), DMAC_GetFIFOState().

3) DMA 割り込み設定、DMA 割り込み要求のクリア:
DMAC_ClearTxINTReq(), DMAC_SetTxINTConfig()
4) DMA ソフトウェア要求の設定、および取得:
DMAC_SetSWBurstReq(), DMAC_GetSWBurstReqState(), DMAC_SetLinkedList()
5) その他の設定:
DMAC_SetDMAHalt(), DMAC_SetLockedTx()

5.2.3 関数仕様

5.2.3.1 DMAC_Enable

DMA 回路動作の許可

関数のプロトタイプ宣言:
void
DMAC_Enable(void);

機能:
DMA 回路動作を許可します。

補足:
DMAC を使用する際、まず本関数をコールして DMA 回路を動作させてください。DMA 回路用レジスタは、DMA 回路が動作していないと書き込み/読み出しができません。

戻り値:
なし

5.2.3.2 DMAC_Disable

DMA 回路動作の禁止

関数のプロトタイプ宣言:
void
DMAC_Disable(void);

機能:
DMA 回路動作を禁止します。

戻り値:
なし

5.2.3.3 DMAC_GetINTReq

DMA チャネル割り込みステータスの取得

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

機能:
DMA チャネル割り込み要求状態を取得します。

戻り値:
割り込み要求状態を返します。構造体"DMAC_INTReq"の詳細はデータ構造を参照してください。

5.2.3.4 DMAC_GetTxINTReq

DMA チャネル転送割り込み要求状態の取得

関数のプロトタイプ宣言:
DMAC_TxINTReq
DMAC_GetTxINTReq(DMAC_Channel ***Chx***);

引数:
Chx: 以下から DMA チャネルを選択します。
- **DMAC_CHANNEL_0**: チャネル 0
- **DMAC_CHANNEL_1**: チャネル 1

機能:
DMA チャネル転送割り込み要求状態を取得します。

戻り値:
以下のいずれかの DMA チャネル転送割り込み要求状態を返します。
DMAC_TX_NO_REQ:転送割り込み要求なし
DMAC_TX_END_REQ:転送終了割り込み要求あり
DMAC_TX_ERR_REQ:転送エラー割り込み要求あり
DMAC_TX_REQS:2 つ以上の割り込み要求あり

5.2.3.5 DMAC_ClearTxINTReq

転送割り込み要求のクリア

関数のプロトタイプ宣言:
void
DMAC_ClearTxINTReq(DMAC_Channel ***Chx***,
DMAC_INTSrc ***INTSource***);

引数:
Chx: 以下から DMA チャネルを選択します。
- **DMAC_CHANNEL_0**: チャネル 0
- **DMAC_CHANNEL_1**: チャネル 1

INTSource: 以下からリリース割り込みソースを選択します。
- **DMAC_INT_TX_END**:DMA 転送終了割り込み
- **DMAC_INT_TX_ERR**:DMA 転送エラー割り込み

機能:
転送割り込み要求をクリアします。

戻り値:
なし

5.2.3.6 DMAC_GetRawTxINTReq

DMA チャネルの許可前転送終了割り込み発生状態の取得

関数のプロトタイプ宣言:
DMAC_TxINTReq
DMAC_GetRawTxINTReq(DMAC_Channel ***Chx***);

引数:
Chx: 以下から DMA チャネルを選択します。
- **DMAC_CHANNEL_0**: チャネル 0
- **DMAC_CHANNEL_1**: チャネル 1

機能:
DMA チャネルの許可前転送終了割り込み発生状態を取得します。

戻り値:
以下のいずれかの DMA チャネルの許可前転送終了割り込み発生状態を返します。
DMAC_TX_NO_REQ:転送前の転送終了割り込み発生なし
DMAC_TX_END_REQ:転送終了割り込みあり
DMAC_TX_ERR_REQ:転送エラー割り込みあり
DMAC_TX_REQS :2 つ以上の割り込み要求あり

5.2.3.7 DMAC_GetChannelTxState

DMA チャネル転送状態の取得

関数のプロトタイプ宣言:
WorkState
DMAC_GetChannelTxState(DMAC_Channel ***Chx***);

引数:
Chx: 以下から DMA チャネルを選択します。
- **DMAC_CHANNEL_0**: チャネル 0
- **DMAC_CHANNEL_1**: チャネル 1

機能:
本関数は、***Chx*** が **DMAC_CHANNEL_0** の時、DMA チャネル 0 転送状態を取得します。***Chx*** が **DMAC_CHANNEL_1** の時、DMA チャネル 1 転送状態を取得します。戻り値が **BUSY** の時は、DMA チャネルは有効で、データ送信中であることを示します。戻り値が **DONE** の時は、DMA チャネルは無効で、データ送信は終了していることを示します。

戻り値:
以下どちらかの DMA 転送状態を返します。
BUSY、または DONE

5.2.3.8 DMAC_SetSWBurstReq

ソフトウェアによる DMA バースト転送要求の設定

関数のプロトタイプ宣言:
void

DMAC_SetSWBurstReq(DMACA_ReqNum ***BurstReq***);

引数:
BurstReq: 以下のいずれかのバースト要求番号を選択します。
- **DMAC_SIO0_UART0_RX:** SIO0/UART0 受信
- **DMAC_SIO0_UART0_TX**: SIO0/UART0 送信
- **DMAC_SIO1_UART1_RX**: SIO1/UART1 受信
- **DMAC_SIO1_UART1_TX**: SIO1/UART1 送信
- **DMAC_SIO2_UART2_RX**: SIO2/UART2 受信
- **DMAC_SIO2_UART2_TX**: SIO2/UART2 送信
- **DMAC_SIO3_UART3_RX** : SIO3/UART3 受信
- **DMAC_SIO3_UART3_TX**: SIO3/UART3 送信
- **DMAC_I2C0_RX_TX**: I2C 送受信
- **DMAC_SIO4_UART4_RX:** SIO4/UART4 受信
- **DMAC_SIO4_UART4_TX**: SIO4/UART4 送信
- **DMAC_TMRB0_3**: TMRB(ch0-3)
- **DMAC_TMRB4_7**: TMRB(ch4-7)
- **DMAC_TOP_PRIORITY_ADC**: AD 最優先変換終了/AD 監視 0/AD 監視 1
- **DMAC_AD_CONVERT_COMPLETE:** AD 変換終了

機能:
ソフトウェアによるバースト転送要求を設定します。

戻り値:
なし

5.2.3.9 DMAC_GetSWBurstReqState

ソフトウェアによる DMA バースト要求状態の取得

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

機能:

戻り値:
DMA バースト要求状態を返します。構造体"DMAC_BurstReqState"の詳細はデータ構造を参照してください。

5.2.3.10 DMAC_SetLinkedList

DMA チャネル・コレクションアイテムレジスタの設定

関数のプロトタイプ宣言:
void
DMAC_SetLinkedList(DMAC_Channel ***Chx***,
uint32_t ***LinkedAddr***);

引数:

Chx: 以下から DMA チャネルを選択します。
- **DMAC_CHANNEL_0**: チャネル 0
- **DMAC_CHANNEL_1**: チャネル 1

LinkedAddr: 次の転送開始アドレスを指定します。0xFFFFFFF0 まで指定可能です。

機能:
DMA チャネル・コレクションレジスタを設定します。スキャッター・ギャザー機能が不要な場合は、***LinkedAddr*** を 0 に設定し本関数を呼び出します。

補足:
スキャッター・ギャザー機能を用いる場合、転送ソース、転送先データアドレスは、コレクション（LinkedList）を最初に作成する必要があります。
各設定は LLI (コレクション LinkedList) と呼ばれます。各 LLI はデータブロック転送を制御します。また、DMA が通常設定であることを示し、連続データの転送を制御します。
DMA 転送終了ごとに、DMA 動作を継続するために次の LLI 設定がロードされます。(デイジーチェーン)
コレクションと共に設定されるアイテムは、以下の4ワードで設定されます。
1) DMACCxSrcAddr
2) DMACCxDestAddr
3) DMACCxLLI
4) DMACCxControl

戻り値:
なし

5.2.3.11 DMAC_GetFIFOState

FIFO 状態の取得

関数のプロトタイプ宣言:
WorkState
DMAC_GetFIFOState(DMAC_Channel ***Chx***);

引数:
Chx: 以下から DMA チャネルを選択します。
- **DMAC_CHANNEL_0**: チャネル 0
- **DMAC_CHANNEL_1**: チャネル 1

機能:
FIFO 状態を取得します。
戻り値が **BUSY** の場合は FIFO にデータが存在することを示し、**DONE** の場合は FIFO にデータがないことを示します。

戻り値:
FIFO 状態:
BUSY、または **DONE**

5.2.3.12 DMAC_SetDMAHalt

DMA 要求の設定

関数のプロトタイプ宣言:
void
DMAC_SetDMAHalt(DMAC_Channel ***Chx***,
FunctionalState ***NewState***);

引数:
Chx: 以下から DMA チャネルを選択します。
- **DMAC_CHANNEL_0**: チャネル 0
- **DMAC_CHANNEL_1**: チャネル 1

NewState: 以下から、DMA 要求受付制御を選択します。
- **ENABLE**: DMA 要求 受付
- **DISABLE**: DMA 要求 無視

機能:
DMA 要求受付制御を設定します。

戻り値:
なし

5.2.3.13 DMAC_SetLockedTx

ロック転送の設定

関数のプロトタイプ宣言:
void
DMAC_SetLockedTx(DMAC_Channel ***Chx***,
FunctionalState ***NewState***);

引数:
Chx: 以下から DMA チャネルを選択します。
- **DMAC_CHANNEL_0**: チャネル 0
- **DMAC_CHANNEL_1**: チャネル 1

NewState: 以下から、ロック転送設定を選択します。
- **ENABLE**: ロック転送 許可
- **DISABLE**: ロック転送 禁止

機能:
ロック転送を設定します。

戻り値:
なし

5.2.3.14 DMAC_SetTxINTConfig

転送割り込みの設定

関数のプロトタイプ宣言:
void

DMAC_SetTxINTConfig(DMAC_Channel ***Chx***,
DMAC_INTSrc ***INTSource***,
FunctionalState ***NewState***);

引数:
Chx: 以下から DMA チャネルを選択します。
- **DMAC_CHANNEL_0**: チャネル 0
- **DMAC_CHANNEL_1**: チャネル 1

INTSource: 以下から、割り込みソースを選択します。
- **DMAC_INT_TX_END**: 転送終了割り込み
- **DMAC_INT_TX_ERR**: エラー割り込み

NewState: 以下から、割り込み状態を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
転送割り込みを設定します。

戻り値:
なし

5.2.3.15 DMAC_SetDMAChannel

DMA チャネルの許可/禁止設定

関数のプロトタイプ宣言:
void
DMAC_SetDMAChannel(DMAC_Channel ***Chx***,
FunctionalState ***NewState***);

引数:
Chx: 以下から DMA チャネルを選択します。
- **DMAC_CHANNEL_0**: チャネル 0
- **DMAC_CHANNEL_1**: チャネル 1

NewState: 以下から、DMA チャネルの許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
DMA チャネルの許可/禁止を設定します。
DMA チャネルの初期設定を行った後に本関数をコールし、DMA チャネルを有効にしてください。本関数を使用し、DMA チャネルを無効にすると、FIFO 中のデータが失われます。FIFO 中のデータ喪失を防ぐため、**DMAC_SetDMAHalt()** をコールし、DMA 要求を無視した後、 **DMAC_GetFIFOState()** をコールし、FIFO のステイタスを取得してください。その後、本関数をコールし、DMA チャネルを無効にしてください。

戻り値:
なし

5.2.3.16 DMAC_Init

DMA チャネルの初期設定

関数のプロトタイプ宣言:

```
        void
DMAC_Init(DMAC_Channel Chx,
                  DMAC_InitTypeDef * InitStruct);
```

引数:

Chx: 以下から DMA チャネルを選択します。

- **DMAC_CHANNEL_0**: チャネル 0
- **DMAC_CHANNEL_1**: チャネル 1

InitStruct: 基本的な DMA 設定を含む構造体で、転送元アドレス、転送元アドレスインクリメントステート、転送元ビット幅、転送元バーストサイズ、転送先アドレス、転送先アドレスインクリメントステート、転送先ビット幅、転送先バーストサイズ、転送サイズ、転送方向、転送ペリフェラル、転送割り込み状態が含まれます。(詳細は“データ構造”を参照してください)

機能:

DMA チャネルの初期設定を行います。

補足:

DMAC_SetDMAChannel()をコールする前に、本関数を用いて初期設定を行ってください。

戻り値:

なし

5.2.4 データ構造

5.2.4.1 DMAC_InitTypeDef

メンバ:

uint32_t

***TxDirection*:** 以下から、転送方向を選択します。

- **DMAC_MEMORY_TO_MEMORY**: メモリ->メモリ
- **DMAC_MEMORY_TO_PERIPH**: メモリ->周辺回路
- **DMAC_PERIPH_TO_MEMORY**: 周辺回路->メモリ
- **DMAC_PERIPH_TO_ PERIPH**: 周辺回路->周辺回路

uint32_t

SrcAddr: 転送元アドレスを設定します。

uint32_t

DstAddr: 転送先アドレスを設定します。

FunctionalState

***SrcIncrementState*:** 以下から、転送元アドレスのインクリメント設定を選択します。

ENABLE、または **DISABLE**.

FunctionalState
***DstIncrementState*:** 以下から、転送先アドレスのインクリメント設定を選択します。
ENABLE、または **DISABLE**.

DMAC_BitWidth
***SrcBitWidth*:** 以下から、転送元データの幅を選択します。
- **DMAC_BYTE:** バイト
- **DMAC_HALF_WORD:** ハーフワード
- **DMAC_WORD:** ワード

DMAC_BurstSize
***SrcBurstSize*:** 以下から、転送元のバーストサイズを選択します。
- **DMAC_1_BEAT**: 1 ビート
- **DMAC_4_BEATS**: 4 ビート
- **DMAC_8_BEATS**: 8 ビート
- **DMAC_16_BEATS**: 16 ビート
- **DMAC_32_BEATS**: 32 ビート
- **DMAC_64_BEATS**: 64 ビート
- **DMAC_128_BEATS**: 128 ビート
- **DMAC_256_BEATS**: 256 ビート

DMAC_BurstSize
***DstBurstSize*:** 以下から、転送先のバーストサイズを選択します。
- **DMAC_1_BEAT** : 1 ビート
- **DMAC_4_BEATS** : 4 ビート
- **DMAC_8_BEATS** : 8 ビート
- **DMAC_16_BEATS** : 16 ビート
- **DMAC_32_BEATS** : 32 ビート
- **DMAC_64_BEATS** : 64 ビート
- **DMAC_128_BEATS** : 128 ビート
- **DMAC_256_BEATS** : 256 ビート

uint32_t
TxSize:最大転送数で、最大値は 0x0FFF です。

DMAC_ReqNum
TxDstPeriph:以下から、転送先の周辺回路を設定します。
- **DMAC_SIO0_UART0_RX:** SIO3/UART3 受信
- **DMAC_SIO0_UART0_TX**: SIO3/UART3 送信
- **DMAC_SIO1_UART1_RX**: SIO3/UART3 受信
- **DMAC_SIO1_UART1_TX**: SIO3/UART3 送信
- **DMAC_SIO2_UART2_RX**: SIO3/UART3 受信
- **DMAC_SIO2_UART2_TX**: SIO0/UART0 送信
- **DMAC_SIO3_UART3_RX:** SIO0/UART0 受信
- **DMAC_SIO3_UART3_TX**: SIO0/UART0 送信
- **DMAC_I2C0_RX_TX**: I2C 送受信
- **DMAC_SIO4_UART4_RX:** SIO4/UART4 受信
- **DMAC_SIO4_UART4_TX**: SIO4/UART4 送信
- **DMAC_TMRB0_3**: TMRB(ch0-3)
- **DMAC_TMRB4_7**: TMRB(ch4-7)
- **DMAC_TOP_PRIORITY_ADC**: AD 最優先変換終了/AD 監視 0/AD 監視 1

- **DMAC_AD_CONVERT_COMPLETE:** AD 変換終了

DMAC_ReqNum
TxSrcPeriph:以下から、転送元の周辺回路を設定します。
- **DMAC_SIO0_UART0_RX:** SIO3/UART3 受信
- **DMAC_SIO0_UART0_TX**: SIO3/UART3 送信
- **DMAC_SIO1_UART1_RX**: SIO3/UART3 受信
- **DMAC_SIO1_UART1_TX**: SIO3/UART3 送信
- **DMAC_SIO2_UART2_RX**: SIO3/UART3 受信
- **DMAC_SIO2_UART2_TX**: SIO0/UART0 送信
- **DMAC_SIO3_UART3_RX:** SIO0/UART0 受信
- **DMAC_SIO3_UART3_TX**: SIO0/UART0 送信
- **DMAC_I2C0_RX_TX**: I2C 送受信
- **DMAC_SIO4_UART4_RX:** SIO4/UART4 受信
- **DMAC_SIO4_UART4_TX**: SIO4/UART4 送信
- **DMAC_TMRB0_3**: TMRB(ch0-3)
- **DMAC_TMRB4_7**: TMRB(ch4-7)
- **DMAC_TOP_PRIORITY_ADC**: AD 最優先変換終了/AD 監視 0/AD 監視 1
- **DMAC_AD_CONVERT_COMPLETE:** AD 変換終了

FunctionalState
TxINT: 以下から、転送割り込みステートを選択します。
- **EANBLE**:転送割り込み許可
- **DISABLE**:転送割り込み無効

5.2.4.2 DMAC_INTReq

メンバ:
uint32_t
***All*:** すべての割り込み要求フラグ

ビットフィールド:
uint32_t
CH0_INTReq: 1 Ch0 の割り込み要求フラグ
uint32_t
CH1_INTReq: 1 Ch1 の割り込み要求フラグ

5.2.4.3 DMAC_ BurstReqState

メンバ:
uint32_t
***ALL*:** DMAC 要求ステータス
- **DMAC_SIO0_UART0_RX:** SIO3/UART3 受信
- **DMAC_SIO0_UART0_TX**: SIO3/UART3 送信
- **DMAC_SIO1_UART1_RX**: SIO3/UART3 受信
- **DMAC_SIO1_UART1_TX**: SIO3/UART3 送信
- **DMAC_SIO2_UART2_RX**: SIO3/UART3 受信
- **DMAC_SIO2_UART2_TX**: SIO0/UART0 送信
- **DMAC_SIO3_UART3_RX:** SIO0/UART0 受信
- **DMAC_SIO3_UART3_TX**: SIO0/UART0 送信

- **DMAC_I2C0_RX_TX**: I2C 送受信
- **DMAC_SIO4_UART4_RX:** SIO4/UART4 受信
- **DMAC_SIO4_UART4_TX**: SIO4/UART4 送信
- **DMAC_TMRB0_3**: TMRB(ch0-3)
- **DMAC_TMRB4_7**: TMRB(ch4-7)
- **DMAC_TOP_PRIORITY_ADC**: AD 最優先変換終了/AD 監視 0/AD 監視 1
- **DMAC_AD_CONVERT_COMPLETE:** AD 変換終了

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、256Kbyte 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\tmpm037_fc.c
\Libraries\TX00_Periph_Driver \inc\tmpm037_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:
   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および低消費電力モードの STOP2 解除で初期化されます。

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:
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:
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**: ブロック 0
- **FC_BLOCK_1**: ブロック 1
- **FC_BLOCK_2:** ブロック 2
- **FC_BLOCK_3**: ブロック 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:
FunctionalState
FC_ProgramBlockProtectState(uint8_t ***BlockNum***)

引数:
BlockNum: ブロック番号を選択します。
- **FC_BLOCK_0**: ブロック 0
- **FC_BLOCK_1**: ブロック 1
- **FC_BLOCK_2:** ブロック 2
- **FC_BLOCK_3**: ブロック 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:
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:
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 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:
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:
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:
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)

6.2.4 データ構造

なし

7. GPIO

7.1 概要

本デバイスの汎用 I/O ポートは、入出力はビット単位で指定でき、入出力ポート機能の他に、内蔵する周辺機能に対する入出力端子としても使用されます。

GPIO ドライバ API は各ポートの設定機能を持ち、入出力、プルアップ、プルダウン、オープンドレイン、CMOS などを設定します。

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/ tmpm037 _gpio.c
/Libraries/TX00_Periph_Driver/inc/tmpm037_gpio.h

7.2 API 関数

7.2.1 関数一覧

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

7.2.2 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) 入出力ポートへの書き込み/読み出し:
   GPIO_ReadData(), GPIO_ReadDataBit(), GPIO_WriteData(), GPIO_WriteDataBit()
2) 入出力ポートの初期化と設定:
   GPIO_SetOutput(), GPIO_SetInput(), GPIO_SetOutputEnableReg(),

GPIO_SetInputEnableReg(), GPIO_SetPullUp(), GPIO_SetPullDown(), GPIO_SetOpenDrain(), GPIO_Init()

3) その他:
GPIO_EnableFuncReg(), GPIO_DisableFuncReg()

7.2.3 関数仕様

7.2.3.1 GPIO_ReadData

DATA データレジスタの読み込み

関数のプロトタイプ宣言:
uint8_t
GPIO_ReadData(GPIO_Port ***GPIO_x***)

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G

機能:
DATA レジスタを読み込みます。

戻り値:
DATA レジスタの値

7.2.3.2 GPIO_ReadDataBit

ビット単位での DATA レジスタの読み込み

関数のプロトタイプ宣言:
uint8_t
GPIO_ReadDataBit(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***)

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G

Bit_x: GPIO 端子を選択します。
- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2

- **GPIO_BIT_3:** GPIO pin 3
- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7

機能:
ビット単位で DATA データレジスタを読み込みます。

戻り値:
GPIO 端子の値:
- **GPIO_BIT_VALUE_0**: 0
- **GPIO_BIT_VALUE_1**: 1

7.2.3.3 GPIO_WriteData

DATA レジスタへの書き込み

関数のプロトタイプ宣言:
void
GPIO_WriteData(GPIO_Port ***GPIO_x***,
uint8_t ***Data***)

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G

Data: DATA レジスタに書き込む値を設定します。

機能:
DATA レジスタへ指定された値を書き込みます。

戻り値:
なし

7.2.3.4 GPIO_WriteDataBit

ビット単位での DATA レジスタの書き込み

関数のプロトタイプ宣言:
void
GPIO_WriteDataBit(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***,
uint8_t ***BitValue***)

引数:
GPIO_x: GPIO ポートを選択します。

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

Bit_x: GPIO 端子を選択します。有効ビットの組み合わせが可能です。
- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2
- **GPIO_BIT_3:** GPIO pin 3
- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7
- **GPIO_BIT_ALL:** GPIO pin[0:7]

BitValue: GPIO 端子値
- **GPIO_BIT_VALUE_0**: 0
- **GPIO_BIT_VALUE_1**: 1

機能:
ビット単位で DATA データレジスタを書き込みます。

戻り値:
なし

7.2.3.5 GPIO_Init

GPIO ポートの初期設定

関数のプロトタイプ宣言:
void
GPIO_Init(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***,
GPIO_InitTypeDef * ***GPIO_InitStruct***)

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G

Bit_x: GPIO 端子を選択します。有効ビットの組み合わせが可能です。
- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2
- **GPIO_BIT_3:** GPIO pin 3

- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7
- **GPIO_BIT_ALL:** GPIO pin[0:7]

GPIO_InitStruct: GPIO 基本設定の構造体です。(詳細は"データ構造"を参照)

機能:
GPIO ポートを IO モード、プルアップ、プルダウン、オープンドレインポート、CMOS ポートなどの設定をおこないます。本 API は **GPIO_SetOutput(), GPIO_SetInput(), GPIO_SetPullUp(), GPIO_SetPullDown(), GPIO_SetOpenDrain()**を実行します。

戻り値:
なし

7.2.3.6 GPIO_SetOutput

出力ポートの設定

関数のプロトタイプ宣言:
void
GPIO_SetOutput(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***);

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G

Bit_x: GPIO 端子を選択します。有効ビットの組み合わせが可能です。
- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2
- **GPIO_BIT_3:** GPIO pin 3
- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7
- **GPIO_BIT_ALL:** GPIO pin[0:7]

機能:
出力ポートに設定します。

戻り値:
なし

7.2.3.7 GPIO_SetInput

入力ポートの設定

関数のプロトタイプ宣言:
void
GPIO_SetInput(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***)

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G

Bit_x: GPIO 端子を選択します。有効ビットの組み合わせが可能です。
- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2
- **GPIO_BIT_3:** GPIO pin 3
- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7
- **GPIO_BIT_ALL:** GPIO pin[0:7]

機能:
入力ポートに設定します。

戻り値:
なし

7.2.3.8 GPIO_SetOutputEnableReg

出力ポートの許可/禁止設定

関数のプロトタイプ宣言:
void
GPIO_SetOutputEnableReg(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***,
FunctionalState ***NewState***)

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F

- **GPIO_PG:** GPIO port G

Bit_x: GPIO 端子を選択します。有効ビットの組み合わせが可能です。
- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2
- **GPIO_BIT_3:** GPIO pin 3
- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7
- **GPIO_BIT_ALL:** GPIO pin[0:7]

NewState:
- **ENABLE :** 出力許可
- **DISABLE :** 出力禁止

機能:
GPIO 端子出力の許可/禁止を設定します。
NewState が **ENABLE** の時、出力許可。
NewState が **DISABLE** の時、出力禁止。

戻り値:
なし

7.2.3.9 GPIO_SetInputEnableReg

入力ポートの許可/禁止設定

関数のプロトタイプ宣言:
void
GPIO_SetInputEnableReg(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***,
FunctionalState ***NewState***)

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G

Bit_x: GPIO 端子を選択します。有効ビットの組み合わせが可能です。
- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2
- **GPIO_BIT_3:** GPIO pin 3
- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7

- **GPIO_BIT_ALL:** GPIO pin[0:7]

NewState:
- **ENABLE :** 入力許可
- **DISABLE :** 入力禁止

機能:
GPIO 端子入力の許可/禁止を設定します。
NewState が **ENABLE** の時、入力許可。
NewState が **DISABLE** の時、入力禁止。

戻り値:
なし

7.2.3.10 GPIO_SetPullUp

プルアップポートの設定

関数のプロトタイプ宣言:
void
GPIO_SetPullUp(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***,
FunctionalState ***NewState***)

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G

Bit_x: GPIO 端子を選択します。有効ビットの組み合わせが可能です。
- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2
- **GPIO_BIT_3:** GPIO pin 3
- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7
- **GPIO_BIT_ALL:** GPIO pin[0:7]

NewState:
- **ENABLE :** プルアップ許可
- **DISABLE :** プルアップ禁止

機能:
GPIO 端子プルアップの許可/禁止を設定します。
NewState が **ENABLE** の時、プルアップを許可し、***NewState*** が **DISABLE** の時、プルアップを禁止します。

戻り値:
なし

7.2.3.11 GPIO_SetPullDown

プルダウンポートの設定

関数のプロトタイプ宣言:
void
GPIO_SetPullDown(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***,
FunctionalState ***NewState***)

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G

Bit_x: GPIO 端子を選択します。有効ビットの組み合わせが可能です。
- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2
- **GPIO_BIT_3:** GPIO pin 3
- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7
- **GPIO_BIT_ALL:** GPIO pin[0:7]

NewState:
- **ENABLE :** プルダウン許可
- **DISABLE :** プルダウン禁止

機能:
GPIO 端子プルダウンの許可/禁止を設定します。
NewState が **ENABLE** の時、プルダウン許可。
NewState が **DISABLE** の時、プルダウン禁止。

戻り値:
なし

7.2.3.12 GPIO_SetOpenDrain

CMOS/オープンドレインポートの設定

関数のプロトタイプ宣言:
void

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

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G

Bit_x: GPIO 端子を選択します。有効ビットの組み合わせが可能です。
- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2
- **GPIO_BIT_3:** GPIO pin 3
- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7
- **GPIO_BIT_ALL:** GPIO pin[0:7]

NewState:
- **ENABLE :** オープンドレイン許可
- **DISABLE :** CMOS 許可

機能:
GPIO 端子 CMOS/オープンドレインの許可/禁止を設定します。
NewState が **ENABLE** の時、オープンドレイン許可。
NewState が **DISABLE** の時、CMOS 許可。

戻り値:
なし

7.2.3.13 GPIO_EnableFuncReg

機能ポートの有効設定

関数のプロトタイプ宣言:
void
GPIO_EnableFuncReg(GPIO_Port ***GPIO_x***,
uint8_t ***FuncReg_x***,
uint8_t ***Bit_x***)

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F

- **GPIO_PG:** GPIO port G

FuncReg_x: GPIO 機能レジスタの番号を選択します。
- **GPIO_FUNC_REG_1**: GPIO 機能レジスタ 1
- **GPIO_FUNC_REG_2**: GPIO 機能レジスタ 2

Bit_x: GPIO 端子を選択します。有効ビットの組み合わせが可能です。
- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2
- **GPIO_BIT_3:** GPIO pin 3
- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7

機能:
GPIO 端子の機能を有効に設定します。

戻り値:
なし

7.2.3.14 GPIO_DisableFuncReg

機能ポートの無効設定

関数のプロトタイプ宣言:
```
void
GPIO_DisableFuncReg(GPIO_Port GPIO_x,
                    uint8_t FuncReg_x,
                    uint8_t Bit_x)
```

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G

FuncReg_x: GPIO 機能レジスタの番号を選択します。
- **GPIO_FUNC_REG_1**: GPIO 機能レジスタ 1
- **GPIO_FUNC_REG_2**: GPIO 機能レジスタ 2

Bit_x: GPIO 端子を選択します。有効ビットの組み合わせが可能です。
- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2
- **GPIO_BIT_3:** GPIO pin 3
- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6

- **GPIO_BIT_7:** GPIO pin 7

機能:
GPIO 端子の機能を無効に設定します。

戻り値:
なし

7.2.4 データ構造

7.2.4.1 GPIO_InitTypeDef

メンバ:
uint8_t
IOMode　ポートの入出力設定
- **GPIO_INPUT:** 入力ポートに設定
- **GPIO_OUTPUT:** 出力ポートに設定
- **GPIO_IO_MODE_NONE:** 入出力モードを変更しない

uint8_t
PullUp　プルアップポートの許可/禁止設定
- **GPIO_PULLUP_ENABLE:** プルアップ許可
- **GPIO_PULLUP_DISABLE:** プルアップ禁止
- **GPIO_PULLUP_NONE:** プルアップ機能が無い、または設定変更しない

uint8_t
OpenDrain　オープンドレインポート/CMOS ポートの設定
- **GPIO_OPEN_DRAIN_ENABLE:** オープンドレインポートに設定
- **GPIO_OPEN_DRAIN_DISABLE:** CMOS ポートに設定
- **GPIO_OPEN_DRAIN_NONE:** オープンドレイン機能がない、または設定変更しない

uint8_t
PullDown　プルダウンポートの許可/禁止設定
- **GPIO_PULLDOWN_ENABLE:** プルダウン許可
- **GPIO_PULLDOWN_DISABLE:** プルダウン禁止
- **GPIO_PULLDOWN_NONE:** プルダウン機能がない、または設定変更しない

7.2.4.2 GPIO_RegTypeDef

メンバ:
uint8_t
PinDATA　DATA レジスタのマスク値

uint8_t
PinCR　CR レジスタのマスク値
- **"0"**: 出力ディゼーブル
- **"1"**: 出力イネーブル

uint8_t
PinFR[FRMAX]　FR レジスタのマスク値

uint8_t
PinOD　OD レジスタのマスク値
- **”0”**: CMOS
- **”1”**: オープンドレイン

uint8_t
PinPUP　PUP レジスタのマスク値
- **”0”:** Pull-up ディゼーブル
- **”1”:** Pull-up イネーブル

uint8_t
PinPDN　PDN レジスタのマスク値
- **”0”:** Pull-down ディゼーブル
- **”1”:** Pull-down イネーブル

uint8_t
PinPIE　IE レジスタのマスク値
- **”0”:** 入力ディゼーブル
- **”1”:** 入力イネーブル

7.2.4.3 TSB_Port_TypeDef

メンバ:
__IO uint32_t
DATA　DATA レジスタのリードデータまたはライトデータです。

__IO uint32_t
PinCR　CR レジスタのリードデータまたはライトデータです。

__IO uint32_t
PinFR[FRMAX]　“FR[FRMAX]” レジスタのリードデータまたはライトデータです。

uint32_t
RESERVED0[RESER]　未使用

__IO uint32_t
PinOD　OD レジスタのリードデータまたはライトデータです。

__IO uint32_t
PinPUP　PUP レジスタのリードデータまたはライトデータです。

__IO uint32_t
PinPDN　PDN レジスタのリードデータまたはライトデータです。

uint32_t
RESERVED1[RESER]　未使用

__IO uint32_t
PinPIE　IE レジスタのリードデータまたはライトデータです。

8. I2C

8.1 概要

本デバイスは I2C バスを 1 チャンネル 内蔵しています。
I2C バスは SDA と SCL を通して、外部デバイスがバスに接続されるバスで、複数のデバイスと通信が可能です。
また独自フォーマットのフリーデータフォーマットに対応しています。フリーデータフォーマットにおいて、データはマスタ側がデータ送信を行い、スレーブ側がデータ受信を行います。
I2C ドライバ API は、スレーブアドレスの設定、クロックの周波数選択、ACK のためのクロック発生、スタート/ストップ状態の発生、データの送受信、状態表示各種ステータスの取得を行う関数セットです。
本ドライバ 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/tmpm037_i2c.c
/Libraries/TX00_Periph_Driver/inc/tmpm037_i2c.h

8.2 API 関数

8.2.1 関数一覧

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

8.2.2 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) 共通設定:
   I2C_SetACK(), I2C_SetBitNum(), I2C_SetPrescalerClock(), I2C_Init()
2) 転送設定:
   I2C_ClearINTReq(), I2C_Generatestart(), I2C_Generatestop(), I2C_SetSendData(), I2C_GetReceiveData(), I2C_SetINTReq(), I2C_ClearINTOutput()
3) ステータスの取得:
   I2C_GetState(), I2C_GetSlaveAddrMatchState(), I2C_GetINTStatus()

4) その他:
I2C_SWReset(), I2C_ SetFreeDataMode()

8.2.3 関数仕様

8.2.3.1 I2C_SetACK

ACK クロックの発生/停止

関数のプロトタイプ宣言:
void
I2C_SetACK(FunctionalState ***NewState***)

引数:
NewState 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。
- **ENABLE**: ACK のためのクロックを発生する
- **DISABLE**: ACK のためのクロックを発生しない

機能:
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。

戻り値:
なし

8.2.3.2 I2C_Init

I2C の初期化

関数のプロトタイプ宣言:
void
I2C_Init(I2C_InitTypeDef* ***InitI2CStruct***)

引数:
InitI2CStruct: I2C 初期設定の構造体 (詳細は"データ構造"参照)

機能:
I2C の初期設定 (スレーブアドレスの設定、転送データ長の設定、クロックの周波数選択、ACK のためのクロック発生、動作モードの選択)を行う。

戻り値:
なし

8.2.3.3 I2C_SetBitNum

転送ビット数の設定

関数のプロトタイプ宣言:
void
I2C_SetBitNum(uint32_t ***I2CBitNum***)

引数:
I2CBitNum: 転送ビット数を選択します。

- **I2C_DATA_LEN_8**: データ長は 8
- **I2C_DATA_LEN_1**: データ長は 1
- **I2C_DATA_LEN_2**: データ長は 2
- **I2C_DATA_LEN_3**: データ長は 3
- **I2C_DATA_LEN_4**: データ長は 4
- **I2C_DATA_LEN_5**: データ長は 5
- **I2C_DATA_LEN_6**: データ長は 6
- **I2C_DATA_LEN_7**: データ長は 7

機能:
転送ビット数を選択します。

戻り値:
なし

8.2.3.4 I2C_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
I2C_SWReset(void)

引数:
なし

機能:
ソフトウェアリセットを発生します。ソフトウェアリセット後、すべてのコントロールレジスタとステータスフラグはリセット直後の値となります。

戻り値:
なし

8.2.3.5 I2C_ClearINTReq

INTI2C 割込み要求の解除

関数のプロトタイプ宣言:
void
I2C_ClearINTReq(void)

引数:
なし

機能:
INTI2C 割込み要求を解除します。

戻り値:
なし

8.2.3.6 I2C_GenerateStart

マスタモードの選択とスタートコンディションの発生

関数のプロトタイプ宣言:
void
I2C_GenerateStart(void)

引数:
なし

機能:
マスタモードを選択し、スタートコンディションを発生します。

戻り値:
なし

8.2.3.7 I2C_GenerateStop

マスタモードの選択とストップコンディションの発生

関数のプロトタイプ宣言:
void
I2C_GenerateStop(void)

引数:
なし

機能:
マスタモードを選択し、ストップコンディションを発生します。

戻り値:
なし

8.2.3.8 I2C_GetState

I2C バスステータスの取得

関数のプロトタイプ宣言:
I2C_State
I2C_GetState(void)

引数:
なし

機能:
I2C バスステータスを取得します。本 API は他のプロセスのステータスを間違って取得しないよう、I2C 割込みハンドラ内でコールしてください。

戻り値:
I2C バスステータス

8.2.3.9 I2C_SetSendData

送信データの設定と送信開始

関数のプロトタイプ宣言:
void
I2C_SetSendData(uint32_t ***Data***)

引数:
Data: 送信データを設定します。送信データの最大値は 0xFF です。

機能:
送信データの設定と送信を開始します。

戻り値:
なし

8.2.3.10 I2C_GetReceiveData

受信データの取得

関数のプロトタイプ宣言:
uint32_t
I2C_GetReceiveData(void)

引数:
なし

機能:
受信データを取得します。

戻り値:
受信データ

8.2.3.11 I2C_SetFreeDataMode

I2C フリーデータフォーマットの設定

関数のプロトタイプ宣言:
void
I2C_SetFreeDataMode(FunctionalState ***NewState***)

引数:
NewState: システムが IDLE モードの場合に以下の状態を選択します。
- **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 フリーデータフォーマットを設定します。フリーデータフォーマット時、マスタ時は送信に、スレーブ時は受信に転送方向が固定されます。フリーデータフォーマットを解除するには **I2C_Init()** をコールしてください。

戻り値:
なし

8.2.3.12 I2C_GetSlaveAddrMatchState

スレーブアドレス一致検出およびゼネラルコール検出選択状態の取得

関数のプロトタイプ宣言:
FunctionalState
I2C_ GetSlaveAddrMatchState(void)

引数:
なし

機能:
スレーブアドレス一致検出およびゼネラルコール検出選択状態を取得します。

戻り値:
スレーブアドレス一致検出およびゼネラルコール検出選択状態:
ENABLE:: スレーブ動作時、スレーブアドレス一致及びゼネラルコールを検出します。
DISABLE:: スレーブ動作時、スレーブアドレス一致及びゼネラルコールを検出しません。

8.2.3.13 I2C_SetPrescalerClock

内部 SCL 出力クロックの周波数選択

関数のプロトタイプ宣言:
void
I2C_SetPrescalerClock(uint32_t ***PrescalerClock***)

引数:
PrescalerClock: 内部 SCL 出力クロックの周波数を選択します。
- **I2C_PRESCALER_DIV_1 ～ I2C_PRESCALER_DIV_32**

機能:
内部 SCL 出力クロックの周波数を選択します。
設定範囲は動作周波数(fsys)により変わります。50ns＜プリスケーラクロック幅≦150ns の条件を満たすように、プリスケーラ設定の設定可能範囲を決定してください。詳細は TD の I2C 章「シリアルクロック」を参照してください。

戻り値:
なし

8.2.3.14 I2C_SetINTReq

I2C 割込み出力の許可/禁止の設定

関数のプロトタイプ宣言:
void
I2C _SetINTReq(FunctionalState ***NewState***)

引数:
NewState: I2C 割込み出力の許可/禁止を選択します。
- **ENABLE:** 許可
- **DISABLE:** 禁止

機能:
I2C 割込み出力の許可/禁止を選択します。

戻り値:
なし

8.2.3.15 I2C_GetINTStatus

I2C 割込み状態の取得

関数のプロトタイプ宣言:
FunctionalState
I2C_GetINTStatus(void)

引数:
なし

機能:
I2C 割込み状態を取得します。

戻り値:
I2C 割込み状態:
ENABLE: 割込み発生
DISABLE: 割込みなし

8.2.3.16 I2C_ClearINTOutput

I2C 割込みのクリア

関数のプロトタイプ宣言:
void
I2C_ClearINTOutput(void)

引数:
なし

機能:
I2C 割込み出力(INTI2C)をクリアします。

戻り値:
なし

8.2.4 データ構造

8.2.4.1 I2C_InitTypeDef

メンバ:
uint32_t
I2CSelfAddr スレーブアドレスを 0x1~0xFE までの間で設定します。

uint32_t
I2CDataLen 送信ビット数を選択します。
- **I2C_DATA_LEN_8**: データ長 8
- **I2C_DATA_LEN_1**: データ長 1
- **I2C_DATA_LEN_2**: データ長 2
- **I2C_DATA_LEN_3**: データ長 3
- **I2C_DATA_LEN_4**: データ長 4
- **I2C_DATA_LEN_5**: データ長 5
- **I2C_DATA_LEN_6**: データ長 6
- **I2C_DATA_LEN_7**: データ長 7

uint32_t
I2CClkDiv: プリスケーラクロックの分周値を選択します。
- **I2C_SCK_CLK_DIV_20**: シリアルクロックは fprsck を 20 で割った商の値です。
- **I2C_SCK_CLK_DIV_24**: シリアルクロックは fprsck を 24 で割った商の値です。
- **I2C_SCK_CLK_DIV_32**: シリアルクロックは fprsck を 32 で割った商の値です。
- **I2C_SCK_CLK_DIV_48**: シリアルクロックは fprsck を 48 で割った商の値です。
- **I2C_SCK_CLK_DIV_80**: シリアルクロックは fprsck を 80 で割った商の値です。
- **I2C_SCK_CLK_DIV_144**: シリアルクロックは fprsck を 144 で割った商の値です。
- **I2C_SCK_CLK_DIV_272**: シリアルクロックは fprsck を 272 で割った商の値です。
- **I2C_SCK_CLK_DIV_528**: シリアルクロックは fprsck を 528 で割った商の値です。

uint32_t
PrescalerClkDiv: fprsck を出力するシステムクロックの分周値です。
- **I2C_PRESCALER_DIV_1**: fprsck は、fsys を 1 で割った商の値です。
- **I2C_PRESCALER_DIV_2**: fprsck は、fsys を 2 で割った商の値です。
- **I2C_PRESCALER_DIV_3**: fprsck は、fsys を 3 で割った商の値です。
- **I2C_PRESCALER_DIV_4**: fprsck は、fsys を 4 で割った商の値です。
- **I2C_PRESCALER_DIV_5**: fprsck は、fsys を 5 で割った商の値です。
- **I2C_PRESCALER_DIV_6**: fprsck は、fsys を 6 で割った商の値です。
- **I2C_PRESCALER_DIV_7**: fprsck は、fsys を 7 で割った商の値です。
- **I2C_PRESCALER_DIV_8**: fprsck は、fsys を 8 で割った商の値です。
- **I2C_PRESCALER_DIV_9**: fprsck は、fsys を 9 で割った商の値です。
- **I2C_PRESCALER_DIV_10**: fprsck は、fsys を 10 で割った商の値です。
- **I2C_PRESCALER_DIV_11**: fprsck は、fsys を 11 で割った商の値です。
- **I2C_PRESCALER_DIV_12**: fprsck は、fsys を 12 で割った商の値です。
- **I2C_PRESCALER_DIV_13**: fprsck は、fsys を 13 で割った商の値です。
- **I2C_PRESCALER_DIV_14**: fprsck は、fsys を 14 で割った商の値です。
- **I2C_PRESCALER_DIV_15**: fprsck は、fsys を 15 で割った商の値です。
- **I2C_PRESCALER_DIV_16**: fprsck は、fsys を 16 で割った商の値です。

- **I2C_PRESCALER_DIV_17**: fprsck は、fsys を 17 で割った商の値です。
- **I2C_PRESCALER_DIV_18**: fprsck は、fsys を 18 で割った商の値です。
- **I2C_PRESCALER_DIV_19**: fprsck は、fsys を 19 で割った商の値です。
- **I2C_PRESCALER_DIV_20**: fprsck は、fsys を 20 で割った商の値です。
- **I2C_PRESCALER_DIV_21**: fprsck は、fsys を 21 で割った商の値です。
- **I2C_PRESCALER_DIV_22**: fprsck は、fsys を 22 で割った商の値です。
- **I2C_PRESCALER_DIV_23**: fprsck は、fsys を 23 で割った商の値です。
- **I2C_PRESCALER_DIV_24**: fprsck は、fsys を 24 で割った商の値です。
- **I2C_PRESCALER_DIV_25**: fprsck は、fsys を 25 で割った商の値です。
- **I2C_PRESCALER_DIV_26**: fprsck は、fsys を 26 で割った商の値です。
- **I2C_PRESCALER_DIV_27**: fprsck は、fsys を 27 で割った商の値です。
- **I2C_PRESCALER_DIV_28**: fprsck は、fsys を 28 で割った商の値です。
- **I2C_PRESCALER_DIV_29**: fprsck は、fsys を 29 で割った商の値です。
- **I2C_PRESCALER_DIV_30**: fprsck は、fsys を 30 で割った商の値です。
- **I2C_PRESCALER_DIV_31**: fprsck は、fsys を 31 で割った商の値です。
- **I2C_PRESCALER_DIV_32**: fprsck は、fsys を 32 で割った商の値です。

***補足:** 設定範囲は動作周波数(fsys)により変わります。50ns＜fprsck 幅≦150ns の条件を満たすように設定してください。

FunctionalState
I2CACKState: 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。

- **ENABLE:** ACK のためのクロックを発生する。
- **DISABLE:** ACK のためのクロックを発生しない。

8.2.4.2 I2C_State

メンバ:
uint32_t
All: すべての状態です。

ビットフィールド:

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検出モニタ

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

uint32_t
INTReq: INTI2C 割込み要求状態モニタ

uint32_t
BusState: I2C バス状態モニタ

uint32_t
TRx: トランスミッタ/レシーバ選択状態モニタ

uint32_t
MasterSlave: マスタ/スレーブ選択状態モニタ

9. LVD

9.1 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、電圧の低下/上昇を検出することにより、リセット信号または割り込みを発生させます。

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/tmpm037_lvd.c
/Libraries/TX00_Periph_Driver/inc/tmpm037_lvd.h

9.2 API 関数

9.2.1 関数一覧

- void LVD_EnableVD1(void)
- void LVD_DisableVD1(void)
- void LVD_SetVD1Level(uint32_t ***VDLevel***)
- LVD_VDStatus LVD_GetVD1Status(void)
- void LVD_SetVD1ResetOutput(FunctionalState ***NewState***)
- void LVD_SetVD1INTOutput(FunctionalState ***NewState***)
- uint32_t LVD_GetINTCondition(void)

9.2.2 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_EnableVD1(), LVD_DisableVD1(), LVD_SetVD1Level(),
   LVD_SetVD1ResetOutput(), LVD_SetVD1INTOutput()
2) 電圧検出状態の確認:
   LVD_GetVD1Status(), LVD_GetINTCondition()

9.2.3 関数仕様

9.2.3.1 LVD_EnableVD1

LVDLVL1 の許可

関数のプロトタイプ宣言:
void
LVD_EnableVD1(void)

引数:
なし。

機能:
LVDLVL1 を許可します。

戻り値:
なし

9.2.3.2 LVD_DisableVD1

LVDLVL1 の禁止

関数のプロトタイプ宣言:
void
LVD_DisableVD1(void)

引数:
なし。

機能:
LVDLVL1 を禁止します。

戻り値:
なし

9.2.3.3 LVD_SetVD1Level

LVDLVL1 用電圧レベルの選択

関数のプロトタイプ宣言:
void
LVD_SetVD1Level(uint32_t ***VDLevel***)

引数:
VDLevel: LVDLVL1 用の電圧レベルです。

- **LVD_VDLVL1_240:** 2.40 ± 0.1V
- **LVD_VDLVL1_250:** 2.50 ± 0.1V
- **LVD_VDLVL1_260:** 2.60 ± 0.1V
- **LVD_VDLVL1_270:** 2.70 ± 0.1V
- **LVD_VDLVL1_280:** 2.80 ± 0.1V
- **LVD_VDLVL1_290:** 2.90 ± 0.1V

機能:
LVDLVL1 用電圧レベルを設定します。

戻り値:
なし

9.2.3.4 LVD_GetVD1Status

LVDLVL1 状態の取得

関数のプロトタイプ宣言:
LVD_VDStatus
LVD_GetVD1Status(void)

引数:
なし。

機能:
LVDLVL1 のステータスを取得します。

戻り値:
LVD_VDStatus: LVDLVL1 のステータス。
- ***LVD_VD_UPPER***: 電源電圧は検出電圧以上。
- ***LVD_VD_LOWER***: 電源電圧は検出電圧以下。

9.2.3.5 LVD_SetVD1ResetOutput

LVD1 の RESET 信号の出力

関数のプロトタイプ宣言:
void
LVD_SetVD1ResetOutput(FunctionalState ***NewState***)

引数:
NewState: LVDRST 信号の出力状態を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
LVDRST 信号の出力状態を選択します。

戻り値:
なし

9.2.3.6 LVD_SetVD1INTOutput

LVD1 の INTLVD 信号の出力

関数のプロトタイプ宣言:
void
LVD_SetVD1INTOutput(FunctionalState ***NewState***)

引数:
NewState: INTLVD 信号の出力状態を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
INTLVD 信号の出力状態を選択します。

戻り値:

なし

9.2.3.7 LVD_GetINTCondition

割り込み発生条件の取得

関数のプロトタイプ宣言:
uint32_t
LVD_GetINTCondition(void)

引数:
なし

機能:
割り込み発生条件を取得します。

戻り値:
割り込み発生条件:
LVD_INTSEL_LOWER: 電源電圧が検出電圧を下回った場合に割り込み発生
LVD_INTSEL_LOWER_UPPER: 電源電圧は検出電圧を下回った場合または上回った場合に割り込み発生

9.2.4 データ構造

なし

10. TMR16A

10.1 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/tmpm037_tmr16a.c
/Libraries/TX00_Periph_Driver/inc/tmpm037_tmr16a.h

10.2 API 関数

10.2.1 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***);

10.2.2 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()

10.2.3 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

10.2.3.1 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

10.2.3.2 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

10.2.3.3 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

10.2.3.4 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

10.2.3.5 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

10.2.3.6 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

10.2.3.7 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

10.2.1 データ構造

10.2.1.1 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。

11. TMRB

11.1 概要

本デバイスは、10 チャンネルの多機能 16 ビットタイマ/ イベントカウンタ (TMRB0 ～ TMRB9)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。

- 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/tmpm037_tmrb.c
/Libraries/TX00_Periph_Driver/inc/tmpm037_tmrb.h

11.2 API 関数

11.2.1 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***);
- TMRB_INTMask TMRB_GetINTMask(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_GetRegisterValue(TSB_TB_TypeDef * ***TBx***, uint8_t ***Reg***);
- 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_SetDMAReq(TSB_TB_TypeDef * ***TBx***, FunctionalState ***NewState***, uint8_t ***DMAReq***);

11.2.2 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_GetRegisterValue(), TMRB_GetUpCntValue(), TMRB_GetCaptureValue()
4) その他:
   TMRB_SetFlipFlop(), TMRB_SetINTMask(), TMRB_SetIdleMode(), TMRB_SetSyncMode(), TMRB_SetDoubleBuf(), TMRB_SetExtStartTrg(), TMRB_SetClkInCoreHalt(), TMRB_SetExtInput(), TMRB_SetDMAReq()

11.2.3 関数仕様

補足: 引数に記述されている “TSB_TB_TypeDef* ***TBx***” は特に記載の無い限り以下から選択してください。

TSB_TB0, TSB_TB1, TSB_TB2, TSB_TB3,
TSB_TB4, TSB_TB5, TSB_TB6, TSB_TB7

11.2.3.1 TMRB_Enable

TMRB 機能の許可

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 機能を有効にします。

戻り値:

なし

11.2.3.2 TMRB_Disable

TMRB 機能の禁止

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 機能を無効にします。

戻り値:
なし

11.2.3.3 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

11.2.3.4 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

11.2.3.5 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。
TSB_TB0, TSB_TB1, TSB_TB3, TSB_TB7.

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_FALLING**: TBxIN0↑ TBxIN0↓
- **TMRB_CAPTURE_FF_RISING_FALLING**: TBxFF0↑ TBxFF0↓

機能:
CaptureTiming が **TMRB_CAPTURE_IN_RISING_FALLING** の場合、TBxIN0 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、TBxIN0 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_FF_RISING_FALLING** の場合、TBxFF0 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、TBxFF0 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。

TMRB6, TMRB7 のフリップフロップ出力を他のチャネルのキャプチャトリガとして使用できます。
TMRB0~2: TB6OUT
TMRB3~5: TB7OUT

戻り値:
なし

11.2.3.6 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

11.2.3.7 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
}
```

11.2.3.8 TMRB_GetINTMask

割り込みマスク要因の取得

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

引数:
***TBx*:** TMRB チャネルを指定します。

機能:
割り込みマスク要因を取得します。

戻り値:
マスクする割り込み要因です。
- **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

11.2.3.9 TMRB_SetINTMask

割り込みマスク要因の設定

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

11.2.3.10 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。

11.2.3.11 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:
TrailingTiming は ***LeadingTiming*** より小さくすることはできません。

11.2.3.12 TMRB_GetRegisterValue

レジスタ値の取得

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

引数:
***TBx*:** TMRB チャネルを指定します。

***Reg*:** 以下から取得するレジスタを選択してください。
- **TMRB_Reg_0:** TBxREG0
- **TMRB_Reg_1:** TBxREG1

機能:
レジスタ値を取得します。

戻り値:
レジスタ値

11.2.3.13 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

11.2.3.14 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。

TSB_TB0, TSB_TB1, TSB_TB3, TSB_TB7

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:
キャプチャされた値

11.2.3.15 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。
TSB_TB0, TSB_TB1, TSB_TB3, TSB_TB7.

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

11.2.3.16 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

11.2.3.17 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*:** 以下から TMRB チャネルをします。
TSB_TB1, TSB_TB2, TSB_TB3, TSB_TB5, TSB_TB6, TSB_TB7

***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:
TMRB1～TMRB3 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、TMRB5～TMRB7 を同期モードに設定すると、TMRB4 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, TMRB4 のカウントを開始する前に、**TMRB_SetRunState()** によって TMRB0~TMRB3、TMRB4~TMRB7 をスタートしてください。

11.2.3.18 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:** 許可。
- **DISABLE:** 禁止。

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

11.2.3.19 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。

補足:
NewState が **ENABLE** の場合のみ ***TrgMode*** を選択できます。

戻り値:
なし

11.2.3.20 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

11.2.3.21 TMRB_SetDMAReq

DMA 要求の制御

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

引数:
TBx: TMRB チャネルを選択します。

NewState: 以下から DMA 要求の許可/禁止を選択します。
- **ENABLE:** 許可
- **DISABLE:** 禁止

DMAReq: 以下から DMA 要求の種類を選択します。
- **TMRB_DMA_REQ_CMP_MATCH:**コンペアー致
- **TMRB_DMA_REQ_CAPTURE_1:** インプットキャプチャ 1
- **TMRB_DMA_REQ_CAPTURE_0:** インプットキャプチャ 0

機能:
DMA 要求の制御を行います。

戻り値:
なし

補足:
TBxIM レジスタで割り込みをマスク設定している場合、DMA 要求を許可しても要求は発生しません。

11.2.4 データ構造

11.2.4.1 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。

11.2.4.2 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。

11.2.4.3 TMRB_INTFactor

メンバ:
uint32_t
All: TMRB 割り込み要因

ビットフィールド:
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　　未使用

11.2.4.4 TMRB_INTMask

メンバ:
uint32_t
All: TMRB 割り込み要因マスク

ビットフィールド:
uint32_t
MatchLeadingTimingMask: 1　　デューティとの一致検出マスク

uint32_t
MatchTrailingTimingMask : 1　　周期との一致検出マスク

uint32_t
OverFlowMask : 1　　オーバーフローマスク

uint32_t
Reserverd : 29　　未使用

12. SIO/UART

12.1 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/tmpm037_uart.c
/Libraries/TX00_Periph_Driver/inc/tmpm037_uart.h

12.2 API 関数

12.2.1 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 UART_FIFOConfig(TSB_SC_TypeDef * ***UARTx***, FunctionalState NewState);
- void UART_SetFIFOTransferMode(TSB_SC_TypeDef * ***UARTx,***
  uint32_t ***TransferMode***);
- void UART_TRxAutoDisable(TSB_SC_TypeDef * ***UARTx***,
  UART_TRxAutoDisable ***TRxAutoDisable***);
- void UART_RxFIFOINTCtrl(TSB_SC_TypeDef * ***UARTx***, FunctionalState ***NewState***);
- void UART_TxFIFOINTCtrl(TSB_SC_TypeDef * ***UARTx***, FunctionalState ***NewState***);
- void UART_RxFIFOByteSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***BytesUsed***);
- void UART_RxFIFOFillLevel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***RxFIFOLevel***);
- void UART_RxFIFOINTSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***RxINTCondition***);
- void UART_RxFIFOClear(TSB_SC_TypeDef * ***UARTx***);
- void UART_TxFIFOFillLevel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***TxFIFOLevel***);
- void UART_TxFIFOINTSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***TxINTCondition***);
- void UART_TxFIFOClear(TSB_SC_TypeDef * ***UARTx***);
- void UART_TxBufferClear(TSB_SC_TypeDef * ***UARTx***);
- uint32_t UART_GetRxFIFOFillLevelStatus(TSB_SC_TypeDef * ***UARTx***);
- uint32_t UART_GetRxFIFOOverRunStatus(TSB_SC_TypeDef * ***UARTx***);

- uint32_t UART_GetTxFIFOFillLevelStatus(TSB_SC_TypeDef * ***UARTx***);
- uint32_t UART_GetTxFIFOUnderRunStatus(TSB_SC_TypeDef * ***UARTx***);
- void UART_SetRxDMAReq (TSB_SC_TypeDef* ***UARTx***, FunctionalState ***NewState***)
- void UART_SetTxDMAReq (TSB_SC_TypeDef* ***UARTx***, FunctionalState ***NewState***)
- void UART_SetInputClock(TSB_SC_TypeDef * ***UARTx***, uint32_t ***clock***)
- void SIO_SetInputClock(TSB_SC_TypeDef * ***SIOx***, uint32_t ***Clock***)
- void SIO_Enable(TSB_SC_TypeDef* ***SIOx***)
- void SIO_Disable(TSB_SC_TypeDef* ***SIOx***)
- void SIO_Init(TSB_SC_TypeDef* ***SIOx***, uint32_t ***IOClkSel***,
  UART_InitTypeDef* ***InitStruct***)
- uint8_t SIO_GetRxData(TSB_SC_TypeDef* ***SIOx***)
- void SIO_SetTxData(TSB_SC_TypeDef* ***SIOx***, uint8_t ***Data***)

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) 初期化と設定:
   UART_Enable(), UART_Disable(),UART_SetInputClock(),UART_Init() and
   UART_DefaultConfig(),SIO_Enable(), SIO_Disable(),SIO_SetInputClock(), 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) FIFO モードの設定:
   UART_FIFOConfig(), UART_SetFIFOTransferMode(), UART_TrxAutoDisable(),
   UART_RxFIFOINTCtrl(), UART_TxFIFOINTCtrl(), UART_RxFIFOByteSel(),
   UART_RxFIFOFillLevel(), UART_RxFIFOINTSel(), UART_RxFIFOClear(),
   UART_TxFIFOFillLevel(), UART_TxFIFOINTSel(), UART_TxFIFOClear(),
   UART_TxBufferClear (), UART_GetRxFIFOFillLevelStatus(),
   UART_GetRxFIFOOverRunStatus(), UART_GetTxFIFOFillLevelStatus(),
   UART_GetTxFIFOUnderRunStatus()
4) その他:
   UART_SetRxDMAReq , UART_SetTxDMAReq ,UART_SWReset(),
   UART_SetWakeUpFunc(), UART_SetIdleMode()

12.2.3 関数仕様

補足: 下記関数の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**, **UART4**.
また、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**, **SIO4**.

12.2.3.1 UART_Enable

UART 機能の許可

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 機能を有効にします。

戻り値:
なし

12.2.3.2 UART_Disable

UART 機能の禁止

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 機能を無効にします。

戻り値:
なし

12.2.3.3 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

12.2.3.4 UART_SWReset

ソフトウェアリセットの実行

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を実行します。

戻り値:
なし

12.2.3.5 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

12.2.3.6 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。

12.2.3.7 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

12.2.3.8 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

12.2.3.9 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

12.2.3.10 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

12.2.3.11 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:
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

12.2.3.12 UART_FIFOConfig

FIFO の許可

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

引数:
UARTx: UART チャネルを指定します。

NewState: FIFO の許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
FIFO の許可/禁止を選択します。
NewState が **ENABLE** の場合、FIFO を許可します。**DISABLE** の場合、FIFO を禁止します。

戻り値:
なし

12.2.3.13 UART_SetFIFOTransferMode

転送モードの選択

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

引数:
UARTx: UART チャネルを指定します。

TransferMode: 転送モードを選択します。
- **UART_TRANSFER_PROHIBIT** : 転送禁止
- **UART_TRANSFER_HALFDPX_RX** : 半二重(受信)
- **UART_TRANSFER_HALFDPX_TX** : 半二重(送信)
- **UART_TRANSFER_FULLDPX** : 全二重

機能:
転送モードを選択します。

戻り値:
なし

12.2.3.14 UART_TRxAutoDisable

送信/受信の自動禁止

関数のプロトタイプ宣言:
void
UART_TRxAutoDisable (TSB_SC_TypeDef * ***UARTx,***
UART_TRxDisable ***TRxAutoDisable***)

引数:
UARTx: UART チャネルを指定します。

TRxAutoDisable: 送信/受信の自動禁止機能を制御します。
- **UART_RXTXCNT_NONE**: なし
- **UART_RXTXCNT_AUTODISABLE**: 自動禁止

機能:
送信/受信の自動禁止機能を制御します。

戻り値:
なし

12.2.3.15 UART_RxFIFOINTCtrl

受信 FIFO 使用時の受信割り込み許可

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

引数:
UARTx: UART チャネルを指定します。

NewState: 受信 FIFO 使用時の受信割り込みの許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
受信 FIFO 有効にされている時の受信割り込みの許可/禁止を切り替えます。

戻り値:
なし

12.2.3.16 UART_TxFIFOINTCtrl

送信 FIFO 使用時の送信割り込み許可

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

引数:
UARTx: UART チャネルを指定します。

NewState: 送信 FIFO 使用時の送信割り込みの許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
送信 FIFO 有効にされている時の送信割り込みの許可/禁止を切り替えます。

戻り値:
なし

12.2.3.17 UART_RxFIFOByteSel

受信 FIFO 使用バイト数

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

引数:
UARTx: UART チャネルを指定します。

BytesUsed: 受信 FIFO 使用バイト数を設定します。
- **UART_RXFIFO_MAX**: 最大
- **UART_RXFIFO_RXFLEVEL**: 受信 FIFO の FILL レベルに同じ

機能:
受信 FIFO 使用バイト数を設定します。

戻り値:
なし

12.2.3.18 UART_RxFIFOFillLevel

受信割り込みが発生する受信 FIFO の fill レベルの設定

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

引数:
UARTx: UART チャネルを指定します。

RxFIFOLevel: 受信 FIFO の fill レベルを選択します。

RxFIFOLevel	半二重	全二重
UART_RXFIFO4B_FLEVLE_4_2B	4 バイト	2 バイト
UART_RXFIFO4B_FLEVLE_1_1B	1 バイト	1 バイト
UART_RXFIFO4B_FLEVLE_2_2B	2 バイト	2 バイト
UART_RXFIFO4B_FLEVLE_3_1B	3 バイト	1 バイト

機能:
受信割り込みが発生する受信 FIFO の fill レベルを選択します。

戻り値:
なし

12.2.3.19 UART_RxFIFOINTSel

受信割り込み発生条件の選択

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

引数:
UARTx: UART チャネルを指定します。

RxINTCondition: 受信 割り込み発生条件を選択します。

- **UART_RFIS_REACH_FLEVEL**: FIFO fill レベル==割り込み発生 fill レベル
- **UART_RFIS_REACH_EXCEED_FLEVEL**: FIFO fill レベル≦割り込み発生 fill レベル

機能:
受信割り込み発生条件を選択します。

戻り値:
なし

12.2.3.20 UART_RxFIFOClear

受信 FIFO クリア

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

引数:
UARTx: UART チャネルを指定します。

機能:
受信 FIFO をクリアします。

戻り値:
なし

12.2.3.21 UART_TxFIFOFillLevel

送信割り込みが発生する送信 FIFO の fill レベルの設定

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

引数:
UARTx: UART チャネルを指定します。

TxFIFOLevel: 受信 FIFO の fill レベルを選択します。

TxFIFOLevel	半二重	全二重
UART_TXFIFO4B_FLEVLE_0_0B	Empty	Empty
UART_TXFIFO4B_FLEVLE_1_1B	1 バイト	1 バイト
UART_TXFIFO4B_FLEVLE_2_0B	2 バイト	Empty
UART_TXFIFO4B_FLEVLE_3_1B	3 バイト	1 バイト

機能:
送信割り込みが発生する送信 FIFO の fill レベルを選択します。

戻り値:
なし

12.2.3.22 UART_TxFIFOINTSel

送信割り込み発生条件の選択

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

引数:
UARTx: UART チャネルを指定します。

TxINTCondition: 受信 割り込み発生条件を選択します。

- **UART_TFIS_REACH_FLEVEL**: FIFO fill レベル==割り込み発生 fill レベル
- **UART_TFIS_REACH_EXCEED_FLEVEL**: FIFO fill レベル≦割り込み発生 fill レベル

機能:
送信割り込み発生条件を選択します。

戻り値:
なし

12.2.3.23 UART_TxFIFOClear

送信 FIFO クリア

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

引数:
UARTx: UART チャネルを指定します。

機能:
送信 FIFO をクリアします。

戻り値:
なし

12.2.3.24 UART_TxBufferClear

送信バッファのクリア

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

引数:
UARTx: UART チャネルを指定します。

機能:
送信バッファをクリアします。

戻り値:
なし

12.2.3.25 UART_GetRxFIFOFillLevelStatus

受信 FIFO の fill レベルの取得

関数のプロトタイプ宣言:

uint32_t
UART_GetRxFIFOFillLevelStatus (TSB_SC_TypeDef* ***UARTx***);

引数:
UARTx: UART チャネルを指定します。

機能:
受信 FIFO の fill レベルを取得します。

戻り値:
- **UART_TRXFIFO_EMPTY**: Empty
- **UART_TRXFIFO_1B**: 1 バイト
- **UART_TRXFIFO_2B**: 2 バイト
- **UART_TRXFIFO_3B**: 3 バイト
- **UART_TRXFIFO_4B**: 4 バイト

12.2.3.26 UART_GetRxFIFOOverRunStatus

受信 FIFO オーバーラン状態の取得

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

引数:
UARTx: UART チャネルを指定します。

機能:
受信 FIFO オーバーラン状態を取得します。

戻り値:
UART_RXFIFO_OVERRUN: オーバーラン発生

12.2.3.27 UART_GetTxFIFOFillLevelStatus

送信 FIFO の fill レベルの取得

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

引数:
UARTx: UART チャネルを指定します。

機能:
送信 FIFO の fill レベルの取得

戻り値:
- **UART_TRXFIFO_EMPTY**: Empty
- **UART_TRXFIFO_1B**: 1 バイト
- **UART_TRXFIFO_2B**: 2 バイト

- **UART_TRXFIFO_3B**: 3 バイト
- **UART_TRXFIFO_4B**: 4 バイト

12.2.3.28 UART_GetTxFIFOUnderRunStatus

送信 FIFO アンダーラン状態の取得

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

引数:
UARTx: UART チャネルを指定します。

機能:
送信 FIFO アンダーラン状態を取得します。

戻り値:
UART_TXFIFO_UNDERRUN: アンダーラン発生

12.2.3.29 UART_SetRxDMAReq

受信割り込みによる DMA 要求の設定

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

引数:
UARTx: UART チャネルを指定します。

NewState: 以下から受信割り込みによる DMA 要求の許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
受信割り込みによる DMA 要求 (受信割り込み INTRX 発生により DMA リクエストを発行) を設定します。

戻り値:
なし

12.2.3.30 UART_SetTxDMAReq

送信割り込みによる DMA 要求の設定

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

引数:
UARTx: UART チャネルを指定します。

NewState: 以下から送信割り込みによる DMA 要求の許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
送信割り込みによる DMA 要求 (送信割り込み INTTX 発生により DMA リクエストを発行) を設定します。

戻り値:
なし

12.2.3.31 UART_SetInputClock

入力クロックの設定

関数のプロトタイプ宣言:
```
void
UART_SetInputClock (TSB_SC_TypeDef * UARTx,
                    uint32_t clock)
```
引数:
UARTx: UART チャネルを指定します。

Clock : 以下から、プリスケーラの入力クロックを選択します。
- **0**：ΦT0/2
- **1** :ΦT0

機能:
プリスケーラの入力クロックを選択します。

戻り値:
なし

12.2.3.32 SIO_SetInputClock

入力クロックの設定

関数のプロトタイプ宣言:
```
void
SIO_SetInputClock (TSB_SC_TypeDef * SIOx,
                   uint32_t Clock)
```
引数:
SIOx: SIO チャネルを指定します。

Clock : 以下から、プリスケーラの入力クロックを選択します。
- **SIO_CLOCK_T0_HALF** : ΦT0/2
- **SIO_CLOCK_T0** :ΦT0

機能:
プリスケーラの入力クロックを選択します。

戻り値:
なし

12.2.3.33 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

12.2.3.34 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

12.2.3.35 SIO_GetRxData

受信用バッファの取得

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)

12.2.3.36 SIO_SetTxData

送信用バッファの設定

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

12.2.3.37 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

12.2.4 データ構造

12.2.4.1 UART_InitTypeDef

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 演算子によって組み合わせてください。
- **UART_ENABLE_TX:** 送信許可
- **UART_ENABLE_RX:** 受信許可

uint32_t
FlowCtrl: フロー制御モードを選択します。(**)
- **UART_NONE_FLOW_CTRL**: フロー制御 無効

(*)補足:
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。

(**)補足:
UART_NONE_FLOW_CTRLのみ選択可能です。

12.2.4.2 SIO_InitTypeDef

メンバ:
uint32_t
InputClkEdge: 入力クロックエッジを選択します。
- **SIO_SCLKS_TXDF_RXDR**: SCLKx 端子の立ち下がりエッジで送信バッファのデータを 1bit ずつ TXDx 端子へ出力します。
- **SIO_SCLKS_TXDR_RXDF**: SCLKx 端子の立ち上がりエッジで送信バッファのデータを 1bit ずつ TXDx 端子へ出力します。

uint32_t

TIDLE: 最終ビット出力後の TXDx 端子の状態を選択します。
- **SIO_TIDLE_LOW**: "Low"出力保持
- **SIO_TIDLE_HIGH**: "High"出力保持
- **SIO_TIDLE_LAST**: 最終ビット保持

uint32_t
TXDEMP: アンダーランエラーが発生したときの TXDx 端子の状態を選択します。
- **SIO_TXDEMP_LOW**: "Low"出力
- **SIO_TXDEMP_HIGH**: "High"出力保持

uint32_t
EHOLDTime: クロック入力モードの TXDx 端子の最終ビットホールド時間を設定します。
- **SIO_EHOLD_FC_2**: 2/fc.
- **SIO_EHOLD_FC_4**: 4/fc.
- **SIO_EHOLD_FC_8**: 8/fc.
- **SIO_EHOLD_FC_16**: 16/fc.
- **SIO_EHOLD_FC_32**: 32/fc.
- **SIO_EHOLD_FC_64**: 64/fc.
- **SIO_EHOLD_FC_128**: 128/fc.

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_TS0**: ΦTS0
- **SIO_BR_CLOCK_TS2**: ΦTS2
- **SIO_BR_CLOCK_TS8**: ΦTS8
- **SIO_BR_CLOCK_TS32**: ΦTS32

uint32_t
Divider: 分周値"N"を選択します。
- **SIO_BR_DIVIDER_16**: 16 分周
- **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

13. WDT

13.1 概要

ウォッチドッグタイマ(WDT)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。

WDT ドライバの API は、検出時間、カウンタのオーバーフロ時の出力、IDLE 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\tmpm037_wdt.c
\Libraries/TX00_Periph_Driver\inc\tmpm037_wdt.h

13.2 API 関数

13.2.1 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)

13.2.2 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(), DT_SetOverflowOutput(), WDT_Init(), WDT_Enable(), WDT_Disable(), WDT_WriteClearCode()
2) IDLE モード時の開始・停止:
   WDT_SetIdleMode()

13.2.3 関数仕様

13.2.3.1 WDT_SetDetectTime

WDT 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*: 以下から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

機能:
WDT の検出時間を設定します。

戻り値:
なし

13.2.3.2 WDT_SetIdleMode

IDLE 時の動作選択

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

引数:
NewState: 以下から IDLE 時の WDT 動作を選択します。
- **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 カウンタ停止
NewState が **DISABLE** の時は WDt カウンタ作動

補足:
CPU が IDLE モードに入る前に、引数を選択して本関数を呼び出してください。

戻り値:
なし

13.2.3.3 WDT_SetOverflowOutput

暴走検出後の動作選択

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

引数:
OverflowOutput: 以下から暴走検出後の動作を選択します。
- **WDT_NMIINT**: INTWDT 割り込み要求を発生します。
- **WDT_WDOUT**: マイコンをリセットします。

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、***OverflowOutput***i が **WDT_WDOUT** の時、カウンタオーバーフローが発生するとリセットが発生します。

戻り値:
なし

13.2.3.4 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 設定に関する構造体。(詳細は“データ構造:”を参照)

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。**WDT_SetDetectTime()**, **WDT_SetOverflowOutput()** が呼び出されます。

戻り値:
なし

13.2.3.5 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

13.2.3.6 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

13.2.3.7 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:
クリアコードをライトします。

戻り値:
なし

13.2.4 データ構造

13.2.4.1 WDT_InitTypeDef

メンバ:
uint32_t
DetectTime 以下から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 以下から、カウンタオーバーフロー時の動作を選択します。

- **WDT_WDOUT:** マイコンをリセットします。
- **WDT_NMIINT:** INTNMI 割り込み要求を発生します。