TOSHIBA

TX03 ペリフェラルドライバ ユーザーガイド (TMPM361/362/363/364)

第一版

2017 年 9 月

東芝デバイス＆ストレージ株式会社

CMDR-M361UG-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

本製品は、東芝TX03シリーズマイコン用ペリフェラルドライバセットです。TMPM36x (※参照) ペリフェラルドライバは、東芝TX03ペリフェラルドライバのTMPM36xシリーズMCU用です。

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。

TMPM36x 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。

※ 本ドキュメントではTMPM361/TMPM362/TMPM363/TMPM364を表す部分を"TMPM36x"で表現します。

2. TX03 ペリフェラルドライバの構成

/Libraries
TX03 CMSIS ファイルと TMPM36x 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 TMPM36x 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
TMPM36x 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
TMPM36x 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
TMPM36x 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
TMPM36x 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
TMPM36x 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
TMPM36x ペリフェラルドライバの使用例が格納されています。

/Utilities/ MCBTMPM36x
TMPM36x ボードのハードウェアリソース用の設定ファイル、およびドライバファイル (例: led, key) が格納されています。

3. ADC

3.1 概要

本デバイスは、12 ビット逐次変換方式アナログ/デジタルコンバータ(AD コンバータ)を内蔵し、以下のような特徴があります。

1. 通常AD変換と最優先AD変換
   ソフトウェアによる起動
   外部トリガ(ADTRG)による起動
   内部トリガによる起動
2. 通常AD変換機能の動作モード
   チャネル固定シングル変換モード
   チャネルスキャンシングル変換モード
   チャネル固定リピート変換モード
   チャネルスキャンリピート変換モード
3. 最優先AD変換機能の動作モード
   固定シングル変換モード
4. 通常AD変換終了、最優先AD変換終了時、割込み発生機能
5. 通常AD変換機能、最優先AD変換昨日のステータスフラグ
6. AD監視機能
   任意比較条件と一致した場合、割込みを発生
7. AD変換クロックをfc ~ fc/16まで制御可能
8. VREFのリファレンス電流低減機能

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

全ドライバ API は、アプリで使用する API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm36**x**_adc.c(*)
/Libraries/TX03_Periph_Driver/inc/tmpm36**x**_adc.h(*)

補足: 1, 2, 3, 4 を省略して、"**x**"と表記します。

3.2 TMPM361/362/363/364 の相違点

TMPM361/M363 の入力チャネルは 8 チャネル (AN0~AN7) です。
TMPM362/M364 の入力チャネルは 16 チャネル (AN0~AN15) です。

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(uint32_t ***INTMode***)
- WorkState ADC_GetConvertState(void)
- void ADC_SetInputChannel(ADC_AINx ***InputChannel***)

- void ADC_SetChannelScanMode(ADC_ChannelScanMode ***ScanMode***)
- void ADC_SetIdleMode(FunctionalState ***NewState***)
- void ADC_SetVref(FunctionalState ***NewState***)
- void ADC_SetInputChannelTop(ADC_AINx ***TopInputChannel***)
- void ADC_StartTopConvert(void)
- WorkState ADC_GetTopConvertState(void)
- void ADC_SetMonitor(ADC_CMPCRx ***ADCMPx***, FunctionalState ***NewState)***
- void ADC_SetResultCmpReg(uint16_t ***ADCMPx***, uint8_t ***ResultComparison***)
- void ADC_SetMonitorINT(uint16_t ***ADCMPx***, ADC_ComparisonState ***NewState***)
- void ADC_SetHWTrg(uint32_t ***HWSrc***, FunctionalState ***NewState***)
- void ADC_SetHWTrgTop(uint32_t ***HWSrc***, FunctionalState ***NewState)***
- ADC_ResultTypeDef ADC_GetConvertResult(ADC_REGx ***ADREGx***)
- void ADC_SetCmpValue(uint16_t ***ADCMPx***, uint16_t ***value***)
- void ADC_SetClkSupply(FunctionalState NewState)

3.3.2 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) AD 変換設定:
   ADC_SetClk(), ADC_SetScanMode(), ADC_SetRepeatMode(), ADC_SetINTMode(), ADC_SetInputChannel(), ADC_SetScanChannel(), ADC_SetVref(), ADC_SetInputChannelTop(), ADC_SetMonitor(), ADC_ConfigMonitor(), ADC_SetResultCmpReg(), ADC_SetMonitorINT(), ADC_SetHWTrg(), ADC_SetHWTrgTop(), ADC_SetCmpValue()
2) AD 変換開始:
   ADC_Start(), ADC_StartTopConvert()
3) AD 変換ステータス/結果の読み出し:
   ADC_GetConvertState(), ADC_GetTopConvertState(), ADC_GetConvertResult()
4) その他:
   ADC_SWReset(), ADC_SetAccuracy(),ADC_SetIdleMode()

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(ADC_AINx ***InputChannel***)

引数:
***InputChannel*:** AD 変換入力チャネルを選択します。
[TMPM361, TMPM363 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_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**

[TMPM362, TMPM364 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_12, ADC_AN_13, ADC_AN_14, ADC_AN_15**
- **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, ADC_AN_8_12, ADC_AN_8_13, ADC_AN_8_14, ADC_AN_8_15, ADC_AN_12_13, ADC_AN_12_14, ADC_AN_12_15**

機能:
InputChannel により、AD 変換入力チャネルを設定します。

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(ADC_AINx ***TopInputChannel***)

引数:
TopInputChannel:最優先 AD 変換入力チャネルを選択します。

[TMPM361, TMPM363 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**

[TMPM362, TMPM364 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_12, ADC_AN_13, ADC_AN_14, ADC_AN_15**

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_15** 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(ADC_CMPCRx ***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_CMPCR_0:** ADCMPCR0
- **ADC_CMPCR_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_CMPCR_0:** ADCMPCR0
- **ADC_CMPCR_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
void
ADC_SetMonitorINT(ADC_CMPCRx ***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_CMPCR_0:** ADCMPCR0
- **ADC_CMPCR_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(uint32_t ***HWSrc***,
FunctionalState ***NewState***)

引数:
HWSrc: 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:
HWSrc 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。

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。

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(uint32_t ***HWSrc***,
FunctionalState ***NewState***)

引数:
HWSrc: 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:
HWSrc 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_Result
ADC_GetConvertResult(ADC_REGx ***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。

	AD 変換結果レジスタ			
	チャネル固定リピートモード (1 回変換毎割り込み発生) 及び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		4 回変換毎割り込み発生	8 回変換毎割り込み発生	最優先 AD 変換
ADC_AN_0	ADC_REG_08	ADC_REG_08	ADC_REG_08	ADC_REG_SP-
ADC_AN_1	ADC_REG_19	ADC_REG_08~ ADC_REG_19	ADC_REG_08~ ADC_REG_19	
ADC_AN_2	ADC_REG_2A	ADC_REG_08~ ADC_REG_2A	ADC_REG_08~ ADC_REG_2A	
ADC_AN_3	ADC_REG_3B	ADC_REG_08~ ADC_REG_3B	ADC_REG_08~ ADC_REG_3B	
ADC_AN_4	ADC_REG_4C	ADC_REG_4C	ADC_REG_08~ ADC_REG_4C	
ADC_AN_5	ADC_REG_5D	ADC_REG_4C~ ADC_REG_5D	ADC_REG_08~ ADC_REG_5D	
ADC_AN_6	ADC_REG_6E	ADC_REG_4C~ ADC_REG_6E	ADC_REG_08~ ADC_REG_6E	
ADC_AN_7	ADC_REG_7F	ADC_REG_4C~ ADC_REG_7F	ADC_REG_08~ ADC_REG_7F	
ADC_AN_8	ADC_REG_08	ADC_REG_08	ADC_REG_08	
ADC_AN_9	ADC_REG_19	ADC_REG_08~ ADC_REG_19	ADC_REG_08~ ADC_REG_19	
ADC_AN_10	ADC_REG_2A	ADC_REG_08~ ADC_REG_2A	ADC_REG_08~ ADC_REG_2A	
ADC_AN_11	ADC_REG_3B	ADC_REG_08~ ADC_REG_3B	ADC_REG_08~ ADC_REG_3B	
ADC_AN_12	ADC_REG_4C	ADC_REG_4C	ADC_REG_08~ ADC_REG_4C	
ADC_AN_13	ADC_REG_5D	ADC_REG_4C~ ADC_REG_5D	ADC_REG_08~ ADC_REG_5D	
ADC_AN_14	ADC_REG_6E	ADC_REG_4C~ ADC_REG_6E	ADC_REG_08~ ADC_REG_6E	
ADC_AN_15	ADC_REG_7F	ADC_REG_4C~ ADC_REG_7F	ADC_REG_08~ ADC_REG_7F	

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_CMPCR_0:** ADCMPCR0
- **ADC_CMPCR_1:** ADCMPCR1

value: 変換結果レジスタの値と比較する値を設定します。

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

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

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

戻り値:
なし

3.3.3.23 ADC_SetClkSupply

AD 変換クロック供給の許可/禁止

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

引数:
NewState: 以下から AD 変換クロックの発振許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
AD 変換クロック供給の許可/禁止を選択します。

戻り値:

なし

3.3.4 データ構造

3.3.4.1 ADC_Result

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. CAN

4.1 概要

本デバイス は CAN (Controller Area Network) を 1 チャネル内蔵し、CAN のデータ・リンク・レイヤのハードウェア処理を行います。メッセージフィルタと事前にロードされたメッセージ・データを使用し設定できますので、バス上で自律的にメッセージの送受信し、適宜アプリケーションに通知することができます。

TXCAN の概要:

- CAN 2.0 B アクティブ準拠
- 標準フォーマット、拡張フォーマット、両フォーマットに対応
- 各フォーマット上でデータフレーム、リモートフレームに対応
- 32 メールボックス (31 送受信 + 1 受信のみ)
- 最少 48 MHz システムクロックの CAN バス上で、最大 1 Mbps のボーレート
- Intel 82527™ 同等のビットタイミングパラメータ
- ボーレートプリスケーラ内蔵
- 送信メッセージの内部アービトレーションの選択機能
  a) メールボックス数の少なさ
  b) 高優先度のメールボックス ID
- 送受信メッセージのタイムスタンプ
- 動作モード
  a) 標準動作モード
  b) 設定モード
  c) スリープモード: CAN バスが動作 (MCR<WUBA>=1)、または MCU が MCR レジスタにアクセスすると Wake-up します。
  d) 一時停止モード: CAN バス上で機能停止
  e) テストループバックモード: 自己診断
  f) テストエラーモード: 書き込み可能なエラーカウンタ
- 受信用 マスク機能 2 種
  a) プログラマブルグローバル受信マスク(メールボックス 0-30 用)
  b) プログラマブルローカル受信マスク(メールボックス 31 用)
- 識別し拡張ビット用受信マスクビット
- IRQ
  a) INTCANRX: 受信完了割り込み
  b) INTCANTX: 送信完了割り込み
  c) INTCANGB: グローバル割り込み (警告レベル、エラーパッシブ、バスオフ等)

CAN ドライバ API は以下の機能を設定する関数セットです。その機能は、メールボックス、バスのタイミング、メッセージ送受信、CAN コントローラモード、グローバル状態、割り込み、タイムスタンプカウンタ等です。

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/tmpM36**x**_can.c(*)
/Libraries/TX03_Periph_Driver/inc/tmpM36**x**_can.h(*)

補足: 3, 4 を省略して"**x**"と記載します。

4.2 TMPM361/362/363/364 の相違点

なし。

4.3 API 関数

4.3.1 関数一覧

- void CAN_SetMbxMsg(uint32_t ***MbxNum***, CAN_MsgTypeDef * ***MsgStruct***);
- void CAN_GetMbxMsg(uint32_t ***MbxNum***, CAN_MsgTypeDef * ***MsgStruct***);
- void CAN_SetMbxDirection(uint32_t ***MbxNum***, uint8_t ***MbxDirection***);
- void CAN_EnableMbx(uint32_t ***MbxNum***);
- void CAN_DisableMbx(uint32_t ***MbxNum***);
- void CAN_SetTxReq(uint32_t ***MbxNum***, FunctionalState ***NewState***);
- CAN_MbxState CAN_GetMbxTRxState(uint8_t ***StateID***);
- void CAN_ClearMbxTRxState(uint8_t ***StateID***);
- void CAN_SetTxLock(uint32_t ***MbxNum***, FunctionalState ***NewState***);
- void CAN_SetRxMaskID(CAN_RxMaskIDTypeDef * ***MaskIDStruct***);
- void CAN_SetMode(uint32_t ***Mode***);
- void CAN_SetTestMode(uint8_t ***TestModeID***, FunctionalState ***NewState***);
- void CAN_SetWakeUpOnBusActivity(FunctionalState ***NewState***);
- void CAN_SetTxOrder(uint8_t ***TxOrder***);
- void CAN_ClearTimeStampCnt(void);
- void CAN_SWReset(void);
- void CAN_ConfigBitTiming(CAN_BitTimingTypeDef * ***BitTimingStruct***);
- void CAN_SetTimeStampCnt(uint16_t ***CntValue***, uint8_t ***Prescaler***);
- void CAN_GetErrCnt(uint8_t * ***TxErrCnt***, uint8_t * ***RxErrCnt***);
- void CAN_SetErrCnt(uint8_t ***TRxErrCnt***);
- CAN_GlobalState CAN_GetGlobalState(void);
- CAN_INTFactor CAN_GetINTFlag(uint8_t ***INTTypeID***);
- void CAN_ClearINTFlag(uint8_t ***INTTypeID***);
- void CAN_SetINTMask(uint8_t ***INTMaskType***, CAN_INTFactor * ***INTMaskStruct***);

4.3.2 関数の種類

関数は 6 種類に分類されます。

1) メールボックスの Read/Write:
   CAN_SetMbxMsg(), CAN_GetMbxMsg()
2) メールボックス設定:
   CAN_SetMbxDirection(), CAN_EnableMbx(), CAN_DisableMbx()
3) メッセージ送受信:
   CAN_SetTxReq(), CAN_SetTxLock(), CAN_SetRxMaskID(), CAN_SetTxOrder()
4) CAN コントローラバス、モードコントロール:
   CAN_SetMode(), CAN_SetTestMode(), CAN_SetWakeUpOnBusActivity(),
   CAN_SWReset(), CAN_ConfigBitTiming(), CAN_GetErrCnt(), CAN_SetErrCnt()
5) メールボックス、グローバル状態、割り込み制御:
   CAN_GetMbxTRxState(), CAN_ClearMbxTRxState(), CAN_GetGlobalState(),
   CAN_GetINTFlag(), CAN_ClearINTFlag(), CAN_SetINTMask()
6) タイムスタンプ機能:
   CAN_SetTimeStampCnt(), CAN_ClearTimeStampCnt()

4.3.3 関数仕様

4.3.3.1 CAN_SetMbxMsg

メッセージボックスのメッセージ・データ、ID、コントロールビットの設定。

関数のプロトタイプ宣言:
void
CAN_SetMbxMsg(uint32_t ***MbxNum***,
CAN_MsgTypeDef * ***MsgStruct***)

引数:
MbxNum : 以下からメッセージメールボックスを選択します。
CAN_MBX_0~CAN_MBX_31,
CAN_MBX_ALL
または上記の組み合わせ

MsgStruct : メッセージ・データ、ID, タイプ、制御情報を含む構造体です。(詳細は“データ構造”を参照してください)

機能:
MbxNum で設定されるメールボックスに、メッセージ・データ、ID, タイプ、制御情報を設定します。

戻り値:
なし

4.3.3.2 CAN_GetMbxMsg

メールボックスからメッセージ・データ、ID、コントロールビットを取得

関数のプロトタイプ宣言:
void
CAN_GetMbxMsg(uint32_t ***MbxNum***,
CAN_MsgTypeDef * ***MsgStruct***)

引数:
MbxNum : メッセージメールボックスを下記から選択します。

CAN_MBX_0~CAN_MBX_31(組み合わせ不可)

MsgStruct : メッセージ・データ、ID、タイプ、制御情報を取得するために使用する構造体です。(詳細は “データ構造” を参照してください)

機能:
MbxNum により指定されるメールボックスから、メッセージ・データ、ID、タイプ、制御情報を取得します。

戻り値:
なし。

4.3.3.3 CAN_SetMbxDirection

メールボックスの送受信設定

関数のプロトタイプ宣言:
void
CAN_SetMbxDirection(uint32_t ***MbxNum***,
uint8_t ***MbxDirection***)

引数:
MbxNum :メールボックスの選択

MbxDirection が **CAN_MBX_RX** の時、 引数は次のいずれか、または組み合わせ。
- **CAN_MBX_0~ CAN_MBX_31**, **CAN_MBX_ALL**

MbxDirection が **CAN_MBX_TX** の時、引数は次のいずれか、または組み合わせ。
- **CAN_MBX_0~ CAN_MBX_30**

MbxDirection :メッセージボックスを下記のいずれかに設定。
- **CAN_MBX_TX:** 送信メールボックス
- **CAN_MBX_RX:** 受信メールボックス

機能:
メールボックスを送信メールボックス、または受信メールボックスのいずれかに設定します。

戻り値:
なし

4.3.3.4 CAN_EnableMbx

メールボックスの有効設定

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

引数:
MbxNum:メッセージメールボックスを下記から選択します。
- **CAN_MBX_0~CAN_MBX_31、CAN_MBX_ALL** または上記の組み合わせ

機能:
MbxNum により設定されるメールボックスを有効にします。

戻り値:
なし

4.3.3.5 CAN_DisableMbx

メールボックスの無効設定

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

引数:
MbxNum:下記からメールボックスを選択します。
- **CAN_MBX_0~CAN_MBX_31**, **CAN_MBX_ALL** または上記の組見合わせ

機能:
MbxNum により指定されるメールボックスを無効にします。

戻り値:
なし

4.3.3.6 CAN_SetTxReq

特定のメールボックスの送信設定、または送信取消し

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

引数:
MbxNum:下記からメールボックスを選択します。
- **CAN_MBX_0~CAN_MBX_31**, **CAN_MBX_ALL** または上記の組見合わせ

NewState:メッセージメールボックスの送信要求の状態を下記から選択します。
- **ENABLE:** 送信要求を設定
- **DISABLE:** 送信要求を取消し

機能:
MbxNum により設定されるメールボックスの送信要求を設定、または取り消します。

戻り値:

なし。

4.3.3.7 CAN_GetMbxTRxState

全メールボックスの送受信状態の取得

関数のプロトタイプ宣言:
CAN_MbxState
CAN_GetMbxTRxState(uint8_t ***StateID***)

引数:
StateID :下記のいずれかの送受信状態 。
- **CAN_STATE_ID_TX_ACK**: 送信アクノリッジフラグ。メールボックス “n” の送信に成功すると、戻り値 Mbxn が設定されます。
- **CAN_STATE_ID_TX_ABORT**: アボートアクノリッジフラグ。メールボックス “n” の送信がアボートされると、戻り値 Mbxn が設定されます。
- **CAN_STATE_ID_RX_ACK**: 受信メッセージペンディングフラグ。メールボックス “n” に受信メッセージがある時、戻り値 Mbxn が設定されます。
- **CAN_STATE_ID_RX_LOST**: 受信メッセージロストフラグ。メールボックス “n” がオーバーロード状態にある時、引数 Mbxn が設定されます。
- **CAN_STATE_ID_RX_RMT_FRAME**: リモートフレームペンディングフラグ。リモートフレームが、受信メールボックスに設定されたメールボックスに受信されると、戻り値 Mbxn が設定されます。

機能:
StateID により指定される送受信状態を読み込みます。

戻り値:
CAN メールボックスの状態。各メールボックスは、ビット Mbxn (bit0~bit31) または全ビットの状態 mailbox0~31(All) により指定されます。(詳細は “データ構造” を参照してください)

4.3.3.8 CAN_ClearMbxTRxState

全メールボックスの送受信状態のクリア

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

引数:
StateID: 下記のいずれかの送受信状態。
- **CAN_STATE_ID_TX_ACK**: 送信アクノリッジフラグ
- **CAN_STATE_ID_TX_ABORT**: アボートアクノリッジフラグ
- **CAN_STATE_ID_RX_ACK**: 受信メッセージペンディングフラグ
- **CAN_STATE_ID_RX_LOST**: 受信メッセージロストフラグ
- **CAN_STATE_ID_RX_RMT_FRAME**: リモートフレームペンディングフラグ

機能:
全メールボックスの指定された送受信状態をクリアします。

戻り値:
なし

4.3.3.9 CAN_SetTxLock

指定メールボックスの送信要求のロック、アンロック

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

引数:
MbxNum : 下記からメッセージメールボックスを選択。
- **CAN_MBX_0~CAN_MBX_31**, **CAN_MBX_ALL** または上記の組見合わせ

NewState : メッセージボックスの送信要求ロックの状態を下記から選択。
- **ENABLE:** 送信要求をロック
- **DISABLE:** 送信要求をアンロック

機能:
送信要求ロックを設定します。ロックしたメールボックスへの送信要求は無視されます。リモートフレーム使用時に便利で、リモートフレームの自動応答機能 (RFH ビット設定) に設定され、送信メールボックスのデータフィールドを更新します。本関数の使用により、データの整合性を気にせずデータフィールドを更新できます。

戻り値:
なし

4.3.3.10 CAN_SetRxMaskID

受信メールボックスの受信フィルタ設定

関数のプロトタイプ宣言:
void
CAN_SetRxMaskID(CAN_RxMaskIDTypeDef * ***MaskIDStruct***)

引数:
MaskIDStruct: メールボックスのタイプ、マスクされる ID ビット、拡張ビットのマスク状態、ID ビットのマスク状態を指定する構造体。

機能:

受信メールボックスの受信フィルタの設定をします。

戻り値:
なし

4.3.3.11 CAN_SetMode

CAN モジュールの動作モード設定

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

引数:
Mode:CAN モジュールの動作モードを下記から選択。
- **CAN_NORMAL_MODE:** ノーマルオペレーションモード
- **CAN_SUSPEND_MODE:** サスペンドモード
- **CAN_CONFIG_MODE:** コンフィグレーションモード
- **CAN_SLEEP_MODE:** スリープモード

機能:
CAN モジュールの動作モードを設定します。

戻り値:
なし

4.3.3.12 CAN_SetTestMode

テストループバックモード、テストエラーモードの許可/禁止設定

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

引数:
TestModeID : 下記からテストモードを選択します。
- **CAN_TEST_LOOP_BACK_MODE:** テストループバックモード。このモードでは、CAN モジュールは自分が送信したメッセージを受信し、アクノリッジを発生します。他の CAN ノードは必要ありません。
- **CAN_TEST_ERR_MODE:** テストエラーモード。CAN モジュールがテストモードの時のみ、エラーカウンタの書き込みが可能のです。

NewState:テストモードの状態を下記から選択します。
- **ENABLE:** 許可

- **DISABLE:** 禁止

機能:
テストループバックモード、テストエラーモードの許可/禁止の設定をします。

戻り値:
なし

4.3.3.13 CAN_SetWakeUpOnBusActivity

バスアクティブ状態の検出により、スリープモード解除機能を有効/無効に設定

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

引数:
NewState : バスアクティブ状態の検出によるウェイクアップを下記より選択。
- **ENABLE:** バスアクティブ状態の検出により、スリープモードを解除。
- **DISABLE:** バスアクティブ状態の検出により、スリープモードを解除せず。

機能:
バスアクティブ状態の検出により、スリープモード解除機能を有効/無効に設定します。

戻り値:
なし

4.3.3.14 CAN_SetTxOrder

メールボックス番号、またはメッセージ ID により、メールボックス送信順序を選択。

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

引数:
TxOrder: メールボックス送信順序を下記より選択。
- **CAN_TX_ORDER_MBX_NUM:** メールボックス番号によるメールボックス送信順序。メールボックス番号の小さい順に送信。
- **CAN_TX_ORDER_ID:** ID 優先度によるメールボックス送信順序。ID 優先度の高い順に送信されます。

機能:

メールボックスの送信順序をメールボックス番号、またはメッセージ ID 優先度により設定します。

戻り値:
なし

4.3.3.15 CAN_ClearTimeStampCnt

タイムスタンプカウンタのクリア

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

引数:
なし

機能:
タイムスタンプカウンタをクリアします。

戻り値:
なし

4.3.3.16 CAN_SWReset

モジュールのソフトウェアリセット (すべての引数が初期値に戻されます)

関数のプロトタイプ宣言:
void
CAN_SWReset(void)

引数:
なし

機能:
ソフトウェアリセットを実行し、CAN モジュールの引数は全て初期値に戻されます。

戻り値:
なし

4.3.3.17 CAN_ConfigBitTiming

CAN モジュールのビットタイミング、サンプリングの引数を設定します。

関数のプロトタイプ宣言:
void
CAN_ConfigBitTiming(CAN_BitTimingTypeDef * ***BitTimingStruct***)

引数:
BitTimingStruct:ボーレートプリスケーラ、ビットタイミング引数 Sjw、Tseg1, Tseg2 (Intel 82527™)、サンプリング方法を設定する構造体です。(詳細は"データ構造"を参照してください)

機能:
CAN モジュールのビットタイミング、サンプリングの引数を設定します。

戻り値:
なし

補足:
タイミングに関して下記の制約があります。
1) Brp <= 0x3FF
2) Tseg1 >= Tseg2
3) Tseg2 >= Sjw
4) Brp = 0 の時, Tseg2 >= **CAN_TIMING_TSEG2_3TQ**
5) Brp < 4 の時, <Sam> ビットは **CAN_SINGLE_SAMPLING** のみ

4.3.3.18 CAN_SetTimeStampCnt

タイムスタンプカウンタ値とプリスケーラの設定

関数のプロトタイプ宣言:
void
CAN_SetTimeStampCnt(uint16_t ***CntValue***,
uint8_t ***Prescaler***)

引数:
CntValue:16 ビットタイムスタンプカウンタ値

Prescaler:タイムスタンプカウンタ、プリスケーラの設定をします。この時の値は、0xF 以下である必要があります。

機能:
タイムスタンプカウンタ値とプリスケーラの設定をします。

戻り値:
なし

4.3.3.19 CAN_GetErrCnt

送受信エラーカウンタ値の取得

関数のプロトタイプ宣言:
void

CAN_GetErrCnt(uint8_t * ***TxErrCnt***,
uint8_t * ***RxErrCnt***)

引数:
TxErrCnt:送信エラーカウンタ値を保存するポインタです。

RxErrCnt:受信エラーカウンタ値を保存するポインタです。

機能:
送受信エラーカウンタの値を取得します。

戻り値:
なし

4.3.3.20 CAN_SetErrCnt

テストエラーモードでの送受信エラーカウンタ値の設定

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

引数:
TRxErrCnt:送信エラーカウンタ、受信エラーカウンタに設定する値です。これら 2 つのカウンタには、テストエラーモードで同時に同じ値を設定する必要があります。

機能:
テストエラーモードでの送信エラーカウンタ値、受信エラーカウンタ値を設定します。CAN コントローラがテストエラーモードの場合、2 つのカウンタには同時に同じ値が書き込まれます。エラーカウンタに書き込まれる最大値はあ 255 です。したがって、CAN モジュールをバスオフモードにする値 “256” は、エラーカウンタに書き込まれません。

戻り値:
なし

4.3.3.21 CAN_GetGlobalState

モジュールのグローバル状態の取得

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

引数:
なし

機能:
モジュールのグローバル状態を取得します。

戻り値:
グローバル状態を保存する構造体です。特定の状態になった時、ビットが設定されます。(詳細は“データ構造”を参照してください)

4.3.3.22 CAN_GetINTFlag

グローバル割り込みフラグ、メールボックス割り込みフラグの取得

関数のプロトタイプ宣言:
CAN_INTFactor
CAN_GetINTFlag(uint8_t ***INTTypeID***)

引数:
INTTypeID:グローバル割り込みフラグ、メールボックス送信/受信割り込みフラグを下記から選択します。

- **CAN_INT_TYPE_GLOBAL:** グローバル割り込みフラグ
- **CAN_INT_TYPE_TX:** メールボックス送信割り込みフラグ
- **CAN_INT_TYPE_RX:** メールボックス受信割り込みフラグ

機能:
グローバル割り込みフラグ、メールボックス割り込みフラグを取得します。

戻り値:
割り込みフラグ要因。特定の割り込みが発生すると、フラグがセットされます。
(詳細は“データ構造”を参照してください)

4.3.3.23 CAN_ClearINTFlag

グローバル割り込みフラグ、メールボックス割り込みフラグのクリア

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

引数:
INTTypeID:クリアするグローバル割り込みフラグ、メールボックス送信割り込みフラグ、メールボックス受信割り込みフラグを、下記から選択します。

- **CAN_INT_TYPE_GLOBAL:** グローバル割り込みフラグ
- **CAN_INT_TYPE_TX:** メールボックス送信割り込みフラグ
- **CAN_INT_TYPE_RX:** メールボックス受信割り込みフラグ

機能:
グローバル割り込みフラグ、メールボックス割り込みフラグをクリアします。

戻り値:
なし

4.3.3.24 CAN_SetINTMask

グローバル割り込みマスク、メールボックス割り込みマスクの設定

関数のプロトタイプ宣言:
void

CAN_SetINTMask(uint8_t ***INTMaskType***,
CAN_INTFactor * ***INTMaskStruct***)

引数:
INTMaskType:マスクする割り込みを、下記から選択。

- **CAN_INT_GLOBAL_MASK:** グローバル割り込み
- **CAN_INT_MBX_MASK:** メールボックス割り込み

INTMaskStruct:特定の割り込みを許可/禁止する構造体です。特定のビットがセットされると、割り込みが許可されます。(詳細は“データ構造”を参照してください)

機能:
グローバル割り込みマスク、メールボックス割り込みマスクを設定します。

戻り値:
なし

4.3.4 データ構造

4.3.4.1 CAN_MsgTypeDef

メンバ:

uint32_t
MsgID :メッセージ ID です。メッセージが標準フレームの場合、11 ビット ID (0x7FF 以下)。拡張フレームの場合、29 ビット ID (0x1FFFFFFF 以下)。

uint8_t
MsgData[8]: メッセージ・データを保存する 8 バイトのバッファー

uint8_t
MsgDataLen :MsgData[8] フィールドの有効データ長を 0~8 で指定。

uint8_t
MsgFormat :下記からメッセージフォーマットを指定。
- **CAN_STD_FORMAT**: 11 ビット ID
- **CAN_EXT_FORMAT**: 29 ビット ID

uint8_t
MsgFrameType **:** 下記からメッセージフレームタイプを指定。
- **CAN_DATA_FRAME**: データフレーム
- **CAN_RMT_FRAME**: リモートフレーム

uint16_t
MsgTimeStamp :タイムスタンプカウンタ値。CAN_SetMbxMsg() の API を使用し、メールボックスを設定する際はこの値を設定する必要はありません。メールボックスを読み出す時のみ使用します。

uint8_t
RmtFrameHandleMode **:**リモートフレームを操作するモードを下記から指定します。
- **CAN_RMT_FRAME_SW_HANDLE**: ソフトウェアでリモートフレームを操作。
- **CAN_RMT_FRAME_AUTO_HANDLE**: メールボックスが自動的にリモートフレームに応答。

FunctionalState
RxMaskState **:**アクセプタンスマスクを使用するかどうかを選択。
- **ENABLE**: アクセプタンスマスクを使用する
- **DISABLE**: アクセプタンスマスクを使用しない

4.3.4.2 CAN_MbxState

メンバ:

uint32_t
All: メールボックスの状態要素

ビットフィールド:

uint32_t
Mbxn: 1(n: 0~31) ビットフィールドはメールボックス “n” (n = 0~31) の状態を表します。指定された状態になると、フィールドがセットされます。

4.3.4.3 CAN_RxMaskIDTypeDef

メンバ:

uint8_t
MbxType **:**メールボックスタイプを下記から指定します。
- **CAN_MBX_GLOBAL**: グローバルメールボックス (メールボックス 0~30)
- **CAN_MBX_LOCAL**: ローカルボックス (メールボックス 31)

uint32_t
MaskIDBit : マスクする ID ビットを指定します。
CAN_ID_BIT_0 ~ **CAN_ID_BIT_28**, **CAN_ID_BIT_ALL**、またはこの組み合わせ

FunctionalState
ExtIDState **:**ID 拡張ビットマスク状態を指定します。
- **ENABLE**: 標準、拡張フレームの受信可能
- **DISABLE**: メールボックス内の ID 拡張ビットがどちらかを決定。

FunctionalState
MaskState: ID ビットマスク状態。
- **ENABLE**: 指定された ID ビットをマスク
- **DISABLE**: 指定された ID ビットをマスクしない

4.3.4.4 CAN_GlobalState

メンバ:

uint32_t
All: グローバルステータス要因

ビットフィールド:

uint32_t
WarningStatus: 1 ワーニングステータス
1 = 送信または受信エラーカウンタが 96 に到達。
0 = 送信エラーカウンタ、受信エラーカウンタの両方で 96 未満。

uint32_t
ErrPassiveStatus: 1 エラーパッシブステータス
1 = CAN モジュールがエラーパッシブモード
0 = CAN モジュールがエラーアクティブモード

uint32_t
BusOffStatus: 1 バスオフステータス
1 = バスオフ
0 = ノーマルオペレーション

uint32_t
TimeStampOverFlow: 1 タイムスタンプオーバーフローフラグ
1 = 少なくとも 1 回はオーバーフロー発生
0 = オーバーフローの発生なし

uint32_t
Reserverd2:2 未使用

uint32_t
SleepModeAck: 1 スリープモードアクノリッジ
1 = CAN モジュールはスリープモード
0 = ノーマルオペレーション

uint32_t
ChangeConfig: 1 チェンジコンフィグレーションエネーブル
1 = CAN モジュールはコンフィグレーションモード
0 = CAN モジュールはコンフィグレーションモードではない

uint32_t
SuspendModeAck: 1　サスペンドモードアクノリッジ
1 = CAN モジュールはサスペンドモード
0 = CAN モジュールはサスペンドモードではない

uint32_t
Reserverd1:1　未使用

uint32_t
TxMode: 1　トランスミットモード
1 = CAN モジュールは送信中
0 = CAN モジュールは送信中ではない

uint32_t
RxMode: 1　レシーブモード
1 = CAN モジュールは受信中
0 = CAN モジュールは受信中ではない

uint32_t
MsgInSlot: 5　メッセージインスロット中のメッセージ
11111 = スロットに送信メッセージなし
00000 = メッセージ 0
…
11110 = メッセージ 30

uint32_t
Reserverd0:15　未使用

4.3.4.5 CAN_BitTimingTypeDef

メンバ:

uint16_t

Brp :ボーレートプリスケーラ。0x3FF 以下。

uint8_t

Tseg1:ビットタイミング引数 1　は下記から設定。

CAN_TIMING_TSEG1_2TQ~CAN_TIMING_TSEG1_16TQ

uint8_t
Tseg2 **:** ビットタイミング引数 1　は下記から設定。

CAN_TIMING_TSEG2_2TQ~CAN_TIMING_TSEG2_8TQ

uint8_t
Sam :サンプリング方法を下記から設定。

- **CAN_SINGLE_SAMPLING**: シングルサンプリング
- **CAN_TRIPLE_SAMPLING**: トリプルサンプリング

uint8_t
Sjw :再同期時に、長くしたり短くしたり調節が可能な時間量 (TQ) のセットで、以下から選択されます。

- **CAN_TIMING_SJW_1TQ~CAN_TIMING_SJW_4TQ**

4.3.4.6 CAN_INTFactor

メンバ:

uint32_t

All: グローバル割り込みフラグ、またはメールボックス割り込みフラグ

グローバルビット:

uint32_t
WarningLevel: 1　ワーニングレベル割り込みフラグ
1 = 少なくともエラーカウンタの 1 つが 96 に到達
0 = 両方のエラーカウンタとも 96 に到達していない

uint32_t
ErrPassive: 1　エラーパッシブ割り込みフラグ
1 = CAN モジュールがエラーパッシブモード
0 = CAN モジュールがアクティブモード

uint32_t
BusOff: 1　バスオフ割り込みフラグ
1 = CAN モジュールはバスオフモード
0 = CAN モジュールはバスオンモード

uint32_t
TimeStampOverFlow: 1 タイムスタンプカウンタオーバーフロー
1 = タイムスタンプカウンタのオーバーフローが少なくとも 1 回以上発生
0 = タイムスタンプカウンタのオーバーフローがない

uint32_t
TxAbort: 1　送信アボートフラグ
1 = 送信停止されました
0 = 送信は停止されていません

uint32_t
RxLost: 1　受信メッセージロスト割り込み
1 = 少なくとも受信メッセージロストが 1 回以上発生
0 =受信メッセージロストが発生していない

uint32_t
WakeUp: 1　ウェイクアップ割り込みフラグ
1 = CAN モジュールはスリープモードから解除
0 = CAN モジュールはスリープモード、ノーマルモード

uint32_t
RmtFramePending: 1 リモートフレームペンディングフラグ
1 = リモートフレームを受信
0 = リモートフレームを受信していない

uint32_t
Reserverd: 24　未使用

メールボックスビット:

uint32_t

Mbxn: 1(n: 0~31)　　メールボックス“n” (n = 0~31) の割り込みフラグを指定。指定された割り込みの発生時、フィールドがセットされます。

補足:

構造体 CAN_INTFactor は割り込みマスクとしても使用されます。ビットが “1” にセットされると、指定の割り込みを許可します。ビットが “0” にセットされると、ビットはクリアされ、指定の割り込みが禁止されます。

5. CEC

5.1 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 クロックまたは 16 ビットタイマフリップフロップ出力(TBxOUT)でサンプリング
  1. ノイズキャンセル時間を調整可能
- 1byte 毎にデータを受信
  1. データサンプリングポイントを調整可能
  2. 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/tmpm36**x**_cec.c(*)
/Libraries/TX03_Periph_Driver/inc/tmpm36**x**_cec.h(*)

補足: 1, 2, 3, 4 を"**x**"で記載します。

5.2 TMPM361/362/363/364 の相違点

なし。

5.3 API 関数

5.3.1 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***)
- 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***)
- Result CEC_SetRxStartBitDetect(FunctionalState ***NewState***)
- void CEC_SetSamplingClk(CEC_SamplingClockSrc ***ClkSrc***)

5.3.2 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(), CEC_SetRxStartBitDetect(), CEC_SetSamplingClk()

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()

5.3.3 関数仕様

5.3.3.1 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

5.3.3.2 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

5.3.3.3 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

5.3.3.4 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

5.3.3.5 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

5.3.3.6 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

5.3.3.7 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

5.3.3.8 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)

5.3.3.9 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

5.3.3.10 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

5.3.3.11 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

5.3.3.12 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)

5.3.3.13 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

5.3.3.14 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

5.3.3.15 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

5.3.3.16 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

5.3.3.17 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

5.3.3.18 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

5.3.3.19 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

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

5.3.3.20 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

5.3.3.21 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+2cycle
- **CEC_VALID_TIME_2**: 1.05ms+4cycle
- **CEC_VALID_TIME_3**: 1.05ms+6cycle
- **CEC_VALID_TIME_4**: 1.05ms-2cycle
- **CEC_VALID_TIME_5**: 1.05ms-4cycle
- **CEC_VALID_TIME_6**: 1.05ms-6cycle

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

5.3.3.22 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

5.3.3.23 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

5.3.3.24 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

5.3.3.25 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

5.3.3.26 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-2cycles
- **CEC_START_BIT_RISING_TIME_MIN_3**: 3.5ms-3cycles
- **CEC_START_BIT_RISING_TIME_MIN_4**: 3.5ms-4cycles
- **CEC_START_BIT_RISING_TIME_MIN_5**: 3.5ms-5cycles
- **CEC_START_BIT_RISING_TIME_MIN_6**: 3.5ms-6cycles
- **CEC_START_BIT_RISING_TIME_MIN_7**: 3.5ms-7cycles

RisingTimeMax: 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

5.3.3.27 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

5.3.3.28 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

***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

5.3.3.29 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

5.3.3.30 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

5.3.3.31 CEC_SetRxStartBitDetect

受信時のスタートビット割り込み設定

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

引数:
NewState: 以下から受信時のスタートビット割り込み許可/禁止を選択します。
- **ENABLE:** 許可
- **DISABLE:** 禁止

機能:
受信時のスタートビット割り込み許可/禁止を選択します。
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

5.3.3.32 CEC_SetSamplingClk

サンプリングクロックの選択

関数のプロトタイプ宣言:
void
void CEC_SetSamplingClk(CEC_SamplingClockSrc ***ClkSrc***)

引数:
ClkSrc: 以下からサンプリングクロックを選択します。
- **CEC_SAMPLING_CLK_SRC_LOW_SPEED**: 低速クロック (fs)
- **CEC_SAMPLING_CLK_SRC_TBXOUT**: タイマ出力 (TBxOUT)

機能:
サンプリングクロックを選択します。

戻り値:

なし

5.3.4 データ構造

5.3.4.1 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”

5.3.4.2 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

6. CG

6.1 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: 1, 2, 3, 4 を"**x**"と記載します。

6.2 TMPM361/362/363/364 の相違点

1) INT ソースの違い
   - M361 に INT8, INT9, INTA, INTB, INTC, INTD, INTJ はありません。
   - M363 に INT8, INT9, INTA, INTB, INTC, INTD, INTE, INTF, INTJ はありません。

6.3 API 関数

6.3.1 関数一覧

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

- CG_DivideLevel CG_GetPhiT0Level(void);
- void CG_SetSCOUTSrc(CG_SCOUTSrc ***Source***) ;
- CG_SCOUTSrc CG_GetSCOUTSrc(void);
- void CG_SetWarmUpTime(CG_WarmUpSrc ***Source***, uint16_t ***Time***);
- void CG_StartWarmUp(void);
- WorkState CG_GetWarmUpState(void);
- Result CG_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_SetPortKeep(FunctionalState ***NewState***);
- 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);
- void CG_SetClkMulTimes(CG_ClkMul ***MulTimes***);

6.3.2 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) クロックの選択:
   CG_SetFgearLevel(), CG_GetFgearLevel(),CG_SetPhiT0Src(),CG_GetPhiT0Src(),
   CG_SetPhiT0Level(), CG_GetPhiT0Level(), CG_SetSCOUTSrc(),
   CG_GetSCOUTSrc(), CG_SetWarmUpTime(), CG_StartWarmUp(),
   CG_GetWarmUpState(), CG_SetPLL(), CG_GetPLLState(), CG_SetFosc(),
   CG_GetFoscState(), CG_SetFs(), CG_GetFsState(), CG_SetFcSrc(),
   CG_GetFcSrc(), CG_SetFsysSrc(), CG_GetFsysSrc(), CG_SetClkMulTimes()

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(), CG_SetPortKeep()

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

6.3.3 関数仕様

6.3.3.1 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

6.3.3.2 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

6.3.3.3 CG_SetPhiT0Src

PhiT0(ΦT0) ,fc 間の PhiT0(ΦT0) ソースの設定

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

引数:
PhiT0Src: 以下から PhiT0 ソースを選択します。
- **CG_PHIT0_SRC_FPSCALER**: ΦT0 ソースは fprescaler
- **CG_PHIT0_SRC_FS**: ΦT0 ソースは fs

機能:
PhiT0 (ΦT0) ソースを選択します。

戻り値:
なし

6.3.3.4 CG_GetPhiT0Src

PhiT0 (ΦT0) ソースの取得

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

引数:
なし

機能:
PhiT0 (ΦT0) ソースを取得します。

戻り値:
- **CG_PHIT0_SRC_FPSCALER**: ΦT0 ソースは fprescaler
- **CG_PHIT0_SRC_FS**: ΦT0 ソースは fs

6.3.3.5 CG_SetPhiT0Level

PhiT0 (ΦT0) と fc 間の分周レベルの設定

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

引数:
DividePhiT0FromFc: PhiT0 (ΦT0) と fc 間の分周レベルを下記の値から設定します。
- **CG_DIVIDE_1**: ΦT0 = fc
- **CG_DIVIDE_2**: ΦT0 = fc/2
- **CG_DIVIDE_4**: ΦT0 = fc/4
- **CG_DIVIDE_8**: ΦT0 = fc/8
- **CG_DIVIDE_16**: ΦT0 = fc/16
- **CG_DIVIDE_32**: ΦT0 = fc/32
- **CG_DIVIDE_64**: ΦT0 = fc/64
- **CG_DIVIDE_128**: ΦT0 = fc/128
- **CG_DIVIDE_256**: ΦT0 = fc/256

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

6.3.3.6 CG_GetPhiT0Level

PhiT0(ΦT0) ,fc 間の分周レベルの取得

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

引数:
なし

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

戻り値:
PhiT0(ΦT0) ,fc 間の分周レベル:
- **CG_DIVIDE_1**: ΦT0 = fc
- **CG_DIVIDE_2**: ΦT0 = fc/2
- **CG_DIVIDE_4**: ΦT0 = fc/4
- **CG_DIVIDE_8**: ΦT0 = fc/8
- **CG_DIVIDE_16**: ΦT0 = fc/16
- **CG_DIVIDE_32**: ΦT0 = fc/32
- **CG_DIVIDE_64**: ΦT0 = fc/64
- **CG_DIVIDE_128**: ΦT0 = fc/128
- **CG_DIVIDE_256**: ΦT0 = fc/256
- **CG_DIVIDE_UNKNOWN**: 無効データ

6.3.3.7 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

6.3.3.8 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:
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

6.3.3.9 CG_SetWarmUpTime

ウォームアップ時間の設定

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

引数:
Source: 以下から、ウォームアップカウンタのソースクロックを選択します。
- **CG_WARM_UP_SRC_OSC_X1**: fosc
- **CG_WARM_UP_SRC_OSC_XT1**: fs

Time: ウォーミングアップカウンタ値を設定します。設定可能な値は ***Time*** が
CG_WARM_UP_SRC_OSC_X1 の場合 0U から 0x1000U、
CG_WARM_UP_SRC_OSC_XT1 の場合は 0U から 0xFFFFU です。

機能:
ウォームアップサイクル数の計算式は下記になります。
ウォーミングアップサイクル数 = (ウォーミングアップ時間) / (ウォーミングアップクロック周期) / 16

高速発振子 8MHz 使用時、ウォーミングアップ時間 100us を設定する場合の計算例:
(ウォーミングアップ時間)/(ウォーミングアップクロック) = 100us/(1/8MHz)/16 = 0x32

戻り値:
なし

6.3.3.10 CG_StartWarmUp

ウォームアップ開始

関数のプロトタイプ宣言:

void
CG_StartWarmUp(void)

引数:
なし

機能:
ウォームアップを開始します。

戻り値:
なし

6.3.3.11 CG_GetWarmUpState

ウォーミングアップ動作状態 (動作中、完了)の確認

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

引数:
なし

機能:
ウォーミングアップ動作状態を確認します。

```
Example of using warm-up timer:
CG_SetWarmUpTime(CG_WARM_UP_SRC_OSC_EXT_HIGH, 0x32);
/* start warm up */
CG_StartWarmUp();
/* check warm up is finished or not*/
While(CG_GetWarmUpState() == BUSY);
```

戻り値:
ウォーミングアップ状態:

- **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

6.3.3.12 CG_SetPLL

PLL 回路の設定

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

引数:
NewState:

- **ENABLE**: PLL 有効
- **DISABLE**: PLL 無効

機能:
PLL 回路の有効/無効を設定します。

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

6.3.3.13 CG_GetPLLState

PLL 回路の状態の取得

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

引数:
なし

機能:
PLL 回路の状態を取得します。

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

6.3.3.14 CG_SetFosc

高速発振器(fosc)の有効/無効設定

関数のプロトタイプ宣言:
Result
CG_SetFosc(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。

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

6.3.3.15 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

6.3.3.16 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

6.3.3.17 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

6.3.3.18 CG_SetSTBYMode

スタンバイモードの選択

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

引数:
Mode: 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_IDLE2**: IDLE2 モード (CPU、SSP が停止)
- **CG_STBY_MODE_BACKUP_STOP**: バックアップ STOP モード (一部の機能を保持して内部電源を遮断)
- **CG_STBY_MODE_BACKUP_SLEEP**: バックアップ SLEEP モード (一部の機能を保持して内部電源を遮断)
- **CG_STBY_MODE_IDLE1**: IDLE1 モード(IDLE2 モードより低消費電力を実現)

機能:
スタンバイモードを選択します。

戻り値:
なし

6.3.3.19 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_IDLE2**: IDLE2 モード
- **CG_STBY_MODE_BACKUP_STOP**: バックアップ STOP モード
- **CG_STBY_MODE_BACKUP_SLEEP**: バックアップ SLEEP モード
- **CG_STBY_MODE_IDLE1**: IDLE1 モード
- **CG_STBY_MODE_UNKNOWN**: 無効なモード

6.3.3.20 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

6.3.3.21 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

6.3.3.22 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

6.3.3.23 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

6.3.3.24 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

6.3.3.25 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

6.3.3.26 CG_SetPortKeep

バックアップモード中の I/O 制御信号保持状態の設定

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

引数:
NewState:
- **DISABLE:** ポートによる制御
- **ENABLE:** ENABLE 設定時の状態を保持

バックアップモード中の I/O 制御信号保持の詳細については、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:
バックアップモード中の I/O 制御信号保持の有効/無効を切り替えます。

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

6.3.3.27 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

6.3.3.28 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

6.3.3.29 CG_SetClkMulTimes

PLL 逓倍回路の逓倍数設定

関数のプロトタイプ宣言:
void
CG_SetClkMulTimes (CG_ClkMul ***MulTimes***);

引数:
MulTimes:以下から PLL 逓倍回路の逓倍数を設定します。
- **CG_MUL_4TIMES**: 4 逓倍
- **CG_MUL_8TIMES**: 8 逓倍

機能:
PLL 逓倍回路の逓倍数を設定します。

補足:
PLL による逓倍後の最大周波数は 64MHz です。

戻り値:
なし

6.3.3.30 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

6.3.3.31 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

6.3.3.32 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
- **CG_INT_SRC_6**: INT6
- **CG_INT_SRC_7**: INT7
- **CG_INT_SRC_8**: INT8
- **CG_INT_SRC_9**: INT9
- **CG_INT_SRC_A**: INT10
- **CG_INT_SRC_B**: INT11
- **CG_INT_SRC_C**: INT12
- **CG_INT_SRC_D**: INT13
- **CG_INT_SRC_E**: INT14
- **CG_INT_SRC_F**: INT15
- **CG_INT_SRC_CEC_RX**: CEC 受信
- **CG_INT_SRC_CEC_TX**: CEC 送信
- **CG_INT_SRC_RMC_RX_0**: INTRMCRX0
- **CG_INT_SRC_RMC_RX_1**: INTRMCRX1
- **CG_INT_SRC_RTC**: INTRTC
- **CG_INT_SRC_KWUP**: INTKWUP

ActiveState: 解除トリガのアクティブ状態を選択します。

割り込み要因	選択できるアクティブレベル	説明
CG_INT_SRC_RTC	**CG_INT_ACTIVE_STATE_FALLING**	↓エッジ
CG_INT_SRC_CEC_RX, ***CG_INT_SRC_CEC_TX,*** ***CG_INT_SRC_RMC_RX_0,*** ***CG_INT_SRC_RMC_RX_1***	**CG_INT_ACTIVE_STATE_RISGING**	↑エッジ
CG_INT_SRC_KWUP	**CG_INT_ACTIVE_STATE_H**	"High"レベル
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。
補足:
M361、M363 では、**CG_INT_SRC_8, CG_INT_SRC_9, CG_INT_SRC_A, CG_INT_SRC_B, CG_INT_SRC_C, CG_INT_SRC_D, CG_INT_SRC_RMC_RX_1** は選択できません。
M364 では、**CG_INT_SRC_E**, **CG_INT_SRC_F** は選択できません。

戻り値:
なし

6.3.3.33 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_INT_ActiveState
CG_GetSTBYReleaseINTSrc(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, CG_INT_SRC_6, CG_INT_SRC_7, CG_INT_SRC_8, CG_INT_SRC_9, CG_INT_SRC_A, CG_INT_SRC_B, CG_INT_SRC_C, CG_INT_SRC_D, CG_INT_SRC_E , CG_INT_SRC_F, CG_INT_SRC_CEC_RX, CG_INT_SRC_CEC_TX, CG_INT_SRC_RMC_RX_0, CG_INT_SRC_RMC_RX_1, CG_INT_SRC_RTC, CG_INT_SRC_KWUP**

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。

補足:
M361、M363 では、**CG_INT_SRC_8, CG_INT_SRC_9, CG_INT_SRC_A, CG_INT_SRC_B, CG_INT_SRC_C, CG_INT_SRC_D, CG_INT_SRC_RMC_RX_1** は選択できません。
M364 では、**CG_INT_SRC_E**, **CG_INT_SRC_F** は選択できません。

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

6.3.3.34 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, CG_INT_SRC_6, CG_INT_SRC_7, CG_INT_SRC_8, CG_INT_SRC_9, CG_INT_SRC_A, CG_INT_SRC_B, CG_INT_SRC_C, CG_INT_SRC_D, CG_INT_SRC_E , CG_INT_SRC_F, CG_INT_SRC_CEC_RX, CG_INT_SRC_CEC_TX, CG_INT_SRC_RMC_RX_0, CG_INT_SRC_RMC_RX_1, CG_INT_SRC_RTC, CG_INT_SRC_KWUP**

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

6.3.3.35 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_NMI_Factor
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

6.3.3.36 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
- **KBPRese**t(Bit3) バックアップモード解除によるリセット
- **DebugReset** (Bit 4) <SYSRESETREQ>によるリセット

6.3.4 データ構造

6.3.4.1 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

6.3.4.2　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
BKPReset(Bit3)　バックアップモード解除によるリセット

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

uint32_t
Reserved (Bit5~bit31)　未使用

7. DMAC

7.1 概要

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

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

全ドライバ API は、アプリ使用の API 定義を格納する以下のファイルで構成されています。
\Libraries\TX03_Periph_Driver\src\tmpm36**x**_dmac.c(*)
\Libraries/TX03_Periph_Driver\inc\tmpm36**x**_dmac.h(*)

補足: 1, 2, 3, 4,を“x”と記載します。

7.1 TMPM361/362/363/364 の違い

なし

7.2 API 関数

7.2.1 関数一覧

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

7.2.2 関数の種類

関数は、主に以下の 5 種類に分かれています。

1) DMAC 基本設定:
   DMAC_Enable(),DMAC_Disable(),DMAC_SetDMAChannel(), DMAC_Init()
2) DMA 転送割り込みステータス、FIFO または DMA チャネル状態:
   DMAC_GetINTReq(), DMAC_GetTxINTReq(), DMAC_GetRawTxINTReq(), DMAC_GetChannelTxState(), DMAC_GetFIFOState()
3) DMA 割り込み設定、DMA 割り込み要求のクリア:
   DMAC_ClearTxINTReq(), DMAC_SetTxINTConfig()
4) DMA ソフトウェア要求の設定、および取得:
   DMAC_SetSWBurstReq(), DMAC_GetSWBurstReqState(), DMAC_SetSWSingleReq(), DMAC_GetSWSingleReqState(), DMAC_SetLinkedList()
5) その他の設定:
   DMAC_SetDMAHalt(), DMAC_SetLockedTx()

7.2.3 関数仕様

7.2.3.1 DMAC_Enable

DMA 回路動作の許可

関数のプロトタイプ宣言:
void
DMAC_Enable(void);

引数:
なし

機能:
DMA 回路動作を許可します。

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

戻り値:
なし

7.2.3.2 DMAC_Disable

DMA 回路動作の禁止

関数のプロトタイプ宣言:
void
DMAC_Disable(void);

引数:
なし

機能:

DMA 回路動作を禁止します。

戻り値:
なし

7.2.3.3 DMAC_GetINTReq

DMA チャネル割り込みステータスの取得

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

引数:
なし

機能:
DMA チャネル割り込み要求状態を取得します。

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

7.2.3.4 DMAC_GetTxINTReq

DMA チャネル転送割り込み要求状態の取得

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

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

機能:
DMA チャネル転送割り込み要求状態を取得します。

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

7.2.3.5 DMAC_ClearTxINTReq

転送割り込み要求のクリア

関数のプロトタイプ宣言:

void
DMAC_ClearTxINTReq(DMAC_Channel ***Chx***,
DMAC_INTSrc ***INTSource***);

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

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

機能:
転送割り込み要求をクリアします。

戻り値:
なし

7.2.3.6 DMAC_GetRawTxINTReq

DMA チャネルの許可前転送終了割り込み発生状態の取得

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

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

機能:
DMA チャネルの許可前転送終了割り込み発生状態を取得します。

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

7.2.3.7 DMAC_GetChannelTxState

DMA チャネル転送状態の取得

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

引数:
Chx: 以下から DMA チャネルを選択します。

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

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

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

7.2.3.8 DMAC_SetSWBurstReq

ソフトウェアによる DMA バースト転送要求の設定

関数のプロトタイプ宣言:
void
DMACA_SetSWBurstReq(DMAC_ReqNum ***BurstReq***);

引数:
BurstReq: 以下のいずれかのバースト要求番号を選択します。
- **DMAC_SIO_0_RTX:** SIO0 受信/送信
- **DMAC_SIO_1_RTX**: SIO1 受信/送信
- **DMAC_SIO_2_RTX**: SIO2 受信/送信
- **DMAC_SIO_3_RTX**: SIO3 受信/送信
- **DMAC_SIO_4_RTX**: SIO4 受信/送信
- **DMAC_SIO_5_RTX**: SIO5 受信/送信
- **DMAC_SIO_6_RTX**: SIO6 受信/送信
- **DMAC_SIO_7_RTX**: SIO7 受信/送信
- **DMAC_SIO_8_RTX**: SIO8 受信/送信
- **DMAC_SIO_9_RTX**: SIO9 受信/送信
- **DMAC_SIO_10_RTX**: SIO10 受信/送信
- **DMAC_SIO_11_RTX**: SIO11 受信/送信
- **DMAC_SSP_TX**: SSP 送信
- **DMAC_SSP_RX**: SSP 受信
- **DMAC_AD_CONVERT_END**: A/D 変換終了

機能:
ソフトウェアによる DMA ユニット A のバースト転送要求を設定します。
ソフトウェアでの DMA 要求とハードウェアからの同時実行は禁止です。

戻り値:
なし

7.2.3.9 DMAC_GetSWBurstReqState

ソフトウェアによる DMA バースト要求状態の取得

関数のプロトタイプ宣言:

DMAC_BurstReqState
DMAC_GetSWBurstReqState(void);

引数:
なし

機能:
ソフトウェアによる DMA バースト要求状態を取得します。

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

7.2.3.10 DMAC_SetSWSingleReq

ソフトウェアによる DMA シングル転送要求の設定

関数のプロトタイプ宣言:
void
DMAC_SetSWSingleReq(DMAC_ReqNum ***SingleReq***);

引数:
SingleReq: 以下から、シングル要求番号を選択します。
- **DMACB_SSP_RX**: SSP 受信

機能:
ソフトウェアによる DMA シングル転送要求を設定します。ソフトウェアでの DMA 要求とハードウェアからの同時実行は禁止です。

戻り値:
なし

7.2.3.11 DMAC_GetSWSingleReqState

ソフトウェアによる DMA シングル要求状態の取得

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

引数:
なし

機能:
ソフトウェアによる DMA シングル要求状態を取得します。

戻り値:
DMA シングル要求状態です。構造体" DMAC_SingleReqState"の詳細は"データ構造"を参照してください。

7.2.3.12 DMAC_SetLinkedList

DMA チャネル・コレクションアイテムレジスタの設定

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

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

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

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

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

戻り値:
なし

7.2.3.13 DMAC_GetFIFOState

FIFO 状態の取得

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

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

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

戻り値:
FIFO 状態:
BUSY、または **DONE**

7.2.3.14 DMAC_SetDMAHalt

DMA 要求の設定

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

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

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

機能:
DMA 要求受付制御を設定します。

戻り値:
なし

7.2.3.15 DMAC_SetLockedTx

ロック転送の設定

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

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

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

機能:
ロック転送を設定します。

戻り値:

なし

7.2.3.16 DMAC_SetTxINTConfig

転送割り込みの設定

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

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

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

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

機能:
転送割り込みを設定します。

戻り値:
なし

7.2.3.17 DMAC_SetDMAChannel

DMA チャネルの許可/禁止設定

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

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

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

機能:
DMA チャネルの許可/禁止を設定します。

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

戻り値:
なし

7.2.3.18 DMAC_Init

DMA チャネルの初期設定

関数のプロトタイプ宣言:
void
DMAC_Init(DMAC_Channel ***Chx***,
DMAC_InitTypeDef * ***InitStruct***);

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

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

機能:
DMA チャネルの初期設定を行います。

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

戻り値:
なし

7.2.4 データ構造

7.2.4.1 DMAC_InitTypeDef

メンバ:
uint32_t
***TxDirection*:** 以下から、転送方向を選択します。
- **DMAC_MEMORY_TO_MEMORY**: メモリ->メモリ
- **DMAC_MEMORY_TO_PERIPH**: メモリ->周辺回路
- **DMAC_PERIPH_TO_MEMORY**: 周辺回路->メモリ
- **DMAC_PERIPH_TO_PERIPH**: 周辺回路->周辺回路

uint32_t
SrcAddr: 転送元アドレスを設定します。

uint32_t
DstAddr: 転送先アドレスを設定します。

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

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

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

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

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

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

DMAC_ReqNum
TxPeriph: 以下のいずれかのバースト要求番号を選択します。
- **DMAC_SIO_0_RTX:** SIO0 受信/送信

- **DMAC_SIO_1_RTX**: SIO1 受信/送信
- **DMAC_SIO_2_RTX**: SIO2 受信/送信
- **DMAC_SIO_3_RTX**: SIO3 受信/送信
- **DMAC_SIO_4_RTX**: SIO4 受信/送信
- **DMAC_SIO_5_RTX**: SIO5 受信/送信
- **DMAC_SIO_6_RTX**: SIO6 受信/送信
- **DMAC_SIO_7_RTX**: SIO7 受信/送信
- **DMAC_SIO_8_RTX**: SIO8 受信/送信
- **DMAC_SIO_9_RTX**: SIO9 受信/送信
- **DMAC_SIO_10_RTX**: SIO10 受信/送信
- **DMAC_SIO_11_RTX**: SIO11 受信/送信
- **DMAC_SSP_TX**: SSP 送信
- **DMAC_SSP_RX**: SSP 受信
- **DMAC_AD_CONVERT_END**: A/D 変換終了.

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

7.2.4.1 DMAC_INTReq

メンバ:
uint32_t
All: DMAC 全チャネルの割り込み発生状態です
ビットフィールド
uint32_t
CH0_INTReq : 1 DMAC チャネル 0 の割り込み発生状態です。
uint32_t
CH1_INTReq : 1 DMAC チャネル 1 の割り込み発生状態です。

7.2.4.2 DMAC_BurstReqState

メンバ:
uint32_t
All: 全ての DMAC バースト要求状態
ビットフィールド
uint32_t
SIO0_RTx : 1 SIO0 受信/送信.
uint32_t
SIO1_RTx : 1 SIO1 受信/送信.
uint32_t
SIO2_RTx : 1 SIO2 受信/送信.
uint32_t
SIO3_RTx : 1 SIO3 受信/送信.
uint32_t
SIO4_RTx : 1 SIO4 受信/送信.
uint32_t
SIO5_RTx : 1 SIO5 受信/送信.
uint32_t
SIO6_RTx : 1 SIO6 受信/送信.
uint32_t
SIO7_RTx : 1 SIO7 受信/送信.

uint32_t
SIO8_RTx : 1 SIO8 受信/送信.
uint32_t
SIO9_RTx : 1 SIO9 受信/送信.
uint32_t
SIO10_RTx : 1 SIO10 受信/送信.
uint32_t
SIO11_RTx : 1 SIO11 受信/送信.
uint32_t
SSP_Tx : 1 SSP 送信
uint32_t
SSP_Rx : 1 SSP 受信
uint32_t
Reserved : 1 未使用
uint32_t
AD_Convert_End : 1 A/D 変換終了

7.2.4.3 DMAC_SingleReqState

メンバ:

uint32_t

All: 全ての DMAC シングル要求状態

ビットフィールド:

uint32_t

Reserved : 13 未使用

uint32_t

SSP_Rx : 1 SSP 受信

8. FC

8.1 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 TMPM36xF10 の場合 1024Kbyte 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\tmpm36**x**_fc.c(*)
\Libraries\TX03_Periph_Driver\inc\ tmpm36**x**_fc.h(*)

補足: 1, 2, 3, 4 を"x"と記載します。

8.2 API 関数

8.2.1 関数一覧

- void FC_SetSecurityBit(FunctionalState ***NewState***)
- FunctionalState FC_GetSecurityBit(void)
- WorkState FC_GetBusyState(void)
- FunctionalState FC_GetBlockProtectState(uint8_t ***BlockNum***)

8.2.2 関数の種類

関数は、主に以下の 2 種類に分かれています:
1) 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()

8.2.3 関数仕様

8.2.3.1 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 (FLCS<BLPROn>)を”1” にします。
2) SECBIT<SECBIT>を”1”にします。

上記の 2 つの条件が成立すると、セキュリティ機能が有効になります。セキュリティ機能が有効な状態の制限内容は次の通りです。

- ROM 領域のデータの読み出し。
- JTAG/SW、トレースの通信

したがって、この API を使用する場合は、注意して実行してください。

SECBIT<SECBIT>はパワーオンリセットおよび低消費電力モードの STOP2 解除で初期化されます。

戻り値:

なし

8.2.3.2 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

8.2.3.3 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

8.2.3.4 FC_GetBlockProtectState

ブロックのプロテクト状態の取得

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

引数:
BlockNum: ブロック番号を選択します。
- FC_BLOCK_1 ~ FC_BLOCK_9

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

8.2.4 データ構造

なし

9. GPIO

9.1 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"と記載します。

9.2 TMPM361/362/363/364 の違い

Port C/D/H/K/O は TMPM362/M364 のみ選択できます。

9.3 API 関数

9.3.1 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_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***);

9.3.2 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_SetOpenDrain(), GPIO_Init()
3) その他:
GPIO_EnableFuncReg(), GPIO_DisableFuncReg()

9.3.3 関数仕様

9.3.3.1 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 (TMPM362/M364 のみ)
- **GPIO_PD:** GPIO port D (TMPM362/M364 のみ)
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H (TMPM362/M364 のみ)
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K (TMPM362/M364 のみ)
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PO:** GPIO port O (TMPM362/M364 のみ)
- **GPIO_PP:** GPIO port P

機能:
DATA レジスタを読み込みます。

戻り値:
DATA レジスタの値

9.3.3.2 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 (TMPM362/M364 のみ)
- **GPIO_PD:** GPIO port D (TMPM362/M364 のみ)
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H (TMPM362/M364 のみ)
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K (TMPM362/M364 のみ)
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PO:** GPIO port O (TMPM362/M364 のみ)
- **GPIO_PP:** GPIO port P

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

9.3.3.3 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_PC:** GPIO port C (TMPM362/M364 のみ)
- **GPIO_PD:** GPIO port D (TMPM362/M364 のみ)
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H (TMPM362/M364 のみ)
- **GPIO_PI:** GPIO port I
- **GPIO_PK:** GPIO port K (TMPM362/M364 のみ)

- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PO:** GPIO port O (TMPM362/M364 のみ)
- **GPIO_PP:** GPIO port P

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

9.3.3.4 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_PC:** GPIO port C (TMPM362/M364 のみ)
- **GPIO_PD:** GPIO port D (TMPM362/M364 のみ)
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H (TMPM362/M364 のみ)
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PO:** GPIO port O (TMPM362/M364 のみ)
- **GPIO_PP:** GPIO port P

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

BitValue: 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

9.3.3.5 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 (TMPM362/M364 のみ)
- **GPIO_PD:** GPIO port D (TMPM362/M364 のみ)
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H (TMPM362/M364 のみ)
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K (TMPM362/M364 のみ)
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PO:** GPIO port O (TMPM362/M364 のみ)
- **GPIO_PP:** GPIO port P

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

戻り値:
なし

9.3.3.6 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_PC:** GPIO port C (TMPM362/M364 のみ)
- **GPIO_PD:** GPIO port D (TMPM362/M364 のみ)
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H (TMPM362/M364 のみ)
- **GPIO_PI:** GPIO port I
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PO:** GPIO port O (TMPM362/M364 のみ)
- **GPIO_PP:** GPIO port P

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

機能:
出力ポートに設定します。

戻り値:
なし

9.3.3.7 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 (TMPM362/M364 のみ)
- **GPIO_PD:** GPIO port D (TMPM362/M364 のみ)
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H (TMPM362/M364 のみ)
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K (TMPM362/M364 のみ)
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PO:** GPIO port O (TMPM362/M364 のみ)
- **GPIO_PP:** GPIO port P

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

機能:
入力ポートに設定します。

戻り値:
なし

9.3.3.8 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_PC:** GPIO port C (TMPM362/M364 のみ)

- **GPIO_PD:** GPIO port D (TMPM362/M364 のみ)
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H (TMPM362/M364 のみ)
- **GPIO_PI:** GPIO port I
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PO:** GPIO port O (TMPM362/M364 のみ)
- **GPIO_PP:** GPIO port P

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

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

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

戻り値:
なし

9.3.3.9 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 (TMPM362/M364 のみ)
- **GPIO_PD:** GPIO port D (TMPM362/M364 のみ)
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H (TMPM362/M364 のみ)

- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K (TMPM362/M364 のみ)
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PO:** GPIO port O (TMPM362/M364 のみ)
- **GPIO_PP:** GPIO port P

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

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

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

戻り値:
なし

9.3.3.10 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 (TMPM362/M364 のみ)
- **GPIO_PD:** GPIO port D (TMPM362/M364 のみ)
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H (TMPM362/M364 のみ)
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K (TMPM362/M364 のみ)

- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PO:** GPIO port O (TMPM362/M364 のみ)
- **GPIO_PP:** GPIO port P

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

NewState:
- **ENABLE:** 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

9.3.3.11 GPIO_SetOpenDrain

CMOS/オープンドレインポートの設定

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

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C (TMPM362/M364 のみ)
- **GPIO_PD:** GPIO port D (TMPM362/M364 のみ)
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H (TMPM362/M364 のみ)
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PO:** GPIO port O (TMPM362/M364 のみ)
- **GPIO_PP:** GPIO port P

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

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

機能:
GPIO 端子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

9.3.3.12 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_PC:** GPIO port C (TMPM362/M364 のみ)
- **GPIO_PD:** GPIO port D (TMPM362/M364 のみ)
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H (TMPM362/M364 のみ)
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K (TMPM362/M364 のみ)
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PO:** GPIO port O (TMPM362/M364 のみ)
- **GPIO_PP:** GPIO port P

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

- **GPIO_FUNC_REG_3** GPIO 機能レジスタ 3

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

機能:
GPIO 端子の機能を有効に設定します。

戻り値:
なし

9.3.3.13 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_PC:** GPIO port C (TMPM362/M364 のみ)
- **GPIO_PD:** GPIO port D (TMPM362/M364 のみ)
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H (TMPM362/M364 のみ)
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K (TMPM362/M364 のみ)
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PO:** GPIO port O (TMPM362/M364 のみ)
- **GPIO_PP:** GPIO port P

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
- **GPIO_FUNC_REG_3** GPIO 機能レジスタ 3

Bit_x: GPIO 端子を選択します。有効ビットの組み合わせが可能です。

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

機能:
GPIO 端子の機能を無効に設定します。

戻り値:
なし

9.3.4 データ構造

9.3.4.1 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
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。

10. KWUP

10.1 概要

KWUP ドライバ API は、KWUP 入力、KWUP 動作、KWUP 割り込みなどを設定します。

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/tmpm36x_kwup.c(*)
/Libraries/TX03_Periph_Driver/inc/tmpm36x_kwup.h(*)

補足: 1, 2, 3, 4を"x"と記載します。

10.2 TMPM361/362/363/364 の違い

なし。

10.3 API 関数

10.3.1 関数一覧

- void KWUP_SetConfig(KWUP_SettingTypeDef* Settings);
- KWUP_PortStatus KWUP_GetPortStatus(void);
- void KWUP_SetPullUpConfig(KWUP_PullUpCycles T1, KWUP_PullUpCycles T2);
- void KWUP_ClearINTReq(void);
- KWUP_INTStatus KWUP_GetINTStatus(void);

10.3.2 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) KWUP の設定:

KWUP_SetConfig(), KWUP_SetPullUpConfig()

2) KWUP 入力の状態および割り込み状態の取得:

KWUP_GetPortStatus(), KWUP_GetINTStatus()

3) KWUP 割り込み要求のクリア:

KWUP_ClearINTReq()

10.3.3 関数仕様

10.3.3.1 KWUP_SetConfig

KWUP の初期設定

関数のプロトタイプ宣言:
void
KWUP_SetConfig(KWUP_SettingTypeDef * ***Settings***);

引数:
Settings: KWUP の初期化を行う構造体です。詳細は"データ構造"を参照してください。

機能:
KWUP の初期設定を行います。

戻り値:
なし

10.3.3.2 KWUP_GetPortStatus

KWUP 入力の端子状態の取得

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

引数:
なし。

機能:
KWUP 入力端子状態を取得します。

戻り値:
なし

10.3.3.3 KWUP_SetPullUpConfig

ダイナミックプルアップ期間とダイナミックプルアップ周期の設定

関数のプロトタイプ宣言:
void
KWUP_SetPullUpConfig(KWUP_PullUpCycles T1, KWUP_PullUpCycles T2);

引数:
T1: 以下からダイナミックプルアップ期間を選択します。
- **KWUP_CYCLES_2_FS**: 2/fs
- **KWUP_CYCLES_4_FS**: 4/fs
- **KWUP_CYCLES_8_FS**: 8/fs
- **KWUP_CYCLES_16_FS**: 16/fs

T2: 以下からダイナミックプルアップ周期を選択します。
- **KWUP_CYCLES_256_FS**: 256/fs,
- **KWUP_CYCLES_512_FS**: 512/fs,
- **KWUP_CYCLES_1024_FS**: 1024/fs,
- **KWUP_CYCLES_2048_FS**: 2048/fs,

機能:
ダイナミックプルアップ期間とダイナミックプルアップ周期を設定します。

戻り値:
なし

10.3.3.4 KWUP_ClearINTReq

すべての KWUP 割り込み要求のクリア

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

引数:
なし。

機能:
すべての KWUP 割り込み要求をクリアします。

戻り値:
なし

10.3.3.5 KWUP_GetINTStatus

KWUP 割り込み発生時に検出された KWUP 入力状態の取得

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

引数:
なし。

機能:
KWUP 割り込み発生時に検出された KWUP 入力状態を取得します。

戻り値:
KWUP 割り込み発生時に検出された KWUP 入力状態: “0” なし、“1” あり
Key0(Bit 0): KEYINT0
Key1(Bit 1): KEYINT1
Key2(Bit 2): KEYINT2
Key3(Bit 3): KEYINT3

10.3.4 データ構造

10.3.4.1 KWUP_SettingTypeDef

メンバ:
KWUP_Input
KeyN: KWUP 入力
- **KWUP_INPUT_0** :KWUP0
- **KWUP_INPUT_1** :KWUP1
- **KWUP_INPUT_2** :KWUP2
- **KWUP_INPUT_3** :KWUP3

KWUP_PullUpCtrl
PullUpCtrl: 以下からスタティックプルアップ、またはダイナミックプルアップを選択します。

- **KWUP_PUP_CTRL_BY_STATIC** : スタティックプルアップ
- **KWUP_PUP_CTRL_BY_DYNAMIC** : ダイナミックプルアップ

KWUP_ActiveState
ActiveState: 以下から KWUP 入力を検出するアクティブ状態を選択します。

- **KWUP_ACTIVE_BY_L_LEVEL** : "Low"レベル
- **KWUP_ACTIVE_BY_H_LEVEL** : "High"レベル
- **KWUP_ACTIVE_BY_RISING_EDGE** : 立ち上がりエッジ
- **KWUP_ACTIVE_BY_FALLING_EDGE**: 立下りエッジ
- **KWUP_ACTIVE_BY_BOTH_EDGES**: 両エッジ

FunctionalState
INTNewState: 以下から KWUP 割り込み要求を選択します。

- **DISABLE**: 禁止
- **ENABLE**: 許可

10.3.4.2 KWUP_PortStatus

メンバ:
uint32_t
All KWUPn(n=0~3)のポート状態

ビットフィールド:
uint32_t
Key0(Bit 0) KEY0 のポート状態

uint32_t
Key1(Bit 1) KEY1 のポート状態

uint32_t
Key2(Bit 2) KEY2 のポート状態

uint32_t
Key3(Bit 3) KEY3 のポート状態

10.3.4.3 KWUP_INTStatus

メンバ:
uint32_t
All KWUPn(n=0~3)の割り込み要求状態

ビットフィールド:
uint32_t
Key0(Bit 0) KEY0 の割り込み要求状態

uint32_t
Key1(Bit 1) KEY1 の割り込み要求状態

uint32_t

Key2(Bit 2) KEY2 の割り込み要求状態

uint32_t
Key3(Bit 3) KEY3 の割り込み要求状態

11. RC

11.1 概要

RAM の 0x2000_4000 ～ 0x2000_BFFF はリセット解除後 1WAIT に設定されています。0WAIT としても使用可能です。

本ドライバ API は、アプリで使用する API 定義を格納する以下のファイルで構成されています。
\Libraries\TX03_Periph_Driver\src\tmpm36**x**_rc.c(*)
\Libraries\TX03_Periph_Driver \inc\tmpm36**x**_rc.h(*)

補足: 1, 2, 3, 4 を"**x**"と記載します。

11.2 API 関数

11.2.1 関数一覧

- void RC_SetRAMWait(RC_RAMWait RAMWait);
- RC_RAMWait RC_GetRAMWait(void);

11.2.2 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) RCWAIT レジスタの設定

   RC_SetRAMWait ().

2) RCWAIT レジスタ設定値の取得

   RC_GetRAMWait().

11.2.3 関数仕様

11.2.3.1 RC_SetRAMWait

RAMWAIT 値の設定

関数のプロトタイプ宣言:
void
RC_SetRAMWait(RC_RAMWait ***RAMWait***);

引数:
RAMWait: 以下から RAMWAIT 値を選択します。
- **RC_RAMWAIT_0:** 0WAIT
- **RC_RAMWAIT_1:** 1WAIT

機能:
RAM (0x2000_4000 to 0x2000_BFFF)の WAIT 値を選択します。

戻り値:
なし

11.2.3.2 RC_GetRAMWait

RAMWAIT 値の取得

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

引数:
なし。

機能:
RAM (0x2000_4000 to 0x2000_BFFF)の WAIT 値を取得します。

戻り値:
RAM の WAIT 値:
RC_RAMWAIT_0: **0WAIT**
RC_RAMWAIT_1: **1WAIT**

11.2.4 データ構造

なし

12. RMC

12.1 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/tmpm36**x**_rmc.c(*)
/Libraries/TX03_Periph_Driver/inc/tmpm36**x**_rmc.h(*)

補足: 1, 2, 3, 4 を"**x**"と記載します。

12.2 TMPM361/362/363/364 の違い

TMPM362/TMPM364 は 2 チャネルの RMC を持っています。(RMC0, RMC1)
TMPM361/TMPM363 は 1 チャネルの RMC を持っています。

12.3 API 関数

12.3.1 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***)
- void RMC_SetRxEndBitNum(TSB_RMC_TypeDef * ***RMCx***,
  RMC_RxEndBitsReg ***Reg_x***, uint8_t ***BitNum***)
- void RMC_SetSrcClk(TSB_RMC_TypeDef * ***RMCx***, RMC_SrcClk ***Clk***)

12.3.2 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:
   RMC_SetLeaderDetection(), SetFallingEdgeINT(), RMC_SetSignalRxMethod(), RMC_SetRxTrg(), RMC_ SetThreshold(), RMC_SetInputSignalReversed(), RMC_SetNoiseCancellation(), RMC_SetRxEndBitNum(), RMC_SetSrcClk()
3) その他:
   RMC_GetINTFactor(), RMC_GetLeader(), RMC_GetRxData()

12.3.3 関数仕様

補足:** 引数“TSB_RMC_TypeDef * ***RMCx”は以下のいずれかをを指定してください。
TSB_RMC0, TSB_RMC1 (TMPM362/TMPM364 のみ)

12.3.3.1 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

12.3.3.2 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

12.3.3.3 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

12.3.3.4 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

12.3.3.5 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)

12.3.3.6 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

12.3.3.7 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

12.3.3.8 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

12.3.3.9 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

12.3.3.10 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

12.3.3.11 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

12.3.3.12 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

12.3.3.13 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)

12.3.3.14 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

12.3.3.15 RMC_SetRxEndBitNum

受信終了ビット数の設定

関数のプロトタイプ宣言:
void
RMC_SetRxEndBitNum(TSB_RMC_TypeDef * ***RMCx***,
RMC_RxEndBitsReg ***Reg_x***,
uint8_t ***BitNum***)

引数:
RMCx : RMC チャネルを指定します。
Reg_x: 受信終了ビット数レジスタを選択します。
- **RMC_RX_END_BITS_REG_1:** RMCxEND1 レジスタ。
- **RMC_RX_END_BITS_REG_2:** RMCxEND2 レジスタ。
- **RMC_RX_END_BITS_REG_3:** RMCxEND3 レジスタ。

BitNum: 受信するデータのビット数を設定します。

機能:
受信終了ビット数を設定します。

戻り値:
なし

12.3.3.16 RMC_SetSrcClk

RMC サンプリングクロックの選択

関数のプロトタイプ宣言:
void
RMC_SetSrcClk(TSB_RMC_TypeDef * ***RMCx***,
RMC_SrcClk ***Clk***)

引数:
RMCx : RMC チャネルを指定します。
Clk: RMC サンプリングクロックを選択します。
- **RMC_CLK_LOW_FREQUENCY:** 低速クロック(32KHz)
- **RMC_CLK_TB1OUT:** タイマ出力(TB1OUT).

機能:
RMC サンプリングクロックを選択します。

戻り値:
なし

12.3.4 データ構造

12.3.4.1 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)

12.3.4.2 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。

12.3.4.3 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。

12.3.4.4 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

13. RTC

13.1 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
- 1MHz クロック出力機能

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

本ドライバ は、アプリで使用する API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm36**x**_rtc.c(*)
/Libraries/ TX03_Periph_Driver/inc/tmpm36**x**_rtc.h(*)

補足: 1, 2, 3, 4 を"**x**"と記載します。

13.2 TMPM361/362/363/364 の違い

なし。

13.3 API 関数

13.3.1 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_ResetAlarm(void) ;
- void RTC_ResetClockSec(void);
- RTC_ReqState RTC_GetResetClockSecReq(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***);

13.3.2 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_GetResetClockSecReq(), RTC_SetClockValue(), RTC_GetClockValue()
4) RTC(alarm)の設定:
   RTC_EnableAlarm(), RTC_DisableAlarm(), RTC_SetAlarmValue(), RTC_ResetAlarm(), RTC_GetAlarmValue()
5) その他:
   RTC_SetAlarmOutput(), RTC_SetRTCINT()

13.3.3 関数仕様

13.3.3.1 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

13.3.3.2 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

13.3.3.3 RTC_SetMin

時計/アラームの分桁設定

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

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

Min: 最大 59 までの分桁を設定します。

機能:
NewMode が **RTC_CLOCK_MODE** の場合、時計の分桁を設定します。
NewMode が **RTC_ALARM_MODE** の場合、アラームの分桁を設定します。
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

13.3.3.4 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

13.3.3.5 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

13.3.3.6 RTC_SetHour24

24 時間モードの時計/アラーム時桁設定

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

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

Hour: 最大 23 までの時桁を設定します。

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

戻り値:
なし

13.3.3.7 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

13.3.3.8 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

13.3.3.9 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

13.3.3.10 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

13.3.3.11 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

13.3.3.12 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

13.3.3.13 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

13.3.3.14 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

13.3.3.15 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

13.3.3.16 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

13.3.3.17 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

13.3.3.18 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

13.3.3.19 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

13.3.3.20 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

13.3.3.21 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

13.3.3.22 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

13.3.3.23 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

13.3.3.24 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

13.3.3.25 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

13.3.3.26 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

13.3.3.27 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

13.3.3.28 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

13.3.3.29 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

13.3.3.30 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

13.3.3.31 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

13.3.3.32 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

13.3.3.33 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

13.3.3.34 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

13.3.3.35 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

13.3.3.36 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

13.3.3.37 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

13.3.3.38 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

13.3.3.39 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

13.3.4 データ構造

13.3.4.1 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

13.3.4.2 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。

13.3.4.3 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)。

14. SBI

14.1 概要

本デバイスはシリアルバスインターフェースチャンネルを有し、各チャンネルはマルチマスタが可能 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/tmpm36**x**_sbi.c(*)
/Libraries/TX03_Periph_Driver/inc/tmpm36**x**_sbi.h(*)

補足: 1, 2, 3, 4 を“**x**” と記載します。

14.2 TMPM361/362/363/364 の違い

TMPM362/M364 は 5 チャネルの SBI を持っています。(SBI0, SBI1, SBI2, SBI3, SBI4)
TMPM361/M363 は 4 チャネルの SBI を持っています。(SBI0, SBI1, SBI2, SBI3)

14.3 API 関数

14.3.1 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***);

14.3.2 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()

14.3.3 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_SBI3, TSB_SBI4 (**TSB_SBI4** は TMPM362/TMPM364 のみ選択可能です)

14.3.3.1 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

14.3.3.2 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

14.3.3.3 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

14.3.3.4 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

14.3.3.5 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

14.3.3.6 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

14.3.3.7 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

14.3.3.8 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

14.3.3.9 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

14.3.3.10 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

14.3.3.11 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

14.3.3.12 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

14.3.3.13 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:
受信データ

14.3.3.14 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

14.3.4 データ構造

14.3.4.1 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。

14.3.4.2 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

15. SMC

15.1 概要

非同期型の外部メモリ(NOR フラッシュメモリ 、SRAM 等)を制御する SMC(Static Memory Controller)を内蔵しています。
SMC データバス幅は 16 ビットで、マルチプレクスバスに対応しています。AC タイミング(tTR, tPC, tWP, tCEOE, tWC, tRC)は内蔵レジスタにて制御します。

64MB空間のアクセス空間をサポートします。
CS0：0x6000_0000～0x60FF_FFFF (16MB)
CS1：0x6100_0000～0x61FF_FFFF (16MB)
CS2：0x6200_0000～0x62FF_FFFF (16MB)
CS3：0x6300_0000～0x63FF_FFFF (16MB)

SMC ドライバ API は、バスモード、サイクル、動作モードを含む外部スタティックメモリ用 SMC の設定を行う関数セットを提供します。

全ドライバ API は、アプリで使用する API 定義、マクロ、データタイプ、構造を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm36**x**_smc.c(*)
/Libraries/TX03_Periph_Driver/inc/tmpm36**x**_smc.h(*)

補足: 1, 2, 3, 4 を“**x**” と記載します。

15.2 TMPM361/362/363/364 の違い

TMPM361, TMPM363 はマルチプレクスバスのみ選択可能です。
TMPM362, TMPM364 はセパレートバスとマルチプレクスバスのいずれかを選択可能です。

15.3 API 関数

15.3.1 関数一覧

- void SMC_SetBusMode(uint8_t ***BusMode***);
- void SMC_GetIFConfig(SMC_IFConfigTypeDef * ***Config***);
- void SMC_SendDirectCMD(SMC_DirectCMDTypeDef * ***Cmd***);
- void SMC_SetCycles(SMC_CyclesTypeDef * ***Cycles***);
- void SMC_SetOpMode(SMC_OpModeTypeDef * ***OpMode***);
- void SMC_GetCycles(uint8_t ***ChipSelect***, SMC_CyclesTypeDef * ***Cycles***);
- void SMC_GetOpMode(uint8_t ***ChipSelect***, SMC_OpModeTypeDef * ***OpMode***);

15.3.2 関数の種類

SMC の設定:

SMC_SetBusMode(), SMC_GetIFConfig(), SMC_SendDirectCMD(), SMC_SetCycles(), SMC_SetOpMode(), SMC_GetCycles(), SMC_GetOpMode()

15.3.3 関数仕様

15.3.3.1 SMC_SetBusMode

SMC の動作モード設定

関数のプロトタイプ:
void
SMC_SetBusMode(uint8_t ***BusMode***);

引数:
BusMode: SMC の動作モードを選択します。
[TMPM362/M364]
- **SMC_BUS_SEPARATE**: セパレートバスモード
- **SMC_BUS_MULTIPLEX**: マルチプレクスバスモード

[TMPM361/M363]
- **SMC_BUS_MULTIPLEX**: マルチプレクスバスモード

機能:
SMC のバスモードを設定します。動作モードはセパレートバスモード、またはマルチプレクスバスモードを選択できます。

補足:
TMPM361/M363 は、セパレートバスモードを選択できません。

戻り値:
なし

15.3.3.2 SMC_GetIFConfig

SMC メモリインタフェース設定状態の取得

関数のプロトタイプ:
void
SMC_GetIFConfig(SMC_IFConfigTypeDef * ***Config***);

引数:
Config: SMC メモリインタフェース設定状態を格納する構造体です。(詳細は"データ構造"を参照してください。

機能:
SMC メモリインタフェース設定状態を取得します。

補足:
TMPM361/M363 は、セパレートバスモードを選択できないため、動作モードはマルチプレスクバスモード(**SMC_MULTIPLEX_SRAM**)となります。TMPM362/M364 は、セパレートバスモード(**SMC_SEPARATE_SRAM**)または、マルチプレクスバスモード(**SMC_MULTIPLEX_SRAM**)のいずれかとなります。

戻り値:

なし

15.3.3.3 SMC_SendDirectCMD

Static Memory へのダイレクトコマンド転送

関数のプロトタイプ:
void
SMC_SendDirectCMD(SMC_DirectCMDTypeDef * Cmd);

引数:
Cmd: Static Memory へのダイレクトコマンドを格納した構造体です。(詳細は"データ構造"を参照してください)

機能:
Static Memory へのダイレクトコマンドを転送します。

戻り値:
なし

15.3.3.4 SMC_SetCycles

Static Memory のアクセスサイクル設定

関数のプロトタイプ:
void
SMC_SetCycles(SMC_CyclesTypeDef * ***Cycles***);

引数:
Cycles: Static Memory のアクセスサイクル(tTR, tPC, tWP, tCEOE, tWC, tRC)を格納した構造体です。(詳細は"データ構造"を参照してください)

機能:
Static Memory のアクセスサイクルを設定します。アクセスサイクルは、tTR, tPC, tWP, tCEOE, tWC, tRC です。

補足:
Static Memory が要求する AC 特性に合わせて設定してください。SMCCLK は fsys/2 です。設定を有効にするには、**SMC_SendDirectCMD()**にて **SMC_CMD_UPDATEREGS** を設定する必要があります。

サイクル設定時は以下の制約があります。
セパレートバスモード:
tCEOE + SMCCLK*1clk <= tRC
tWP + SMCCLK*2clk <= tWC
マルチプレクスバスモード:
tCEOE + SMCCLK*2clk <= tRC
tWP + SMCCLK*3clk <= tWC

マルチプレクスバスモードではページアクセスは未サポートですので、tPC はセパレートバスモードのみ有効です。

戻り値:
なし

15.3.3.5 SMC_SetOpMode

Static Memory の動作モード設定

関数のプロトタイプ:
void
SMC_SetOpMode(SMC_OpModeTypeDef * ***OpMode***);

引数:
OpMode: 動作モードを格納した構造体です。(詳細は、"データ構造"を参照してください)

機能:
Static Memory の動作モードを設定します。動作モードは、データバス幅、メモリリード時のバースト長、ALE フィールドを含みます。

補足:
SMC 動作モードの設定を有効にするには、**SMC_SendDirectCMD()**にて **SMC_CMD_UPDATEREGS** を実行する必要があります。ALE はセパレートバスモードでは無効にしてください。

戻り値:
なし

15.3.3.6 SMC_GetCycles

Static Memory のアクセスサイクル設定状態の取得

関数のプロトタイプ:
void
SMC_GetCycles(uint8_t ***ChipSelect***,
SMC_CyclesTypeDef * ***Cycles***);

引数:
ChipSelect: チップ番号を選択します。
- **SMC_CS0**: chip 0
- **SMC_CS1**: chip 1
- **SMC_CS2**: chip 2
- **SMC_CS3**: chip 3

Cycles: Static Memory のアクセスサイクル(tTR, tPC, tWP, tCEOE, tWC, tRC)設定状態を格納する構造体です。(詳細は、"データ構造"を参照してください)

機能:
Static Memory アクセスサイクル(tTR, tPC, tWP, tCEOE, tWC, tRC)を取得します。

戻り値:
なし

15.3.3.7 SMC_GetOpMode

Static Memory の動作モード設定状態の取得

関数のプロトタイプ:
void

SMC_GetOpMode(uint8_t ***ChipSelect***,
SMC_OpModeTypeDef * ***OpMode***);

引数:
ChipSelect: チップ番号を選択します。
- **SMC_CS0**: chip 0
- **SMC_CS1**: chip 1
- **SMC_CS2**: chip 2
- **SMC_CS3**: chip 3

OpMode: Static Memory の動作モード(データバス幅、リード時のバースト長、ALE フィールド)設定状態を格納する構造体です。(詳細は、“データ構造”を参照してください)

機能:
Static Memory の動作モード(データバス幅、リード時のバースト長、ALE フィールド)設定状態を取得します。

戻り値:
なし

15.3.4 データ構造

15.3.4.1 SMC_IFConfigTypeDef

メンバ:
uint8_t
MemoryType: サポートするメモリの種類を選択します。
[TMPM362/M364]
- **SMC_SEPARATE_SRAM**: セパレートバスタイプ SRAM
- **SMC_ MULTIPLEX_SRAM**: マルチプレクスタイプ SRAM

[TMPM361/M363]
- **SMC_ MULTIPLEX_SRAM**: マルチプレクスタイプ SRAM

uint8_t
MemoryChips: サポートするメモリの CS 数を選択します。
- **SMC_MEMORY_CHIPS_1**: 1 chip
- **SMC_MEMORY_CHIPS_2**: 2 chip
- **SMC_MEMORY_CHIPS_3**: 3 chip
- **SMC_MEMORY_CHIPS_4**: 4 chip

uint8_t
MemoryWidth: 外部 SMC メモリバス幅の最大値を選択します。
- **SMC_DATA_BUS_16BIT**: 16 ビット

15.3.4.2 SMC_DirectCMDTypeDef

メンバ:
uint8_t
CmdType: ダイレクトコマンドを設定します。
- **SMC_CMD_UPDATEREGS**: レジスタ更新

uint8_t
ChipSelect: CS を選択します。
- **SMC_CS0**: chip 0

- **SMC_CS1**: chip 1
- **SMC_CS2**: chip 2
- **SMC_CS3**: chip 3

15.3.4.3 SMC_CyclesTypeDef

メンバ:

uint8_t

RC_Time: リードサイクル時間(tRC)を選択します。

- **SMC_READ_CYCLE_TIME_2:** SMCCLK×2clk
- **SMC_READ_CYCLE_TIME_3:** SMCCLK×3clk
- **SMC_READ_CYCLE_TIME_4:** SMCCLK×4clk
- **SMC_READ_CYCLE_TIME_5:** SMCCLK×5clk
- **SMC_READ_CYCLE_TIME_6:** SMCCLK×6clk
- **SMC_READ_CYCLE_TIME_7:** SMCCLK×7clk
- **SMC_READ_CYCLE_TIME_8:** SMCCLK×8clk
- **SMC_READ_CYCLE_TIME_9:** SMCCLK×9clk
- **SMC_READ_CYCLE_TIME_10:** SMCCLK×10clk
- **SMC_READ_CYCLE_TIME_11:** SMCCLK×11clk
- **SMC_READ_CYCLE_TIME_12:** SMCCLK×12clk
- **SMC_READ_CYCLE_TIME_13:** SMCCLK×13clk
- **SMC_READ_CYCLE_TIME_14:** SMCCLK×14clk
- **SMC_READ_CYCLE_TIME_15:** SMCCLK×15clk

uint8_t

WC_Time: ライトサイクル時間(tWC)を選択します。

- **SMC_WRITE_CYCLE_TIME_3:** SMCCLK×3clk
- **SMC_WRITE_CYCLE_TIME_4:** SMCCLK×4clk
- **SMC_WRITE_CYCLE_TIME_5:** SMCCLK×5clk
- **SMC_WRITE_CYCLE_TIME_6:** SMCCLK×6clk
- **SMC_WRITE_CYCLE_TIME_7:** SMCCLK×7clk
- **SMC_WRITE_CYCLE_TIME_8:** SMCCLK×8clk
- **SMC_WRITE_CYCLE_TIME_9:** SMCCLK×9clk
- **SMC_WRITE_CYCLE_TIME_10:** SMCCLK×10clk
- **SMC_WRITE_CYCLE_TIME_11:** SMCCLK×11clk
- **SMC_WRITE_CYCLE_TIME_12:** SMCCLK×12clk
- **SMC_WRITE_CYCLE_TIME_13:** SMCCLK×13clk
- **SMC_WRITE_CYCLE_TIME_14:** SMCCLK×14clk
- **SMC_WRITE_CYCLE_TIME_15:** SMCCLK×15clk

uint8_t

CEOE_Time: OEn のディレイ時間(tCEOE)を選択します。

- **SMC_CEOE_CYCLE_TIME_1:** SMCCLK×1clk
- **SMC_CEOE_CYCLE_TIME_2:** SMCCLK×2clk
- **SMC_CEOE_CYCLE_TIME_3:** SMCCLK×3clk
- **SMC_CEOE_CYCLE_TIME_4:** SMCCLK×4clk
- **SMC_CEOE_CYCLE_TIME_5:** SMCCLK×5clk
- **SMC_CEOE_CYCLE_TIME_6:** SMCCLK×6clk
- **SMC_CEOE_CYCLE_TIME_7:** SMCCLK×7clk

uint8_t

WP_Time: WEn の幅(tWP)を選択します。

- **SMC_WE_PULSE_CYCLE_TIME_1:** SMCCLK×1clk
- **SMC_WE_PULSE_CYCLE_TIME_2:** SMCCLK×2clk
- **SMC_WE_PULSE_CYCLE_TIME_3:** SMCCLK×3clk
- **SMC_WE_PULSE_CYCLE_TIME_4:** SMCCLK×4clk
- **SMC_WE_PULSE_CYCLE_TIME_5:** SMCCLK×5clk
- **SMC_WE_PULSE_CYCLE_TIME_6:** SMCCLK×6clk

- **SMC_WE_PULSE_CYCLE_TIME_7:** SMCCLK×7clk

uint8_t
PC_Time: ページサイクル時間(tPC)を選択します。
- **SMC_PAGE_CYCLE_TIME_1:** SMCCLK×1clk
- **SMC_PAGE_CYCLE_TIME_2:** SMCCLK×2clk
- **SMC_PAGE_CYCLE_TIME_3:** SMCCLK×3clk
- **SMC_PAGE_CYCLE_TIME_4:** SMCCLK×4clk
- **SMC_PAGE_CYCLE_TIME_5:** SMCCLK×5clk
- **SMC_PAGE_CYCLE_TIME_6:** SMCCLK×6clk
- **SMC_PAGE_CYCLE_TIME_7:** SMCCLK×7clk

uint8_t
TR_Time: SRAM chip 設定のターンアラウンド時間(tTR)を選択します。
- **SMC_TURN_AROUND_CYCLE_TIME_1:** SMCCLK×1clk
- **SMC_TURN_AROUND_CYCLE_TIME_2:** SMCCLK×2clk
- **SMC_TURN_AROUND_CYCLE_TIME_3:** SMCCLK×3clk
- **SMC_TURN_AROUND_CYCLE_TIME_4:** SMCCLK×4clk
- **SMC_TURN_AROUND_CYCLE_TIME_5:** SMCCLK×5clk
- **SMC_TURN_AROUND_CYCLE_TIME_6:** SMCCLK×6clk
- **SMC_TURN_AROUND_CYCLE_TIME_7:** SMCCLK×7clk

15.3.4.4 SMC_OpModeTypeDef

メンバ:
uint8_t
BusWidth: メモリデータバス幅を設定します。
- **SMC_DATA_BUS_16BIT**: 16 ビット

uint8_t
ReadBurstLen: リード用メモリのバースト長を選択します。
- **SMC_READ_BURST_1BEAT**: 1 ビート
- **SMC_READ_BURST_4BEAT**: 4 ビート

FunctionalState
ALE: アドレスラッチイネーブル信号(ALEn)の使用可否を選択します。
- **DISABLE:** ALEn を使用しない
- **ENABLE:** ALEn を使用する

uint8_t
StartAddr: スタートアドレスの設定値を選択します。
- **SMC_START_ADDR_CHIP0**: CS0 のスタートアドレス
- **SMC_START_ADDR_CHIP1**: CS1 のスタートアドレス
- **SMC_START_ADDR_CHIP2**: CS2 のスタートアドレス
- **SMC_START_ADDR_CHIP3**: CS3 のスタートアドレス

16. SSP

16.1 概要

本デバイスは、同期式シリアルインタフェースを (SSP：Synchronous Serial Port) を 1 チャネル内蔵しています。

同期式シリアルインタフェースは、周辺デバイスとシリアル通信を、3 タイプの同期式シリアルインタフェースで行います。

同期式シリアルインタフェースは、周辺デバイスから受信したデータのシリアル-パラレル変換を行います。送信パスは、送信モードの 16 ビット幅、8 層の送信 FIFO のデータをバッファリングし、受信パスは受信モードの 16 ビット幅、8 層の受信 FIFO のデータをバッファリングします。シリアルデータは SPDO で送信され、SPDI で受信されます。 SSP はプログラマブルプリスケーラを内蔵し、入力クロック fSYS からシリアル出力クロック(CPCLK)を出力します。生成します。動作モード、フレームフォーマット、SSP のデータサイズは制御レジスタにプログラムされています。

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/tmpm36**x**_ssp.c
/Libraries/TX03_Periph_Driver/inc/tmpm36**x**_ssp.h

補足: 1, 2, 3, 4 を"**x**"と記載します。

16.2 TMPM361/362/363/364 の違い

なし。

16.3 API 関数

16.3.1 関数一覧

- void SSP_Enable(void);
- void SSP_Disable(void);
- void SSP_Init(SSP_InitTypeDef * ***InitStruct***);
- void SSP_SetClkPreScale(uint8_t ***PreScale***, uint8_t ***ClkRate***);
- void SSP_SetFrameFormat(SSP_FrameFormat ***FrameFormat***);
- void SSP_SetClkPolarity(SSP_ClkPolarity ***ClkPolarity***);
- void SSP_SetClkPhase(SSP_ClkPhase ***ClkPhase***);
- void SSP_SetDataSize(uint8_t ***DataSize***);
- void SSP_SetSlaveOutputCtrl(FunctionalState ***NewState***);
- void SSP_SetMSMode(SSP_MS_Mode ***Mode***);
- void SSP_SetLoopBackMode(FunctionalState ***NewState***);
- void SSP_SetTxData(uint16_t ***Data***);
- uint16_t SSP_GetRxData(void);
- WorkState SSP_GetWorkState(void);
- SSP_FIFOState SSP_GetFIFOState(SSP_Direction ***Direction***);
- void SSP_SetINTConfig(uint32_t ***IntSrc***);
- SSP_INTState SSP_GetINTConfig(void);
- SSP_INTState SSP_GetPreEnableINTState(void);

- SSP_INTState SSP_GetPostEnableINTState(void);
- void SSP_ClearINTFlag(uint32_t ***IntSrc***);
- void SSP_SetDMACtrl(SSP_Direction ***Direction***, FunctionalState ***NewState***);

16.3.2 関数の種類

関数は、主に以下の 6 種類に分かれています:

1) 共通関数:
   SSP_Init(),SSP_SetClkPreScale(), SSP_SetFrameFormat(), SSP_SetClkPolarity(), SSP_SetClkPhase(), SSP_SetDataSize(), SSP_SetMSMode()
2) データ送受信:
   SSP_SetTxData(), SSP_GetRxData()
3) SSP 割り込み関連:
   SSP_SetINTConfig(), SSP_GetINTConfig(), SSP_GetPreEnableINTState(), SSP_GetPostEnableINTState(), SSP_ClearINTFlag()
4) 状態の取得:
   SSP_GetWorkState(), SSP_GetFIFOState()
5) モジュールの有効/無効設定:
   SSP_Enable(), SSP_Disable()
6) その他:
   SSP_SetSlaveOutputCtrl(), SSP_SetLoopBackMode(), SSP_SetDMACtrl()

16.3.3 関数仕様

16.3.3.1 SSP_Enable

同期式シリアルインタフェース動作の許可

関数のプロトタイプ宣言:
void
SSP_Enable(void)

引数:
なし

機能:
SSP 動作を有効にします。

戻り値:
なし

16.3.3.2 SSP_Disable

同期式シリアルインタフェース動作の禁止

関数のプロトタイプ宣言:
void
SSP_Disable(void)

引数:
なし

機能:

SSP 動作を無効にします。

戻り値:
なし

16.3.3.3 SSP_Init

SSP 通信の初期化

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

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

機能:
SSP 通信の初期化を行います。
本 API がコールする API は以下の通りです。
SSP_SetFrameFormat(),
SSP_SetClkPreScale(),
SSP_SetClkPolarity(),
SSP_SetClkPhase(),
SSP_SetDataSize(),
SSP_SetMSMode().

戻り値:
なし

16.3.3.4 SSP_SetClkPreScale

送受信のビットレート設定

関数のプロトタイプ宣言:
void
SSP_SetClkPreScale(uint8_t ***PreScale***,
uint8_t ***ClkRate***)

引数:
PreScale: クロックプリスケール除数を 2～254 の間で設定します。
ClkRate: シリアルクロックレートを 0～255 の間で設定します。

機能:
送受信のビットレートを設定します。**SSP_Init()** によりコールされます。
Tx と Rx 用の本ビットレートは下記計算式で求めることができます。
BitRate = fSYS / (PreScale × (1 + ClkRate))
fSYS はシステム周波数

戻り値:
なし

16.3.3.5 SSP_SetFrameFormat

フレームフォーマットの選択

関数のプロトタイプ宣言:
void
SSP_SetFrameFormat(SSP_FrameFormat ***FrameFormat***)

引数:
FrameFormat: フレームフォーマットを選択します。
- **SSP_FORMAT_SPI:** SPI フレームフォーマット
- **SSP_FORMAT_SSI:** SSI シリアルフレームフォーマット
- **SSP_FORMAT_MICROWIRE:** Microwire フレームフォーマット

機能:
フレームフォーマットを選択します。**SSP_Init()** からコールされます。

戻り値:
なし

16.3.3.6 SSP_SetClkPolarity

SPxCLK 極性の選択

関数のプロトタイプ宣言:
void
SSP_SetClkPolarity(SSP_ClkPolarity ***ClkPolarity***)

引数:
SSPx*:* SSP チャネルを指定します。
ClkPolarity: SPxCLK 極性を選択します。
- **SSP_POLARITY_LOW:** SPxCLK は Low 状態。
- **SSP_POLARITY_HIGH:** SPxCLK は High 状態。

機能:
SPxCLK 極性を選択します。**SSP_Init()** からコールされます。

戻り値:
なし

16.3.3.7 SSP_SetClkPhase

SPxCLK フェーズの選択

関数のプロトタイプ宣言:
void
SSP_SetClkPhase(SSP_ClkPhase ***ClkPhase***)

引数:
ClkPhase: SPxCLK フェーズを選択します。
- **SSP_PHASE_FIRST_EDGE:** 1st クロックエッジでデータを取り込み
- **SSP_PHASE_SECOND_EDGE:** 2nd クロックエッジでデータを取り込み

機能:
SPxCLK フェーズを選択します。**SSP_Init()** からコールされます。

戻り値:
なし

16.3.3.8 SSP_SetDataSize

データサイズの選択

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

引数:
DataSize: データサイズを 4～16 の間で選択します。

機能:
データサイズを選択します。**SSP_Init()** からコールれます。

戻り値:
なし

16.3.3.9 SSP_SetSlaveOutputCtrl

スレーブモード SPxDO 出力の制御

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

引数:
NewState: スレーブモード SPxDO 出力の許可/禁止を選択します。

- **ENABLE:** 許可。
- **DISABLE:** 禁止。

機能:
スレーブモード SPxDO 出力の許可/禁止を選択します。

戻り値:
なし

16.3.3.10 SSP_SetMSMode

マスタ/ スレーブモードの選択

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

引数:

Mode: マスタ/ スレーブモードを選択します。
- **SSP_MASTER:** デバイスがマスタ。
- **SSP_SLAVE:** デバイスがスレーブ。

機能:
マスタ/ スレーブモードを選択します。

戻り値:
なし

16.3.3.11 SSP_SetLoopBackMode

ループバックモードの制御

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

引数:
NewState: ループバックモードの許可/禁止を選択します。
- **ENABLE**: 許可。
- **DISABLE**: 禁止。

機能:
ループバックモードを設定します。
例えば、ループバックモードが有効の場合、送受信間にセルフテストを行います。

戻り値:
なし

16.3.3.12 SSP_SetTxData

送信 FIFO のデータ設定

関数のプロトタイプ宣言:
void
SSP_SetTxData(uint16_t ***Data***)

引数:
Data: 送信データを 4～16 ビットの間で設定します。

機能:
送信 FIFO にデータを設定します。

戻り値:
なし

16.3.3.13 SSP_GetRxData

受信 FIFO からのデータ読み込み

関数のプロトタイプ宣言:
uint16_t
SSP_GetRxData(void)

引数:
なし。

機能:
受信 FIFO から受信データを読み込みます。

戻り値:
受信データ

16.3.3.14 SSP_GetWorkState

ビジーフラグの読み込み

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

引数:
なし。

機能:
ビジーフラグを読み込みます。

戻り値:
ビジーフラグ
BUSY: ビジー
DONE: アイドル

16.3.3.15 SSP_GetFIFOState

送受信 FIFO の読み込み

関数のプロトタイプ宣言:
SSP_FIFOState
SSP_GetFIFOState(SSP_Direction ***Direction***)

引数:
Direction: 送受信方向を選択します。

- **SSP_RX**: 受信 FIFO
- **SSP_TX**: 送信 FIFO

機能:
送受信 FIFO の状態を読み込みます。
例えば、送信 FIFO の状態を判断した後でのデータ送信処理は次の通り。

```
SSP_FIFOState  fifoState;

fifoState = SSP_GetFIFOState(TSB_SSP0, SSP_TX);
if ((fifoState == SSP_FIFO_EMPTY) || (fifoState == SSP_FIFO_NORMAL))
```

```
{ SSP_SetTxData(SSP0, data_to_be_sent ); }
```

戻り値:
送受信 FIFO の状態:
SSP_FIFO_EMPTY: FIFO が空の状態。
SSP_FIFO_NORMAL: FIFO がフル、かつ空ではない状態。
SSP_FIFO_INVALID: FIFO が無効の状態。
SSP_FIFO_FULL: FIFO がフルの状態。

16.3.3.16 SSP_SetINTConfig

割り込みの制御

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

引数:
IntSrc: 割り込みの許可/禁止を選択します。
- **SSP_INTCFG_NONE:** すべて禁止。
- **SSP_INTCFG_ALL** :すべて許可。

任意の割り込みを“ **|** ”で選択します。
- **SSP_INTCFG_RX_OVERRUN:** 受信オーバーラン割り込み。
- **SSP_INTCFG_RX_TIMEOUT:** 受信タイムアウト割り込み。
- **SSP_INTCFG_RX:** 受信 FIFO 割り込み(受信 FIFO の半分以上がフル)
- **SSP_INTCFG_TX:** 送信 FIFO 割り込み(送信 FIFO の半分以上がフル)

機能:
割り込みの許可/ 禁止を選択します。
例えば、送受信割り込みを設定する処理は次の通り。

```
SSP_SetINTConfig(TSB_SSP, SSP_INTCFG_RX | SSP_INTCFG_TX )
```

戻り値:
なし

16.3.3.17 SSP_GetINTConfig

割り込み制御の読み込み

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

引数:
なし。

機能:
割り込みの許可/禁止状態を取得します。
例えば、SSP_SetINTConfig()で許可または禁止した割り込みソースを確認することができます。

戻り値:
SSP_INTState: 割り込み設定状態。詳細は"データ構造"を参照。

16.3.3.18 SSP_GetPreEnableINTState

許可前の割り込み状態の読み込み

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

引数:
なし。

機能:
許可前の割り込み状態を読み込みます。

戻り値:
SSP_INTState: 許可前の割り込み状態。詳細は"データ構造"を参照。

16.3.3.19 SSP_GetPostEnableINTState

許可後の割り込み状態の読み込み

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

引数:
なし。

機能:
禁止前の割り込み状態を読み込みます。

戻り値:
SSP_INTState: 許可前の割り込み状態。詳細は"データ構造"を参照。

16.3.3.20 SSP_ClearINTFlag

割り込みフラグのクリア

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

引数:
IntSrc: クリアする割り込みフラグを選択します。

- **SSP_INTCFG_RX_OVERRUN:** 受信オーバーラン割り込みフラグ。
- **SSP_INTCFG_RX_TIMEOUT:** 受信タイムアウト割り込みフラグ
- **SSP_INTCFG_ALL:** すべての割り込みフラグ。

機能:
割り込みフラグをクリアします。

戻り値:
なし

16.3.3.21 SSP_SetDMACtrl

送受信 FIFO の DMA 制御

関数のプロトタイプ宣言:
void
SSP_SetDMACtrl(SSP_Direction ***Direction***,
FunctionalState ***NewState***)

引数:
Direction: 送受信方向を選択します。
- **SSP_RX**: 受信。
- **SSP_TX**: 送信。

NewState: DMA FIFO の状態。
- **ENABLE**: 許可。
- **DISABLE**: 禁止。

機能:
送受信 FIFO の DMA 許可/禁止を選択します。

戻り値:
なし

16.3.4 データ構造

16.3.4.1 SSP_InitTypeDef

メンバ:
SSP_FrameFormat
FrameFormat: フレームフォーマットを選択します。
- **SSP_FORMAT_SPI**: SPI フレームフォーマット
- **SSP_FORMAT_SSI**: SSI フレームフォーマット
- **SSP_FORMAT_MICROWIRE**: Microwire フレームフォーマット

uint8_t
PreScale: クロックプリスケール除数を 2～254 の間で設定します。

SSP_ClkPolarity
ClkPolarity: SPxCLK 極性を選択します。
- **SSP_POLARITY_LOW**: SPxCLK 極性は Low 状態。
- **SSP_POLARITY_HIGH**: SPxCLK 極性は High 状態。

SSP_ClkPhase
ClkPhase: SPxCLK フェーズを設定します。
- **SSP_PHASE_FIRST_EDGE**: 1st クロックエッジでデータを取り込み

- **SSP_PHASE_SECOND_EDGE**: 2nd クロックエッジでデータを取り込み

uint8_t
DataSize: データを 4～16 ビットの間で設定します。

SSP_MS_Mode
Mode: マスタ/ スレーブモードを選択します。
- **SSP_MASTER**: デバイスがマスタ
- **SSP_SLAVE**: デバイスがスレーブ

16.3.4.2 SSP_INTState

メンバ:
uint32_t
All: 割り込み要因

ビットフィールド:

uint32_t		
OverRun:	1	オーバー欄割り込み
uint32_t		
TimeOut:	1	受信タイムアウト
uint32_t		
Rx:	1	受信
uint32_t		
Tx:	1	送信
uint32_t		
Reserved:	1	未使用

17. TMRB

17.1 概要

本デバイスは、16 チャンネルの多機能 16 ビットタイマ/ イベントカウンタ (TMRB0 ～ TMRBF)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/tmpm36**x**_tmrb.c(*)
/Libraries/TX03_Periph_Driver/inc/tmpm36**x**_tmrb.h(*)

補足: 1, 2, 3, 4 を"**x**"と記載します。

17.2 TMPM361/362/363/364 の違い

ありません。

17.3 API 関数

17.3.1 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***, uint8_t ***WriteRegMode***)

17.3.2 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()

17.3.3 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, TSB_TBA, TSB_TBB, TSB_TBC, TSB_TBD, TSB_TBE, TSB_TBF

17.3.3.1 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。
チャネルが MPT の場合、本関数は、タイマモードとして MPT チャネルも選択します。

戻り値:
なし

17.3.3.2 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

17.3.3.3 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

17.3.3.4 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

17.3.3.5 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_TB1, TSB_TB2, TSB_TB3, TSB_TB5, TSB_TB6, TSB_TB7, TSB_TB9, TSB_TBA, TSB_TBB, TSB_TBD, TSB_TBE, TSB_TBF**

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。(TMRB1~3: TB4OUT、TMRB5~7: TB0OUT、TMRB9~B: TBCOUT、TMRBD~F: TB8OUT)
- **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:
(*)TBxIN0, TBxIN1 端子入力については、TMRB1, 2, 5～7, 9～B, D～F が対象です。

17.3.3.6 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

17.3.3.7 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
}
```

17.3.3.8 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

17.3.3.9 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。

17.3.3.10 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。

17.3.3.11 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

17.3.3.12 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

17.3.3.13 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

17.3.3.14 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

17.3.3.15 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, TSB_TB9, TSB_TBA, TSB_TBB, TSB_TBD, TSB_TBE, TSB_TBF
***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、TMRB9 ～TMRBB を同期モードに設定すると、TMRB8 のスタートに同期して動作がスタートし、TMRBD ～TMRBF を同期モードに設定すると、TMRBC 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, TMRB8, TMRBC のカウントを開始する前に、**TMRB_SetRunState()** によって TMRB1 ～ TMRB3、TMRB5 ～ TMRB7、TMRB9 ～ TMRBB、TMRBD ～ TMRBF をスタートしてください。

17.3.3.16 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

17.3.4 データ構造

17.3.4.1 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。

17.3.4.2 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。

17.3.4.3 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

18. SIO/UART

18.1 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/tmpm36**x**_uart.c(*)
/Libraries/TX03_Periph_Driver/inc/tmpm36**x**_uart.h(*)

補足: 1, 2, 3, 4 を"**x**"と記載します。

18.2 TMPM361/362/363/364 の違い

TMPM362、TMPM364 の場合、SC0(UART0)～SC11(UART11)を選択可能です。
TMPM361、TMPM363 の場合、SC0(UART0)～SC4(UART4)を選択可能です。

18.3 API 関数

18.3.1 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_TRxAutoDisable ***TRxAutoDisable***);
- void UART_RxFIFOINTCtrl(TSB_SC_TypeDef * ***UARTx***, FunctionalState ***NewState***);
- void UART_TxFIFOINTCtrl(TSB_SC_TypeDef * ***UARTx***, FunctionalState ***NewState***);
- void UART_RxFIFOByteSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***BytesUsed***);
- void UART_RxFIFOFillLevel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***RxFIFOLevel***);
- void UART_RxFIFOINTSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***RxINTCondition***);
- void UART_RxFIFOClear(TSB_SC_TypeDef * ***UARTx***);

- void UART_TxFIFOFillLevel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***TxFIFOLevel***);
- void UART_TxFIFOINTSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***TxINTCondition***);
- void UART_TxFIFOClear(TSB_SC_TypeDef * ***UARTx***);
- 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_SetInputClock(TSB_SC_TypeDef * ***SIOx***, uint32_t ***Clock***)
- 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***,
  UART_InitTypeDef* ***InitStruct***)

18.3.2 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()

18.3.3 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。
TMPM362, TMPM364 の場合: **UART0～UART11**
TMPM361, TMPM363 の場合: **UART0～UART4**
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。
TMPM362, TMPM364 の場合: **SIO0～SIO11**
TMPM361, TMPM363 の場合: **SIO0～SIO4**

18.3.3.1 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

18.3.3.2 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

18.3.3.3 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

18.3.3.4 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

18.3.3.5 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

18.3.3.6 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

18.3.3.7 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

18.3.3.8 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: なし
送受信有効。ボーレートジェネレータはソースクロックとして使用。

戻り値:
なし

18.3.3.9 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

18.3.3.10 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

18.3.3.11 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

18.3.3.12 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

18.3.3.13 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

18.3.3.14 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

18.3.3.15 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

18.3.3.16 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

18.3.3.17 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

18.3.3.18 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

18.3.3.19 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

18.3.3.20 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

18.3.3.21 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

18.3.3.22 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

18.3.3.23 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

18.3.3.24 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

18.3.3.25 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

18.3.3.26 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

18.3.3.27 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

18.3.3.28 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

18.3.3.29 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

18.3.3.30 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)

18.3.3.31 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

18.3.3.32 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

18.3.4 データ構造

18.3.4.1 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。

18.3.4.2 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

19. WDT

19.1 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\tmpm36**x**_wdt.c(*)
\Libraries/TX03_Periph_Driver\inc\tmpm36**x**_wdt.h(*)

補足: 1, 2, 3, 4 を"x"と記載します。

19.2 TMPM361/362/363/364 の違い

なし。

19.3 API 関数

19.3.1 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)

19.3.2 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().

19.3.3 関数仕様

19.3.3.1 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

19.3.3.2 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

19.3.3.3 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

19.3.3.4 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

19.3.3.5 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

19.3.3.6 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

19.3.3.7 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

19.3.4 データ構造

19.3.4.1 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