TOSHIBA

TX03 ペリフェラルドライバ
ユーザガイド
(TMPM330/332/333)

第一版

2017 年 10 月

東芝デバイス＆ストレージ株式会社

CMDR-M330UG-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

本ソフトウェアは、東芝製TX03シリーズマイコンTMPM33x(*)用ペリフェラルドライバセットです。

TX03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。

TMPM33x 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。

※1 本ドキュメントにおいて、"TMPM33x"はTMPM330/332/333を表します。

2. TX03 ペリフェラルドライバの構成

/Libraries
TX03 CMSIS ファイルと TMPM33x ペリフェラルドライバが格納されています。

/Libraries/ TX03_CMSIS
このフォルダには TMPM33x CMSIS ファイルのデバイス・ペリフェラル・アクセス・レイヤーが格納されています。

/Libraries/TX03_Periph_Driver
TMPM33x 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/TX03_Periph_Driver/inc
TMPM33x 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/TX03_Periph_Driver/src
TMPM33x 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
TMPM33x 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
TMPM33x 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
TMPM33x ペリフェラルドライバの使用例が格納されています。

/Utilities/TMPM330-SK

TMPM330-SK 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

本製品は、10ビットのADコンバーターを12チャネル内蔵しています。
- 内部タイマトリガー、外部タイマトリガーによりスタートします。
- チャネル固定/スキャンモード
- 信号/リピートモード
- AD 監視機能 2チャネル
- 変換速度 1.15usec(@fsys = 40MHz)

ADC API は、ADC モジュールを使用するための機能を備えています。
ADC チャネル設定、モード設定, 監視機能設定, 割込み設定, ADC ステータス読み込み, ADC 結果値設定 等 を含みます。

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/TX03_Periph_Driver\src\tmpm33x_adc.c
/Libraries/TX03_Periph_Driver\inc\tmpm33x_adc.h

3.2 TMPM330、TMPM332、TMPM333 の相違点

TMPM330/M333 は 12 チャネルの ADC を内蔵しています。(AN0~AN11)
TMPM332 は 8 チャネルの ADC を内蔵しています(AN4~AN11)

3.3 API 関数

3.3.1 関数一覧

- void ADC_SWReset(void)
- void ADC_SetAccuracy(void)
- void ADC_SetClk(uint32_t ***Sample_HoldTime***, uint32_t ***Prescaler_Output***)
- void ADC_Start(void)
- void ADC_SetScanMode(FunctionalState ***NewState***)
- void ADC_SetRepeatMode(FunctionalState ***NewState***)
- void ADC_SetINTMode(uint8_t ***INTMode***)
- WorkState ADC_GetConvertState(void)
- void ADC_SetInputChannel(uint8_t ***InputChannel***)
- void ADC_SetChannelScanMode(ADC_ChannelScanMode ***ScanMode***)

- void ADC_SetIdleMode(FunctionalState ***NewState***)
- void ADC_SetVref(FunctionalState ***NewState***)
- void ADC_SetInputChannelTop(uint8_t ***TopInputChannel***)
- void ADC_StartTopConvert(void)
- WorkState ADC_GetTopConvertState(void)
- void ADC_SetMonitor(uint16_t ***ADCMPx***, FunctionalState ***NewState***)
- void ADC_SetResultCmpReg(uint16_t ***ADCMPx***, uint8_t ***ResultComparison***)
- void ADC_SetMonitorINT(uint16_t ***ADCMPx***, ADC_ComparisonState ***NewState***)
- void ADC_SetHWTrg(uint8_t ***HwSource***, FunctionalState ***NewState***)
- void ADC_SetHWTrgTop(uint8_t ***HwSource***, FunctionalState ***NewState***)
- ADC_ResultTypeDef ADC_GetConvertResult (uint8_t ***ADREGx***)
- void ADC_SetCmpValue(uint16_t ***ADCMPx***, uint16_t ***value***)

3.3.2 関数の種類

Functions listed above can be divided into four parts:

1) ADC setting by ADC_SetClk(), ADC_SetScanMode(), ADC_SetRepeatMode(), ADC_SetINTMode(), ADC_SetInputChannel(), ADC_SetChannelScanMode(), ADC_SetVref(), ADC_SetInputChannelTop(), ADC_SetMonitor(), ADC_SetResultCmpReg(), ADC_SetMonitorINT(), ADC_SetHWTrg(), ADC_SetHWTrgTop(), ADC_SetCmpValue().
2) ADC function start by ADC_Start(), ADC_StartTopConvert().
3) ADC state or data read functions by ADC_GetConvertState(), ADC_GetTopConvertState(), ADC_GetConvertResult ().
4) ADC_SWReset(), ADC_SetAccuracy(), and ADC_SetIdleMode() handle other specified functions.

3.3.3 関数仕様

3.3.3.1 ADC_SWReset

AD 変換機能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:
AD 変換機能をソフトウェアリセットします。

補足:
ソフトウェアリセットは ADCLK<ADCLK>を除くすべてのレジスタを初期化します
ソフトウェアリセットによる初期化には 3μs かかります。

戻り値:
なし

3.3.3.2 ADC_SetAccuracy

変換精度の設定

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

引数:
なし

機能:
変換精度を設定します。

戻り値:
なし

3.3.3.3 ADC_SetClk

AD 変換サンプルホールド時間とプリスケーラ出力の設定

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

引数:
Sample_HoldTime: 以下から ADC サンプルホールド時間を選択します。

- **ADC_HOLD_CLK_8**, **ADC_HOLD_CLK_16**, **ADC_HOLD_CLK_24**, **ADC_HOLD_CLK_32**, **ADC_HOLD_CLK_64**, **ADC_HOLD_CLK_128**, **ADC_HOLD_CLK_512**

Prescaler_Output: 以下から ADC プリスケーラ出力(ADCLK)を選択します。

- **ADC_FC_DIVIDE_LEVEL_1:** fc
- **ADC_FC_DIVIDE_LEVEL_2:** fc / 2
- **ADC_FC_DIVIDE_LEVEL_4:** fc / 4

- **ADC_FC_DIVIDE_LEVEL_8:** fc / 8
- **ADC_FC_DIVIDE_LEVEL_16:** fc / 16

機能:
Sample_HoldTime で ADC サンプルホールド時間を設定し、***Prescaler_Output*** でプリスケーラ出力を設定します。

補足:
AD変換中にこの関数を使用して、AD変換用クロックの設定を変更しないでください。
ADC_GetConvertState() を使用して、AD変換状態が **BUSY** 以外のときに本関数を実行してください。

戻り値:
なし

3.3.3.4 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:
通常(ソフト)AD 変換を開始します。

補足:
本関数を使用する前に、予め変換モードを指定してください。
AD 変換をスタートさせる場合は、**ADC_SetVref (ENABLE)** を実行して Vref を有効にし、内部回路状態が安定するまで 3 μs 待ってからを **ADC_Start()**を実行してください。

戻り値:
なし

3.3.3.5 ADC_SetScanMode

AD 変換スキャンモードの有効/無効切り替え

関数のプロトタイプ宣言:

void
ADC_SetScanMode(FunctionalState ***NewState***)

引数:
NewState: AD 変換スキャンモードの状態を指定します。
- **ENABLE:** スキャンモードを有効
- **DISABLE:** スキャンモードを無効

機能:
AD 変換スキャンモードの有効/無効を切り替えます。

戻り値:
なし

3.3.3.6 ADC_SetRepeatMode

AD 変換リピートモードの有効/無効切り替え

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

引数:
NewState: AD 変換リピートモードの状態を指定します。
- **ENABLE:** リピートモードを有効
- **DISABLE:** リピートモードを無効

機能:
AD 変換リピートモードの有効/無効を切り替えます。

戻り値:
なし

3.3.3.7 ADC_SetINTMode

チャネル固定リピート変換モードにおける AD 変換 割り込みモードの設定

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

引数:

***INTMode*:** AD 変換割り込みモードを選択します。

- **ADC_INT_SINGLE:** 1 回変換ごと割り込み発生。
- **ADC_INT_CONVERSION_4:** 4 回変換ごと割り込み発生。
- **ADC_INT_CONVERSION_8:** 8 回変換ごと割り込み発生。

機能:

INTMode 設定により、チャネル固定リピート変換モードにおける AD 変換 割り込みモードを設定します。

補足:

本関数はチャネル固定リピート変換モード設定後に使用してください。

以下はチャネル固定リピートモードの例です。

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

戻り値:

なし

3.3.3.8 ADC_GetConvertState

通常 AD 変換状態の取得

関数のプロトタイプ宣言:

WorkState
ADC_GetConvertState(void)

引数:

なし。

機能:

通常 AD 変換状態を取得します。

戻り値:

通常 AD 変換状態:

- **DONE**：通常 AD 変換完了
- **BUSY**：通常 AD 変換中

3.3.3.9 ADC_SetInputChannel

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_SetInputChannel(uint8_t ***InputChannel***)

引数:
***InputChannel*:** AD 変換入力チャネルを選択します。
[TMPM330, TMPM333 の場合]

- **ADC_AN_0, ADC_AN_1, ADC_AN_2, ADC_AN_3, ADC_AN_4, ADC_AN_5, ADC_AN_6, ADC_AN_7, ADC_AN_8, ADC_AN_9, ADC_AN_10, ADC_AN_11**
- **ADC_AN_0_1, ADC_AN_0_2, ADC_AN_0_3, ADC_AN_0_4, ADC_AN_0_5, ADC_AN_0_6, ADC_AN_0_7, ADC_AN_4_5, ADC_AN_4_6, ADC_AN_4_7, ADC_AN_8_9, ADC_AN_8_10, ADC_AN_8_11**

[TMPM332 の場合]

- **ADC_AN_4, ADC_AN_5, ADC_AN_6, ADC_AN_7, ADC_AN_8, ADC_AN_9, ADC_AN_10, ADC_AN_11**
- **ADC_AN_4_5, ADC_AN_4_6, ADC_AN_4_7, ADC_AN_8_9, ADC_AN_8_10, ADC_AN_8_11**

機能:
InputChannel により、AD 変換入力チャネルを設定します。
入力チャネルはモード設定に依存します。
チャネル固定モード(**ADC_SetScanMode(DISABLE)**)の場合、入力チャネルは 12 チャネルの中から 1 チャネル選択可能です。
チャネルスキャンモード(**ADC_SetScanMode(ENABLE)**)の場合、入力チャネルは 4 チャネル((**ADC_SetChannelScanMode(ADC_SCAN_4CH)時**)、または 8 チャネル(**ADC_SetChannelScanMode(ADC_SCAN_8CH)時**)を選択可能です。

戻り値:
なし

3.3.3.10 ADC_SetChannelScanMode

チャネルスキャンモード時の動作設定

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モード時の動作を選択します。

- **ADC_SCAN_4CH**: 4ch スキャン
- **ADC_SCAN_8CH**: 8ch スキャン

機能:
チャネルスキャンモード時の動作を設定します。

戻り値:
なし

3.3.3.11 ADC_SetIdleMode

IDLE モード時の ADC 動作制御の指定

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

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

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

戻り値:
なし

3.3.3.12 ADC_SetVref

ADC Vref アプリケーションの回路 ON/OFF 制御

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

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

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

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

戻り値:
なし

3.3.3.13 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:最優先 AD 変換入力チャネルを選択します。
[TMPM330, TMPM333 の場合]

- **ADC_AN_0, ADC_AN_1, ADC_AN_2, ADC_AN_3, ADC_AN_4, ADC_AN_5, ADC_AN_6, ADC_AN_7, ADC_AN_8, ADC_AN_9, ADC_AN_10, ADC_AN_11**

[TMPM332 の場合]

- **ADC_AN_4, ADC_AN_5, ADC_AN_6, ADC_AN_7, ADC_AN_8, ADC_AN_9, ADC_AN_10, ADC_AN_11**

機能:
TopInputChannel により最優先 AD 変換入力チャネルを設定します。

補足:
最優先 AD 変換入力チャネルには、**ADC_AN_0~ADC_AN_11** のうちの一つを選ぶことができます。

戻り値:
なし

3.3.3.14 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:
本関数を実行する前に、**ADC_SetInputChannelTop()** を実行してください。

戻り値:
なし

3.3.3.15 ADC_GetTopConvertState

最優先 AD 変換状態の取得

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

引数:
なし。

機能:
最優先 AD 変換状態を取得します。

戻り値:
最優先 AD 変換状態:
- **DONE**：最優先 AD 変換完了
- **BUSY**：最優先 AD 変換中

3.3.3.16 ADC_SetMonitor

AD 監視機能の設定

関数のプロトタイプ宣言:
void
ADC_SetMonitor(uint16_t ***ADCMPx***,
FunctionalState ***NewState***)

引数:
ADCMPx: AD 監視機能の比較レジスタを選択します。

- **ADC_CMP_0:** ADCMPCR0
- **ADC_CMP_1:** ADCMPCR1

NewState: AD 監視機能の有効/無効を選択します。

- **ENABLE:** ADC 監視の有効
- **DISABLE:** ADC 監視の無効

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、***NewState*** で有効/無効の設定をします。

戻り値:

なし

3.3.3.17 ADC_SetResultCmpReg

AD 変換機能使用時に、比較対象とする変換結果レジスタの選択

関数のプロトタイプ宣言:

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

引数:

ADCMPx: AD 監視機能の比較レジスタを選択します。

- **ADC_CMP_0:** ADCMPCR0
- **ADC_CMP_1:** ADCMPCR1

ResultComparison: 以下から AD 変換機能使用時に比較対象とする変換結果レジスタを選択します。

- **ADC_REG_08**, **ADC_REG_19**, **ADC_REG_2A**, **ADC_REG_3B, ADC_REG_4C**, **ADC_REG_5D**, **ADC_REG_6E**, **ADC_REG_7F, ADC_REG_SP**

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、***ResultComparison*** で AD 変換機能使用時に、比較対象とする変換結果レジスタを選択します。

戻り値:

なし

3.3.3.18 ADC_SetMonitorINT

AD 監視機能割り込みの設定

関数のプロトタイプ宣言:

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

引数:

ADCMPx: AD 監視機能の比較レジスタを選択します。

- **ADC_CMP_0:** ADCMPCR0
- **ADC_CMP_1:** ADCMPCR1

NewState: 以下から AD 監視機能割り込みを設定します。

- **ADC_COMPARISON_SMALLER:** 変換結果レジスタの値が、変換結果比較レジスタ 0 の値より小さい場合割り込み発生
- **ADC_COMPARISON_LARGER:** 変換結果レジスタの値が、変換結果比較レジスタ 0 の値より大きい場合割り込み発生

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、***NewState*** で AD 監視割り込みを設定します。

戻り値:

なし

3.3.3.19 ADC_SetHWTrg

通常 AD 変換のハードウェア起動と起動ソースの設定

関数のプロトタイプ宣言:

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

引数:

HwSource: 通常 AD 変換の起動ソースを選択します。

- **ADC_EXT_TRG:** 外部トリガ入力
- **ADC_MATCH_TB6RG0:** TB6RG0 一致割り込み

NewState: 通常 AD 変換のハードウェア起動の有効/無効を選択します。

- **ENABLE:** ハードウェアトリガを有効
- **DISABLE:** ハードウェアトリガを無効

機能:

HwSource により、通常 AD 変換のハードウェア起動と起動ソースを設定します。また ***NewState*** により、通常 AD 変換のハードウェアトリガの有効/無効を指定します。
この機能は TB6 が接続されています。

補足:
本デバイスは外部トリガ入力がないため、**ADC_SetHWTrg(ADC_EXT_TRG, NewState)** を選択できません。
最優先 AD 変換のハードウェア起動を使用する場合、通常 AD 変換のハードウェア起動用の外部トリガを使用することはできません。

ハードウエア起動を許可するには、通常 AD 変換では ADMOD4<ADHTG>、最優先 AD 変換では ADMOD4<HADHTG>に"1"をセットします。AD 変換を行うには AD 変換停止中に次の 3 つを行います。

- ハードウェアトリガの選択
- AD 変換のハードウェアトリガ起動の許可
- 変換開始

最優先 AD 変換のハードウェア起動ソースに外部トリガを使用しているときは、通常 AD 変換ハードウェア起動としては外部トリガを設定できません。

戻り値:
なし

3.3.3.20 ADC_SetHWTrgTop

最優先 AD 変換のハードウェア起動と起動ソースの設定

関数のプロトタイプ宣言:

```
void
ADC_SetHWTrgTop(uint8_t HwSource,
                FunctionalState NewState)
```

引数:
HwSource: 最優先 AD 変換の起動ソースを選択します。

- **ADC_EXT_TRG:** 外部トリガ入力
- **ADC_MATCH_TB5RG0:** TB5RG0 一致割り込み

NewState: 最優先常 AD 変換のハードウェア起動の有効/無効を選択します。

- **ENABLE:** ハードウェアトリガを有効
- **DISABLE:** ハードウェアトリガを無効

機能:
HwSource により、最優先 AD 変換のハードウェア起動と起動ソースを設定します。また ***NewState*** により、最優先 AD 変換のハードウェアトリガの有効/無効を指定します。

***補足:**
本デバイスは外部トリガ入力がないため、**ADC_SetHWTrgTop(ADC_EXT_TRG, NewState)** を選択できません。
最優先AD変換のハードウェア起動を使用する場合、通常AD変換のハードウェア起動用の外部トリガを使用することはできません。

戻り値:
なし

3.3.3.21 ADC_GetConvertResult

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

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

引数:
ADREGx: AD 変換結果レジスタを選択します。

- **ADC_REG_08**, **ADC_REG_19**, **ADC_REG_2A**, **ADC_REG_3B, ADC_REG_4C**, **ADC_REG_5D**, **ADC_REG_6E**, **ADC_REG_7F, ADC_REG_SP**

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

補足:
アナログ入力チャネルと AD 変換結果レジスタの関係を下表に示します。

チャネル (port A)	AD 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モード	チャネル固定リピートモード	チャネル固定リピートモード(8 回変換毎)
ADC_AN_0	ADC_REG_08	ADC_REG_08 固定	ADC_REG_08--> ADC_REG_3B	ADC_REG_08--> ADC_REG_7F
ADC_AN_1	ADC_REG_19			
ADC_AN_2	ADC_REG_2A			
ADC_AN_3	ADC_REG_3B			
ADC_AN_4	ADC_REG_4C			
ADC_AN_5	ADC_REG_5D			
ADC_AN_6	ADC_REG_6E			
ADC_AN_7	ADC_REG_7F			

ADC_AN_8	ADC_REG_08			
ADC_AN_9	ADC_REG_19			
ADC_AN_10	ADC_REG_2A			
ADC_AN_11	ADC_REG_3B			

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

戻り値:
AD 変換結果: 詳細は"データ構造"を参照してください。

3.3.3.22 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(uint16_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_CMP_0:** ADCMPCR0
- **ADC_CMP_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

3.3.4 データ構造

3.3.4.1 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

4. CEC

4.1 概要

本デバイスは 1 チャネルの CEC を内蔵しています。本機能はは Consumer Electronics Control (以下 CEC) プロトコルのデータ送受信を行います。 (HDMI 規格 Version 1.3a に準拠)

受信

- 32kHz クロックでサンプリング
    1. ノイズキャンセル時間を調整可能
    2. 1byte 毎にデータを受信
- データサンプリングポイントを調整可能
    1. ディスティネーションアドレス不一致でも受信可能
- エラー検出
    1. 周期違反(最少/最大)
    2. ACK 衝突
    3. 波形エラー

送信

- 1byte 毎にデータを送信
    1. バスフリーを自動判定し送信開始
- 送信波形の調整
    1. 立ち上がりタイミング、周期を調整可能
- エラー検出
    1. アービトレーションロスト
    2. ACK 違反

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/TX03_Periph_Driver/src/tmpm33**x**_cec.c(*)
/Libraries/TX03_Periph_Driver/inc/tmpm33**x**_cec.h(*)

補足: "**x**"は 0、または 2 です。

4.2 TMPM330, TMPM332, TMPM333 の相違点

TMPM333 は CEC 機能を未サポートです。

4.3 API 関数

4.3.1 関数一覧

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

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

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

4.3.2 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) CEC の初期化と各種設定:
   CEC_Enable(), CEC_Disable(), CEC_DefaultConfig(), CEC_SetLogicalAddr(), CEC_AddLogicalAddr(), CEC_RemoveLogicalAddr(), CEC_GetLogicalAddr(), CEC_SetACKResponseMode(), CEC_SetNoiseCancellation(), CEC_SetCycleConfig(), CEC_SetDataValidTime(), CEC_SetTimeOutMode(), CEC_SetRxErrINTSuspend(), CEC_SetSnoopMode(), CEC_SetRxDetectWaveConfig(), CEC_SetRxStartBitWaveConfig(), CEC_SetRxWaveErrDetect(), CEC_SetTxWaveConfig(), CEC_SetTxBroadcast(), CEC_SetBusFreeTime()
2) 送受信とエラーチェック:
   CEC_SetRxCtrl(), CEC_StartTx(), CEC_StopTx(), CEC_GetRxData(), CEC_SetTxData(), CEC_GetRxState(), CEC_GetTxState(), CEC_GetRxINTState(), CEC_GetTxINTState()
3) その他:
   CEC_SWReset(), CEC_SetIdleMode()

4.3.3 関数仕様

4.3.3.1 CEC_Enable

CEC 動作の許可

関数のプロトタイプ宣言:
void
CEC_Enable(void)

引数:
なし

機能:
CEC 動作を許可します。使用前に CEC を許可してください。

戻り値:
なし

4.3.3.2 CEC_Disable

CEC 動作の禁止

関数のプロトタイプ宣言:
void
CEC_Disable(void)

引数:
なし

機能:
CEC 動作を禁止します。

戻り値:
なし

4.3.3.3 CEC_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
CEC_SWReset(void)

引数:
なし

機能:
CEC 回路を初期化するリセット信号を発生します。リセット後、すべての制御レジスタやステータスフラグはリセット後の値に初期化されます。
本関数をコールした後、リセットが終了したか確認可能な以下の関数を呼び出すことが可能です。
CEC_GetRxState() (戻り値は **DISABLE** になります)
CEC_GetTxState() (戻り値は **CEC_TX_STOPED** になります)

戻り値:
なし

4.3.3.4 CEC_DefaultConfig

デフォルト値の設定

関数のプロトタイプ宣言:

Result
CEC_DefaultConfig(void)

引数:

なし

機能:

CEC を以下の値で設定します。
Idle Mode:　on
Noise Cancellation Time: H: 1 cycle L: 1 cycle
Cycle Range: 2.05ms~2.75ms
Data Valid Time: 1.05ms
Time Out: 1 Bit
Rx Start Wave configure: Min of start: 3.5ms; Max of start: 3.9ms
Min of cycle: 4.3ms; Max of cycle: 4.7ms
Receive Bit Wave configure: Min of “1”: 0.4ms; Max of “1”: 0.8ms
Min of “0”:1.3ms; Max of “0”: 1.7
Send Bit Wave configure: RV
Bus free configure: 5 bit cycle
Snoop mode: On
CEC が受信許可、または、送信中の場合、本関数は **ERROR** を返します。

戻り値:

- **SUCCESS**　成功
- **ERROR**　エラー

4.3.3.5 CEC_SetLogicalAddr

ロジカルアドレスの設定

関数のプロトタイプ宣言:

Result
CEC_SetLogicalAddr(CEC_LogicalAddr ***LogicalAddr***)

引数:

LogicalAddr: 以下から設定するロジカルアドレスを選択します。

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

機能:
ロジカルアドレスを設定します。
本関数が実行されると、以前設定されたロジカルアドレスはクリアされ、新しいロジカルアドレスが設定されます。
CEC が受信許可、または送信中の場合、本関数の戻り値は**ERROR**となり、設定変更は行いません。

戻り値:
- **SUCCESS** 成功
- **ERROR** エラー

4.3.3.6 CEC_AddLogicalAddr

ロジカルアドレスの追加

関数のプロトタイプ宣言:
Result
CEC_AddLogicalAddr(CEC_LogicalAddr ***LogicalAddr***)

引数:
LogicalAddr: 以下から追加するロジカルアドレスを指定します。
- **CEC_TV**, **CEC_RECORDING_DEVICE_1**, **CEC_RECORDING_DEVICE_2**, **CEC_STB_1**, **CEC_DVD_1**, **CEC_AUDIO_SYSTEM**, **CEC_STB_2**, **CEC_STB_3**, **CEC_DVD_2**, **CEC_RECORDING_DEVICE_3**, **CEC_FREE_USE**, **CEC_BROADCAST**

機能:
ロジカルアドレスを追加します。本関数が実行されると、以前に設定されたロジカルアドレスを残したまま、新しいロジカルアドレスを追加します。
CEC が受信許可、または、送信中の場合、本関数の戻り値は **ERROR** となり、設定を変更しません。

戻り値:
- **SUCCESS** 成功
- **ERROR** エラー

4.3.3.7 CEC_RemoveLogicalAddr

ロジカルアドレスの削除

関数のプロトタイプ宣言:
Result

CEC_RemoveLogicalAddr(CEC_LogicalAddr ***LogicalAddr***)

引数:

LogicalAddr: 以下から削除するロジカルアドレスを指定します。

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

機能:

ロジカルアドレスを削除します。本関数が実行されると、選択されたロジカルアドレスのみ削除され、それ以外のロジカルアドレスは削除されません。

CEC が受信許可、または、送信中の場合、本関数は **ERROR** を返し、設定を変更しません。

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.3.3.8 CEC_GetLogicalAddr

ロジカルアドレスの取得

関数のプロトタイプ宣言:

CEC_AddrListTypeDef
CEC_GetLogicalAddr(void)

引数:

なし

機能:

ロジカルアドレス情報を取得します。

戻り値:

ロジカルアドレス情報:

- **CEC_AddrListTypeDef**: ロジカルアドレスリスト構造体(詳細は “4.3.4 データ構造” を参照してください)

4.3.3.9 CEC_SetRxCtrl

データ受信制御

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

引数:
NewState: データ受信機能有効 / 無効を選択します。
- **ENABLE**：有効
- **DISABLE :** 無効

機能:
CEC 受信を制御します。<CECREN> ビットへの設定が実際に反映されるまでには若干の時間を要します。本関数を呼び出した後、**CEC_GetRxState()**を実行することで、CEC 受信の有効/無効の状態を確認できます。

戻り値:
なし

4.3.3.10 CEC_StartTx

送信の開始

関数のプロトタイプ宣言:
Result
CEC_StartTx(void)

引数:
なし

機能:
送信を開始します。すでに送信中の場合の戻り値は **ERROR** となります。

戻り値:
- **SUCCESS** 成功
- **ERROR** エラー

4.3.3.11 CEC_StopTx

送信の停止

関数のプロトタイプ宣言:
void
CEC_StopTx(void)

引数:
なし

機能:
送信を停止します。

戻り値:
なし

4.3.3.12 CEC_GetRxData

受信データの読み出し

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

引数:
なし

機能:
受信データを読み出します。受信した 1 バイト分のデータが読めます。ビット 7 は MSB です。また受信したACKビットとEOMビットが読めます。受信割り込み発生後、なるべく早く読み込んでください。

戻り値:
受信バッファからの受信データ

- **CEC_DataTypeDef**: 受信データ構造体 (詳細は “データ構造” を参照してください)

4.3.3.13 CEC_SetTxData

送信データの設定

関数のプロトタイプ宣言:
void
CEC_SetTxData(uint8_t ***Data***,
CEC_EOMBit ***EOM_Flag***)

引数:
Data: 送信データを指定します。データ長は 1 バイトです。

EOM_Flag: 送信する EOM ビットを設定します。
- **CEC_EOM**: フレームの最終データの場合に設定します。
- **CEC_NO_EOM**: フレームの最終データ以外の場合に設定します。

機能:
送信データを設定します。**CEC_StartTx()** を実行する前に、本関数によってフレームの最初のデータを設定してください。最初の 1 ビットデータの送信が開始されると、送信割り込みが発生します。送信割り込み発生後、本関数によって次のデータを設定することができます。データ送信は、***EOM_Bit*** に **CEC_EOM** がセットされるまで続きます。

戻り値:
なし

4.3.3.14 CEC_SetIdleMode

IDLE モード時の CEC 動作制御の指定

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

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

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

戻り値:
なし

4.3.3.15 CEC_GetRxState

受信状態の取得

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

引数:

なし

機能:
受信状態を取得します。

戻り値:
- **ENABLE**: 動作中
- **DISABLE**: 停止中

4.3.3.16 CEC_GetTxState

送信状態の取得

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

引数:
なし

機能:
送信状態を取得します。

戻り値:
- **BUSY**: 送信中
- **DONE**: 送信していない

4.3.3.17 CEC_GetRxINTState

受信割り込みステータスの取得

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

引数:
なし

機能:
受信割り込みステータスを取得します。

戻り値:

受信割り込みステータス:

- **RxEnd**(Bit 0) ： 1 byte データ受信終了
- **RxStartBit**(Bit 1)：開始ビットの検出
- **MAXCycleErr**(Bit 2)： 最大サイクルエラー検出
- **MINCycleErr**(Bit 3)： 最小サイクルエラー検出
- **ACKCollision**(Bit 4)： ACK 不一致検出
- **BufOverrun**(Bit 5)： 受信バッファオーバーラン検出
- **WaveformErr**(Bit 6)： 波形エラー検出

4.3.3.18 CEC_GetTxINTState

送信割り込みステータスの取得

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

引数:
なし

機能:
送信割り込みステータスを取得します。

戻り値:
送信割り込みステータス:

- **TxStart**(Bit 0)： 1 バイトデータの送信開始
- **TxEnd**(Bit 1)：EOM ビットを含むデータ送信の完了
- **ArbitrationLost**(Bit 2)：アービトレーションロストの発生
- **ACKErr**(Bit 3)： ACK エラーの検知
- **BufUnderrun**(Bit 4)：送信バッファアンダーランの検知

4.3.3.19 CEC_SetACKResponseMode

ACK 応答モードの設定

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

引数:
NewState: 以下から ACK モードを設定します。

- **ENABLE**： 許可

- **DISABLE**： 禁止

機能:

ACK 応答モードを設定します。ディスティネーションアドレスが設定済みのロジカルアドレスと一致する時に、データブロックに対して論理 "0" の ACK 応答をするかどうかを設定します。ヘッダブロックに対しては、このビットの設定によらず、アドレスが一致すると論理"0"の ACK 応答を行います。詳細はデータシートの CEC 章「(5) ACK 応答」を参照してください。CEC が受信許可、または、送信中の場合、本関数の戻り値は **ERROR** となります。

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.3.3.20 CEC_SetNoiseCancellation

ノイズキャンセルモードの設定

関数のプロトタイプ宣言:

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

引数:

LowCancellation: "Low"検出ノイズキャンセル時間を設定します。

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

HighCancellation: "High"検出ノイズキャンセル時間を設定します。

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

機能:

ノイズキャンセル時間を ***LowCancellation, HighCancellation*** に設定します。検出時間 (1 または 0) は個々に設定できます。指定数がサンプリングできない場合、ノイズとみなされます。詳細はデータシートの CEC 章「 (2) ノイズキャンセル時間」を参照ください。CEC が受信許可、または、送信中の場合、本関数の戻り値は **ERROR** となります。

戻り値:

- **SUCCESS**　成功
- **ERROR**　　エラー

4.3.3.21　CEC_SetCycleConfig

周期違反検出時間の設定

関数のプロトタイプ宣言:

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

引数:

CycleMin: 以下から最少周期違反検出時間を選択します。

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

CycleMax: 以下から最大周波数違反検出時間を選択します。

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

機能:

周期違反を検出します。 1/fs 単位で-4～+3/fs まで設定可能です。データ受信中に違反を検出すると、エラー割り込みが発生し、CEC は次の開始ビットを待ちます。受信データは破棄されます。

CEC が受信許可、または、送信中の場合、本関数の戻り値は **ERROR** となります。

戻り値:

- **SUCCESS**　成功

- **ERROR** エラー

4.3.3.22 CEC_SetDataValidTime

データ 0/1 判別タイミングの設定

関数のプロトタイプ宣言:
Result
CEC_SetDataValidTime(CEC_ValidTime ***ValidTime***)

引数:
ValidTime: 以下から、データ 0/1 判別タイミングを選択します。
- **CEC_VALID_TIME_0**: 1.05ms
- **CEC_VALID_TIME_1**: 1.05ms+2cycles
- **CEC_VALID_TIME_2**: 1.05ms+4cycles
- **CEC_VALID_TIME_3**: 1.05ms+6cycles
- **CEC_VALID_TIME_4**: 1.05ms-2cycles
- **CEC_VALID_TIME_5**: 1.05ms-4cycles
- **CEC_VALID_TIME_6**: 1.05ms-6cycles

機能:
データ 0/1 判別タイミングを設定します。データの論理"0"/論理"1"判別を行うポイントを設定します。約 1.05 ms を基準に、2/fs 単位で±6/fs まで設定可能です。
CEC が受信許可、または、送信中の場合、本関数の戻り値は **ERROR** となります。詳細はデータシートの CEC 章「 (4) 判別タイミング」を参照ください。

戻り値:
- **SUCCESS** 成功
- **ERROR** エラー

4.3.3.23 CEC_SetTimeOutMode

タイムアウト判定時間の設定

関数のプロトタイプ宣言:
Result
CEC_SetTimeOutMode(CEC_TimeOut ***TimeOut***)

引数:
TimeOut: 以下から、タイムアウト判定時間を選択します。
- **CEC_TIME_OUT_1_BIT**: 1 bit cycle
- **CEC_TIME_OUT_2_BIT**: 2 bit cycle

- **CEC_TIME_OUT_3_BIT**: 3 bit cycle

機能:
タイムアウト判定時間を設定します。本設定は、受信エラー割り込み保留で使用されます。CEC が受信許可、または、送信中の場合、本関数の戻り値は **ERROR** となります。

戻り値:
- **SUCCESS** 成功
- **ERROR** エラー

4.3.3.24 CEC_SetRxErrINTSuspend

受信エラー割り込み保留設定

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

引数:
NewState: 以下から、受信エラー割り込み保留の有効/無効を選択します。
- **ENABLE:** 受信エラー割り込みを有効にする
- **DISABLE:** 受信エラー割り込みを無効にする

機能:
受信エラー割り込み (最大周期違反、バッファオーバラン、波形エラー) を保留にするか設定します。**ENABLE** に設定されていると、エラー検出時点では割り込みは発生しません。CEC が受信許可、または送信中の場合、本関数の戻り値は **ERROR** となります。

戻り値:
- **SUCCESS** 成功
- **ERROR** エラー

4.3.3.25 CEC_SetSnoopMode

ロジカルアドレス不一致時に、データ受信動作を行うかどうかを設定

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

引数:
NewState: スヌープモード有効 / 無効を設定します。

- **ENABLE:** スヌープモード有効
- **DISABLE:** スヌープモード無効

機能:

ディスティネーションアドレスが、ロジカルアドレスと異なる場合にもデータの受信を行うかどうかを設定します。この場合、受信動作は通常の場合と同様に行い、違反が検出されれば割り込みも発生しますが、ACK 応答はヘッダブロック、データブロックとも行いません。ブロードキャストメッセージは、本設定にかかわらず受信します。CEC が受信許可、または送信中の場合、本関数の戻り値は **ERROR** となります。

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.3.3.26 CEC_SetRxDetectWaveConfig

波形エラー検出範囲の設定

関数のプロトタイプ宣言:

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

引数:

***Logical1RisingTimeMin*:** 以下から、論理 "1" 波形の立ち上がりタイミングより早い場合にエラー検出を行うための設定を選択します。

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

Logical1RisingTimeMax: 以下から、論理 "1" 波形の立ち上がりタイミングより遅い場合にエラー検出を行うための設定を選択します。

- **CEC_LOGICAL_1_RISING_TIME_MAX_0**: 0.8ms
- **CEC_LOGICAL_1_RISING_TIME_MAX_1**: 0.8ms+1cycle
- **CEC_LOGICAL_1_RISING_TIME_MAX_2**: 0.8ms+2cycle
- **CEC_LOGICAL_1_RISING_TIME_MAX_3**: 0.8ms+3cycle

- **CEC_LOGICAL_1_RISING_TIME_MAX_4**: 0.8ms+4cycle
- **CEC_LOGICAL_1_RISING_TIME_MAX_5**: 0.8ms+5cycle
- **CEC_LOGICAL_1_RISING_TIME_MAX_6**: 0.8ms+6cycle
- **CEC_LOGICAL_1_RISING_TIME_MAX_7**: 0.8ms+7cycle

Logical0RisingTimeMin: 以下から、論理 "0" 波形の立ち上がりタイミングより早い場合にエラー検出を行うための設定を選択します。

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

Logical0RisingTimeMax: 以下から、論理 "0" 波形の立ち上がりタイミングより遅い場合にエラー検出を行うための設定を選択します。

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

機能:

受信波形が設定された波形エラー検出範囲外の場合に、エラー検出をするかどうかを設定します。検出時間は、***Logical1RisingTimeMin***, ***Logical1RisingTimeMax***, ***Logical0RisingTimeMin, Logical0RisingTimeMax*** で設定します。
詳細はデータシートの CEC 章「 (10) 波形エラー検出」を参照ください。
CEC が受信許可、または送信中の場合、本関数の戻り値は **ERROR** となります。

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.3.3.27 CEC_SetRxStartBitWaveConfig

スタートビット検出時の立ち上がりタイミングを設定

関数のプロトタイプ宣言:

Result
CEC_SetRxStartBitWaveConfig(

CEC_StartBitRisingTimeMin ***RisingTimeMin***,
CEC_StartBitRisingTimeMax ***RisingTimeMax***,
CEC_StartBitCycleMin ***CycleMin***,
CEC_StartBitCycleMax ***CycleMax***)

引数:

RisingTimeMin: 以下から、スタートビット検出時の立ち上がりタイミングの最小値の条件を選択します。

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

RisingTimeMax: 以下から、スタートビット検出時の立ち上がりタイミングの最大 値の条件を選択します。

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

CycleMin: 以下から、スタートビット検出時の周期の最小値の条件を選択します。

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

CycleMax: 以下から、スタートビット検出時の周期の最大値の条件を選択します。

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

- **CEC_START_BIT_CYCLE_MAX_7**: 4.7ms +7cycle

機能:
立ち上がりのタイミングとスタートビット検出条件を設定します。
RisingTimeMin :立ち上がりタイミングの最小値
RisingTimeMax :立ち上がりタイミングの最大値
CycleMin: スタートビット検出時の周期の最小値
CycleMax :スタートビット検出時の周期の最大値
詳細はデータシートの CEC 章「 (9) スタートビット検出」を参照ください。
CEC が受信許可、または送信中の場合、本関数の戻り値は **ERROR** となります。

戻り値:
- **SUCCESS** 成功
- **ERROR** エラー

4.3.3.28 CEC_SetRxWaveErrDetect

波形エラー検出の有効/無効設定、および波形エラー割り込みの発生の設定

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

引数:
NewState: 以下から、波形エラー検出の有効/無効を選択します。
- **ENABLE:** 波形エラー検出許可
- **DISABLE:** 波形エラー検出禁止

機能:
受信データ波形が規格から外れたことを検出し、波形エラー割り込みを発生させる、波形エラー検出を設定します。CEC が受信許可、または送信中の場合、本関数の戻り値は **ERROR** となります。

戻り値:
- **SUCCESS** 成功
- **ERROR** エラー

4.3.3.29 CEC_SetTxWaveConfig

送信波形の設定

関数のプロトタイプ宣言:

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

引数:

DataBitCycle: 以下から、データビットの周期を選択します。

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

DataBitRisingTime: 以下から、データビットの立ち上がりタイミングを選択します。

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

StartBitCycle: 以下から、スタートビットの周期を選択します。

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

***StartBitRisingTime*:** 以下から、スタートビットの立ち上がりタイミングを選択します。

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

機能:

送信波形のデータビット/スタートビットの周期と立ち上がりタイミングを設定します。
DataBitCycle, ***DataBitRisingTime***, ***StartBitCycle, StartBitRisingTime*** で、立ち上がりと周期の最も早いタイミングから標準値の間で設定をします。詳細はデータシートの CEC 章「(3) 送信波形調整」を参照ください。
CEC が受信許可、または送信中の場合、本関数の戻り値は **ERROR** となります。

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.3.3.30 CEC_SetTxBroadcast

ブロードキャスト送信の設定

関数のプロトタイプ宣言:

Result
CEC_SetTxBroadcast(FunctionalState ***NewState***)

引数:

NewState: 以下から、ブロードキャスト送信モードの有効/無効を選択します。

- **ENABLE:** ブロードキャスト送信
- **DISABLE:** ブロードキャスト送信しない

機能:

ブロードキャスト送信時には、本関数を呼び出し、***NewState*** を **ENABLE** に設定します。***NewState*** が **ENABLE** の時、ACK サイクルで論理 “0” の応答があるとエラーになります。***NewState*** が **DISABLE** の時、ACK サイクルで論理 “1” の応答があるとエラーになります。CEC が受信許可、または送信中の場合、本関数の戻り値は **ERROR** となります。

戻り値:

- **SUCCESS** 成功

- **ERROR** エラー

4.3.3.31 CEC_SetBusFreeTime

送信開始前に確認するバスフリー時間の設定

関数のプロトタイプ宣言:
Result
CEC_SetBusFreeTime (CEC_BusFree ***BusFree***)

引数:
BusFree: 以下から、バスフリー待ち時間を選択します。
- **CEC_BUS_FREE_1_BIT**: 1 bit cycle
- **CEC_BUS_FREE_2_BIT**: 2 bit cycle
- **CEC_BUS_FREE_3_BIT**: 3 bit cycle
- **CEC_BUS_FREE_4_BIT**: 4 bit cycle
- **CEC_BUS_FREE_5_BIT**: 5 bit cycle
- **CEC_BUS_FREE_6_BIT**: 6 bit cycle
- **CEC_BUS_FREE_7_BIT**: 7 bit cycle
- **CEC_BUS_FREE_8_BIT**: 8 bit cycle
- **CEC_BUS_FREE_9_BIT**: 9 bit cycle
- **CEC_BUS_FREE_10_BIT**: 10 bit cycle
- **CEC_BUS_FREE_11_BIT**: 11 bit cycle
- **CEC_BUS_FREE_12_BIT**: 12 bit cycle
- **CEC_BUS_FREE_13_BIT**: 13 bit cycle
- **CEC_BUS_FREE_14_BIT**: 14 bit cycle
- **CEC_BUS_FREE_15_BIT**: 15 bit cycle
- **CEC_BUS_FREE_16_BIT**: 16 bit cycle

機能:
送信開始前に確認するバスフリー時間の設定を行います。1 サイクルから 16 サイクルの間で設定します。バスフリー状態の確認は、最終ビットから開始します。
CEC_BUS_FREE_1_BIT がバスフリーの場合、送信開始します。詳細はデータシートの CEC 章「(3) バスフリー待ち時間」を参照ください。
送信中の場合、本関数の戻り値は **ERROR** となります。

戻り値:
- **SUCCESS** 成功
- **ERROR** エラー

4.3.4 データ構造

4.3.4.1 CEC_DataTypeDef

メンバ:

uint8_t

Data: 受信データの 1 バイト分を読みます。ビット 7 が MSB です。

CEC_ACKState

ACKBit: 受信 ACK ビットです。

- **CEC_ACK**: ACK ビットが "1"
- **CEC_NO_ACK**: ACK ビットが"0"

CEC_EOMBit

EOMBit: 受信 EOM ビットです。

- **CEC_EOM**: EOM ビットが “1”
- **CEC_NO_EOM**: EOM ビットが “0”

4.3.4.2 CEC_AddrListTypeDef

メンバ:

uint8_t

AddrNumber: ロジカルアドレス番号

CEC_LogicalAddr

AddrList[16]: ロジカルアドレス一覧です。以下のいずれかの値を選択します。

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

5. CG

5.1 概要

本 CG API は TMPM36x CG における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/TX03_Periph_Driver¥src¥tmpm36**x**_cg.c(*)
/Libraries/TX03_Periph_Driver¥inc¥tmpm36**x**_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
fs : XT1とXT2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
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)

補足: 0, 2, 3 を"**x**"と記載します。

5.2 API 関数

5.2.1 関数一覧

- void CG_SetFgearLevel(CG_DivideLevel ***DivideFgearFromFc***)
- CG_DivideLevel CG_GetFgearLevel(void)
- Result CG_SetPhiT0Level(CG_DivideLevel ***DividePhiT0FromFc***)
- CG_DivideLevel CG_GetPhiT0Level(void)
- void CG_SetSCOUTSrc(CG_SCOUTSrc ***Source***)

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

5.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています。

1) クロックの選択:
   CG_SetFgearLevel(), CG_GetFgearLevel(), CG_SetPhiT0Level(), CG_GetPhiT0Level(), CG_SetSCOUTSrc(), CG_GetSCOUTSrc(), CG_SetWarmUpTime(), CG_StartWarmUp(), CG_GetWarmUpState(), CG_SetPLL(), CG_GetPLLState(), CG_SetFosc(), CG_GetFoscState(), CG_SetFs(), CG_GetFsState(), CG_SetFcSrc(), CG_GetFcSrc(), CG_SetFsysSrc(),CG_GetFsysSrc()
2) スタンバイモードの設定:
   CG_SetSTBYMode(), CG_GetSTBYMode( ), CG_SetExitStopModeFosc(), CG_GetExitStopModeFoscState(), CG_SetExitStopModeFs(),

CG_GetExitStopModeFsState(), CG_SetPinStateInStopMode(),
CG_GetPinStateInStopMode()

3) 割り込みの設定:
CG_SetSTBYReleaseINTSrc(), CG_GetSTBYReleaseINTState(), CG_ClearINTReq(),
CG_GetNMIFlag(), CG_GetResetFlag()

5.2.3 関数仕様

5.2.3.1 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

機能:
fgear,fc 間の分周レベルを設定します。

戻り値:
なし

5.2.3.2 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_UNKNOWN:** 無効なデータ

5.2.3.3 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

機能:

PhiT0(ΦT0) ,fc 間の分周レベルを設定します。

戻り値:

- **SUCCESS**: 成功
- **ERROR**: 失敗

5.2.3.4 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_UNKNOWN**: 無効データ

5.2.3.5 CG_SetSCOUTSrc

SCOUT ソースクロック設定

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

引数:
Source: 以下から、SCOUT のソースクロックを選択します。

- **CG_SCOUT_SRC_FS**: fs
- **CG_SCOUT_SRC_HALF_FSYS**: fsys/2
- **CG_SCOUT_SRC_FSYS**: fsys
- **CG_SCOUT_SRC_PHIT0**: φT0

機能:
SCOUT のソースクロックを設定します。

戻り値:
なし

5.2.3.6 CG_GetSCOUTSrc

SCOUT ソースクロック設定の取得

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

引数:
なし

機能:
SCOUT のソースクロック設定を取得します。

戻り値:
SCOUT のソースクロック:

- **CG_SCOUT_SRC_FS**: fs
- **CG_SCOUT_SRC_HALF_FSYS**: fsys/2
- **CG_SCOUT_SRC_FSYS**: fsys
- **CG_SCOUT_SRC_PHIT0**: φT0

5.2.3.7 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:
Result
CG_SetWarmUpTime (CG_WarmUpSrc ***Source***,
CG_WarmUpTime ***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_X1**: fosc
- **CG_WARM_UP_SRC_XT1**: fs

Time: ウォーミングアップカウンタ値を設定します。

機能:
Time は以下のいずれかとなります。

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

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

Time が正しく設定されない場合、本 API は **ERROR** を返却します。
ウォームアップカウンタの詳細は以下を参照してください。

表 ウォームアップカウンタ(fosc=10MHz, fs=32.768kHz)

ウォームアップカウンタ	***Source*** = **CG_WARM_UP_SRC_X1**	***Source*** = **CG_WARM_UP_SRC_XT1**
CG_WARM_UP_TIME_NONE	ウォームアップなし	ウォームアップなし
CG_WARM_UP_TIME_EXP_6	-	1.953 ms
CG_WARM_UP_TIME_EXP_7	-	3.906 ms
CG_WARM_UP_TIME_EXP_8	-	7.813 ms
CG_WARM_UP_TIME_EXP_10	102.4 us	-
CG_WARM_UP_TIME_EXP_11	204.8 us	-
CG_WARM_UP_TIME_EXP_12	409.6 us	-
CG_WARM_UP_TIME_EXP_13	819.2 us	-
CG_WARM_UP_TIME_EXP_14	1.638 ms	-
CG_WARM_UP_TIME_EXP_15	3.277 ms	1.0 s
CG_WARM_UP_TIME_EXP_16	6.554 ms	2.0 s
CG_WARM_UP_TIME_EXP_17	-	4.0 s
CG_WARM_UP_TIME_EXP_18	-	8.0 s

戻り値:

- **SUCCESS**: 成功
- **ERROR**: 失敗

5.2.3.8 CG_StartWarmUp

ウォームアップ開始

関数のプロトタイプ宣言:
void

CG_StartWarmUp (void)

引数:
なし

機能:
ウォームアップを開始します。

戻り値:
なし

5.2.3.9 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:

```
/* set up warm time 100us */
CG_SetWarmUpTime(CG_WARM_UP_SRC_X1,
CG_WARM_UP_TIME_EXP_10);
/* start warm up */
CG_StartWarmUp();
/* check warm up is finished or not*/
While(CG_GetWarmUpState() == BUSY);
```

戻り値:
ウォーミングアップ状態:

- **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

5.2.3.10 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 有効
- **DISABLE**: PLL 無効

機能:
PLL 回路の有効/無効を設定します。
PLL として fc が選択されている場合は PLL を無効にできないため、本 API は **ERROR** を返却します。

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

5.2.3.11 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 回路の設定状態:
- **ENABLE:** PLL 有効
- **DISABLE:** PLL 無効

5.2.3.12 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 (FunctionalState ***NewState***)

引数:
NewState 高速発振器の有効/無効を選択します。
- **ENABLE**: 有効
- **DISABLE**: 無効

機能:
高速発信器の有効/無効を設定します。
システムクロック(fsys)として fgear が選択されている場合、高速発振(fosc)は無効にできないため、本 API は **ERROR** を返却します。

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

5.2.3.13 CG_GetFoscState

高速発信器の状態

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

引数:
なし

機能:
高速発信器の状態を取得します。

戻り値:
fosc の状態:
- **ENABLE**: 有効
- **DISABLE**: 無効

5.2.3.14 CG_SetFs

低速発振器(fs)の設定

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

引数:

NewState: 以下から、低速発振器の有効/無効を設定します。

- **ENABLE**: 有効
- **DISABLE**: 無効

機能:

低速発振器(fs)の有効/無効を設定します。

戻り値:

- **SUCCESS**: 成功
- **ERROR**: 失敗

5.2.3.15 CG_GetFsState

低速発振器(fs)の状態取得

関数のプロトタイプ宣言:

FunctionalState
CG_GetFsState (void)

引数:

なし

機能:

低速発振器(fs)の状態を取得します。

戻り値:

fs の状態です。

- **ENABLE**: 有効
- **DISABLE**: 無効

5.2.3.16 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: スタンバイモードを選択します。

- **CG_STBY_MODE_STOP**: STOP 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_SLEEP**: SLEEP モード (低速発振器と RTC、CEC、RMC のみ動作)
- **CG_STBY_MODE_IDLE**: IDLE モード(CPU のみ停止)

機能:
スタンバイモードを選択します。

戻り値:
なし

5.2.3.17 CG_GetSTBYMode

スタンバイモード設定状態の取得

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

引数:
なし

機能:
スタンバイモード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:
スタンバイモード:
CG_STBY_MODE_STOP: STOP モード
CG_STBY_MODE_SLEEP: SLEEP モード
CG_STBY_MODE_IDLE: IDLE モード
CG_STBY_MODE_UNKNOWN：無効なモード

5.2.3.18 CG_SetExitStopModeFosc

STOP モード解除後の高速発振器の動作選択

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

引数:
NewState： 以下から STOP モード解除後の高速発振器の動作を選択します。
- **ENABLE** ： 発振
- **DISABLE**： 停止

機能:
STOP モード解除後の高速発振器の動作を選択します。

戻り値:
なし

5.2.3.19 CG_GetExitStopModeFoscState

STOP モード解除後の高速発振器の動作選択状態の取得

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

引数:
なし

機能:
STOP モード解除後の高速発振器の動作選択状態を取得します。

戻り値:
STOP モード解除後の高速発振器の動作選択状態
- **ENABLE :** 発振
- **DISABLE:** 停止

5.2.3.20 CG_SetExitStopModeFs

STOP モード解除後の低速発振器の動作選択

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

引数:
NewState: 以下から STOP モード解除後の低速発振器の動作を選択します。
- **ENABLE :** 発振
- **DISABLE:** 停止

機能:
STOP モード解除後の低速発振器の動作を選択します。

戻り値:
なし

5.2.3.21 CG_GetExitStopModeFsState

STOP モード解除後の低速発振器の動作選択状態の取得

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

引数:
なし

機能:
STOP モード解除後の低速発振器の動作選択状態を取得します。

戻り値:
STOP モード解除後の低速発振器の動作選択状態
- **ENABLE :** 発振
- **DISABLE:** 停止

5.2.3.22 CG_SetPinStateInStopMode

STOP モード中の端子状態の設定

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

引数:
NewState:
- **DISABLE:** STOP モード中端子をドライブしません
- **ENABLE:** STOP モード中端子をドライブします
  STOP モード中の端子状態制御については、MCU データシートの“低消費電力モード”を参照してくださいい。

機能:
STOP モード時の端子状態を設定します。

戻り値:
なし

5.2.3.23 CG_GetPinStateInStopMode

STOP モード中の端子状態の取得

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

引数:
なし

機能:
STOP モード中の端子状態を取得します。

戻り値:
STOP モード中の端子状態:
- **DISABLE**: STOP モード中端子をドライブしません
- **ENABLE**: STOP モード中端子をドライブします

5.2.3.24 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 使用
- **CG_FC_SRC_FPLL**: fpll 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

5.2.3.25 CG_GetFcSrc

fc ソースの設定状態取得

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

引数:
なし

機能:
fc ソースの設定状態を取得します。

戻り値:
fc ソースの設定状態
- **CG_FC_SRC_FOSC** : fosc 使用
- **CG_FC_SRC_FPLL**: fpll 使用

5.2.3.26 CG_SetFsysSrc

システムクロックの選択

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

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

機能:
システムクロックを選択します。
CG_FSYS_SRC_FGEAR を選択する場合、事前に高速発振器(X1)を発振状態にしてください。**CG_FSYS_SRC_FS** を選択する場合、事前に低速発振器(TX1)を発振状態にしてください。上記のようになっていない場合、本 API は **ERROR** を返却します。

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

5.2.3.27 CG_GetFsysSrc

システムクロックの選択状態の取得

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

引数:
なし

機能:
システムクロックの選択状態を取得します。

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

5.2.3.28 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: 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 (M330/M333 のみ選択可能)
- **CG_INT_SRC_CEC_RX** : CEC 受信割り込み (M330/M332 のみ選択可能)
- **CG_INT_SRC_RMC_RX_0** : RMC0 受信割り込み(M330/332 のみ選択可能)
- **CG_INT_SRC_RTC** : RTC 割り込み
- **CG_INT_SRC_6** : INT6(M330/333 のみ選択可能)
- **CG_INT_SRC_7** : INT7(M330/333 のみ選択可能)
- **CG_INT_SRC_RMC_RX_1** : RMC1 受信割り込み(M330 のみ選択可能)

- **CG_INT_SRC_CEC_TX** : CEC 転送割り込み(M330/332 のみ選択可能)

ActiveState: 解除トリガのアクティブ状態を選択します。

割り込み要因	選択できるアクティブレベル	説明
CG_INT_SRC_RTC **CG_INT_SRC_CEC_TX**	**CG_INT_ACTIVE_STATE_FALLING**	↓エッジ
CG_INT_SRC_CEC_RX **CG_INT_SRC_RMC_RX_0** **CG_INT_SRC_RMC_RX_1**	**CG_INT_ACTIVE_STATE_RISGING**	↑エッジ
上記以外	**CG_INT_ACTIVE_STATE_L**	"Low"レベル
	CG_INT_ACTIVE_STATE_H	"High"レベル
	CG_INT_ACTIVE_STATE_FALLING	↓エッジ
	CG_INT_ACTIVE_STATE_RISGING	↑エッジ
	CG_INT_ACTIVE_STATE_BOTH_EDGES	両エッジ

NewState: 解除トリガの有効/無効を選択します。

- **ENABLE**: 許可
- **DISABLE:** 禁止

機能:

スタンバイモードの解除割り込みソースを設定します。

戻り値:

なし

5.2.3.29 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: 解除割り込みソースの選択
CG_INT_SRC_0, CG_INT_SRC_1, CG_INT_SRC_2,
CG_INT_SRC_3, CG_INT_SRC_4,
CG_INT_SRC_5(M330/333 のみ選択可能),
CG_INT_SRC_CEC_RX (M330/332 のみ選択可能),
CG_INT_SRC_RMC_RX_0 (M330/332 のみ選択可能),
CG_INT_SRC_RTC, CG_INT_SRC_6 (M330/333 のみ選択可能),
CG_INT_SRC_7 (M330/333 のみ選択可能),
CG_INT_SRC_RMC_RX_1 (M330 のみ選択可能)
CG_INT_SRC_CEC_TX (M330/332 のみ選択可能)

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_FALLING**: ↓エッジ
- **CG_INT_ACTIVE_STATE_RISING**: ↑エッジ
- **CG_INT_ACTIVE_STATE_BOTH_EDGES**: 両エッジ
- **CG_INT_ACTIVE_STATE_INVALID**: 無効な値

5.2.3.30 CG_ClearINTReq

スタンバイ解除割り込み要求のクリア

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

引数:
INTSource: 解除割り込みソースを選択します。
CG_INT_SRC_0, CG_INT_SRC_1, CG_INT_SRC_2,
CG_INT_SRC_3, CG_INT_SRC_4,
CG_INT_SRC_5(M330/333 のみ選択可能),
CG_INT_SRC_CEC_RX (M330/332 のみ選択可能),
CG_INT_SRC_RMC_RX_0 (M330/332 のみ選択可能),
CG_INT_SRC_RTC, CG_INT_SRC_6 (M330/333 のみ選択可能),
CG_INT_SRC_7 (M330/333 のみ選択可能),
CG_INT_SRC_RMC_RX_1 (M330 のみ選択可能)
CG_INT_SRC_CEC_TX (M330/332 のみ選択可能)

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

5.2.3.31 CG_GetNMIFlag

NMI 発生要因フラグの取得

関数のプロトタイプ宣言:
CG_NMIFactor

CG_ GetNMIFlag (void)

引数:
なし

機能:
NMI 発生要因フラグを取得します。

戻り値:
NMI 発生要因
- **WDT** (Bit 0) :WDT による NMI 発生
- **NMIPin** (Bit 1) :NMI 端子による NMI 発生

5.2.3.32 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:
- **PowerOn** (Bit0) パワーオンによるリセット
- **ResetPin** (Bit1) RESET 端子によるリセット
- **WDTReset** (Bit 2) WDT によるリセット
- **DebugReset** (Bit 4) <SYSRESETREQ>によるリセット

5.2.4 データ構造

5.2.4.1 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
NMIPin(Bit 1)　　NMI 端子による NMI 発生

uint32_t
Reserved (Bit2~bit31)　未使用

5.2.4.2 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
PowerOn (Bit0)　　パワーオンによるリセット

uint32_t
ResetPinBit1)　　RESET 端子によるリセット

uint32_t
WDTReset(Bit2)　　WDT によるリセット

uint32_t
Reserved (Bit3)　　未使用

uint32_t
DebugReset(Bit4)　<SYSRESETREQ>によるリセット

uint32_t
Reserved (Bit5~bit31) 未使用

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 TMPM330FDFG、TMPM333FDFG の場合 512Kbyte、
TMPM332FWFG の場合 128Kbyte です。

オンボードプログラミングにおいて、CPU はソフトウェアを実行し、flash メモリへのデータ書き込み / 削除 を行います。データ書き込み / 削除は JEDEC 標準型コマンドに従って行います。また、Flash メモリをモニタ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\TX03_Periph_Driver\src\tmpm33x_fc.h(*)
\Libraries\TX03_Periph_Driver\inc\tmpm33x_fc.h(*)

補足: 0, 2, 3 を"x"と表します。

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(FC_BlockNum ***BlockNum***)

6.2.2 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) セキュリティ設定:
   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().

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 (FCFLCS<BLPRO>)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。

FCSECBIT<SECBIT>はコールドリセットで初期化されます。

戻り値:
なし

6.2.3.2 FC_GetSecurityBit

セキュリティビットの設定状態の取得

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

引数:
なし

機能:
セキュリティビットの設定状態を取得します。

戻り値:
セキュリティビットの設定状態:
- **DISABLE**: セキュリティ機能設定不可

- **ENABLE**: セキュリティビット設定可能

6.2.3.3 FC_GetBusyState

自動動作状態の取得

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

引数:
なし

機能:
自動動作状態を取得します。

戻り値:
自動動作状態:
- **BUSY**: 自動動作中
- **DONE**: 自動動作終了

6.2.3.4 FC_GetBlockProtectState

ブロックのプロテクト状態の取得

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

引数:
BlockNum: ブロック番号を選択します。
- **FC_BLOCK_0** block 0
- **FC_BLOCK_1** block 1
- **FC_BLOCK_2** block 2
- **FC_BLOCK_3** block 3
- **FC_BLOCK_4** block 4(M330FDFG, M333 FDFG のみ選択可能です)
- **FC_BLOCK_5** block 5(M330FDFG, M333 FDFG のみ選択可能です)

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

6.2.4 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/TX03_Periph_Driver/src/tmpm36**x**_gpio.c(*)
/Libraries/TX03_Periph_Driver/inc/tmpm36**x**_gpio.h(*)

補足: 1, 2, 3, 4 を”x"と表します。

7.2 TMPM330, TMPM332, TMPM333 の相違点

TMPM330/TMPM333 の入出力ポート: Port A~Port K
TMPM332 の入出力ポート: Port A~Port B, Port D~Port K

7.3 API 関数

7.3.1 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***)

7.3.2 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()

7.3.3 関数仕様

7.3.3.1 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 (TMPM330/TMPM333 のみ選択可能)
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K

機能:

DATA レジスタを読み込みます。

戻り値:
DATA レジスタの値

7.3.3.2 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 (TMPM330/TMPM333 のみ選択可能)
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K

Bit_x: 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 端子値
- **GPIO_BIT_VALUE_0**: 0

- **GPIO_BIT_VALUE_1**: 1

7.3.3.3 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_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K

Data: DATA レジスタへのライトデータを指定します。

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

7.3.3.4 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_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K

Bit_x: 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

BitValue: 設定ビットを指定します。

- **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

7.3.3.5 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 (TMPM330/TMPM333 のみ選択可能)
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K

Bit_x: 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
- **GPIO_BIT_ALL:** GPIO pin0~pin7

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

7.3.3.6 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_PE:** GPIO port E

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

Bit_x: 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
- **GPIO_BIT_ALL:** GPIO pin0~pin7

機能:
出力ポートに設定します。

戻り値:
なし

7.3.3.7 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 (TMPM330/TMPM333 のみ選択可能)
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI:** GPIO port I

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

Bit_x: 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
- **GPIO_BIT_ALL:** GPIO pin0~pin7

機能:

入力ポートに設定します。

戻り値:

なし

7.3.3.8 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_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K

Bit_x: 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
- **GPIO_BIT_ALL:** GPIO pin0~pin7

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** の時は出力許可、***NewState*** が **DISABLE** の時は出力禁止です。

戻り値:

なし

7.3.3.9 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 (TMPM330/TMPM333 のみ選択可能)
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K

Bit_x: 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
- **GPIO_BIT_ALL:** GPIO pin0~pin7

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** の時は入力許可、***NewState*** が **DISABLE** の時は入力禁止です。

戻り値:

なし

7.3.3.10 GPIO_SetPullUp

内蔵プルアップの設定

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 (TMPM330/TMPM333 のみ選択可能)
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K

Bit_x: 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
- **GPIO_BIT_ALL:** GPIO pin0~pin7

NewState:

- **ENABLE:** 内蔵プルアップ有効
- **DISABLE:** 内蔵プルアップ無効

機能:

GPIO 端子の内蔵プルアップ有効/無効を設定します。***NewState*** が **ENABLE** の時は内蔵プルアップ許可、***NewState*** が **DISABLE** の時は内蔵プルアップ禁止です。

戻り値:

なし

7.3.3.11 GPIO_SetPullDown

内蔵プルダウンの設定

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

Bit_x: GPIO 端子を選択します。

- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_ALL:** GPIO pin0

NewState:

- **ENABLE:** 内蔵プルダウン有効
- **DISABLE:** 内蔵プルダウン無効

機能:

GPIO 端子の内蔵プルダウン有効/無効を設定します。***NewState*** が **ENABLE** の時は内蔵プルダウン許可、***NewState*** が **DISABLE** の時は内蔵プルダウン禁止です。

戻り値:
なし

7.3.3.12 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_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
- **GPIO_BIT_ALL:** GPIO pin0~pin7

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

機能:
GPIO 端子のオープンドレイン有効/無効を設定します。***NewState*** が **ENABLE** の時はオープンドレイン許可、***NewState*** が **DISABLE** の時は CMOS 許可です。

戻り値:
なし

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

FuncReg_x: 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。

- **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

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

FuncReg_x: 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。

- **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

7.3.4 データ構造

7.3.4.1 GPIO_InitTypeDef

メンバ:

uint8_t

IOMode ポートの入出力を選択します。

- **GPIO_INPUT:** 入力ポートに設定します。
- **GPIO_OUTPUT:** 出力ポートに設定します。
- **GPIO_IO_MODE_NONE:** 入出力モードを変更しません。

uint8_t

PullUp 内蔵プルアップの有効/無効を選択します。

- **GPIO_PULLUP_ENABLE**: 内蔵プルアップを有効にします。
- **GPIO_PULLUP_DISABLE**: 内蔵プルアップを無効にします。
- **GPIO_PULLUP_NONE**: 内蔵プルアップ機能がない、または設定変更しません。

uint8_t

PullDown 内蔵プルダウンの有効/無効を選択します。

- **GPIO_PULLDOWN_ENABLE:** 内蔵プルダウンを有効にします。
- **GPIO_PULLDOWN_DISABLE:** 内蔵プルダウンを無効にします。
- **GPIO_PULLDOWN_NONE:** 内蔵プルダウン機能がない、または設定変更しません。

uint8_t

OpenDrain オープンドレインポート/CMOS ポートを選択します。

- **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、または設定変更しません。

8. RMC

8.1 概要

本デバイスはリモコン判定機能(RMC)を内蔵しています。

リモコン受信:
- サンプリングクロックは低周波クロック(32KHz)とタイマ出力を選択可能。
- ノイズキャンセル時間を調整可能。
- リーダ検出。
- 最大 72bit まで一括受信。

RMC ドライバ API ではチャネル毎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/TX03_Periph_Driver/src/tmpm33**x**_rmc.c(*)
/Libraries/TX03_Periph_Driver/inc/tmpm33**x**_rmc.h(*)

補足: "**x**"は 0、2 を表します。

8.2 TMPM330, TMPM332, TMPM333 の相違点

TMPM330: 2 チャネル (RMC0, RMC1)
TMPM332: 1 チャネル (RMC0)
TMPM333: ありません

8.3 API 関数

8.3.1 関数一覧

- void RMC_Enable(TSB_RMC_TypeDef * ***RMCx***);
- void RMC_Disable(TSB_RMC_TypeDef * ***RMCx***);
- void RMC_Init(TSB_RMC_TypeDef * ***RMCx***, RMC_InitTypeDef * ***RMC_InitStruct***);
- void RMC_SetRxCtrl(TSB_RMC_TypeDef * ***RMCx***, FunctionalState ***NewState***);
- RMC_RxDataTypeDef RMC_GetRxData(TSB_RMC_TypeDef * ***RMCx***);
- void RMC_SetLeaderDetection(TSB_RMC_TypeDef * ***RMCx***,
  RMC_LeaderParameterTypeDef ***LeaderPara***);
- void RMC_SetFallingEdgeINT(TSB_RMC_TypeDef * ***RMCx***,
  FunctionalState ***NewState***);

- void RMC_SetSignalRxMethod(TSB_RMC_TypeDef * ***RMCx***,
  RMC_RxMethod ***Method***);
- void RMC_SetRxTrg(TSB_RMC_TypeDef * ***RMCx***, uint8_t ***LowWidth***,
  uint8_t ***MaxDataBitCycle***);
- void RMC_SetThreshold(TSB_RMC_TypeDef * ***RMCx***, uint8_t ***LargerThreshold***,
- uint8_t ***SmallerThreshold***);
- void RMC_SetInputSignalReversed(TSB_RMC_TypeDef * ***RMCx***,
  FunctionalState ***NewState***);
- void RMC_SetNoiseCancellation(TSB_RMC_TypeDef * ***RMCx***,
  uint8_t ***NoiseCancellationTime***);
- RMC_INTFactor RMC_GetINTFactor(TSB_RMC_TypeDef * ***RMCx***);
- RMC_LeaderDetection RMC_GetLeader(TSB_RMC_TypeDef * ***RMCx***);

8.3.2 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) RMC の初期化と設定:
   RMC_Enable(), RMC_Disable(), RMC_Init(), RMC_SetRxCtrl()
2) RMC 基本状態の設定:
   SetLeaderDetection (), SetFallingEdgeINT (), RMC_SetSignalRxMethod(), RMC_
   SetRxTrg(), RMC_ SetThreshold(), RMC_SetInputSignalReversed(), RMC_
   SetNoiseCancellation()
3) その他:
   RMC_GetINTFactor(), RMC_GetLeader(), RMC_GetRxData()

8.3.3 関数仕様

補足:** 引数“TSB_RMC_TypeDef * ***RMCx”は以下を指定してください。
TMPM330: TSB_RMC0, TSB_RMC1 のいずれか
TMPM332: TSB_RMC0

8.3.3.1 RMC_Enable

RMC 機能の許可

関数のプロトタイプ宣言:
void
RMC_Enable(TSB_RMC_TypeDef * ***RMCx***)

引数:
RMCx : RMC チャネルを指定します。

機能:

RMC 機能を許可します。

戻り値:
なし

8.3.3.2 RMC_Disable

RMC 機能の禁止

関数のプロトタイプ宣言:
void
RMC_Disable(TSB_RMC_TypeDef * ***RMCx***)

引数:
RMCx : RMC チャネルを指定します。

機能:
RMC 機能を禁止します。

戻り値:
なし

8.3.3.3 RMC_Init

RMC レジスタの初期化

関数のプロトタイプ宣言:
void
RMC_Init(TSB_RMC_TypeDef * ***RMCx***, RMC_InitTypeDef * ***RMC_InitStruct***)

引数:
RMCx : RMC チャネルを指定します。.
RMC_InitStruct : RMC 動作の初期値です。(詳細は“データ構造説明”を参照)

機能:
RMC チャネルの初期化を行います。

戻り値:
なし

8.3.3.4 RMC_SetRxCtrl

受信動作の設定

関数のプロトタイプ宣言:
void
RMC_SetRxCtrl(TSB_RMC_TypeDef * ***RMCx***, FunctionalState ***NewState***)

引数:
RMCx : RMC チャネルを指定します。
NewState: RMC 機能の受信動作を指定します。

- **ENABLE :** 許可。
- **DISABLE:** 禁止。

機能:
RMC 判定機能動作の許可/禁止を選択します。

戻り値:
なし

8.3.3.5 RMC_GetRxData

受信データの取得

関数のプロトタイプ宣言:
RMC_RxDataTypeDef
RMC_GetRxData(TSB_RMC_TypeDef * ***RMCx***)

引数:
RMCx : RMC チャネルを指定します。

機能:
受信データを取得します。

戻り値:
RMC_RxDataDef: RMC 受信バッファの構造体。(詳細は“データ構造説明”を参照)

8.3.3.6 RMC_ SetLeaderDetection

リーダ検出の設定

関数のプロトタイプ宣言:
void

RMC_SetLeaderDetection(TSB_RMC_TypeDef * ***RMCx***,
RMC_LeaderParameterTypeDef ***LeaderPara***)

引数:
RMCx : RMC チャネルを指定します。
LeaderPara: リーダ検出を設定します。(詳細は“データ構造説明”を参照)

機能:
RMC リーダ検出を設定します。

戻り値:
なし

8.3.3.7 RMC_SetFallingEdgeINT

リモコン入力立下りエッジ割り込み発生の許可

関数のプロトタイプ宣言:
void
RMC_SetFallingEdgeINT(TSB_RMC_TypeDef * ***RMCx***,
FunctionalState ***NewState***)

引数:
RMCx: RMC チャネルを指定します。
NewState: リモコン入力立下りエッジ割り込み発生の許可/禁止を選択します。

- **ENABLE :** 許可。
- **DISABLE:** 禁止。

機能:
NewState が **ENABLE** の場合、リモコン入力立ち下がりエッジ割り込みが有効になります。***NewState*** が **DISABLE** の場合、無効になります。

戻り値:
なし

8.3.3.8 RMC_SetSignalRxMethod

位相方式のリモコン受信モード選択

関数のプロトタイプ宣言:
void
RMC_SetSignalRxMethod(TSB_RMC_TypeDef * ***RMCx***,

RMC_RxMethod ***Method***)

引数:
RMCx : RMC チャネルを指定します。
Method: 位相方式のリモコン受信モードを選択します。
- **RMC_RX_IN_CYCLE_METHOD**: 周期方式で受信。
- **RMC_RX_IN_PHASE_METHOD**: 位相方式で受信。

機能:
位相方式のリモコン受信モードを選択します。

戻り値:
なし

8.3.3.9 RMC_SetRxTrg

受信終了/割り込み設定

関数のプロトタイプ宣言:
void
RMC_SetRxTrg(TSB_RMC_TypeDef * ***RMCx***,
uint8_t ***LowWidth***,
uint8_t ***MaxDataBitCycle***)

引数:
RMCx : RMC チャネルを指定します。
LowWidth: Low 幅の検出による受信終了/割込み発生のタイミングを設定します。
MaxDataBitCycle: データビットの周期 MAX で受信終了/割り込みを設定します。

機能:
RMC チャネルのトリガ設定を行います。
LowWidth を RMCRCR2<RMCLL7:0> に設定した場合は、Low 幅の検出による受信終了/割込み発生のタイミングを設定します。Low 幅検出時に受信が完了し、割り込みが発生します。<RMCLL7:0> = 11111111b の時は検出しません。
計算式: RMCLLx1/fs[s]
MaxDataBitCycle を RMCRCR2<RMCDMAX7:0> に設定した場合は、データ bit の周期 MAX 検出のしきい値を設定します。データ bit 周期の値がしきい値以上であれば検出となります。<RMCMAX7:0> = 11111111b の時は検出しません。
計算式:RMCDMAX x 1/fs[s]

戻り値:
なし

8.3.3.10 RMC_SetThreshold

位相方式のしきい値の設定

関数のプロトタイプ宣言:

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

引数:

RMCx : RMC チャネルを指定します。

LargerThreshold:位相方式のリモコン信号の3値判定の1.5Tと 2T のしきい値の設定をします。 データビットの測定結果がしきい値以上でデータを“10”、しきい値未満でデータ“01” と判別します。

しきい値計算式:RMCDATHx1/fs[s]

LargerThreshold には0x80 より小さい値を設定してください。

SmallerThreshold: 2種類のしきい値の設定:データビットの0/1 判定のしきい値および、位相方式のリモコン信号の3 値判定の1T と1.5T のしきい値の設定をします。
データビットの0/1 判定の場合、測定結果がしきい値以上でデータ"1"、しきい値未満でデータ"0"と判別します。

しきい値の計算式: RMCDATL×1/fs[s]

位相方式のリモコン信号の3 値判定の場合、データビットの測定結果がしきい値以上でデータを"01"、しきい値未満でデータ"00"と判別します。

データビットの0/1 判定: RMCDATLx1/fs[s]

RMCRCR3<RMCDATH0-6> <RMCDATL0-6> ビットで設定します。
しきい値下位は 0x80 以下となります。

機能:

位相方式のリモコン信号のしきい値を設定します。本設定が有効になるのは、位相方式のリモコン受信が次のように許可されているときのみです。<RMCPHM> = “1”

戻り値:

なし

8.3.3.11 RMC_SetInputSignalReversed

リモコン入力信号の極性設定

関数のプロトタイプ宣言:

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

引数:
RMCx : RMC チャネルを指定します。
NewState:リモコン入力信号の極性を選択します。

- **ENABLE :** 負極。
- **DISABLE:** 正極。

機能:
NewState が **ENABLE** の場合、RMC チャネルのリモコン入力信号の極性反転は有効(負極)となり、**DISABLE** の時は無効(正極)となります。

戻り値:
なし

8.3.3.12 RMC_SetNoiseCancellation

ノイズ除去時間の設定

関数のプロトタイプ宣言:
void
RMC_SetNoiseCancellation(TSB_RMC_TypeDef * ***RMCx***,
uint8_t ***NoiseCancellationTime***)

引数:
RMCx: RMC チャネルを指定します。
NoiseCancellationTime: ノイズ除去時間を設定します。0x10 よりも小さい値を設定してください。

機能:
ノイズ除去時間を設定します。
<RMCNC3:0> = 0000b の場合は、ノイズを除去しません。
ノイズキャンセル時間の計算式:RMCNC x 1/fs[s]

戻り値:
なし

8.3.3.13 RMC_GetINTFactor

割り込み要因の取得

関数のプロトタイプ宣言:
RMC_INTFactor

RMC_GetINTFactor(TSB_RMC_TypeDef * ***RMCx***)

引数:
RMCx : RMC 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:
RMC_INTFactor: 割り込み要因の構造体です。(詳細は“データ構造説明”を参照)

8.3.3.14 RMC_GetLeader

リーダ検出の取得

関数のプロトタイプ宣言:
RMC_LeaderDetection
RMC_GetLeader(TSB_RMC_TypeDef * ***RMCx***)

引数:
RMCx : RMC チャネルを指定します。

機能:
リーダ検出を取得します。

戻り値:
RMC_LeaderDetection: リーダ検出結果
- **RMC_LEADER_DETECTED**: リーダ検出あり
- **RMC_NO_LEADER**: リーダ検出なし

8.3.4 データ構造

8.3.4.1 RMC_RxDataDef

メンバ:
uint8
***RxDataBits*:** 受信データビット数

uint32_t
***RxBuf1*:** 受信バッファ 1(<MCRBUF31:0>から 4 バイトデータを読み出します)

uint32_t
***RxBuf2*:** 受信バッファ 2(<MCRBUF63:32>から 4 バイトデータを読み出します)

uint8_t
***RxBuf3*:** 受信バッファ 3(<MCRBUF71:64>から 1 バイトデータを読み出します)

8.3.4.2 RMC_LeaderParameterTypeDef

メンバ:
FunctionalState
***LeaderDetectionState*:** リーダ検出のあり/なしを選択します。
- **ENABLE:** リーダ検出あり。
- **DISABLE:** リーダ検出なし。

uint8_t
***MaxCycle*:** リーダ検出の周期期間の上限。

uint8_t
***MinCycle*:** リーダ検出の周期期間の下限。

uint8_t
***MaxLowWidth*:** リーダ検出の LOW 期間の上限。

uint8_t
***MinLowWidth*:** リーダ検出の LOW 期間の下限。

FunctionalState
***LeaderINTState*:** リーダ検出割り込み発生の許可/禁止を選択します。
- **ENABLE**: 割り込み発生する。
- **DISABLE**: 割り込み発生しない。

8.3.4.3 RMC_InitTypeDef

メンバ:
RMC_LeaderParameterTypeDef
***LeaderPara*:** リーダ検出設定

FunctionalState
***FallingEdgeINTState*:** リモコン入力立ち下がりエッジ割り込みの有効/無効を選択します。
- **ENABLE**: 割り込み発生する。
- **DISABLE**: 割り込み発生しない。

RMC_RxMethod

***SignalRxMethod*:** 位相方式のリモコン受信モードを設定します。

- **RMC_RX_IN_CYCLE_METHOD**:周期方式で受信。
- **RMC_RX_IN_PHASE_METHOD**:位相方式で受信。

FunctionalState

***InputSignalReversedState*:** リモコン入力信号の極性選択を選択します。

- **ENABLE**: 負極。
- **DISABLE**: 正極。

uint8_t

***NoiseCancellationTime*:** ノイズ除去時間を設定します。0x10 よりも小さい値を設定してください。

uint8_t

***LowWidth*:** Low 幅の検出による受信終了/割込み発生のタイミングを設定します。

uint8_t

***MaxDataBitCycle*:** 受信終了/割り込み発生の周期の最大値を設定します。

uint8_t

***LargerThreshold*:** 位相方式のリモコン信号におけるデータビットの 3 値判定のしきい値の上位を設定します。0x80 より小さい値を設定してください。

uint8_t

***SmallerThreshold*:** 位相方式のリモコン信号におけるデータビットの 0/1 判別および 3 値判定のしきい値の下位を設定します。0x80 より小さい値を設定してください。

8.3.4.4 RMC_INTFactor

メンバ:

uint32_t

All: すべてのデータ

ビットフィールド:

uint32_t

Reserved : 12 未使用

uint32_t

InputFallingEdge : 1 立ち下がりエッジ割り込み要因フラグ

uint32_t

MaxDataBitCycle : 1 データビット周期 MAX 割り込み要因フラグ

uint32_t
LowWidthDetection : 1 Low 幅検出割り込み要因フラグ

uint32_t
LeaderDetection : 1 リーダ検出割り込み要因フラグ

9. RTC

9.1 概要

RTC の機能概略は以下です。

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

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

本ドライバ は、アプリで使用する API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm33**x**_rtc.c(*)
/Libraries/ TX03_Periph_Driver/inc/tmpm33**x**_rtc.h(*)

補足: "**x**"は 0、2、3 を表します。

9.2 TMPM330, TMPM332, TMPM333 の相違点

なし。

9.3 API 関数

9.3.1 関数一覧

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

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

9.3.2 関数の種類

関数は、主に以下の 5 種類に分かれています:

1) RTC 機能の年月日の設定:
   RTC_SetDay(), RTC_GetDay(), RTC_SetDate(), RTC_GetDate(), RTC_SetMonth(), RTC_GetMonth(), RTC_SetYear(), RTC_GetYear(), RTC_SetLeapYear(), RTC_GetLeapYear(), RTC_SetDateValue(), RTC_GetDateValue()
2) RTC 機能の時間の設定:
   RTC_SetSec(), RTC_GetSec(), RTC_SetMin(), RTC_GetMin(), RTC_SetHour24(), RTC_SetHour12(), RTC_GetHour(), RTC_SetHourMode(), RTC_GetHourMode, RTC_GetAMPM(), RTC_SetTimeValue(), RTC_GetTimeValue()
3) RTC(clock)の設定:
   RTC_EnableClock(), RTC_DisableClock(), RTC_SetTimeAdjustReq(),

RTC_GetTimeAdjustReq(), RTC_ResetClockSec(), RTC_GetResetClockSec(), RTC_SetClockValue(), RTC_GetClockValue()

4) RTC(alarm)の設定:
RTC_EnableAlarm(), RTC_DisableAlarm(), RTC_ResetAlarm(), RTC_SetAlarmValue(), RTC_GetAlarmValue()

5) その他:
RTC_SetAlarmOutput(), RTC_SetRTCINT()

9.3.3 関数仕様

9.3.3.1 RTC_SetSec

時計の秒桁設定

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

引数:
Sec:最大 59 までの秒桁設定の値。

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

戻り値:
なし

9.3.3.2 RTC_GetSec

時計の秒桁設定

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

引数:
なし。

機能:
時計の秒桁の値を返します。

戻り値:

時計の秒桁:

- 0 ~ 59

9.3.3.3 RTC_SetMin

時計/アラームの分析設定

関数のプロトタイプ宣言:

void
RTC_SetMin(RTC_FuncMode ***NewMode***,
uint8_t ***Min***);

引数:

NewMode: RTC モードを選択します。

- **RTC_CLOCK_MODE**: 時計機能
- **RTC_ALARM_MODE**: アラーム機能

Min: 最大 59 までの分析を設定します。

機能:

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

戻り値:

なし

9.3.3.4 RTC_GetMin

時計/アラームの分析読み込み

関数のプロトタイプ宣言:

uint8_t
RTC_GetMin(RTC_FuncMode ***NewMode***);

引数:

NewMode: RTC モードを選択します。

- **RTC_CLOCK_MODE**: 時計機能
- **RTC_ALARM_MODE**: アラーム機能

機能:

NewMode が **RTC_CLOCK_MODE** の場合、時計の分析の値を返します。

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

戻り値:
分析:
- 0 ~ 59

9.3.3.5 RTC_GetAMPM

12 時間モードの AM/PM 読み込み

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

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

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

戻り値:
時計モード:
RTC_AM_MODE: AM
RTC_PM_MODE: PM

9.3.3.6 RTC_SetHour24

24 時間モードの時計/アラーム時析設定

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

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

Hour: 最大 23 までの時桁を設定します。

機能:
24 時間モードの時計/アラームの時桁を設定します。
NewMode が **RTC_CLOCK_MODE** の場合、時計機能の時桁を設定し、
NewMode が **RTC_ALARM_MODE** の場合、アラームの時桁を設定します。
RTC レジスタは、INTRTC のタイミングに同期して書換えられます。この関数の実行後、1Hz 割り込みが発生するのを待つ必要があります。
*12 時間モードから 24 時間モードに変更する場合、本関数 **RTC_SetHour24()** によって HOURR レジスタを再設定してください。

戻り値:
なし

9.3.3.7 RTC_SetHour12

12 時間モードの時計/アラーム時桁設定

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

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

Hour: 最大 11 までの時桁を設定します。
AmPm: 以下から時間モードを選択します。
- **RTC_AM_MODE**: 12H モードの AM モード
- **RTC_PM_MODE**: 12H モードの PM モード

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

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

戻り値:
なし

9.3.3.8 RTC_GetHour

時計/アラームの時桁読み込み

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

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

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

戻り値:
24 時間モードでの時桁:
- 0 ~ 23

12H 時間モードでの時桁:
- 0 ~ 11

9.3.3.9 RTC_SetDay

時計/アラームの曜日設定

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

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

Day:曜日を選択します。
- **RTC_SUN:** 日曜日

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

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

戻り値:
なし

9.3.3.10 RTC_GetDay

時計/アラームの曜日の読み込み

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

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

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

戻り値:
曜日の値:
- 0 ~ 6

9.3.3.11 RTC_SetDate

時計/アラームの日桁設定

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

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

Date: 1 から 31 の日桁を設定します。

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

戻り値:
なし

9.3.3.12 RTC_GetDate

時計/アラームの日桁読み込み

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

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

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

戻り値:
日桁:
- 1 ~ 31

9.3.3.13 RTC_SetMonth

時計の月桁設定

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

引数:
Month: 1 から 12 の月桁を設定します。

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

戻り値:
なし

9.3.3.14 RTC_GetMonth

時計の月桁読み込み

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

引数:
なし。

機能:
時計の月桁の値を返します。

戻り値:
月桁:
- 1 ~ 12

9.3.3.15 RTC_SetYear

時計の年桁設定

関数のプロトタイプ宣言:
void

RTC_SetYear(uint8_t ***Year***);

引数:
Year: 最大 99 までの年の値

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

戻り値:
なし

9.3.3.16 RTC_GetYear

時計の年桁の読み込み

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

引数:
なし。

機能:
時計の年桁の値を返します。

戻り値:
年桁:
- 0 ~ 99

9.3.3.17 RTC_SetHourMode

24 時間時計/12 時間時計の選択

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

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

- **RTC_24_HOUR_MODE**.: 24 時間時計

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

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

戻り値:
なし

9.3.3.18 RTC_GetHourMode

時計モードの読み込み

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

引数:
なし。

機能:
時計モードを読み込みます。

戻り値:
時計モード:
- **RTC_24_HOUR_MODE:** 24 時間時計
- **RTC_12_HOUR_MODE:** 12 時間時計

9.3.3.19 RTC_SetLeapYear

うるう年の設定

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

引数:

LeapYear: 以下からうるう年を選択します。

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

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

戻り値:
なし

9.3.3.20 RTC_GetLeapYear

うるう年の読み込み

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

引数:
なし。

機能:
うるう年の状態を返します。

戻り値:
うるう年の状態を表す値

9.3.3.21 RTC_SetTimeAdjustReq

+/– 30 秒の補正

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

引数:

なし。

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

戻り値:
なし

9.3.3.22 RTC_GetTimeAdjustReq

ADJUST 要求状態の読み込み

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

引数:
なし。

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

戻り値:
ADJUST 要求状態を読み込みます。

- **RTC_NO_REQ :** ADJUST 要求なし
- **RTC_REQ:** ADJUST 要求あり

9.3.3.23 RTC_EnableClock

時計機能の起動

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

引数:
なし。

機能:

時計機能を有効にします。

戻り値:
なし

9.3.3.24 RTC_DisableClock

時計機能の終了

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

引数:
なし。

機能:
時計機能を無効にします。

戻り値:
なし

9.3.3.25 RTC_EnableAlarm

アラーム機能の起動

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

引数:
なし。

機能:
アラーム機能を有効にします。

戻り値:
なし

9.3.3.26 RTC_DisableAlarm

アラーム機能の終了

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

引数:
なし。

機能:
アラーム機能を無効にします。

戻り値:
なし

9.3.3.27 RTC_SetRTCINT

INTRTC 割り込みの有効/無効設定

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

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

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

戻り値:
なし

9.3.3.28 RTC_SetAlarmOutput

ALARM 端子の出力設定

関数のプロトタイプ宣言:
void

RTC_SetAlarmOutput(uint8_t ***Output***);

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

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

戻り値:
なし

9.3.3.29 RTC_ResetClockSec

時計秒カウンタのリセット

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

引数:
なし。

機能:
時計秒カウンタをリセットします。

戻り値:
なし

9.3.3.30 RTC_GetResetClockSecReq

時計秒カウンタのリセット要求状態の読み込み

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

引数:
なし。

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

戻り値:
リセット要求状態

- **RTC_NO_REQ**: リセット要求なし
- **RTC_REQ**: リセット要求あり

9.3.3.31 RTC_ResetAlarm

アラームリセット

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

引数:
なし

機能:
アラームレジスタ(分、時、日、週桁レジスタ)を初期化します。

戻り値:
なし

9.3.3.32 RTC_SetDateValue

時計の日付設定

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

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

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

戻り値:
なし

9.3.3.33 RTC_GetDateValue

時計の日付の読み込み

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

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

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

戻り値:
なし

9.3.3.34 RTC_SetTimeValue

時計の時刻設定

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

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

機能:

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

戻り値:
なし

9.3.3.35 RTC_GetTimeValue

時計の時刻の読み込み

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

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

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

戻り値:
なし

9.3.3.36 RTC_SetClockValue

時計の日時設定

関数のプロトタイプ宣言:
void
RTC_SetClockValue(RTC_DateTypeDef * ***DateStruct***,
RTC_TimeTypeDef * ***TimeStruct***);

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

機能:

時計の日付(うるう年、年、月、曜日、日)、および、時刻(時間モード、時間、12 時間モードの AM/PM モード、分、秒)を設定します。
RTC_SetLeapYear(), **RTC_SetYear()**, **RTC_SetMonth()**, **RTC_SetDate()**, **RTC_SetDay()**, **RTC_SetHourMode()**, **RTC_SetHour24()**, **RTC_SetHour12()**, **RTC_SetMin(), RTC_SetSec()** を実行します。

戻り値:
なし

9.3.3.37 RTC_GetClockValue

時計の日時の読み込み

関数のプロトタイプ宣言:
void
RTC_GetClockValue(RTC_DateTypeDef * ***DateStruct***,
RTC_TimeTypeDef * ***TimeStruct***);

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

機能:
時計の日付(うるう年、年、月、曜日、日)、および、時刻(時間モード、時間、12 時間モードの AM/PM モード、分、秒)を設定します。
RTC_GetLeapYear(), **RTC_GetYear()**, **RTC_GetMonth()**, **RTC_GetDate()**, **RTC_GetDay()**, **RTC_GetHourMode()**, **RTC_GetHour()**,**RTC_GetAMPM()**, **RTC_GetMin(), RTC_GetSec()** を実行します。

戻り値:
なし

9.3.3.38 RTC_SetAlarmValue

アラームの日時設定

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

引数:

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

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

戻り値:
なし

9.3.3.39 RTC_GetAlarmValue

アラームの日時の取得

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

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

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

戻り値:
なし

9.3.4 データ構造

9.3.4.1 RTC_DateTypeDef

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

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

- **RTC_LEAP_YEAR_3:** 現在がうるう年から 3 年目

uint8_t
Year 年桁の値(0～99)。

uint8_t
Month 月桁の値(1～12)。

uint8_t
Date 日桁の値(1～31)。

uint8_t
Day 週の値を設定します。
- **RTC_SUN:** 日曜日
- **RTC_MON:** 月曜日
- **RTC_TUE:** 火曜日
- **RTC_WED:**水曜日
- **RTC_THU:**木曜日
- **RTC_FRI:** 金曜日
- **RTC_SAT:** 土曜日

9.3.4.2 RTC_TimeTypeDef

メンバ:

uint8_t
HourMode 24 時間時計、12 時間時計のモード選択の値:
- **RTC_12_HOUR_MODE:** 12 時間モード
- **RTC_24_HOUR_MODE:** 24 時間モード

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

uint8_t
AmPm 12 時間モード時の AM/PM の値:
- **RTC_AM_MODE:** AM モード
- **RTC_PM_MODE:** PM モード
- **RTC_AMPM_INVALID:** 24 時間モード

uint8_t
Min 0～59 までの分析の値。

uint8_t

Sec 0～59 までの秒桁の値。

9.3.4.3 RTC_AlarmTypeDef

メンバ:

uint8_t

Date アラーム機能有効時の日桁の値(1～31)。

uint8_t

Day アラーム機能有効時の週桁の値。

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

uint8_t

Hour アラーム機能有効時の時間桁の値。

uint8_t

AmPm アラーム機能有効時の AM/PM 選択の値:

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

uint8_t

Min アラーム機能有効時の分桁の値(0～59)。

10. SBI

10.1 概要

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

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

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm33**x**_sbi.c(*)
/Libraries/TX03_Periph_Driver/inc/tmpm33**x**_sbi.h(*)

補足: "**x**"は 0、2、3 を表します。

10.2 TMPM330, TMPM332, TMPM333 の相違点

TMPM330/TMPM333: 3 チャネル内蔵しています。 (SBI0, SBI1, SBI2)
TMPM332: 2 チャネル内蔵しています。 (SBI0, SBI1)

10.3 API 関数

10.3.1 関数一覧

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

- void SBI_SetIdleMode(TSB_SBI_TypeDef* ***SBIx***, FunctionalState ***NewState***)
- void SBI_SetSendData(TSB_SBI_TypeDef* ***SBIx***, uint32_t ***Data***)
- uint32_t SBI_GetReceiveData(TSB_SBI_TypeDef* ***SBIx***)
- void SBI_SetI2CFreeDataMode(TSB_SBI_TypeDef* ***SBIx,***
  FunctionalState ***NewState***)

10.3.2 関数の種類

関数は、主に以下の 4 種類に分かれています。

1) 共通機能の設定:
   SBI_Enable(), SBI_Disable(), SBI_SetI2CACK(), SBI_SetI2CBitNum(), SBI_InitI2C()
2) 転送制御:
   SBI_ClearI2CINTReq(), SBI_GenerateI2Cstart(), SBI_GenerateI2Cstop(),
   SBI_IsI2ClastRxBitSet(), SBI_GetReceiveData()
3) ステータス確認:
   SBI_GetI2CState()
4) その他:
   SBI_SWReset(), SBI_SetIdleMode(), SBI_EnableI2CfreeDataMode()

10.3.3 関数仕様

補足: 下記の全 API において、パラメータ"TSB_SBI_TypeDef* ***SBIx***" は 以下のいずれかを選択してください。

TSB_SBI0, TSB_SBI1, TSB_SBI2 (**TSB_SBI2** は TMPM330/TMPM333 のみ選択可能です)

10.3.3.1 SBI_Enable

SBI 動作の許可

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

引数:
SBIx: SBI チャネルを指定します。

機能:
SBI 動作を有効にします。

戻り値:
なし

10.3.3.2 SBI_Disable

SBI 動作の禁止

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

引数:
SBIx: SBI チャネルを指定します。

機能:
SBI 動作を無効にします。

戻り値:
なし

10.3.3.3 SBI_SetI2CACK

I2C バスモードにおける ACK 選択

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

引数:
SBIx: SBI チャネルを指定します。
NewState: ACK の発生有無を選択します。

- **ENABLE**: 発生する。
- **DISABLE**: 発生しない。

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

戻り値:
なし

10.3.3.4 SBI_InitI2C

I2C バスモードにおける通信の初期化

関数のプロトタイプ宣言:

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

引数:

SBIx: SBI チャネルを指定します。
InitI2CStruct: SBI に関する構造体です。(詳細は"データ構造"を参照)

機能:

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

戻り値:

なし

10.3.3.5 SBI_SetI2CBitNum

I2C バスモードにおける転送ビット数の選択

関数のプロトタイプ宣言:

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

引数:

SBIx: SBI チャネルを指定します。
I2CBitNum: 転送ビット数(1～8)を選択します。

- **SBI_I2C_DATA_LEN_8:** データ長 8
- **SBI_I2C_DATA_LEN_1:** データ長 1
- **SBI_I2C_DATA_LEN_2:** データ長 2
- **SBI_I2C_DATA_LEN_3:** データ長 3
- **SBI_I2C_DATA_LEN_4:** データ長 4
- **SBI_I2C_DATA_LEN_5:** データ長 5
- **SBI_I2C_DATA_LEN_6:** データ長 6
- **SBI_I2C_DATA_LEN_7:** データ長 7

機能:

転送ビット数を選択します。

戻り値:

なし

10.3.3.6 SBI_SWReset

ソフトウェアリセットの発生

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

引数:
SBIx: SBI チャネルを指定します。

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

戻り値:
なし

10.3.3.7 SBI_ClearI2CINTReq

I2C バスモードにおける INTSBIx 割り込み要求解除

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

引数:
SBIx: SBI チャネルを指定します。

機能:
SBI 割り込み要求を解除します。

戻り値:
なし

10.3.3.8 SBI_GenerateI2CStart

I2C バスモードにおけるスタート状態の発生

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

引数:
SBIx: SBI チャネルを指定します。

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

戻り値:
なし

10.3.3.9 SBI_GenerateI2CStop

I2C バスモードにおけるストップ状態の発生

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

引数:
SBIx: SBI チャネルを指定します。

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

戻り値:
なし

10.3.3.10 SBI_GetI2CState

I2C バスモードにおける SBI チャネルの状態の読み込み

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

引数:
SBIx: SBI チャネルを指定します。

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

戻り値:

I2C モードでの SBI チャネルの状態

10.3.3.11 SBI_SetIdleMode

IDLE モード時の動作の許可/禁止

関数のプロトタイプ宣言:

```
void
SBI_SetIdleMode(TSB_SBI_TypeDef* SBIx,
                FunctionalState NewState)
```

引数:

SBIx: SBI チャネルを指定します。
NewState: システムが idle モードの時の動作を指定します。

- **ENABLE:** 許可。
- **DISABLE:** 禁止。

機能:

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

戻り値:

なし

10.3.3.12 SBI_SetSendData

データ送信

関数のプロトタイプ宣言:

```
void
SBI_SetSendData(TSB_SBI_TypeDef* SBIx,
                uint32_t Data)
```

引数:

SBIx: SBI チャネルを指定します。
Data: 送信データ。(最大値は 0xFF です)

機能:

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

戻り値:

なし

10.3.3.13 SBI_GetReceiveData

データ受信

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

引数:
SBIx: SBI チャネルを指定します。

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

戻り値:
Data which has been received

10.3.3.14 SBI_SetI2CFreeDataMode

アドレス認識モードの指定

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

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

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

戻り値:

なし

10.3.4 データ構造

10.3.4.1 SBI_InitI2CTypeDef

メンバ:

uint32_t

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

uint32_t

***I2CDataLen*:** I2C モードにおける SBI チャネルの転送ビット数を指定します。

- **SBI_I2C_DATA_LEN_8:** データ長 8
- **SBI_I2C_DATA_LEN_1:** データ長 1
- **SBI_I2C_DATA_LEN_2:** データ長 2
- **SBI_I2C_DATA_LEN_3:** データ長 3
- **SBI_I2C_DATA_LEN_4:** データ長 4
- **SBI_I2C_DATA_LEN_5:** データ長 5
- **SBI_I2C_DATA_LEN_6:** データ長 6
- **SBI_I2C_DATA_LEN_7:** データ長 7

uint32_t

***I2CClkDiv*:** I2C 転送のソースクロックを選択します。

- **SBI_I2C_CLK_DIV_104:** fsys/104
- **SBI_I2C_CLK_DIV_136:** fsys/136
- **SBI_I2C_CLK_DIV_200:** fsys/200
- **SBI_I2C_CLK_DIV_328:** fsys/328
- **SBI_I2C_CLK_DIV_584:** fsys/584
- **SBI_I2C_CLK_DIV_1096:** fsys/1096
- **SBI_I2C_CLK_DIV_2120:** fsys/2120

FunctionalState

***I2CACKState*:** ACK の有効/無効を選択します。

- **ENABLE:** 有効。
- **DISABLE:** 無効。

10.3.4.2 SBI_I2CState

メンバ:

uint32_t

***All*:** I2C モードの全ての状態

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モニタ
uint32_t
***ArbitrationLost*:** アービトレーションロスト検出モニタ
uint32_t
***INTReq*:** 割り込み要求状態モニタ
uint32_t
***BusState*:** バス状態モニタ
uint32_t
***TRx*:** 送信/受信選択状態モニタ
uint32_t
***MasterSlave*:** マスタ/スレーブ選択状態モニタ

11. 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 (TB0 ～ TB9)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
- 時間差測定

本デバイスは、16 ビットの多目的タイマ (MPT)を内蔵しており、MPT はタイマーモードで動作する場合、TMRB と同一の動作を行います。

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/TX03_Periph_Driver/src/tmpm33**x**_tmrb.c(*)
/Libraries/TX03_Periph_Driver/inc/tmpm33**x**_tmrb.h(*)

補足: "**x**"は 0, 2, 3 を表します。

11.2 TMPM330, TMPM332, TMPM333 の相違点

ありません。

11.3 API 関数

11.3.1 関数一覧

◆ void TMRB_Enable(TSB_TB_TypeDef* ***TBx***)

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

11.3.2 関数の種類

関数は、主に以下の 4 種類に分かれています:

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

11.3.3 関数仕様

補足:** 引数に記述されている “TSB_TB_TypeDef* ***TBx” は下記から選択してください。
TSB_TB0, TSB_TB1, TSB_TB2, TSB_TB3, TSB_TB4, TSB_TB5, TSB_TB6, TSB_TB7, TSB_TB8, TSB_TB9

11.3.3.1 TMRB_Enable

TMRB 動作の許可

関数のプロトタイプ宣言:
void

TMRB_Enable(TSB_TB_TypeDef* ***TBx***)

引数:
***TBx*:** TMRB チャネルを指定します。

機能:
TMRB 動作を有効にします。

戻り値:
なし

11.3.3.2 TMRB_Disable

TMRB 動作の禁止

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

引数:
***TBx*:** TMRB チャネルを指定します。

機能:
TMRB 動作を無効にします。

戻り値:
なし

11.3.3.3 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

11.3.3.4 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

11.3.3.5 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_TB2, TSB_TB3, TSB_TB4, TSB_TB5, TSB_TB6

CaptureTiming: キャプチャタイミングを選択します。

- **TMRB_CAPTURE_IN_RISING**: 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、TBxIN1 端子入力の立ち上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。
- **TMRB_CAPTURE_IN0_RISING_IN1_FALLING**: 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。(*)
- **TMRB_CAPTURE_OUTPUT_EDGE**: 16 ビットタイマー致出力(TBxOUT)の立ち上がりでキャプチャレジスタ 0 (TBxCP0)にカウント値を取り込み、TBxOUT の立ち下がりでキャプチャレジスタ 1 (TBxCP1)にカウント値を取り込みます。(**)
- **TMRB_DISABLE_CAPTURE**: キャプチャ禁止

機能:
キャプチャタイミングとアップカウンタのクリアタイミングを設定します。

戻り値:
なし

補足:
(*) TMRB7, TMRB8, TMRB9 に TBxIN0/TBxIN1 端子入力はありません。
(**)TMRB0~1: TB7OUT、TMRB2~4: TB8OUT、TMRB5~6: TB9OUT

11.3.3.6 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

11.3.3.7 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
}
```

11.3.3.8 TMRB_SetINTMask

割り込みマスクの設定

関数のプロトタイプ宣言:

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

引数:

***TBx*:** TMRB チャネルを指定します。

***INTMask*:** マスクする割り込みを選択します。

- **TMRB_MASK_MATCH_TRAILINGTIMING_INT**: 一致 (TBxRG0) 割り込み
- **TMRB_MASK_MATCH_LEADINGTIMING_INT**: 一致 (TBxRG1) 割り込み
- **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_TRAILINGTIMING_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_LEADINGTIMING_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、オーバー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

11.3.3.9 TMRB_ChangeLeadingTiming

デューティの設定

関数のプロトタイプ宣言:

```
void
TMRB_ ChangeLeadingTiming(TSB_TB_TypeDef* TBx,
                          uint32_t LeadingTiming)
```

引数:

***TBx*:** TMRB チャネルを指定します。

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

機能:

デューティを設定します。実際のデューティのインターバルは、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。

11.3.3.10 TMRB_ChangeTrailingTiming

周期の設定

関数のプロトタイプ宣言:

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

引数:

***TBx*:** TMRB チャネルを指定します。
TrailingTiming: 周期を設定します。最大は 0xFFFF です。

機能:

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

戻り値:

なし

補足:

TrailingTiming は ***LeadingTiming*** より小さくすることはできません。 また PPG モード時、TBxRG0/1 は TBxRG0 < TBxRG1 を満たす必要があります。

11.3.3.11 TMRB_GetUpCntValue

アップカウンタ値の読み込み

関数のプロトタイプ宣言:

uint16_t
TMRB_GetUpCntValue(TSB_TB_TypeDef* ***TBx***)

引数:

***TBx*:** TMRB チャネルを指定します。

機能:

アップカウンタ値の読み込みを行います。

戻り値:

アップカウンタ値

11.3.3.12 TMRB_GetCaptureValue

キャプチャレジスタの読み込み

関数のプロトタイプ宣言:

uint16_t
TMRB_GetCaptureValue(TSB_TB_TypeDef* ***TBx***,
uint8_t ***CapReg***)

引数:

***TBx*:** TMRB チャネルを指定します。

CapReg: キャプチャレジスタを選択します。

- **TMRB_CAPTURE_0:** キャプチャレジスタ 0
- **TMRB_CAPTURE_1:** キャプチャレジスタ 1

機能:

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

戻り値:

キャプチャレジスタの値

11.3.3.13 TMRB_ExecuteSWCapture

ソフトウェアキャプチャの実行

関数のプロトタイプ宣言:

void
TMRB_ExecuteSWCapture(TSB_TB_TypeDef* ***TBx***)

引数:

***TBx*:** TMRB チャネルを指定します。

機能:

キャプチャレジスタ 0 (TBxCP0)にカウント値を取り込みます。

戻り値:

なし

11.3.3.14 TMRB_SetIdleMode

IDLE 時の動作設定

関数のプロトタイプ宣言:

void
TMRB_SetIdleMode(TSB_TB_TypeDef* ***TBx***,
FunctionalState ***NewState***)

引数:

***TBx*:** TMRB チャネルを指定します。

NewState: IDLE 時の動作を指定します。

- **ENABLE:** 動作
- **DISABLE:** 停止

機能:

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

戻り値:

なし

11.3.3.15 TMRB_SetSyncMode

同期モードの切り替え

関数のプロトタイプ宣言:

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

引数:

***TBx*:** TMRB チャネルを以下から選択します。

TSB_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TMRB1 ～ TMRB3、TMRB5 ～ TMRB7をスタートしてください。

11.3.3.16 TMRB_SetDoubleBuf

ダブルバッファ動作の制御

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

引数:
***TBx*:** TMRB チャネルを指定します。
***NewState*:** ダブルバッファの有効/無効を選択します。
- **ENABLE:** 有効。
- **DISABLE:** 無効。

機能:
TBxRG0 レジスタ(***LeadingTiming***)と TBxRG1 (***TrailingTiming***)およびこれらのバッファは、同一アドレスへ割り付けられます。ダブルバッファがディセーブルの場合、同一の値はレジスタとそのバッファに書き込まれます。
ダブルバッファがイネーブルの場合、その値は各レジスタのバッファのみに書き込まれます。そのため初期値をレジスタ(TBxRG0 (***LeadingTiming***) および TBxRG1 (***TrailingTiming***))へ書き込むためには、ダブルバッファは **DISABLE** に設定してください。その後、イネーブルのダブルバッファには、レジスタへ書き込む次のデータが書き込まれます。データは対応する割り込みが発生した場合に自動的にロードされます。

戻り値:
なし

11.3.4 データ構造

11.3.4.1 TMRB_InitTypeDef

メンバ:
uint32_t
Mode: タイマモードを選択します。
- **TMRB_INTERVAL_TIMER**: インターバルタイマ
- **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

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。

11.3.4.2 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。

11.3.4.3 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

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/TX03_Periph_Driver/src/tmpm33**x**_uart.c(*)
/Libraries/TX03_Periph_Driver/inc/tmpm33**x**_uart.h(*)

補足: "**x**"は 0, 2, 3 を表します。

12.2 TMPM330, TMPM332, TMPM333 の相違点

TMPM330、TMPM333: 3 チャネルのシリアルチャネルを内蔵します。
TMPM332: 2 チャネルのシリアルチャネルを内蔵します。

12.3 API 関数

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

12.3.2 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(), 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) その他:
   UART_SWReset(), UART_SetWakeUpFunc(), UART_SetIdleMode()
4) 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_GetRxFIFOFillLevelStatus(), UART_GetRxFIFOOverRunStatus(), UART_GetTxFIFOFillLevelStatus(), UART_GetTxFIFOUnderRunStatus()

12.3.3 関数仕様

補足: 引数に記述している“TSB_SC_TypeDef* ***UARTx***” は、以下から選択してください。
TMPM330, TMPM333 の場合: **UART0～UART3**
TMPM332 の場合: **UART0～UART2**
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。
TMPM330, TMPM333 の場合: **SIO0～SIO3**
TMPM332 の場合: **SIO0～SIO2**

12.3.3.1 UART_Enable

UART 動作の許可

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

引数:
UARTx: UART チャネルを指定します。

機能:
UART 動作を許可します。

戻り値:
なし

12.3.3.2 UART_Disable

UART 動作の禁止

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

引数:
UARTx: UART チャネルを指定します。

機能:
UART 動作を禁止します。

戻り値:
なし

12.3.3.3 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

12.3.3.4 UART_SWReset

ソフトウェアリセット

関数のプロトタイプ宣言:

void
UART_SWReset(TSB_SC_TypeDef* ***UARTx***)

引数:

UARTx: UART チャネルを指定します。

機能:

ソフトウェアリセットを行います。

戻り値:

なし

12.3.3.5 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

12.3.3.6 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

12.3.3.7 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

12.3.3.8 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:	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

12.3.3.9 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

12.3.3.10 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

12.3.3.11 UART_SetIdleMode

IDLE 時の動作

関数のプロトタイプ宣言:

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

引数:

UARTx: UART チャネルを指定します。
NewState: IDLE 時の動作を選択します。

- **ENABLE**: 動作
- **DISABLE**: 停止

機能:

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

戻り値:

なし

12.3.3.12 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

12.3.3.13 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

12.3.3.14 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

12.3.3.15 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

12.3.3.16 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

12.3.3.17 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

12.3.3.18 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

12.3.3.19 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

12.3.3.20 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

12.3.3.21 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

12.3.3.22 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

12.3.3.23 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

12.3.3.24 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:
受信 FIFO の fill レベル:

- **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

12.3.3.25 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:
受信 FIFO オーバーラン状態:
- **UART_RXFIFO_OVERRUN:** オーバーラン発生
- **0:** オーバーランは発生していない

12.3.3.26 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:
送信 FIFO の fill レベル:
- **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

12.3.3.27 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:
送信 FIFO アンダーラン状態:
- **UART_TXFIFO_UNDERRUN**: アンダーラン発生
- **0**: アンダーランは発生していない

12.3.3.28 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

12.3.3.29 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

12.3.3.30 SIO_GetRxData

受信データの取得

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:
受信データを取得します。

戻り値:
受信データ(値の範囲は 0x00 ~ 0xFF です)

12.3.3.31 SIO_SetTxData

送信データの設定

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: 送信データ

機能:
送信データを設定します。

戻り値:
なし

12.3.3.32 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

12.3.4 データ構造

12.3.4.1 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 演算子によって接続して指定してください。

- **UART_ENABLE_TX:** 送信許可
- **UART_ENABLE_RX:** 受信許可

uint32_t

FlowCtrl: フローコントロールモードを選択します(**)。

- **UART_NONE_FLOW_CTRL**:CTS 無効

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

**:本バージョンのドライバでは、ハンドシェイク機能に対応していないため、CTSUART_NONE_FLOW_CTRL のみ選択できます。

12.3.4.2 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**: SCxSCLK 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 SCxTXD 端子へ出力します。SCxSCLK 端子の立ち上がりエッジで SCxRXD 端子のデータを 1bit ずつ受信バッファに取り込みます。この時、SCxSCLK 端子は High レベルからスタートします(立ち上がりモード)
- **SIO_SCLKS_TXDR_RXDF**: SCxSCLK 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 SCxTXD 端子へ出力します。SCxSCLK 端子の立ち下がりエッジで SCxRXD 端子のデータを 1bit ずつ受信バッファに取り込みます。この時、SCxSCLK 端子は Low レベルからスタートします。(立ち下りモード)

uint32_t

IntervalTime: 連続転送時のインターバル時間を選択します。

- **SIO_SINT_TIME_NONE**: なし
- **SIO_SINT_TIME_SCLK_1**: 1*SCLK
- **SIO_SINT_TIME_SCLK_2**: 2*SCLK
- **SIO_SINT_TIME_SCLK_4**: 4*SCLK
- **SIO_SINT_TIME_SCLK_8**: 8*SCLK
- **SIO_SINT_TIME_SCLK_16**: 16*SCLK
- **SIO_SINT_TIME_SCLK_32**: 32*SCLK
- **SIO_SINT_TIME_SCLK_64**: 64*SCLK

uint32_t

TransferMode: 転送モードを選択します。

- **SIO_TRANSFER_PROHIBIT**: 転送禁止
- **SIO_TRANSFER_HALFDPX_RX**: 半二重(受信)
- **SIO_TRANSFER_HALFDPX_TX**: 半二重(送信)
- **SIO_TRANSFER_FULLDPX**: 全二重

uint32_t

TransferDir: 転送方向を選択します。

- **SIO_LSB_FRIST**: LSB FRIST
- **SIO_MSB_FRIST**: MSB FRIST

uint32_t

Mode: 送受信を制御します。有効ビットの組み合わせが可能です。

- **SIO_ENABLE_TX**: 送信許可
- **SIO_ENABLE_RX**: 受信許可

uint32_t

DoubleBuffer: ダブルバッファの許可/禁止を選択します。

- **SIO_WBUF_ENABLE**: 許可
- **SIO_WBUF_DISABLE**: 禁止

uint32_t

BaudRateClock: ボーレートジェネレータ入力クロックを選択します。

- **SIO_BR_CLOCK_T1**: φTS0
- **SIO_BR_CLOCK_T4**: φTS2
- **SIO_BR_CLOCK_T16**: φTS8
- **SIO_BR_CLOCK_T64**: 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は、ノイズなどの原因により CPU が誤動作(暴走)を始めた場合、これを検出し正常な状態に戻すことを目的としています。

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

本ドライバは、以下のファイルで構成されています。
\Libraries\TX03_Periph_Driver\src\tmpm33**x**_wdt.c(*)
\Libraries/TX03_Periph_Driver\inc\tmpm33**x**_wdt.h(*)

補足: "x"は 0, 2, 3 を表します。

13.2 TMPM330, TMPM332, TMPM333 の相違点

ありません。

13.3 API 関数

13.3.1 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)

13.3.2 関数の種類

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

13.3.3 関数仕様

13.3.3.1 WDT_SetDetectTime

検出時間の設定

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

引数:
***DetectTime*:** 検出時間を選択します。

- **WDT_DETECT_TIME_EXP_15:** 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

13.3.3.2 WDT_SetIdleMode

IDLE モード時の動作

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

引数:
NewState: IDLE 時の動作の有効/無効を選択します。

- **ENABLE**: 動作
- **DISABLE**. 停止

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

補足:
CPU が IDLE モードに入る前に、設定してください。

戻り値:
なし

13.3.3.3 WDT_SetOverflowOutput

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

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

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

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

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

戻り値:
なし

13.3.3.4 WDT_Init

WDT の初期化

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

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

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

戻り値:
なし

13.3.3.5 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

13.3.3.6 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

13.3.3.7 WDT_WriteClearCode

クリアコードの書き込み

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

引数:
なし

機能:
WDT カウンタにクリアコードを書き込みます。

戻り値:
なし

13.3.4 データ構造

13.3.4.1 WDT_InitTypeDef

メンバ:
uint32_t
DetectTime 検出時間を選択します。

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

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

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