TOSHIBA

TX04 ペリフェラルドライバ
ユーザーガイド
(TMPM461)

第一版

2017 年 9 月

東芝デバイス＆ストレージ株式会社

CMDR-M461UG-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TX04シリーズマイコン用ペリフェラルドライバセットです。TMPM461x (※2参照) ペリフェラルドライバは、東芝TX04ペリフェラルドライバのTMPM461シリーズMCU用です。

TX04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。

TMPM461 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TMPM461F10FG/TMPM461F15FGを表す部分を"TMPM461x"で表現します。

2. TX04 ペリフェラルドライバの構成

/Libraries
TX04 CMSIS ファイルと TMPM461 ペリフェラルドライバが格納されています。

/Libraries/ TX04_CMSIS
このフォルダには TMPM461 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/TX04_Periph_Driver
TMPM461 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/TX04_Periph_Driver/inc
TMPM461 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/TX04_Periph_Driver/src
TMPM461 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
TMPM461 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
TMPM461 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
TMPM461 ペリフェラルドライバの使用例が格納されています。

/Utilities/TMPM461-EVAL
TMPM461 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 1 ユニット内蔵し、合計 15 チャネルの通常アナログ入力と 5 チャネルの拡張アナログ入力を有します。
15 チャネルのアナログ入力端子(AIN0 ~ AIN14)と、5 チャネルの拡張アナログ入力端子 (AIN15 ~ AIN19) は、入出力ポートと兼用です。
12 ビット A/D コンバータは、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/TX04_Periph_Driver/src/tmpm461_adc.c
/Libraries/TX04_Periph_Driver/inc/tmpm461_adc.h

補足:
1 拡張チャネルの通常AD変換機能は、固定シングル変換モード、固定リピート変換モードのみです。
2 AD変換のAD入力としてポートHを使うためには、PHIEの入力禁止とPHPUPのプルアップ設定禁止を行ってください。

3.2 API 関数

3.2.1 関数一覧

- void ADC_SWReset(TSB_AD_TypeDef * ***ADx***)
- void ADC_SetClk(TSB_AD_TypeDef * ***ADx***,
  uint32_t ***Sample_HoldTime***,
  uint32_t ***Prescaler_Output***)
- void ADC_Start(TSB_AD_TypeDef * ***ADx***)

- void ADC_SetScanMode(TSB_AD_TypeDef * ***ADx***,
  FunctionalState ***NewState***)
- void ADC_SetRepeatMode(TSB_AD_TypeDef * ***ADx***,
  FunctionalState ***NewState***)
- void ADC_SetINTMode(TSB_AD_TypeDef * ***ADx***, uint32_t ***INTMode***)
- void ADC_SetInputChannel(TSB_AD_TypeDef * ***ADx***, ADC_AINx ***InputChannel***)
- void ADC_SetScanChannel(TSB_AD_TypeDef * ***ADx***,
  ADC_AINx ***StartChannel***,
  uint32_t ***Range***)
- void ADC_SetVrefCut(TSB_AD_TypeDef * ***ADx***, uint32_t ***VrefCtrl***)
- void ADC_SetIdleMode(TSB_AD_TypeDef * ***ADx***, FunctionalState ***NewState***)
- void ADC_SetVref(TSB_AD_TypeDef * ***ADx***, FunctionalState ***NewState***)
- void ADC_SetInputChannelTop(TSB_AD_TypeDef * ***ADx***,
  ADC_AINx ***TopInputChannel***)
- void ADC_StartTopConvert(TSB_AD_TypeDef * ***ADx***)
- void ADC_SetMonitor(TSB_AD_TypeDef * ***ADx***,
  ADC_CMPCRx ***ADCMPx***,
  FunctionalState ***NewState***)
- void ADC_ConfigMonitor(TSB_AD_TypeDef * ***ADx***,
  ADC_CMPCRx ***ADCMPx***,
  ADC_MonitorTypeDef * ***Monitor***)
- void ADC_SetHWTrg(TSB_AD_TypeDef * ***ADx***,
  uint32_t ***HWSrc***,
  FunctionalState ***NewState***)
- void ADC_SetHWTrgTop(TSB_AD_TypeDef * ***ADx***,
  uint32_t ***HWSrc***,
  FunctionalState ***NewState***)
- ADC_State ADC_GetConvertState(TSB_AD_TypeDef * ***ADx***)
- ADC_Result ADC_GetConvertResult(TSB_AD_TypeDef * ***ADx***,
  ADC_REGx ***ADREGx***)
- void ADC_EnableTrigger(void)
- void ADC_DisableTrigger(void)
- ADC_SetTriggerStartup(ADC_TRGx ***TriggerStartup***)
- ADC_SetTriggerStartupTop(ADC_TRGx ***TopTriggerStartup***)
- ADC_DisableExtension(void)
- ADC_EnableExtension(void)
- ADC_SetExtChannel(ADC_EXT_AINx ***ExtChannel***)
- ADC_SetExtClkSource(uint32_t ***ClkSrc***)
- ADC_SetExtClk (uint32_t ***Sample_HoldTime***)

3.2.2 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) 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_SetHWTrg(), ADC_SetHWTrgTop()
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_GetConvertResult()
4) その他:
   ADC_SWReset(), ADC_SetVrefCut(), ADC_SetIdleMode()

5) AD 変換起動:
ADC_EnableTrigger(), ADC_DisableTrigger(), ADC_SetTriggerStartup(), ADC_SetTriggerStartupTop()
6) 拡張 AD 変換の設定:
ADC_DisableExtension(), ADC_EnableExtension(), ADC_SetExtChannel(), ADC_SetExtClkSource(), ADC_SetExtClk()

3.2.3 関数仕様

3.2.3.1 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(TSB_AD_TypeDef * ***ADx***)

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

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

3.2.3.2 ADC_SetClk

AD 変換サンプルホールド時間とプリスケーラ出力の設定

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

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

Sample_HoldTime: 以下から ADC サンプルホールド時間を選択します。
- **ADC_CONVERSION_CLK_10:** 10 x <ADCLK>
- **ADC_CONVERSION_CLK_20:** 20 x <ADCLK>
- **ADC_CONVERSION_CLK_30:** 30 x <ADCLK>
- **ADC_CONVERSION_CLK_40:** 40 x <ADCLK>
- **ADC_CONVERSION_CLK_80:** 80 x <ADCLK>
- **ADC_CONVERSION_CLK_160:** 160 x <ADCLK>
- **ADC_CONVERSION_CLK_320:** 320 x <ADCLK>

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

3.2.3.3 ADC_Start

AD 変換の開始

関数のプロトタイプ宣言:
void
ADC_Start(TSB_AD_TypeDef * ***ADx***)

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

機能:
通常(ソフト)AD 変換を開始します。

補足:
通常 AD 変換には次の 4 種類の動作モードが用意されています。本関数を使用する前に、予め下記変換モードを指定してください:

チャネル固定シングル変換モード
チャネルスキャンシングル変換モード
チャネル固定リピート変換モード
チャネルスキャンリピート変換モード

詳細は下記関数の機能を参照してください。

ADC_SetScanMode(), ADC_SetRepeatMode(), ADC_SetInputChannel(), ADC_SetScanChannel()

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

3.2.3.4 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(TSB_AD_TypeDef * ***ADx***,
FunctionalState ***NewState***)

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

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

3.2.3.5 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(TSB_AD_TypeDef * ***ADx***,
FunctionalState ***NewState***)

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

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

3.2.3.6 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(TSB_AD_TypeDef * ***ADx***,
uint32_t ***INTMode***)

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

***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_2:** 2 回変換ごと割り込み発生。
- **ADC_INT_CONVERSION_3:** 3 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_5:** 5 回変換ごと割り込み発生。
- **ADC_INT_CONVERSION_6:** 6 回変換ごと割り込み発生。
- **ADC_INT_CONVERSION_7:** 7 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

3.2.3.7 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(TSB_AD_TypeDef * ADx,
                    ADC_AINx InputChannel)
```

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

InputChannel: AD 変換入力チャネルを選択します。
- **ADC_AN_00, ADC_AN_01, ADC_AN_02, ADC_AN_03,ADC_AN_04, ADC_AN_05, ADC_AN_06, ADC_AN_07, ADC_AN_08, ADC_AN_09, ADC_AN_10, ADC_AN_11, ADC_AN_12, ADC_AN_13, ADC_AN_14, ADC_AN_EXT**

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

***補足:**
1 **ADC_AN_00**~**ADC_AN_EXT** の内 1 チャネルだけ通常変換入力を選択可能です。
2 InputChannel が **ADC_AN_EXT** の場合、**AIN15** から **AIN19** のいずれか 1 つを選択してください。(**ADC_SetExtChannel()**の説明を参照してください)

戻り値:
なし

3.2.3.8 ADC_SetScanChannel

AD 変換スキャンチャネルの選択

関数のプロトタイプ宣言:
void
ADC_SetScanChannel(TSB_AD_TypeDef * ***ADx***,
ADC_AINx ***StartChannel***,
uint32_t ***Range***)

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

StartChannel: スキャン開始チャネルを指定します。
- **ADC_AN_00, ADC_AN_01, ADC_AN_02, ADC_AN_03, ADC_AN_04, ADC_AN_05, ADC_AN_06, ADC_AN_07, ADC_AN_08, ADC_AN_09, ADC_AN_10, ADC_AN_11, ADC_AN_12, ADC_AN_13, ADC_AN_14**

Range: チャネルスキャンの範囲を設定します。
- **1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 (補足: StartChannel + Range <= 15)**

機能:
StartChannel の指定により AD 変換開始チャネルを指定し、***Range*** の指定によりチャネルスキャン範囲を指定します。

補足:
有効なチャネルスキャンの設定値を以下に示します:

StartChannel	***Range*** ***(***指定可能なチャネルスキャン値の範囲***)***
ADC_AN_00	1 ～ 15
ADC_AN_01	1 ～ 14
ADC_AN_02	1 ～ 13
ADC_AN_03	1 ～ 12
ADC_AN_04	1 ～ 11
ADC_AN_05	1 ～ 10
ADC_AN_06	1 ～ 9
ADC_AN_07	1 ～ 8
ADC_AN_08	1 ～ 7
ADC_AN_09	1 ～ 6
ADC_AN_10	1 ～ 5
ADC_AN_11	1 ～ 4
ADC_AN_12	1 ～ 3
ADC_AN_13	1 ～ 2
ADC_AN_14	1

上記以外の設定を行った場合は、**ADC_Start()** が呼び出されても AD 変換は行われません。

戻り値:
なし

3.2.3.9 ADC_SetVrefCut

AVREFH-AVREFL 間のリファレンス電流制御

関数のプロトタイプ宣言:
void
ADC_SetVrefCut(TSB_AD_TypeDef * ***ADx***,
uint32_t ***VrefCtrl***)

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

VrefCtrl: AVREFH-AVREFL 間のリファレンス電流の制御方法を指定します。
- **ADC_APPLY_VREF_IN_CONVERSION:** 変換中のみ通電。
- **ADC_APPLY_VREF_AT_ANY_TIME:** リセット時以外常時通電。

機能:
VrefCtrl の設定により AVREFH-AVREFL 間のリファレンス電流を制御します。

戻り値:
なし

3.2.3.10 ADC_SetIdleMode

IDLE モード時の ADC 動作制御の指定

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

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

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

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

戻り値:
なし

3.2.3.11 ADC_SetVref

ADC Vref アプリケーションの回路 ON/OFF 制御

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

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

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

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

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

戻り値:
なし

3.2.3.12 ADC_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(TSB_AD_TypeDef * ***ADx***,
ADC_AINx ***TopInputChannel***)

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

TopInputChannel:最優先 AD 変換入力チャネルを選択します。
- **ADC_AN_00, ADC_AN_01, ADC_AN_02, ADC_AN_03, ADC_AN_04, ADC_AN_05, ADC_AN_06, ADC_AN_07, ADC_AN_08, ADC_AN_09, ADC_AN_10, ADC_AN_11, ADC_AN_12, ADC_AN_13, ADC_AN_14, ADC_AN_EXT**

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:
1 最優先 AD 変換入力チャネルには、**ADC_AN_0~ADC_AN_EXT** のうちの一つを選ぶことができます。
2 TopInputChannel が **ADC_AN_EXT** の場合、**AIN15** から **AIN19** のいずれか 1 つを選択してください。(**ADC_SetExtChannel()**の説明を参照してください)

戻り値:

なし

3.2.3.13 ADC_StartTopConvert

最優先 AD 変換の開始

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

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

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

3.2.3.14 ADC_SetMonitor

AD 監視機能の設定

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

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

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

3.2.3.15 ADC_ConfigMonitor

ADC 監視モジュールの設定

関数のプロトタイプ宣言:
void
ADC_ConfigMonitor(TSB_AD_TypeDef * ***ADx***,
ADC_CMPCRx ***ADCMPx***,
ADC_MonitorTypeDef * ***Monitor***)

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

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

Monitor: AD 監視設定の構造体を指定します。ADC_MonitorTypeDef 構造体の詳細は"データ構造"を参照してください。

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、***Monitor*** で AD 監視設定を行います。

補足: 本関数の実行前に AD 監視モジュールを無効にしてください。

戻り値:
なし

3.2.3.16 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(TSB_AD_TypeDef * ***ADx***,
uint32_t ***HWSrc***,
FunctionalState ***NewState***)

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

HWSrc: 通常 AD 変換の起動ソースを選択します。
- **ADC_EXTERADTRG:** ADTRG 端子
- **ADC_INTERTRIGGER:** 内部トリガ (ADILVTRGSEL<TRGSEL>にて選択)

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:**
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

3.2.3.17 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(TSB_AD_TypeDef * ***ADx***,
uint32_t ***HWSrc***,
FunctionalState ***NewState***)

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

HWSrc: 最優先 AD 変換の起動ソースを選択します。
- **ADC_EXTERADTRG:** $\overline{\text{ADTRG}}$ 端子
- **ADC_INTERTRIGGER:** 内部トリガ (ADILVTRGSEL<HPTRGSEL>にて選択)

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:**
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

3.2.3.18 ADC_GetConvertState

通常 AD 変換完了フラグおよび最優先 AD 変換完了フラグの確認

関数のプロトタイプ宣言:
ADC_State
ADC_GetConvertState(TSB_AD_TypeDef * ***ADx***)

引数:
ADx: AD 変換のユニットを指定します。
- **TSB_AD:** AD コンバータユニット

機能:

通常 AD 変換ステートおよび最優先 AD 変換ステートを確認します。本関数を実行することで、AD 変換が終了したかどうか確認できます。

戻り値:
通常 AD 変換状態:

- **NormalBusy**(Bit 0) : 通常 AD 変換中の場合、'1'がセットされます。
- **NormalComplete** (Bit 1) : 通常 AD 変換完了の場合、'1'がセットされます。
- **TopBusy**(Bit 2) : 最優先 AD 変換中の場合、'1'がセットされます。
- **TopComplete** (Bit 3) : 最優先 AD 変換完了の場合、'1'がセットされます。

3.2.3.19 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(TSB_AD_TypeDef * ***ADx***,
ADC_REGx ***ADREGx***)

引数:
ADx: AD 変換のユニットを指定します。

- **TSB_AD:** AD コンバータユニット

ADREGx: AD 変換結果レジスタを選択します。

- **ADC_REG_00**, **ADC_REG_01**, **ADC_REG_02**, **ADC_REG_03,** **ADC_REG_04**, **ADC_REG_05**, **ADC_REG_06**, **ADC_REG_07,** **ADC_REG_08**, **ADC_REG_09**, **ADC_REG_10**, **ADC_REG_11,** **ADC_REG_12**, **ADC_REG_13**, **ADC_REG_14**, **ADC_REG_15,** **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。
アナログ入力チャネルと AD 変換結果レジスタの関係を下表に示します。

チャネル固定シングルモード		
ユニット	チャネル	変換結果レジスタ
TSB_AD	ADC_AN_00	ADC_REG_00
	ADC_AN_01	ADC_REG_01
	ADC_AN_02	ADC_REG_02
	ADC_AN_03	ADC_REG_03
	ADC_AN_04	ADC_REG_04
	ADC_AN_05	ADC_REG_05
	ADC_AN_06	ADC_REG_06
	ADC_AN_07	ADC_REG_07
	ADC_AN_08	ADC_REG_08
	ADC_AN_09	ADC_REG_09
	ADC_AN_10	ADC_REG_10
	ADC_AN_11	ADC_REG_11
	ADC_AN_12	ADC_REG_12
	ADC_AN_13	ADC_REG_13
	ADC_AN_14	ADC_REG_14

	ADC_AN_EXT	ADC_REG_15

チャネル固定リピートモード	
割り込みコード	変換結果レジスタ
1 回変換ごと割り込み発生	ADC_REG_00
2 回変換ごと割り込み発生	ADC_REG_00 - ADC_REG_01
3 回変換ごと割り込み発生	ADC_REG_00 - ADC_REG_02
4 回変換ごと割り込み発生	ADC_REG_00 - ADC_REG_03
5 回変換ごと割り込み発生	ADC_REG_00 - ADC_REG_04
6 回変換ごと割り込み発生	ADC_REG_00 - ADC_REG_05
7 回変換ごと割り込み発生	ADC_REG_00 - ADC_REG_06
8 回変換ごと割り込み発生	ADC_REG_00 - ADC_REG_07

チャネルスキャンシングルモード/ リピートモード			
ユニット	開始チャネル	スキャンチャネル範囲	変換結果レジスタ
TSB_AD	ADC_AN_00	15 チャネル	ADC_REG_00 - ADC_REG_14
	ADC_AN_01	14 チャネル	ADC_REG_01 - ADC_REG_14
	ADC_AN_02	13 チャネル	ADC_REG_02 - ADC_REG_14
	ADC_AN_03	12 チャネル	ADC_REG_03 - ADC_REG_15
	ADC_AN_04	11 チャネル	ADC_REG_04 - ADC_REG_14
	ADC_AN_05	10 チャネル	ADC_REG_05 - ADC_REG_14
	ADC_AN_06	9 チャネル	ADC_REG_06 - ADC_REG_14
	ADC_AN_07	8 チャネル	ADC_REG_07 - ADC_REG_14
	ADC_AN_08	7 チャネル	ADC_REG_08 - ADC_REG_14
	ADC_AN_09	6 チャネル	ADC_REG_09 - ADC_REG_14
	ADC_AN_10	5 チャネル	ADC_REG_10 - ADC_REG_14
	ADC_AN_11	4 チャネル	ADC_REG_11 - ADC_REG_14
	ADC_AN_12	3 チャネル	ADC_REG_12 - ADC_REG_14
	ADC_AN_13	2 チャネル	ADC_REG_13 - ADC_REG_14
	ADC_AN_14	1 チャネル	ADC_REG_14

AD 変換モードの詳細は、関連 API を参照ください。

***補足:**
1 最優先 AD 変換の結果は "ADC_REG_SP" に格納されます。
2 12 ビット拡張 A/D 変換の結果は"ADC_REG_15"に格納されます

戻り値:
AD 変換結果:
ADResult (Bit 0 - Bit 11) : AD 変換値が格納されます。
Stored (Bit 12) : AD 変換値が格納されると‘1’がセットされます。このフラグはリードすると‘0’にクリアされます。
OverRun (Bit 13) 新しい AD 変換値が上書きされると‘1’がセットされます。このフラグはリードすると‘0’にクリアされます。

3.2.3.20 ADC_EnableTrigger

トリガ動作の有効

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

引数:
なし

機能:
トリガを有効にします。

戻り値:
なし

3.2.3.21 ADC_DisableTrigger

トリガ動作の無効

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

引数:
なし

機能:
トリガを無効にします。

戻り値:
なし

3.2.3.22 ADC_SetTriggerStartup

通常 AD 変換起動トリガの選択

関数のプロトタイプ宣言:
void
ADC_SetTriggerStartup(ADC_TRGx ***TriggerStartup***)

引数:
TriggerStartup: 通常 AD 変換起動トリガを選択します。

- **ADC_TRG_00**, **ADC_TRG_01**, **ADC_TRG_02**, **ADC_TRG_03,** **ADC_TRG_04**, **ADC_TRG_05**, **ADC_TRG_06**, **ADC_TRG_07,** **ADC_TRG_08**, **ADC_TRG_09**

機能:
通常 AD 変換起動トリガを選択します。

戻り値:
なし

3.2.3.23 ADC_SetTriggerStartupTop

最優先 AD 変換起動トリガの選択

関数のプロトタイプ宣言:
void
ADC_SetTriggerStartupTop(ADC_TRGx ***TopTriggerStartup***)

引数:
TopTriggerStartup: 最優先 AD 変換起動トリガを選択します。

- **ADC_TRG_00**, **ADC_TRG_01**, **ADC_TRG_02**, **ADC_TRG_03,** **ADC_TRG_04**, **ADC_TRG_05**, **ADC_TRG_06**, **ADC_TRG_07,** **ADC_TRG_08**, **ADC_TRG_09**

機能:
最優先 AD 変換起動トリガを選択します。

戻り値:
なし

3.2.3.24 ADC_DisableExtension

拡張チャネルの使用禁止

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

引数:
なし

機能:
拡張チャネルを使用しません。

戻り値:
なし

3.2.3.25 ADC_EnableExtension

拡張チャネルの使用許可

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

引数:
なし

機能:
拡張チャネルを使用します。

戻り値:

なし

3.2.3.26 ADC_SetExtChannel

拡張チャネルの選択

関数のプロトタイプ宣言:
void
ADC_SetExtChannel(ADC_EXT_AINx ***ExtChannel***)

引数:
ExtChannel: 拡張チャネルを選択します。
- **ADC_EXT_AN_15**, **ADC_EXT_AN_16**, **ADC_EXT_AN_17**, **ADC_EXT_AN_18**, **ADC_EXT_AN_19**

機能:
拡張チャネルを選択します。

戻り値:
なし

3.2.3.27 ADC_SetExtClkSource

拡張 AD 変換のサンプリング期間のソース選択

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

引数:
ClkSrc: 拡張 AD 変換のサンプリング期間のソースを選択します。
- **ADC_EXT_CLK_SRC_EXCR**: EXCR0 の設定のサンプリング期間を選択します。
- **ADC_EXT_CLK_SRC_ADCLK**: ADCLK<ADSH[3:0]>の設定を選択します。

機能:
拡張 AD 変換のサンプリング期間のソースを選択します。

戻り値:
なし

3.2.3.28 ADC_SetExtClk

拡張チャネルのサンプリング期間を選択します。

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

引数:
Sample_HoldTime: 拡張 AD 変換のサンプリング期間を選択します。

- **ADC_CONVERSION_CLK_10:** 10 x <ADCLK>
- **ADC_CONVERSION_CLK_20:** 20 x <ADCLK>

機能:
拡張 AD 変換のサンプリング期間を選択します。

戻り値:
なし

3.2.4 データ構造

3.2.4.1 ADC_MonitorTypeDef

メンバ:

ADC_AINx
CmpChannel : ADC チャネルを指定します。
ADC_AN_00 - ADC_AN_14, ADC_AN_EXT (16 チャネル)

uint32_t
CmpCnt 大小判定カウント数を設定します。1 - 16 回まで指定できます。

ADC_CmpCondition
Condition 大小判定を設定します。
- **ADC_LARGER_THAN_CMP_REG:** 比較レジスタ 0 よりも変換結果レジスタの値が大きいと割り込みを発生します。
- **ADC_SMALLER_THAN_CMP_REG:** 比較レジスタ 0 よりも変換結果レジスタの値が小さいと割り込みを発生します。

ADC_CmpCntMode
CntMode 判定カウント条件を指定します。
- **ADC_SEQUENCE_CMP_MODE:** 連続方式
- **ADC_CUMULATION_CMP_MODE:** 蓄積方式

uint32_t
CmpValue ADCMP0 または ADCMP1 に設定する比較値を指定します。値は 0 - 4095 まで指定できます。

(補足: 詳細はデータシートの“AD モニタ機能”を参照してください。)

3.2.4.2 ADC_State

メンバ:

uint32_t
All AD 変換の状態を指定します。

ビットフィールド
uint32_t
NormalBusy(Bit 0) 通常 AD 変換 BUSY フラグ(ADBF)
'1': 変換中
'0': 変換停止

uint32_t
NormalComplete (Bit 1)　通常 AD 変換終了フラグ (EOCF).
'1': 変換終了
'0': 変換前または変換中

uint32_t
TopBusy(Bit 2)　最優先 AD 変換 BUSY フラグ(HPADBF)
'1': 変換中
'0': 変換停止

uint32_t
TopComplete (Bit 3)　最優先 AD 変換終了フラグ(HPEOCF)
'1': 変換終了
'0': 変換前または変換中

uint32_t
Reserved (Bit 4 - Bit 31)　未使用

3.2.4.3 ADC_Result

メンバ:
uint32_t
All　AD 変換結果

ビットフィールド:
uint32_t
ADResult (Bit 0 - Bit 11)　AD 変換結果値

uint32_t
Stored (Bit 12)　AD 変換結果格納フラグ
'1': 変換結果あり
'0': 変換結果なし

uint32_t
OverRun (Bit 13)　オーバーランフラグ
'1': 発生あり
'0': 発生なし

uint32_t
Reserved (Bit 14 - Bit 31)　未使用

4. CEC

4.1 概要

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

受信

- 32kHz クロックまたは 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/TX04_Periph_Driver/src/tmpm461_cec.c
/Libraries/TX04_Periph_Driver/inc/tmpm461_cec.h

4.2 API 関数

4.2.1 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***)

4.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています:

1) 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()

4.2.3 関数仕様

4.2.3.1 CEC_Enable

CEC 動作の許可

関数のプロトタイプ宣言:

void
CEC_Enable(void)

引数:

なし

機能:

CEC 動作を許可します。使用前に CEC を許可してください。

戻り値:

なし

4.2.3.2 CEC_Disable

CEC 動作の禁止

関数のプロトタイプ宣言:

void
CEC_Disable(void)

引数:

なし

機能:

CEC 動作を禁止します。

戻り値:

なし

4.2.3.3 CEC_SWReset

ソフトウェアリセットの発生

関数のプロトタイプ宣言:

void
CEC_SWReset(void)

引数:

なし

機能:

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

戻り値:

なし

4.2.3.4 CEC_DefaultConfig

デフォルト値の設定

関数のプロトタイプ宣言:

Result
CEC_DefaultConfig(void)

引数:

なし

機能:

CEC を以下の値で設定します。
Idle Mode: on
Noise Cancellation Time: H: 1 cycle L: 1 cycle
Cycle Range: 2.05ms~2.75ms
Data Valid Time: 1.05ms
Time Out: 1 Bit
Rx Start Wave configure: Min of start: 3.5ms; Max of start: 3.9ms
Min of cycle: 4.3ms; Max of cycle: 4.7ms
Receive Bit Wave configure: Min of “1”: 0.4ms; Max of “1”: 0.8ms
Min of “0”:1.3ms; Max of “0”: 1.7
Send Bit Wave configure: RV
Bus free configure: 5 bit cycle

Snoop mode: On
CEC が受信許可、または、送信中の場合、本関数は **ERROR** を返します。

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.2.3.5 CEC_SetLogicalAddr

ロジカルアドレスの設定

関数のプロトタイプ宣言:

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

引数:

LogicalAddr: 以下から設定するロジカルアドレスを選択します。

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

機能:

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

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.2.3.6 CEC_AddLogicalAddr

ロジカルアドレスの追加

関数のプロトタイプ宣言:

Result
CEC_AddLogicalAddr(CEC_LogicalAddr ***LogicalAddr***)

引数:

LogicalAddr: 以下から追加するロジカルアドレスを指定します。

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

機能:

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

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.2.3.7 CEC_RemoveLogicalAddr

ロジカルアドレスの削除

関数のプロトタイプ宣言:

Result
CEC_RemoveLogicalAddr(CEC_LogicalAddr ***LogicalAddr***)

引数:

LogicalAddr: 以下から削除するロジカルアドレスを指定します。

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

機能:

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

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

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.2.3.8 CEC_GetLogicalAddr

ロジカルアドレスの取得

関数のプロトタイプ宣言:

CEC_AddrListTypeDef
CEC_GetLogicalAddr(void)

引数:

なし

機能:

ロジカルアドレス情報を取得します。

戻り値:

ロジカルアドレス情報:

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

4.2.3.9 CEC_SetRxCtrl

データ受信制御

関数のプロトタイプ宣言:

void
CEC_SetRxCtrl(FunctionalState ***NewState***)

引数:

NewState: データ受信機能有効 / 無効を選択します。

- **ENABLE** : 有効
- **DISABLE :** 無効

機能:

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

戻り値:

なし

4.2.3.10 CEC_StartTx

送信の開始

関数のプロトタイプ宣言:

Result
CEC_StartTx(void)

引数:

なし

機能:

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

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.2.3.11 CEC_StopTx

送信の停止

関数のプロトタイプ宣言:

void
CEC_StopTx(void)

引数:

なし

機能:

送信を停止します。

戻り値:

なし

4.2.3.12 CEC_GetRxData

受信データの読み出し

関数のプロトタイプ宣言:

CEC_DataTypeDef
CEC_GetRxData (void)

引数:

なし

機能:

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

戻り値:

受信バッファからの受信データ
- **CEC_DataTypeDef**: 受信データ構造体 (詳細は “データ構造” を参照してください)

4.2.3.13 CEC_SetTxData

送信データの設定

関数のプロトタイプ宣言:

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

引数:

Data: 送信データを指定します。データ長は 1 バイトです。

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

機能:

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

戻り値:

なし

4.2.3.14 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

4.2.3.15 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

4.2.3.16 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

4.2.3.17 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

4.2.3.18 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

4.2.3.19 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 cycles
- **CEC_LOW_CANCELLATION_3**: 3 cycles
- **CEC_LOW_CANCELLATION_4**: 4 cycles

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 cycles
- **CEC_HIGH_CANCELLATION_3**: 3 cycles
- **CEC_HIGH_CANCELLATION_4**: 4 cycles

機能:

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

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.2.3.20 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+2cycles
- **CEC_CYCLE_MIN_3**: 2.05ms+3cycles
- **CEC_CYCLE_MIN_4**: 2.05ms-1cycles
- **CEC_CYCLE_MIN_5**: 2.05ms-2cycles
- **CEC_CYCLE_MIN_6**: 2.05ms-3cycles
- **CEC_CYCLE_MIN_7**: 2.05ms-4cycles

CycleMax: 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+1cycles
- **CEC_CYCLE_MAX_2**: 2.75ms+2cycles
- **CEC_CYCLE_MAX_3**: 2.75ms+3cycles
- **CEC_CYCLE_MAX_4**: 2.75ms-1cycles
- **CEC_CYCLE_MAX_5**: 2.75ms-2cycles
- **CEC_CYCLE_MAX_6**: 2.75ms-3cycles
- **CEC_CYCLE_MAX_7**: 2.75ms-4cycles

機能:

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

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

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.2.3.21 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+2cycles
- **CEC_VALID_TIME_2**: 1.05ms+4cycles
- **CEC_VALID_TIME_3**: 1.05ms+6cycles
- **CEC_VALID_TIME_4**: 1.05ms-2cycles
- **CEC_VALID_TIME_5**: 1.05ms-4cycles
- **CEC_VALID_TIME_6**: 1.05ms-6cycles

機能:

データ 0/1 判別タイミングを設定します。データの論理"0"/論理"1"判別を行うポイントを設定します。約 1.05 ms を基準に、2/fs 単位で±6/fs まで設定可能です。

CEC が受信許可、または、送信中の場合、本関数の戻り値は **ERROR** となります。詳細はデータシートの CEC 章「 (4) 判別タイミング」を参照ください。

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.2.3.22 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

4.2.3.23 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

4.2.3.24 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

4.2.3.25 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-2cycles
- **CEC_LOGICAL_1_RISING_TIME_MIN_3**: 0.4ms-3cycles
- **CEC_LOGICAL_1_RISING_TIME_MIN_4**: 0.4ms-4cycles
- **CEC_LOGICAL_1_RISING_TIME_MIN_5**: 0.4ms-5cycles
- **CEC_LOGICAL_1_RISING_TIME_MIN_6**: 0.4ms-6cycles
- **CEC_LOGICAL_1_RISING_TIME_MIN_7**: 0.4ms-7cycles

Logical1RisingTimeMax: 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+2cycles
- **CEC_LOGICAL_1_RISING_TIME_MAX_3**: 0.8ms+3cycles
- **CEC_LOGICAL_1_RISING_TIME_MAX_4**: 0.8ms+4cycles
- **CEC_LOGICAL_1_RISING_TIME_MAX_5**: 0.8ms+5cycles
- **CEC_LOGICAL_1_RISING_TIME_MAX_6**: 0.8ms+6cycles
- **CEC_LOGICAL_1_RISING_TIME_MAX_7**: 0.8ms+7cycles

Logical0RisingTimeMin: 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 -2cycles
- **CEC_LOGICAL_0_RISING_TIME_MIN_3**: 1.3ms -3cycles
- **CEC_LOGICAL_0_RISING_TIME_MIN_4**: 1.3ms -4cycles
- **CEC_LOGICAL_0_RISING_TIME_MIN_5**: 1.3ms -5cycles
- **CEC_LOGICAL_0_RISING_TIME_MIN_6**: 1.3ms -6cycles
- **CEC_LOGICAL_0_RISING_TIME_MIN_7**: 1.3ms -7cycles

Logical0RisingTimeMax: 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 +2cycles
- **CEC_LOGICAL_0_RISING_TIME_MAX_3**: 1.7ms +3cycles
- **CEC_LOGICAL_0_RISING_TIME_MAX_4**: 1.7ms +4cycles
- **CEC_LOGICAL_0_RISING_TIME_MAX_5**: 1.7ms +5cycles
- **CEC_LOGICAL_0_RISING_TIME_MAX_6**: 1.7ms +6cycles
- **CEC_LOGICAL_0_RISING_TIME_MAX_7**: 1.7ms +7cycles

機能:

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

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.2.3.26 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+2cycles
- **CEC_START_BIT_RISING_TIME_MAX_3**: 3.9ms+3cycles
- **CEC_START_BIT_RISING_TIME_MAX_4**: 3.9ms+4cycles
- **CEC_START_BIT_RISING_TIME_MAX_5**: 3.9ms+5cycles
- **CEC_START_BIT_RISING_TIME_MAX_6**: 3.9ms+6cycles
- **CEC_START_BIT_RISING_TIME_MAX_7**: 3.9ms+7cycles

CycleMin: 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 -2cycles
- **CEC_START_BIT_CYCLE_MIN_3**: 4.3ms -3cycles
- **CEC_START_BIT_CYCLE_MIN_4**: 4.3ms -4cycles
- **CEC_START_BIT_CYCLE_MIN_5**: 4.3ms -5cycles
- **CEC_START_BIT_CYCLE_MIN_6**: 4.3ms -6cycles
- **CEC_START_BIT_CYCLE_MIN_7**: 4.3ms -7cycles

CycleMax: 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 +2cycles
- **CEC_START_BIT_CYCLE_MAX_3**: 4.7ms +3cycles
- **CEC_START_BIT_CYCLE_MAX_4**: 4.7ms +4cycles
- **CEC_START_BIT_CYCLE_MAX_5**: 4.7ms +5cycles
- **CEC_START_BIT_CYCLE_MAX_6**: 4.7ms +6cycles
- **CEC_START_BIT_CYCLE_MAX_7**: 4.7ms +7cycles

機能:

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

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.2.3.27 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

4.2.3.28 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-2cycles
- **CEC_TX_DATA_BIT_CYCLE_3**: RV-3cycles
- **CEC_TX_DATA_BIT_CYCLE_4**: RV-4cycles
- **CEC_TX_DATA_BIT_CYCLE_5**: RV-5cycles

- **CEC_TX_DATA_BIT_CYCLE_6**: RV-6cycles
- **CEC_TX_DATA_BIT_CYCLE_7**: RV-7cycles
- **CEC_TX_DATA_BIT_CYCLE_8**: RV-8cycles
- **CEC_TX_DATA_BIT_CYCLE_9**: RV-9cycles
- **CEC_TX_DATA_BIT_CYCLE_10**: RV-10cycles
- **CEC_TX_DATA_BIT_CYCLE_11**: RV-11cycles
- **CEC_TX_DATA_BIT_CYCLE_12**: RV-12cycles
- **CEC_TX_DATA_BIT_CYCLE_13**: RV-13cycles
- **CEC_TX_DATA_BIT_CYCLE_14**: RV-14cycles
- **CEC_TX_DATA_BIT_CYCLE_15**: RV-15cycles

DataBitRisingTime: 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-2cycles
- **CEC_TX_DATA_BIT_RISING_TIME_3**: RV-3cycles

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-2cycles
- **CEC_TX_START_BIT_CYCLE_3**: RV-3cycles
- **CEC_TX_START_BIT_CYCLE_4**: RV-4cycles
- **CEC_TX_START_BIT_CYCLE_5**: RV-5cycles
- **CEC_TX_START_BIT_CYCLE_6**: RV-6cycles
- **CEC_TX_START_BIT_CYCLE_7**: RV-7cycles

***StartBitRisingTime*:** 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-2cycles
- **CEC_TX_START_BIT_RISING_TIME_3**: RV-3cycles
- **CEC_TX_START_BIT_RISING_TIME_4**: RV-4cycles
- **CEC_TX_START_BIT_RISING_TIME_5**: RV-5cycles
- **CEC_TX_START_BIT_RISING_TIME_6**: RV-6cycles
- **CEC_TX_START_BIT_RISING_TIME_7**: RV-7cycles

機能:

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

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.2.3.29 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

4.2.3.30 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 cycles
- **CEC_BUS_FREE_3_BIT**: 3 bit cycles
- **CEC_BUS_FREE_4_BIT**: 4 bit cycles
- **CEC_BUS_FREE_5_BIT**: 5 bit cycles
- **CEC_BUS_FREE_6_BIT**: 6 bit cycles
- **CEC_BUS_FREE_7_BIT**: 7 bit cycles
- **CEC_BUS_FREE_8_BIT**: 8 bit cycles
- **CEC_BUS_FREE_9_BIT**: 9 bit cycles
- **CEC_BUS_FREE_10_BIT**: 10 bit cycles
- **CEC_BUS_FREE_11_BIT**: 11 bit cycles
- **CEC_BUS_FREE_12_BIT**: 12 bit cycles
- **CEC_BUS_FREE_13_BIT**: 13 bit cycles
- **CEC_BUS_FREE_14_BIT**: 14 bit cycles
- **CEC_BUS_FREE_15_BIT**: 15 bit cycles

- **CEC_BUS_FREE_16_BIT**: 16 bit cycles

機能:

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

送信中の場合、本関数の戻り値は **ERROR** となります。

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

4.2.3.31 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

4.2.3.32 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

4.2.4 データ構造

4.2.4.1 CEC_DataTypeDef

メンバ:

uint8_t
Data: 受信データの 1 バイト分を読みます。ビット 7 が MSB です。

CEC_ACKState
ACKBit: 受信 ACK ビットです。
- **CEC_ACK**: ACK ビットが "1"
- **CEC_NO_ACK**: ACK ビットが"0"

CEC_EOMBit
EOMBit: 受信 EOM ビットです。
- **CEC_EOM**: EOM ビットが “1”
- **CEC_NO_EOM**: EOM ビットが “0”

4.2.4.2 CEC_AddrListTypeDef

メンバ:

uint8_t
AddrNumber: ロジカルアドレス番号

CEC_LogicalAddr
AddrList[16]: ロジカルアドレス一覧です。以下のいずれかの値を選択します。
CEC_TV, CEC_RECORDING_DEVICE_1, CEC_RECORDING_DEVICE_2, CEC_STB_1, CEC_DVD_1, CEC_AUDIO_SYSTEM, CEC_STB_2, CEC_STB_3, CEC_DVD_2, CEC_RECORDING_DEVICE_3, CEC_FREE_USE, CEC_BROADCAST

5. EXB

5.1 概要

本デバイスは、外部にメモリや I/O などを接続するための外部バスインタフェース機能を内蔵しています。外部バスインタフェース回路 (EBIF)、チップセレクト(CS)ウェイトコントローラがこれに相当します。

チップセレクト、ウェイトコントローラは、任意の 4 ブロックアドレス空間のマッピングアドレス指定と、この 4 ブロックアドレス空間に対して、ウェイトおよびデータバス幅(8 ビットまたは 16 ビット)を制御します。

外部バスインタフェース回路(EBIF)は、CS/内蔵ウェイトコントローラの設定にもとづき外部バスのタイミングを制御します。

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/TX04_Periph_Driver/src/tmpm461_exb.c
/Libraries/TX04_Periph_Driver/inc/tmpm461_exb.h

5.2 API 関数

5.2.1 関数一覧

- void EXB_SetBusMode(uint8_t ***BusMode***);
- void EXB_SetBusCycleExtension(uint8_t ***Cycle***);
- void EXB_Enable(uint8_t ***ChipSelect***);
- void EXB_Disable(uint8_t ***ChipSelect***) ;
- void EXB_Init(uint8_t ***ChipSelect*** , EXB_InitTypeDef* ***InitStruct***);
- Result EXB_SetClkOutputDivision(uint8_t ***ClkDiv***);
- void EXB_EnableClkOutput(void);
- void EXB_DisableClkOutput(void);
- FunctionalState EXB_GetClkOutputState(void);

5.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています:

1) EXB バスモード、バスサイクルウェイト拡張、データバス幅、チップセレクタを元にした外部バスサイクルの設定
   EXB_SetBusMode(), EXB_SetBusCycleExtension(), EXB_Init()
2) 許可/禁止制御
   EXB_Enable(), EXB_Disable()
2) クロック出力の設定
   EXB_SetClkOutputDivision(), EXB_EnableClkOutput(),
   EXB_DisableClkOutput(), EXB_GetClkOutputState()

5.2.3 関数仕様

5.2.3.1 EXB_SetBusMode

EXB 外部バスモードの設定

関数のプロトタイプ宣言:

void
EXB_SetBusMode(uint8_t ***BusMode***)

引数:

BusMode :以下から EXB 外部バスモードを選択します。
- **EXB_BUS_MULTIPLEX**:マルチプレクスバスモード
- **EXB_BUS_SEPARATE**:セパレートバスモード

機能:

外部バスモードを設定します。

戻り値:

なし

5.2.3.2 EXB_SetBusCycleExtension

バスサイクルウェイト拡張の設定

関数のプロトタイプ宣言:

void
EXB_SetBusCycleExtension(uint8_t ***Cycle***)

引数:

Cycle: バスサイクルウェイト拡張を指定します。
- **EXB_CYCLE_NONE**: 拡張なし
- **EXB_CYCLE_DOUBLE**: 2 倍
- **EXB_CYCLE_QUADRUPLE**: 4 倍

機能:

バスサイクルのセットアップ、ウエイト、リカバリサイクル機能を 2 倍、4 倍に設定します。

戻り値:

なし

5.2.3.3 EXB_Enable

チップセレクトの許可

関数のプロトタイプ宣言:

void
EXB_Enable(uint8_t ***ChipSelect***)

引数:

ChipSelect : チップセレクトを選択します。
- **EXB_CS0**: CS0
- **EXB_CS1**: CS1
- **EXB_CS2**: CS2
- **EXB_CS3**: CS3

機能:

チップセレクトを許可します。

戻り値:

なし

5.2.3.4 EXB_Disable

チップセレクトの禁止

関数のプロトタイプ宣言:

void
EXB_Disable(uint8_t ***ChipSelect***)

引数:

ChipSelect : チップセレクトを選択します。
- **EXB_CS0**: CS0
- **EXB_CS1**: CS1
- **EXB_CS2**: CS2
- **EXB_CS3**: CS3

機能:

チップセレクトを禁止します。

戻り値:

なし

5.2.3.5 EXB_Init

チップセレクト設定の初期化

関数のプロトタイプ宣言:

void
EXB_Init (uint8_t ***ChipSelect***,
EXB_InitTypeDef* ***InitStruct***)

引数:

ChipSelect : チップセレクトを選択します。
- **EXB_CS0**: CS0
- **EXB_CS1**: CS1
- **EXB_CS2**: CS2
- **EXB_CS3**: CS3

InitStruct: 下記設定を行います。

チップセレクト空間サイズ、スタートアドレス、データバス幅、外部バスサイクル (詳細は、“データ構造”を参照してください)

機能:

チップセレクト設定を初期化します。

戻り値:

なし

5.2.3.6 EXB_SetClkOutputDivision

クロック分周の設定

関数のプロトタイプ宣言:

Result
EXB_SetClkOutputDivision (uint8_t ***ClkDiv***)

引数:

ClkDiv: 以下からクロック分周値を選択します。
- **EXB_CLK_DIVISION_FSYS_2**: fsys/2
- **EXB_CLK_DIVISION_FSYS_4**: fsys/4
- **EXB_CLK_DIVISION_FSYS_8**: fsys/8

機能:

クロック分周を設定します。

***補足:**

本関数をコールする前に **EXB_DisableClkOutput** をコールしてクロック出力を禁止してください。クロック出力が許可されている場合、本関数の戻り値は **ERROR** となります。

戻り値:

- **SUCCESS**: 成功
- **ERROR**: 失敗

5.2.3.7 EXB_EnableClkOutput

外部クロックの許可

関数のプロトタイプ宣言:

void
EXB_EnableClkOutput(void)

引数:

なし

機能:

外部クロックを許可します。

戻り値:

なし

5.2.3.8 EXB_DisableClkOutput

外部クロックの禁止

関数のプロトタイプ宣言:

void
EXB_DisableClkOutput(void)

引数:

なし

機能:

外部クロックを禁止します。

戻り値:

なし

5.2.3.9 EXB_GetClkOutputState

外部クロック制御状態の取得

関数のプロトタイプ宣言:

void
EXB_GetClkOutputState(void)

引数:

なし

機能:

外部クロックの制御状態を取得します。

戻り値:

- **ENABLE:** 外部クロック許可
- **DISABLE:** 外部クロック禁止

5.2.4 データ構造

5.2.4.1 EXB_InitTypeDef

メンバ:

uint8_t
AddrSpaceSize: アドレス空間を設定します。

- **EXB_16M_BYTE**: アドレス空間 16Mbyte
- **EXB_8M_BYTE**: アドレス空間 8Mbyte
- **EXB_4M_BYTE**: アドレス空間 4Mbyte
- **EXB_2M_BYTE**: アドレス空間 2Mbyte
- **EXB_1M_BYTE**: アドレス空間 1Mbyte
- **EXB_512K_BYTE**: アドレス空間 512Kbyte
- **EXB_256K_BYTE**: アドレス空間 256Kbyte
- **EXB_128K_BYTE**: アドレス空間 128Kbyte
- **EXB_64K_BYTE**: アドレス空間 64Kbyte

uint8_t
StartAddr : 開始アドレスを設定します。最大値は 0xFF です。

uint8_t
BusWidth : データバス幅を設定します。
- **EXB_BUS_WIDTH_BIT_8**: データバス幅 8bit,
- **EXB_BUS_WIDTH_BIT_16**: データバス幅 16bit.

EXB_CyclesTypeDef
Cycles: 外部バス周期を設定します。
InternalWait, ***ReadSetupCycle***, ***WriteSetupCycle***, ***ALEWaitCycle*** (マルチプレクスバスモードのみ), ***ReadRecoveryCycle***, ***WriteRecoveryCycle,*** ***ChipSelectRecoveryCycle***. (詳細は “EXB_CyclesTypeDef” を参照)

uint8_t
WaitSignal: ウェイト信号を選択します。
- **EXB_WAIT_SIGNAL_LOW**: "Low"アクティブ
- **EXB_WAIT_SIGNAL_HIGH**: "High"アクティブ

uint8_t
WaitFunction: ウェイト機能を選択します。

- **EXB_WAIT_FUNCTION_INT**: 内部ウェイト
- **EXB_WAIT_FUNCTION_EXT**: 外部ウェイト

5.2.4.2 EXB_CyclesType Def

メンバ:

uint8_t
InternalWait : 内部ウェイト(自動挿入)を設定します。
- **EXB_INTERNAL_WAIT_0**: 0 wait
- **EXB_INTERNAL_WAIT_1**: 1 wait
- **EXB_INTERNAL_WAIT_2**: 2 wait
- **EXB_INTERNAL_WAIT_3**: 3 wait
- **EXB_INTERNAL_WAIT_4**: 4 wait
- **EXB_INTERNAL_WAIT_5**: 5 wait
- **EXB_INTERNAL_WAIT_6**: 6 wait
- **EXB_INTERNAL_WAIT_7**: 7 wait
- **EXB_INTERNAL_WAIT_8**: 8 wait
- **EXB_INTERNAL_WAIT_9**: 9 wait
- **EXB_INTERNAL_WAIT_10**: 10 wait
- **EXB_INTERNAL_WAIT_11**: 11 wait
- **EXB_INTERNAL_WAIT_12**: 12 wait
- **EXB_INTERNAL_WAIT_13**: 13 wait
- **EXB_INTERNAL_WAIT_14**: 14 wait
- **EXB_INTERNAL_WAIT_15**: 15 wait

uint8_t
ReadSetupCycle : リード(RDn)セットアップサイクルを設定します。

- **EXB_CYCLE_0**: 0 cycle
- **EXB_CYCLE_1**: 1 cycle
- **EXB_CYCLE_2**: 2 cycle
- **EXB_CYCLE_4**: 4 cycle

uint8_t
WriteSetupCycle : ライト(WRn)セットアップサイクルを設定します。
- **EXB_CYCLE_0**: 0 cycle
- **EXB_CYCLE_1**: 1 cycle
- **EXB_CYCLE_2**: 2 cycle
- **EXB_CYCLE_4**: 4 cycle

uint8_t
ALEWaitCycle: ALE ウェイトサイクル(マルチプレクスバスモード時)を選択します。
- **EXB_CYCLE_0**: 0 cycle
- **EXB_CYCLE_1**: 1 cycle
- **EXB_CYCLE_2**: 2 cycle
- **EXB_CYCLE_4**: 4 cycle

uint8_t
ReadRecoveryCycle: リード(RDn)リカバリサイクルを選択します。
- **EXB_CYCLE_0**: 0 cycle
- **EXB_CYCLE_1**: 1 cycle
- **EXB_CYCLE_2**: 2 cycle
- **EXB_CYCLE_3**: 3 cycle
- **EXB_CYCLE_4**: 4 cycle
- **EXB_CYCLE_5**: 5 cycle
- **EXB_CYCLE_6**: 6 cycle
- **EXB_CYCLE_8**: 8 cycle

uint8_t
WriteRecoveryCycle: ライト(WRn)リカバリサイクルを選択します。
- **EXB_CYCLE_0**: 0 cycle
- **EXB_CYCLE_1**: 1 cycle
- **EXB_CYCLE_2**: 2 cycle
- **EXB_CYCLE_3**: 3 cycle
- **EXB_CYCLE_4**: 4 cycle
- **EXB_CYCLE_5**: 5 cycle
- **EXB_CYCLE_6**: 6 cycle
- **EXB_CYCLE_8**: 8 cycle

uint8_t
ChipSelectRecoveryCycle : チップセレクト(CSxn)リカバリサイクルを選択します。
- **EXB_CYCLE_0**: 0 cycle
- **EXB_CYCLE_1**: 1 cycle
- **EXB_CYCLE_2**: 2 cycle
- **EXB_CYCLE_4**: 4 cycle

6. CG

6.1 概要

本 CG API は TMPM461x 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/TX04_Periph_Driver\src\tmpm461_cg.c
/Libraries/TX04_Periph_Driver\inc\tmpm461_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。

EHCLKIN : X1 端子より入力されるクロック
EHOSC : 外部高速発振器から出力されるクロック
ELOSC : 外部低速発振器から出力されるクロック
IHOSC : 内部高速発振器から出力されるクロック (SYS 用)
IHOSC2 : 内部高速発振器から出力されるクロック (OFD 基準クロック用)
FOSCHI : CGOSCCR<HOSCON>で選択したクロック
fosc : CGOSCCR<OSCSEL>により選択されたクロック
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)

6.2 API 関数

6.2.1 関数一覧

- void CG_SetFgearLevel(CG_DivideLevel ***DivideFgearFromFc***)
- CG_DivideLevel CG_GetFgearLevel(void)
- void CG_SetPhiT0Src(CG_PhiT0Src ***PhiT0Src***)
- CG_PhiT0Src CG_GetPhiT0Src(void)
- Result CG_SetPhiT0Level(CG_DivideLevel ***DividePhiT0FromFc***)
- CG_DivideLevel CG_GetPhiT0Level(void)
- void CG_SetSCOUTSrc(CG_SCOUTSrc ***Source***)
- CG_SCOUTSrc CG_GetSCOUTSrc(void)
- void CG_SetWarmUpTime(CG_WarmUpSrc ***Source***, uint16_t ***Time***)
- void CG_StartWarmUp(void)
- WorkState CG_GetWarmUpState(void)
- Result CG_SetFPLLValue(CG_FpllValue ***NewValue***)

- CG_FpllValue CG_GetFPLLValue(void)
- Result CG_SetPLL(FunctionalState ***NewState***)
- FunctionalState CG_GetPLLState(void)
- Result CG_SetFosc(CG_FoscSrc Source, FunctionalState ***NewState***)
- void CG_SetFoscSrc(CG_FoscSrc ***Source***)
- CG_FoscSrc CG_GetFoscSrc(void)
- FunctionalState CG_GetFoscState(CG_FoscSrc ***Source***)
- void CG_SetSTBYMode(CG_STBYMode ***Mode***)
- CG_STBYMode CG_GetSTBYMode(void)
- void CG_SetPortKeepInStop2Mode(FunctionalState ***NewState***)
- FunctionalState CG_GetPortKeepInStop2Mode(void)
- Result CG_SetFcSrc(CG_FcSrc ***Source***)
- CG_FcSrc CG_GetFcSrc(void)
- void CG_SetProtectCtrl(FunctionalState ***NewState***)
- void CG_SetSTBYReleaseINTSrc(CG_INTSrc ***INTSource***,
  CG_INTActiveState ***ActiveState***,
  FunctionalState ***NewState***)
- CG_INTActiveState CG_GetSTBYReleaseINTState(CG_INTSrc ***INTSource***)
- void CG_ClearINTReq(CG_INTSrc ***INTSource***)
- CG_NMIFactor CG_GetNMIFlag(void)
- FunctionalState CG_GetIOSCFlashFlag(void)
- CG_ResetFlag CG_GetResetFlag(void)
- void CG_SetADCClkSupply(FunctionalState NewState)
- void CG_SetInternalOscForOFD(FunctionalState ***NewState***)
- void CG_SetFcPeriphA(uint32_t ***Periph***, FunctionalState ***NewState***)
- void CG_SetFcPeriphB(uint32_t ***Periph***, FunctionalState ***NewState***)
- void CG_SetFs(FunctionalState ***NewState***)

6.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。

1) 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_SetFPLLValue(), CG_GetFPLLValue(),CG_SetPLL(),
   CG_GetPLLState(), CG_SetFosc(),CG_SetFoscSrc(), CG_GetFoscSrc(),
   CG_GetFoscState(),CG_SetFcSrc(),CG_GetFcSrc(),CG_SetProtectCtrl()

2) スタンバイモードの設定:
   CG_SetSTBYMode(), CG_GetSTBYMode( ), CG_SetPortKeepInStop2Mode(),
   CG_GetPortKeepInStop2Mode()

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

4) 周辺機能へのクロック供給:
   CG_SetADCClkSupply(), CG_SetInternalOscForOFD(), CG_SetFcPeriphA(),
   CG_SetFcPeriphB(), CG_GetIOSCFlashFlag(), CG_SetFs()

6.2.3 関数仕様

6.2.3.1 CG_SetFgearLevel

fgear,fc 間の分周レベル設定

関数のプロトタイプ宣言:

void
CG_SetFgearLevel(CG_DivideLevel ***DivideFgearFromFc***)

引数:

DivideFgearFromFc: 以下から、fgear,fc 間の分周レベルを選択します。

- **CG_DIVIDE_1**: fgear = fc
- **CG_DIVIDE_2**: fgear = fc/2
- **CG_DIVIDE_4**: fgear = fc/4
- **CG_DIVIDE_8**: fgear = fc/8
- **CG_DIVIDE_16**: fgear = fc/16

機能:

fgear,fc 間の分周レベルを設定します。

戻り値:

なし

6.2.3.2 CG_GetFgearLevel

fgear,fc 間の分周レベルの取得

関数のプロトタイプ宣言:

CG_DivideLevel
CG_GetFgearLevel(void)

引数:

なし

機能:

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

戻り値:

fgear, fc 間の分周レベルで、下記のいずれかの値になります。

- **CG_DIVIDE_1**: fgear = fc

- **CG_DIVIDE_2**: fgear = fc/2
- **CG_DIVIDE_4:** fgear = fc/4
- **CG_DIVIDE_8:** fgear = fc/8
- **CG_DIVIDE_16:** fgear = fc/16
- **CG_DIVIDE_UNKNOWN:** 無効なデータ

6.2.3.3 CG_SetPhiT0Src

PhiT0(ΦT0) ,fc 間の PhiT0(ΦT0) ソースの設定

関数のプロトタイプ宣言:

void
CG_SetPhiT0Src(CG_PhiT0Src ***PhiT0Src***)

引数:

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

機能:

PhiT0 (ΦT0) ソースを選択します。

戻り値:

なし

6.2.3.4 CG_GetPhiT0Src

PhiT0 (ΦT0) ソースの取得

関数のプロトタイプ宣言:

CG_PhiT0Src
CG_GetPhiT0Src(void)

引数:

なし

機能:

PhiT0 (ΦT0) ソースを取得します。

戻り値:

- **CG_PHIT0_SRC_FGEAR**: fgear が PhiT0 ソース

- **CG_PHIT0_SRC_FC**: fc が PhiT0 ソース

6.2.3.5 CG_SetPhiT0Level

PhiT0 (ΦT0) と fc 間の分周レベルの設定

関数のプロトタイプ宣言:

Result
CG_SetPhiT0Level(CG_DivideLevel ***DividePhiT0FromFc***)

引数:

DividePhiT0FromFc: PhiT0 (ΦT0) と fc 間の分周レベルを下記の値から設定します。
- **CG_DIVIDE_1**: ΦT0 = fc
- **CG_DIVIDE_2**: ΦT0 = fc/2
- **CG_DIVIDE_4**: ΦT0 = fc/4
- **CG_DIVIDE_8**: ΦT0 = fc/8
- **CG_DIVIDE_16**: ΦT0 = fc/16
- **CG_DIVIDE_32**: ΦT0 = fc/32
- **CG_DIVIDE_64**: ΦT0 = fc/64
- **CG_DIVIDE_128**: ΦT0 = fc/128
- **CG_DIVIDE_256**: ΦT0 = fc/256
- **CG_DIVIDE_512**: ΦT0 = fc/512

機能:

プリスケーラクロックの分周レベルを設定します。

戻り値:

- **SUCCESS**: 成功
- **ERROR**: 失敗

6.2.3.6 CG_GetPhiT0Level

PhiT0(ΦT0) ,fc 間の分周レベルの取得

関数のプロトタイプ宣言:

CG_DivideLevel
CG_GetPhiT0Level(void)

引数:

なし

機能:

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

戻り値:

PhiT0(ΦT0) ,fc 間の分周レベル:
- **CG_DIVIDE_1**: ΦT0 = fc
- **CG_DIVIDE_2**: ΦT0 = fc/2
- **CG_DIVIDE_4**: ΦT0 = fc/4
- **CG_DIVIDE_8**: ΦT0 = fc/8
- **CG_DIVIDE_16**: ΦT0 = fc/16
- **CG_DIVIDE_32**: ΦT0 = fc/32
- **CG_DIVIDE_64**: ΦT0 = fc/64
- **CG_DIVIDE_128**: ΦT0 = fc/128
- **CG_DIVIDE_256**: ΦT0 = fc/256
- **CG_DIVIDE_512**: ΦT0 = fc/512
- **CG_DIVIDE_UNKNOWN**: 無効データ

6.2.3.7 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_FSYS**: fsys
- **CG_SCOUT_SRC_FS**: fs

機能:

SCOUT のソースクロックを設定します。

戻り値:

なし

6.2.3.8 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_FSYS**: fsys
- **CG_SCOUT_SRC_FS**: fs

6.2.3.9 CG_SetWarmUpTime

ウォームアップ時間の設定

関数のプロトタイプ宣言:

void
CG_SetWarmUpTime(CG_WarmUpSrc ***Source***, uint16_t ***Time***)

引数:

Source: 以下から、ウォームアップカウンタのソースクロックを選択します。
- **CG_WARM_UP_SRC_OSC_INT_HIGH**: 内部高速発振
- **CG_WARM_UP_SRC_OSC_EXT_HIGH**: 外部高速発振
- **CG_WARM_UP_SRC_OSC_EXT_LOW**: 外部低速発振

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 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)

高速発振子 10MHz 使用時、ウォーミングアップ時間 5ms を設定する場合の計算例:

(ウォーミングアップ時間)/(ウォーミングアップクロック) = 5ms/(1/10MHz) = 5000 サイクル = 0xC350

下位 4 ビットを切り捨て、0xC35 を CGOSCCR<WUPT[11:0]>に設定します。

戻り値:

なし

6.2.3.10 CG_StartWarmUp

ウォームアップ開始

関数のプロトタイプ宣言:

void
CG_StartWarmUp(void)

引数:

なし

機能:

ウォームアップを開始します。

戻り値:

なし

6.2.3.11 CG_GetWarmUpState

ウォーミングアップ動作状態 (動作中、完了)の確認

関数のプロトタイプ宣言:

WorkState
CG_GetWarmUpState(void)

引数:

なし

機能:

ウォーミングアップ動作状態を確認します。

```
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

6.2.3.12 CG_SetFPLLValue

PLL 逓倍値の設定

関数のプロトタイプ宣言:

Result
CG_SetFPLLValue(uint32_t ***NewValue***)

引数:

NewValue: 以下から PLL 逓倍値を選択します。

- **CG_8M_MUL_4_FPLL:** 入力クロック 8MHz, 出力クロック 32MHz (4 逓倍)
- **CG_8M_MUL_6_FPLL:** 入力クロック 8MHz, 出力クロック 48MHz (6 逓倍)
- **CG_8M_MUL_8_FPLL:** 入力クロック 8MHz, 出力クロック 64MHz (8 逓倍)
- **CG_8M_MUL_10_FPLL:** 入力クロック 8MHz, 出力クロック 80MHz (10 逓倍)
- **CG_8M_MUL_12_FPLL:** 入力クロック 8MHz, 出力クロック 96MHz (12 逓倍)
- **CG_10M_MUL_4_FPLL:** 入力クロック 10MHz, 出力クロック 40MHz (4 逓倍)
- **CG_10M_MUL_5_FPLL:** 入力クロック 10MHz, 出力クロック 50MHz (5 逓倍)
- **CG_10M_MUL_6_FPLL:** 入力クロック 10MHz, 出力クロック 60MHz (6 逓倍)
- **CG_10M_MUL_10_FPLL:** 入力クロック 10MHz, 出力クロック 100MHz (10 逓倍)
- **CG_10M_MUL_12_FPLL:** 入力クロック 10MHz, 出力クロック 120MHz (12 逓倍)
- **CG_12M_MUL_4_FPLL:** 入力クロック 12MHz, 出力クロック 48MHz (4 逓倍)
- **CG_12M_MUL_10_FPLL:** 入力クロック 12MHz, 出力クロック 120MHz (10 逓倍)
- **CG_16M_MUL_4_FPLL:** 入力クロック 16MHz, 出力クロック 64MHz (4 逓倍)
- **CG_16M_MUL_6_FPLL:** 入力クロック 16MHz, 出力クロック 96MHz (6 逓倍)

機能:

PLL 逓倍値を設定します。

戻り値:

- **SUCCESS**: 成功
- **ERROR**: 失敗

6.2.3.13 CG_GetFPLLValue

PLL 逓倍値の取得

関数のプロトタイプ宣言:

uint32_t
CG_GetFPLLValue(void)

引数:

なし

機能:

PLL 逓倍値を取得します。
取得した値が “Reserved”の場合、**CG_FPLL_MULTIPLY_UNKNOWN** を返却します。

戻り値:

PLL 逓倍値:

- **CG_8M_MUL_4_FPLL:** 入力クロック 8MHz, 出力クロック 32MHz (4 逓倍)
- **CG_8M_MUL_6_FPLL:** 入力クロック 8MHz, 出力クロック 48MHz (6 逓倍)
- **CG_8M_MUL_8_FPLL:** 入力クロック 8MHz, 出力クロック 64MHz (8 逓倍)
- **CG_8M_MUL_10_FPLL:** 入力クロック 8MHz, 出力クロック 80MHz (10 逓倍)
- **CG_8M_MUL_12_FPLL:** 入力クロック 8MHz, 出力クロック 96MHz (12 逓倍)
- **CG_10M_MUL_4_FPLL:** 入力クロック 10MHz, 出力クロック 40MHz (4 逓倍)
- **CG_10M_MUL_5_FPLL:** 入力クロック 10MHz, 出力クロック 50MHz (5 逓倍)
- **CG_10M_MUL_6_FPLL:** 入力クロック 10MHz, 出力クロック 60MHz (6 逓倍)
- **CG_10M_MUL_10_FPLL:** 入力クロック 10MHz, 出力クロック 100MHz (10 逓倍)
- **CG_10M_MUL_12_FPLL:** 入力クロック 10MHz, 出力クロック 120MHz (12 逓倍)
- **CG_12M_MUL_4_FPLL:** 入力クロック 12MHz, 出力クロック 48MHz (4 逓倍)
- **CG_12M_MUL_10_FPLL:** 入力クロック 12MHz, 出力クロック 120MHz (10 逓倍)
- **CG_16M_MUL_4_FPLL:** 入力クロック 16MHz, 出力クロック 64MHz (4 逓倍)
- **CG_16M_MUL_6_FPLL:** 入力クロック 16MHz, 出力クロック 96MHz (6 逓倍)

6.2.3.14 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

6.2.3.15 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

6.2.3.16 CG_SetFosc

高速発振器(fosc)の有効/無効設定

関数のプロトタイプ宣言:

Result
CG_SetFosc(CG_FoscSrc ***Source***,
FunctionalState ***NewState***)

引数:

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

NewState
- **ENABLE**: 高速発信器有効
- **DISABLE**: 高速発信器無効

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

6.2.3.17 CG_SetFoscSrc

高速発振器(fosc)のソース設定

関数のプロトタイプ宣言:

void
CG_SetFoscSrc(CG_FoscSrc Source)

引数:

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

機能:

高速発振器(fosc)のソースを設定します。

戻り値:

なし

6.2.3.18 CG_GetFoscSrc

高速発振器のソース取得

関数のプロトタイプ宣言:

CG_FoscSrc
CG_GetFoscSrc(void)

引数:

なし

機能:

高速発振器のソースを取得します。

戻り値:

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

6.2.3.19 CG_GetFoscState

高速発信器の状態

関数のプロトタイプ宣言:

FunctionalState
CG_GetFoscState(CG_FoscSrc Source)

引数:

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

機能:

高速発信器の状態を取得します。

戻り値:

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

6.2.3.20 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_STOP1**: STOP1 モード (内部発振器も含めてすべての内部回路が停止)
- **CG_STBY_MODE_STOP2**: STOP2 モード (一部の機能を保持して内部電源を遮断)
- **CG_STBY_MODE_IDLE**: IDLE モード(CPU が停止)

機能:

スタンバイモードを選択します。

戻り値:

なし

6.2.3.21 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_STOP1**: STOP1 モード
- **CG_STBY_MODE_STOP2**: STOP2 モード
- **CG_STBY_MODE_IDLE**: IDLE モード
- **CG_STBY_MODE_UNKNOWN**: 無効なモード

6.2.3.22 CG_SetPortKeepInStop2Mode

STOP2 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:

void
CG_SetPortKeepInStop2Mode(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

STOP2 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さい。

機能:

STOP2 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:

なし

6.2.3.23 CG_GetPortKeepInStop2Mode

STOP2 モード中の I/O 制御信号保持状態の取得

関数のプロトタイプ宣言:

FunctionalState
CG_GetPinStateInStopMode(void)

引数:

なし

機能:

STOP2 モード中の I/O 制御信号保持状態を取得します。

戻り値:

STOP2 モード中の I/O 制御信号保持状態:
- **DISABLE:** ポートによる制御
- **ENABLE:** ENABLE 設定時の状態を保持

6.2.3.24 CG_SetFcSrc

fc のソース選択

関数のプロトタイプ宣言:

Result
CG_SetFcSrc(CG_FcSrc ***Source***)

引数:

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

- **CG_FC_SRC_FPLL**: fpll 使用

機能:

fc のソースクロックを選択します。

戻り値:

SUCCESS: 成功
ERROR: 失敗

6.2.3.25 CG_GetFcSrc

fc ソースの設定状態取得

関数のプロトタイプ宣言:

CG_FcSrc
CG_GetFosc(void)

引数:

なし

機能:

fc ソースの設定状態を取得します。

戻り値:

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

6.2.3.26 CG_SetProtectCtrl

CG レジスタの書き込み制御

関数のプロトタイプ宣言:

void
CG_SetProtectCtrl(FunctionalState ***NewState***)

引数:

NewState

- **DISABLE:** 書込み禁止
- **ENABLE:** 書込み許可

機能:

CG レジスタの書き込み許可/禁止を設定します。

戻り値:

なし

6.2.3.27 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**: INTA
- **CG_INT_SRC_B**: INTB
- **CG_INT_SRC_C**: INTC
- **CG_INT_SRC_D**: INTD
- **CG_INT_SRC_E**: INTE
- **CG_INT_SRC_F**: INTF
- **CG_INT_SRC_CECRX**: INTCECRX
- **CG_INT_SRC_CECTX**: INTCECTX
- **CG_INT_SRC_RMCRX0**: INTRMCRX0
- **CG_INT_SRC_RTC**: INTRTC

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_CECRX , **CG_INT_SRC_CECTX ,** ***CG_INT_SRC_RMCRX0***	**CG_INT_ACTIVE_STATE_RISGING**	↑エッジ
上記以外	**CG_INT_ACTIVE_STATE_L**	"Low"レベル
	CG_INT_ACTIVE_STATE_H	"High"レベル

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

NewState: 解除トリガの有効/無効を選択します。

- **ENABLE**: 許可
- **DISABLE:** 禁止

機能:

スタンバイモードの解除割り込みソースを設定します。

戻り値:

なし

6.2.3.28 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_CECRX, CG_INT_SRC_CECTX, CG_INT_SRC_RMCRX0, CG_INT_SRC_RTC**

機能:

スタンバイモードの解除割り込みソースのアクティブ状態を取得します。

戻り値:

解除割り込みソースのアクティブ状態

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

6.2.3.29 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_CECRX, CG_INT_SRC_CECTX, CG_INT_SRC_RMCRX0, CG_INT_SRC_RTC**

機能:

スタンバイ解除割り込み要求をクリアします。

戻り値:

なし

6.2.3.30 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
- **DetectLowVoltage** (Bit 2) : LVD で電源電圧が設定電圧より下がったことによる NMI 発生

- **ReturnLowVoltage** (Bit 3) :LVD で電源電圧が設定電圧より上がったことによる NMI 発生

6.2.3.31 CG_GetIOSCFlashFlag

STOP2 解除後の内部高速発振器停止許可/Flash ROM 消去/プログラム許可フラグの取得

関数のプロトタイプ宣言:

FunctionalState
CG_GetIOSCFlashFlag(void)

引数:

なし

機能:

STOP2 解除後の内部高速発振器停止許可/Flash ROM 消去/プログラム許可状態を取得します。

補足:

STOP2 から NOMAL モードへ遷移後にフラッシュメモリへ書き込みを行う場合は CGRSTFLG<OSCFLF> が"1" であることを確認してください。

戻り値:

内部高速発振器停止/FLASH E/W 可能フラグ:

- **ENABLE:** 許可
- **DISABLE:** 禁止

6.2.3.32 CG_GetResetFlag

リセットフラグの取得とクリア

関数のプロトタイプ宣言:

CG_ResetFlag
CG_GetResetFlag(void)

引数:

なし

機能:

リセットフラグの取得とクリアを行います。

戻り値:

リセットフラグ:
- **ResetPin** (Bit0) RESET 端子によるリセット
- **WDTReset** (Bit 2) WDT によるリセット
- **STOP2Rese**t(Bit3) STOP2 モード解除によるリセット
- **DebugReset** (Bit 4) <SYSRESETREQ>によるリセット
- **OFDReset** (Bit5) OFD によるリセット
- **LVDReset** (Bit6) LVD によるリセット

6.2.3.33 CG_SetADCClkSupply

ADC クロックの選択

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

引数:
NewState: ADC クロックを選択します。
- **ENABLE**: 動作
- **DISABLE**: 停止

機能:
ADC クロックを選択します。

戻り値:
なし

6.2.3.34 CG_SetInternalOscForOFD

内部高速発信器(OFD 用)の発振/停止設定

関数のプロトタイプ宣言:

void
CG_SetInternalOscForOFD(FunctionalState ***NewState***)

引数:

NewState
- **ENABLE**: 発振
- **DISABLE**: 停止

機能:

内部高速発信器(OFD 用)の発振/停止を選択します。

戻り値:

なし

6.2.3.35 CG_SetFcPeriphA

周辺機能へのクロック供給停止設定

関数のプロトタイプ宣言:

void
CG_SetFcPeriphA(uint32_t ***Periph***, FunctionalState ***NewState***)

引数:

Periph: クロック供給を停止する周辺機能を選択します。

- **CG_FC_PERIPH_PORTA:** PORT A
- **CG_FC_PERIPH_PORTB:** PORT B
- **CG_FC_PERIPH_PORTC:** PORT C
- **CG_FC_PERIPH_PORTD:** PORT D
- **CG_FC_PERIPH_PORTE:** PORT E
- **CG_FC_PERIPH_PORTF:** PORT F
- **CG_FC_PERIPH_PORTG:** PORT G
- **CG_FC_PERIPH_PORTH:** PORT H
- **CG_FC_PERIPH_PORTJ:** PORT J
- **CG_FC_PERIPH_PORTK:** PORT K
- **CG_FC_PERIPH_TMRB0:** TMRB0
- **CG_FC_PERIPH_TMRB1:** TMRB1
- **CG_FC_PERIPH_TMRB2:** TMRB2
- **CG_FC_PERIPH_TMRB3:** TMRB3
- **CG_FC_PERIPH_TMRB4:** TMRB4
- **CG_FC_PERIPH_TMRB5:** TMRB5
- **CG_FC_PERIPH_TMRB6:** TMRB6
- **CG_FC_PERIPH_TMRB7:** TMRB7
- **CG_FC_PERIPH_TMRB8:** TMRB8
- **CG_FC_PERIPH_TMRB9:** TMRB9
- **CG_FC_PERIPH_TMRB10:** TMRB10
- **CG_FC_PERIPH_TMRB11:** TMRB11
- **CG_FC_PERIPH_TMRB12:** TMRB12
- **CG_FC_PERIPH_TMRB13:** TMRB13
- **CG_FC_PERIPH_TMRB14:** TMRB14
- **CG_FC_PERIPH_TMRB15:** TMRB15
- **CG_FC_PERIPH_MPT0:** MPT0
- **CG_FC_PERIPH_MPT1:** MPT1
- **CG_FC_PERIPH_TRACE:** TRACE
- **CG_FC_PERIPHA_ALL:** すべて

NewState

- **ENABLE**: 動作
- **DISABLE**: 停止

機能:

周辺機能へのクロック供給を制御します。

戻り値:

なし

6.2.3.36 CG_SetFcPeriphB

周辺機能へのクロック供給停止設定

関数のプロトタイプ宣言:

void
CG_SetFcPeriphB(uint32_t ***Periph***, FunctionalState ***NewState***)

引数:

Periph: クロック供給を停止する周辺機能を選択します。

- **CG_FC_PERIPH_SIO_UART0:** SIO/UART0
- **CG_FC_PERIPH_SIO_UART1:** SIO/UART1
- **CG_FC_PERIPH_SIO_UART2:** SIO/UART2
- **CG_FC_PERIPH_SIO_UART3:** SIO/UART3
- **CG_FC_PERIPH_SIO_UART4:** SIO/UART4
- **CG_FC_PERIPH_SIO_UART5:** SIO/UART5
- **CG_FC_PERIPH_UART0:** UART0
- **CG_FC_PERIPH_UART1:** UART1
- **CG_FC_PERIPH_I2C0:** I2C0
- **CG_FC_PERIPH_I2C1:** I2C1
- **CG_FC_PERIPH_I2C2:** I2C2
- **CG_FC_PERIPH_I2C3:** I2C3
- **CG_FC_PERIPH_I2C4:** I2C4
- **CG_FC_PERIPH_SSP0:** SSP0
- **CG_FC_PERIPH_SSP1:** SSP1
- **CG_FC_PERIPH_SSP2:** SSP2
- **CG_FC_PERIPH_EBIF:** EBIF
- **CG_FC_PERIPH_DMACA:** DMAC A
- **CG_FC_PERIPH_DMACB:** DMAC B
- **CG_FC_PERIPH_DMACC:** DMAC C
- **CG_FC_PERIPH_DMAIF:** DMACIF
- **CG_FC_PERIPH_ADC:** ADC
- **CG_FC_PERIPH_WDT:** WDT
- **CG_FC_PERIPH_OFD:** OFD
- **CG_FC_PERIPHB_ALL:** すべて

NewState

- **ENABLE**: 動作
- **DISABLE**: 停止

機能:

周辺機能へのクロック供給を制御します。

戻り値:

なし

6.2.3.37 CG_SetFs

低速発振器(fs)の動作選択

関数のプロトタイプ宣言:

void
CG_SetFs(FunctionalState ***NewState***)

引数:

NewState
- **ENABLE**: 発振
- **DISABLE**: 停止

機能:

低速発振器(fs)の動作を選択します。

戻り値:

なし

6.2.4 データ構造

6.2.4.1 CG_NMIFactor

メンバ:

uint32_t
All CGNMI ソース起動状態を指定します。

ビットフィールド:

uint32_t
WDT(Bit 0)　WDT による NMI 発生

uint32_t
NMIPin(Bit 1)　NMI 端子による NMI 発生

uint32_t
DetectLowVoltage(Bit 2)　LVD で電源電圧が設定電圧より下がったことによる NMI 発生

uint32_t
Reserved1 (Bit3)　未使用

uint32_t
ReturnLowVoltage(Bit 4)　Flash ECC エラーによる NMI 発生

uint32_t
Reserved2 (Bit5~bit31)　未使用

6.2.4.2　CG_ResetFlag

メンバ:

uint32_t

All CG リセット要因を指定します。

ビットフィールド:

uint32_t
ResetPinBit0)　RESET 端子によるリセット

uint32_t
Reserved1 (Bit1)　未使用

uint32_t
WDTReset(Bit2)　WDT によるリセット

uint32_t
STOP2Reset(Bit3)　STOP2 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
OFDReset(Bit5)　OFD によるリセット

uint32_t
LVDReset(Bit6)　LVD によるリセット

uint32_t
Reserved2 (Bit7~bit31)　未使用

7. FC

7.1 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 TMPM461F10 の場合 1024Kbyte で、TMPM461F15 の場合 1536KByte 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\TX04_Periph_Driver\src\tmpm461_fc.c
\Libraries\TX04_Periph_Driver \inc\ tmpm461_fc.h

7.2 API 関数

7.2.1 関数一覧

- void FC_SetSecurityBit(FunctionalState ***NewState***)
- FunctionalState FC_GetSecurityBit(void)
- WorkState FC_GetBusyState(void)
- FunctionalState FC_GetBlockProtectState(uint8_t ***BlockNum***)
- FunctionalState FC_GetPageProtectState(uint8_t ***PageNum***)
- FunctionalState FC_GetAbortState(void)
- uint32_t FC_GetSwapSize(void);
- uint32_t FC_GetSwapState(void);
- void FC_SelectArea(uint8_t ***AreaNum***, FunctionalState ***NewState***);
- void FC_SetAbortion(void);
- void FC_ClearAbortion(void);
- void FC_SetClkDiv(uint8_t ***ClkDiv***);
- void FC_SetProgramCount(uint8_t ***ProgramCount***);
- void FC_SetEraseCounter(uint8_t ***EraseCounter***);
- FC_Result FC_ProgramBlockProtectState(uint8_t ***BlockNum***);
- FC_Result FC_ProgramPageProtectState(uint8_t ***PageNum***);
- FC_Result FC_EraseProtectState(void);
- FC_Result FC_WritePage(uint32_t ***PageAddr***, uint32_t * ***Data***);
- FC_Result FC_EraseBlock(uint32_t ***BlockAddr***);
- FC_Result FC_EraseArea(uint32_t ***AreaAddr***);
- FC_Result FC_ErasePage(uint32_t ***PageAddr***);
- FC_Result FC_EraseChip(void);
- FC_Result FC_SetSwpsrBit(uint8_t ***BitNum***);
- uint32_t FC_GetSwpsrBitValue(uint8_t ***BitNum***);

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) セキュリティ設定(Flash ROM データの読み出し、デバッグ):
   FC_SetSecurityBit(), FC_GetSecurityBit()
2) 自動動作状態およびプロテクト状態の取得:
   FC_GetBusyState(), FC_GetBlockProtectState(), FC_GetPageProtectState()
3) プロテクト設定:
   FC_ProgramBlockProtectState(), FC_ProgramPageProtectState(), FC_EraseProtectState()
4) 自動動作コマンド:
   FC_WritePage(), FC_EraseBlock(), FC_EraseChip(),FC_EraseArea(), FC_ErasePage(), FC_SetSwpsrBit()
5) その他:
   FC_GetAbortState(), FC_GetSwapSize(), FC_GetSwapState(), FC_SelectArea(), FC_SetAbortion(), FC_ClearAbortion(), FC_SetClkDiv(), FC_SetProgramCount(), FC_SetEraseCounter(), FC_GetSwpsrBitValue()

7.2.3 関数仕様

7.2.3.1 FC_SetSecurityBit

セキュリティビットの設定

関数のプロトタイプ宣言:

void
FC_SetSecurityBit (FunctionalState ***NewState***)

引数:

NewState: セキュリティビットを設定します。

- **DISABLE**: セキュリティ機能設定不可
- **ENABLE**: セキュリティビット設定可能

機能:

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

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

- ROM 領域のデータの読み出し。
- JTAG/SW、トレースの通信

したがって、この API を使用する場合は、注意して実行してください。

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

戻り値:

なし

7.2.3.2 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

7.2.3.3 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

7.2.3.4 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。

TMPM461F10FG:
- FC_BLOCK_1 ~ FC_BLOCK_31

TMPM461F15FG:
- FC_BLOCK_1 ~ FC_BLOCK_47

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

7.2.3.5 FC_GetPageProtectState

ページのプロテクト状態の取得

関数のプロトタイプ宣言:

FunctionalState
FC_GetPageProtectState(uint8_t ***PageNum***)

引数:

PageNum: ページ番号を選択します。
- **FC_PAGE_0 ~ FC_PAGE_7**

機能:

各ページ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:

ページプロテクトの状態:
DISABLE: プロテクト状態ではない。
ENABLE: プロテクト状態

7.2.3.6 FC_GetAbortState

自動実行コマンドの中止状態の取得

関数のプロトタイプ宣言:

FunctionalState
FC_GetAbortState(void)

引数:

なし

機能:

自動実行コマンドの中止状態を取得します。

戻り値:

自動実行コマンドの中止を行うと **DONE** を返します。

7.2.3.7 FC_GetSwapSize

メモリスワップサイズの取得

関数のプロトタイプ宣言:

uint32_t
FC_GetSwapSize(void)

引数:

なし

機能:

メモリスワップサイズを取得します。

戻り値:

メモリスワップサイズ:
FC_SWAP_SIZE_4K: 4K バイト
FC_SWAP_SIZE_8K: 8K バイト

FC_SWAP_SIZE_16K: 16K バイト
FC_SWAP_SIZE_32K: 32K バイト
FC_SWAP_SIZE_AREA1: エリア 0 とエリア 1
FC_SWAP_SIZE_AREA2: エリア 0 とエリア 2

7.2.3.8 FC_GetSwapState

メモリスワップ状態の取得

関数のプロトタイプ宣言:

uint32_t
FC_GetSwapState(void)

引数:

なし

機能:

メモリスワップ状態を取得します。

戻り値:

メモリスワップ状態:
FC_SWAP_RELEASE: スワップ解除
FC_SWAP_PROHIBIT: 設定禁止
FC_SWAPPING: スワップ中
FC_SWAP_INITIAL: スワップ解除(初期化状態)

7.2.3.9 FC_SelectArea

フラッシュメモリ操作コマンドにより実行の対象となるフラッシュメモリの" エリア"選択

関数のプロトタイプ宣言:

void
FC_SelectArea(uint8_t ***AreaNum***)

引数:

AreaNum: 以下のいずれかのエリア番号を選択します。
TMPM461F15FG:
- **FC_AREA_0, FC_AREA_1, FC_AREA_2, FC_AREA_ALL**

TMPM461F10FG:
- **FC_AREA_0, FC_AREA_1, FC_AREA_ALL**

NewState: 選択/非選択を設定します。
- **ENABLE**: 選択

- **DISABLE**: 非選択

機能:

フラッシュメモリ操作コマンドにより実行の対象となるフラッシュメモリの" エリア" を選択します。

戻り値:

なし

7.2.3.10 FC_SetAbortion

自動実行コマンドの中止

関数のプロトタイプ宣言:

void
FC_SetAbortion(void)

引数:

なし

機能:

自動実行コマンドを中止します。

戻り値:

なし

7.2.3.11 FC_ClearAbortion

自動実行コマンド中止フラグのクリア

関数のプロトタイプ宣言:

void
FC_ClearAbortion(void)

引数:

なし

機能:

自動実行コマンド中止フラグをクリアします。

戻り値:

なし

7.2.3.12 FC_SetClkDiv

自動実行中のクロック(WCLK: fsys/(DIV+1)) が 8 ~ 12MHz となる分周比の設定

関数のプロトタイプ宣言:

void
FC_SetClkDiv(uint8_t ***ClkDiv***)

引数:

ClkDiv: 以下のいずれかの分周比を選択します。
- **FC_Clk_Div_1** ~ **FC_Clk_Div_32**

機能:

自動実行中のクロック(WCLK: fsys/(DIV+1)) が 8 ~ 12MHz となる分周比を選択します。

戻り値:

なし

7.2.3.13 FC_SetProgramCount

自動プログラム実行コマンドによる書き込み時間(CNT/WCLK) が 20 ~ 40μsec となるカウント数の設定

関数のプロトタイプ宣言:

void
FC_SetProgramCount(uint8_t ***ProgramCount***)

引数:

ProgramCount: 以下のいずれかのカウント数を選択します。
- **FC_PROG_CNT_250, FC_PROG_CNT_300, FC_PROG_CNT_350**

機能:

自動プログラム実行コマンドによる書き込み時間(CNT/WCLK) が 20 ~ 40μsec となるカウント数を選択します。

戻り値:

なし

7.2.3.14 FC_SetEraseCounter

各自動消去コマンド実行による消去時間(CNT/WCLK) が 100 ~ 130msec となるカウント数の設定

関数のプロトタイプ宣言:

void
FC_SetEraseCounter (uint8_t ***EraseCounter***)

引数:

EraseCounter: 以下のいずれかのカウント数を選択します。

- **FC_ERAS_CNT_85, FC_ERAS_CNT_90**
- **FC_ERAS_CNT_95, FC_ERAS_CNT_100**
- **FC_ERAS_CNT_105,FC_ERAS_CNT_110**
- **FC_ERAS_CNT_115, FC_ERAS_CNT_120**
- **FC_ERAS_CNT_125, FC_ERAS_CNT_130**
- **FC_ERAS_CNT_135, FC_ERAS_CNT_140**

機能:

各自動消去コマンド実行による消去時間(CNT/WCLK) が 100 ~ 130msec となるカウント数を設定します。

戻り値:

なし

7.2.3.15 FC_ProgramBlockProtectState

ブロックのプロテクト設定

関数のプロトタイプ宣言:

FC_Result
FC_ProgramProtectState(uint8_t ***BlockNum***)

引数:

BlockNum: 以下のいずれかのブロック番号を選択します。

TMPM461F10FG:

- **FC_BLOCK_1** ~ **FC_BLOCK_31**

TMPM461F15FG:

- **FC_BLOCK_1** ~ **FC_BLOCK_47**

機能:

ブロックプロテクトを設定します。

戻り値:

実行結果:
FC_SUCCESS: 成功
FC_ERROR_PROTECTED: 設定済
FC_ERROR_OVER_TIME: タイマオーバーフロー

7.2.3.16 FC_ProgramPageProtectState

ページプロテクトの設定

関数のプロトタイプ宣言:

FC_Result
FC_ProgramProtectState(uint8_t ***PageNum***)

引数:

PageNum: 以下のいずれかのページ番号を選択します。
- **FC_PAGE_0** ~ **FC_PAGE_7**

機能:

ページプロテクトを設定します。

戻り値:

実行結果:
FC_SUCCESS: 成功
FC_ERROR_PROTECTED: 設定済
FC_ERROR_OVER_TIME: タイマオーバーフロー

7.2.3.17 FC_EraseProtectState

プロテクトの解除

関数のプロトタイプ宣言:

FC_Result
FC_EraseBlockProtectState(void)

引数:

なし

機能:

プロテクトビットを"0"にすることでプロテクトを解除します。

戻り値:

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

7.2.3.18 FC_WritePage

ページ単位の書き込み

関数のプロトタイプ宣言:

FC_Result
FC_WritePage(uint32_t ***PageAddr***, uint32_t * ***Data***)

引数:

PageAddr: ページの開始アドレスを指定します。

Data: 書き込むデータバッファへのポインタを指定します。サイズは
FC_PAGE_SIZE(4096Byte)です。

機能:

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

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

2 STOP2 モードから NORMAL モードへ復帰後に Flash メモリへの書き込みを行う場合は CG API の CG_GetIOSCFlashFlag()関数をコールし、戻り値が ENABLE であることを確認しておく必要があります。

戻り値:

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

7.2.3.19 FC_EraseBlock

ブロック単位の消去

関数のプロトタイプ宣言:

FC_Result
FC_EraseBlock(uint32_t ***BlockAddr***)

引数:

BlockAddr: ブロック開始アドレスを指定してください。

機能:

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

戻り値:

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

7.2.3.20 FC_EraseArea

エリア単位の消去

関数のプロトタイプ宣言:

FC_Result
FC_EraseArea(uint32_t ***AreaAddr***)

引数:

AreaAddr: エリア開始アドレスを指定してください。

機能:

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

戻り値:

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

7.2.3.21 FC_ErasePage

ページ単位の消去

関数のプロトタイプ宣言:

FC_Result
FC_ErasePage(uint32_t ***PageAddr***)

引数:

PageAddr: ページ開始アドレスを指定してください。

機能:

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

戻り値:

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

7.2.3.22 FC_EraseChip

チップ消去

関数のプロトタイプ宣言:

FC_Result
FC_EraseChip(void)

引数:

なし

機能:

チップ消去を行います。ブロックの一部にプロテクトが設定されている場合、そのブロック以外のブロックを消去します。

戻り値:

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

7.2.3.23 FC_SetSwapsrBit

FCSWPSR[10:0]レジスタの設定

関数のプロトタイプ宣言:

FC_Result
FC_SetSwpsrBit(uint8_t ***BitNum***)

引数:

BitNum: FCSWPSR レジスタに設定するビット番号を以下のいずれかから選択します。
FC_SWPSR_BIT_0 ~ **FC_SWPSR_BIT_10**

機能:

FCSWPSR[10:0]レジスタを設定します。

戻り値:

FCSWPSR ビット変更の実行結果:
FC_SUCCESS: 設定成功
FC_ERROR_OVER_TIME: 設定失敗(自動動作のタイムアウト)

7.2.3.24 FC_GetSwpsrBitValue

FCSWPSR[10:0]レジスタ値の取得

関数のプロトタイプ宣言:

uint32_t
FC_GetSwpsrBitValue(uint8_t ***BitNum***)

引数:

BitNum: FCSWPSR レジスタに設定するビット番号を以下のいずれかから選択します。
FC_SWPSR_BIT_0 ~ **FC_SWPSR_BIT_10**

機能:

FCSWPSR[10:0]レジスタ値を取得します。

戻り値:

指定ビットの値:
FC_BIT_VALUE_0: 0
FC_BIT_VALUE_1: 1

7.2.4　データ構造

なし

8. FUART

8.1 概要

TMPM343 は非同期のシリアルチャネル (Full UART)とモデム制御を内蔵します。本製品は 1 チャネルのフル UART(FUART0 と FUART1)を内蔵します。

FUART ドライバ API は、Full UART チャネルを構成する機能、たとえばボーレート、ビット長、パリティチェック、ストップビット、フロー制御、などの共通パラメータを提供します。また、データの送信/受信、エラーチェックなどのような転送を制御します。

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/TX04_Periph_Driver/src/tmpm461_fuart.c
/Libraries/TX04_Periph_Driver/inc/tmpm461_fuart.h

8.2 API 関数

8.2.1 関数一覧

- void FUART_Enable(TSB_FUART_TypeDef * ***FUARTx***)
- void FUART_Disable(TSB_FUART_TypeDef * ***FUARTx***)
- uint32_t FUART_GetRxData(TSB_FUART_TypeDef * ***FUARTx***)
- void FUART_SetTxData(TSB_FUART_TypeDef * ***FUARTx***, uint32_t ***Data***)
- FUART_Err FUART_GetErrStatus(TSB_FUART_TypeDef * ***FUARTx***)
- void FUART_ClearErrStatus(TSB_FUART_TypeDef * ***FUARTx***)
- WorkState FUART_GetBusyState(TSB_FUART_TypeDef * ***FUARTx***)
- FUART_StorageStatus FUART_GetStorageStatus(
  TSB_FUART_TypeDef * ***FUARTx***, FUART_Direction ***Direction***)
- void FUART_SetIrDADivisor(TSB_FUART_TypeDef * ***FUARTx***, uint32_t ***Divisor***)
- void FUART_Init(
  TSB_FUART_TypeDef * ***FUARTx***, FUART_InitTypeDef * ***InitStruct***)
- void FUART_EnableFIFO(TSB_FUART_TypeDef * ***FUARTx***)
- void FUART_DisableFIFO(TSB_FUART_TypeDef * ***FUARTx***)
- void FUART_SetSendBreak(
  TSB_FUART_TypeDef * ***FUARTx***, FunctionalState ***NewState***)
- void FUART_SetIrDAEncodeMode(
  TSB_FUART_TypeDef * ***FUARTx***, uint32_t ***Mode***)
- Result FUART_EnableIrDA(TSB_FUART_TypeDef * ***FUARTx***)
- void FUART_DisableIrDA(TSB_FUART_TypeDef * ***FUARTx***)
- void FUART_SetINTFIFOLevel(
  TSB_FUART_TypeDef * ***FUARTx***, uint32_t ***RxLevel***, uint32_t ***TxLevel***)
- void FUART_SetINTMask(TSB_FUART_TypeDef * ***FUARTx***, uint32_t ***IntMaskSrc***)
- FUART_INTStatus FUART_GetINTMask(TSB_FUART_TypeDef * ***FUARTx***)
- FUART_INTStatus FUART_GetRawINTStatus(TSB_FUART_TypeDef * ***FUARTx***)
- FUART_INTStatus FUART_GetMaskedINTStatus(TSB_FUART_TypeDef * ***FUARTx***)
- void FUART_ClearINT(

TSB_FUART_TypeDef * ***FUARTx***, FUART_INTStatus ***INTStatus***)

- void FUART_SetDMAOnErr(
  TSB_FUART_TypeDef * ***FUARTx***, FunctionalState ***NewState***)
- void FUART_SetFIFODMA(TSB_FUART_TypeDef * ***FUARTx***,
  FUART_Direction ***Direction***, FunctionalState ***NewState***)
- FUART_AllModemStatus FUART_GetModemStatus(
  TSB_FUART_TypeDef * ***FUARTx***)
- void FUART_SetRTSStatus(
  TSB_FUART_TypeDef * ***FUARTx***, FUART_ModemStatus ***Status***)
- void FUART_SetDTRStatus(
  TSB_FUART_TypeDef * ***FUARTx***, FUART_ModemStatus ***Status***)

8.2.2 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) Full UART 構成と初期化、共通動作:
   FUART_Enable(), FUART_Disable(), FUART_Init(), FUART_GetRxData(), FUART_SetTxData(), FUART_GetErrStatus(), FUART_ClearErrStatus(), FUART_GetBusyState(), FUART_GetStorageStatus(), FUART_SetSendBreak()
2) FIFO と DMA の設定
   FUART_EnableFIFO(), FUART_DisableFIFO(), FUART_SetINTFIFOLevel(), FUART_SetFIFODMA(), FUART_SetDMAOnErr()
3) 割り込み制御と割り込み状態の取得:
   FUART_SetINTMask(), FUART_GetINTMask(), FUART_GetRawINTStatus(), FUART_GetMaskedINTStatus(), FUART_ClearINT()
4) モデム制御:
   FUART_GetModemStatus(), FUART_SetRTSStatus(), FUART_SetDTRStatus()
5) IrDA の設定
   FUART_EnableIrDA(), FUART_DisableIrDA(), FUART_SetIrDAEncodeMode(), FUART_SetIrDADivisor()

8.2.3 関数仕様

補足: 下記の全 API において、パラメータ “TSB_FUART_TypeDef* ***FUARTx***” は、***FUART0*** または ***FUART1*** のいずれかを指定してください。

8.2.3.1 FUART_Enable

Full UART チャネルの有効化

関数のプロトタイプ宣言:

void
FUART_Enable(TSB_FUART_TypeDef * ***FUARTx***)

引数:

FUARTx: Full UART チャネルを指定します。

機能:

Full UART チャネルを有効にします。

戻り値:

なし

8.2.3.2 FUART_Disable

Full UART チャネルの無効化

関数のプロトタイプ宣言:

void
FUART_Disable(TSB_FUART_TypeDef * ***FUARTx***)

引数:

FUARTx: Full UART チャネルを指定します。

機能:

Full UART チャネルを無効にします。

戻り値:

なし

8.2.3.3 FUART_GetRxData

受信データの取得

関数のプロトタイプ宣言:

uint32_t
FUART_GetRxData(TSB_FUART_TypeDef * ***FUARTx***)

引数:

FUARTx: Full UART チャネルを指定します。

機能:

受信データを取得します。
本 **API** は、**FUART_GetStorageStatus(FUARTx, FUART_RX)**の戻り値が **FUART_STORAGE_NORMAL** あるいは **FUART_STORAGE_FULL** の場合に使用してください。

戻り値:

受信データ

8.2.3.4 FUART_SetTxData

送信データの設定

関数のプロトタイプ宣言:

```
void
FUART_SetTxData(TSB_FUART_TypeDef * FUARTx,
                uint32_t Data)
```

引数:

FUARTx: Full UART チャネルを指定します。

Data: 送信データポインタです。データサイズは 0x00 - 0xFF です。

機能:

送信用にデータを設定し、***FUATx*** で選択された Full UART チャネル経由で送信を開始します。

戻り値:

なし

8.2.3.5 FUART_GetErrStatus

受信エラーステータスの取得

関数のプロトタイプ宣言:

```
FUART_Err
FUART_GetErrStatus(TSB_FUART_TypeDef * FUARTx)
```

引数:

FUARTx: Full UART チャネルを指定します。

機能:

本 API は、データ転送後にエラーステータスを取得します。そのため本 API は、**FUART_GetRxData(FUARTx)**の後に実行してください。ただし、このリードシーケンスはエラーステータス情報を取得した直後のみ実行可能です。

戻り値:
FUART_NO_ERR: エラーはありません
FUART_OVERRUN: オーバーランエラー
FUART_PARITY_ERR: パリティエラー
FUART_FRAMING_ERR: フレミングエラー
FUART_BREAK_ERR: ブレークエラー
FUART_ERRS: 2 つ以上のエラー

8.2.3.6 FUART_ClearErrStatus

受信エラーステータスのクリア

関数のプロトタイプ宣言:

void
FUART_ClearErrStatus(TSB_FUART_TypeDef * ***FUARTx***)

引数:

FUARTx: Full UART チャネルを指定します。

機能:

フレーミングエラー、パリティエラー、ブレークエラー、オーバーランエラーの各エラーがクリアされます。

戻り値:

なし

8.2.3.7 FUART_GetBusyState

データ送信状態の取得

関数のプロトタイプ宣言:

WorkState
FUART_GetBusyState(TSB_FUART_TypeDef * ***FUARTx***)

引数:

FUARTx: Full UART チャネルを指定します。

機能:

データ送信中であるか停止中であるか状態を取得します。

戻り値:

データ送信状態:
BUSY: データ送信中
DONE: データ送信が停止中

8.2.3.8 FUART_GetStorageStatus

送受信 FIFO または送受信保持レジスタの取得

関数のプロトタイプ宣言:

FUART_StorageStatus
FUART_GetStorageStatus(TSB_FUART_TypeDef * ***FUARTx***,
FUART_Direction ***Direction***)

引数:

FUARTx: Full UART チャネルを指定します。

Direction: 送信または受信のどちらかを選択します。
- **FUART_RX**: 受信 FIFO または受信保持レジスタ
- **FUART_TX**: 送信 FIFO または送信保持レジスタ

機能:

FIFO が許可されている場合、送受信 FIFO のステータスを取得します。
FIFO が 禁止されている場合、送受信保持レジスタのステータスを取得します。

戻り値:

FUART_STORAGE_EMPTY: FIFO または保持レジスタが empty 状態
FUART_STORAGE_NORMAL: FIFO または保持レジスタが正常状態
FUART_STORAGE_INVALID: FIFO または保持レジスタが無効状態
FUART_STORAGE_FULL: FIFO または保持レジスタが full 状態

8.2.3.9 FUART_SetIrDADivisor

IrDA 低電力除数の設定

関数のプロトタイプ宣言:

void
FUART_SetIrDADivisor(TSB_FUART_TypeDef * ***FUARTx***,
uint32_t ***Divisor***)

引数:

FUARTx: Full UART チャネルを指定します。

Divisor: IrDA 低電力除数を 0x01～0xFF の間で設定します。

機能:

Divisor は、UARTCLK の除算による、IrLPBaud16 シグナル生成に用いられる低電力カウンタ除数値を設定します。

本 API をコールする前に、IrDA 回路を許可してください。

戻り値:

なし

8.2.3.10 FUART_Init

Full UART チャネルの設定

関数のプロトタイプ宣言:

void
FUART_Init(TSB_FUART_TypeDef * ***FUARTx***,
FUART_InitTypeDef * ***InitStruct***)

引数:

FUARTx: Full UART チャネルを指定します。

InitStruct: ボーレート、ワード長、ストップビット、パリティ、転送モード、フロー制御の設定値を格納します。(詳細は“データ構造説明”を参照)

機能:

ボーレート、ワード長、ストップビット、パリティ、転送モード、フロー制御の設定を行います。Full UART 回路を有効にする前に本 API を実行してください。

戻り値:

なし

8.2.3.11 FUART_EnableFIFO

送受信 FIFO の有効化

関数のプロトタイプ宣言:

void
FUART_EnableFIFO(TSB_FUART_TypeDef * ***FUARTx***)

引数:

FUARTx: Full UART チャネルを指定します。

機能:

送受信 FIFO を許可します。

戻り値:

なし

8.2.3.12 FUART_DisableFIFO

送受信 FIFO の無効化

関数のプロトタイプ宣言:

void
FUART_DisableFIFO(TSB_FUART_TypeDef * ***FUARTx***)

引数:

FUARTx: Full UART チャネルを指定します。

機能:

送受信 FIFO を禁止し、モードをキャラクタモードに変更します。

戻り値:

なし

8.2.3.13 FUART_SetSendBreak

ブレーク付き送信の選択

関数のプロトタイプ宣言:

void
FUART_SetSendBreak(TSB_FUART_TypeDef * ***FUARTx***,
FunctionalState ***NewState***)

引数:

FUARTx: Full UART チャネルを指定します。

NewState: ブレーク付き送信の許可/禁止を選択します。

- **ENABLE**: ブレーク送信する。
- **DISABLE**: ブレーク送信しない。

機能:

ブレーク付き送信の許可/禁止を選択します。ブレーク状態を生成するには、最低 1 つ以上のフレームを送信中に、本 API にてイネーブルにしてください。ブレーク状態が生成された場合でも、送信 FIFO には影響を与えません。

戻り値:

なし

8.2.3.14 FUART_SetIrDAEncodeMode

IrDA SIR 低電力モードの設定

関数のプロトタイプ宣言:

```
void
FUART_SetIrDAEncodeMode(TSB_FUART_TypeDef * FUARTx,
                        uint32_t Mode)
```

引数:

FUARTx: Full UART チャネルを指定します。

Mode: IrDA SIR 低電力モードを選択します。
- **FUART_IRDA_3_16_BIT_PERIOD_MODE**: ノーマルモード
- **FUART_IRDA_3_TIMES_IRLPBAUD16_MODE**: 低電力モード

機能:

IrDA SIR 低電力モードを設定します。IrDA SIR 低電力モードとして **FUART_IRDA_3_TIMES_IRLPBAUD16_MODE** を選択すると、消費電力を軽減できますが、送信距離が短くなる可能性があります。

戻り値:

なし

8.2.3.15 FUART_EnableIrDA

SIR 許可

関数のプロトタイプ宣言:

```
Result
FUART_EnableIrDA(TSB_FUART_TypeDef * FUARTx)
```

引数:

FUARTx: Full UART チャネルを指定します。

機能:

IrDA 回路が許可されます。UART が無効の場合、本 API は何もせず、戻り値がエラートなします。

戻り値:

SUCCESS: 成功
ERROR: 失敗

8.2.3.16 FUART_DisableIrDA

SIR 禁止

関数のプロトタイプ宣言:

```
void
FUART_DisableIrDA(TSB_FUART_TypeDef * FUARTx)
```

引数:

FUARTx: Full UART チャネルを指定します。

機能:

フル UART が許可の場合、本 API は IrDA を禁止にします。UART が無効の場合、本 API は何もせず、戻り値がエラートなします。

戻り値:

なし

8.2.3.17 FUART_SetINTFIFOLevel

送受信割り込み FIFO レベルの選択

関数のプロトタイプ宣言:

```
void
FUART_SetINTFIFOLevel(TSB_FUART_TypeDef * FUARTx,
                      uint32_t RxLevel,
                      uint32_t TxLevel)
```

引数:

FUARTx: Full UART チャネルを指定します。

RxLevel: 受信割り込み FIFO レベルを選択します。(受信 FIFO は 32 段です)
- **FUART_RX_FIFO_LEVEL_4**: 受信 FIFO が 4 バイト以上
- **FUART_RX_FIFO_LEVEL_8**: 受信 FIFO が 8 バイト以上
- **FUART_RX_FIFO_LEVEL_16**: 受信 FIFO が 16 バイト以上
- **FUART_RX_FIFO_LEVEL_24**: 受信 FIFO が 24 バイト以上
- **FUART_RX_FIFO_LEVEL_28**: 受信 FIFO が 28 バイト以上

TxLevel: 送信割り込み FIFO レベルを選択します。(送信 FIFO は 32 段です)
- **FUART_TX_FIFO_LEVEL_4**: 送信 FIFO が 4 バイト以上
- **FUART_TX_FIFO_LEVEL_8**: 送信 FIFO が 8 バイト以上
- **FUART_TX_FIFO_LEVEL_16**: 送信 FIFO が 16 バイト以上
- **FUART_TX_FIFO_LEVEL_24**: 送信 FIFO が 24 バイト以上
- **FUART_TX_FIFO_LEVEL_28**: 送信 FIFO が 28 バイト以上

機能:

UARTTXINTR および UARTRXINTR が発生する FIFO レベルを定義します。このレベルを超えると割り込みが発生します。

戻り値:

なし

8.2.3.18 FUART_SetINTMask

割り込み発生要因の設定

関数のプロトタイプ宣言:

```
void
FUART_SetINTMask(TSB_FUART_TypeDef * FUARTx,
uint32_t IntMaskSrc)
```

引数:

FUARTx: Full UART チャネルを指定します。

IntMaskSrc: 割り込み発生要因を選択します。
- **FUART_NONE_INT_MASK**: すべての割り込みを禁止します。
- **FUART_RIN_MODEM_INT_MASK**: RIN モデム割り込みを許可します。
- **FUART_CTS_MODEM_INT_MASK**: CTS モデム割り込みを許可します。
- **FUART_DCD_MODEM_INT_MASK**: DCD モデム割り込みを許可します。
- **FUART_DSR_MODEM_INT_MASK**: DSR モデム割り込みを許可します。
- **FUART_RX_FIFO_INT_MASK**: 受信割り込みを許可します。
- **FUART_TX_FIFO_INT_MASK**: 送信割り込みを許可します。
- **FUART_RX_TIMEOUT_INT_MASK**: 受信タイムアウト割り込みを許可します。

- **FUART_FRAMING_ERR_INT_MASK**: フレーミングエラー割り込みを許可します。
- **FUART_PARITY_ERR_INT_MASK**: パリティエラー割り込みを許可します。
- **FUART_BREAK_ERR_INT_MASK**: ブレークエラー割り込みを許可します。
- **FUART_OVERRUN_ERR_INT_MASK**: オーバーランエラー割り込みを許可します。
- **FUART_ALL_INT_MASK**: すべての割り込みを許可します。

機能:

要因毎に割り込み発生の許可/禁止を設定します。選択されていない要因の割り込みは禁止されます。

戻り値:

なし

8.2.3.19 FUART_GetINTMask

割り込み発生要因の取得

関数のプロトタイプ宣言:

FUART_INTStatus
FUART_GetINTMask(TSB_FUART_TypeDef * ***FUARTx***)

引数:

FUARTx: Full UART チャネルを指定します。

機能:

割り込み発生の許可/禁止状態を要因毎に取得します。

戻り値:

FUART_INTStatus: 割り込み発生要因が格納された変数です。
(詳細は“データ構成説明”を参照)

8.2.3.20 FUART_GetRawINTStatus

割り込み許可/禁止設定前の割り込みステータスの取得

関数のプロトタイプ宣言:

FUART_INTStatus
FUART_GetRawINTStatus(TSB_FUART_TypeDef * ***FUARTx***)

引数:

FUARTx: Full UART チャネルを指定します。

機能:

割り込み許可/禁止設定前の割り込みステータスを取得します。

戻り値:

FUART_INTStatus: 割り込みステータスが格納された変数です。(詳細は“データ構成説明”を参照)

8.2.3.21 FUART_GetMaskedINTStatus

割り込み許可/禁止設定後の割り込みステータスの取得

関数のプロトタイプ宣言:

```
FUART_INTStatus
FUART_GetMaskedINTStatus(TSB_FUART_TypeDef * FUARTx)
```

引数:

FUARTx: Full UART チャネルを指定します。

機能:

割り込み許可/禁止設定後の割り込みステータスを取得します。

戻り値:

FUART_INTStatus: 割り込みステータスが格納された変数です。(詳細は“データ構成説明”を参照)

8.2.3.22 FUART_ClearINT

割り込み要因のクリア

関数のプロトタイプ宣言:

```
void
FUART_ClearINT(TSB_FUART_TypeDef * FUARTx,
               FUART_INTStatus INTStatus)
```

引数:

FUARTx: Full UART チャネルを指定します。

INTStatus: クリア対象の割り込み要因を格納してください。(詳細は“データ構成説明”を参照)

機能:

割り込み要因をクリアします。

戻り値:

なし

8.2.3.23 FUART_SetDMAOnErr

DMA オンエラーの許可/禁止選択

関数のプロトタイプ宣言:

```
void
FUART_SetDMAOnErr(TSB_FUART_TypeDef * FUARTx,
                  FunctionalState NewState)
```

引数:

FUARTx: Full UART チャネルを指定します。

NewState: DMA オンエラーの許可/禁止を選択します。
- **ENABLE**: 許可。
- **DISABLE**: 禁止。

機能:

DMA オンエラーの許可/禁止を設定します。許可が選択されると、データ受信中にエラーが発生したときに DMA 受信要求と UARTxRXDMASREQ と UARTxRXDMABREQ が禁止されます。

戻り値:

なし

8.2.3.24 FUART_SetFIFODMA

送受信 DMA の許可/禁止選択

関数のプロトタイプ宣言:

```
void
FUART_SetFIFODMA(TSB_FUART_TypeDef * FUARTx,
                 FUART_Direction Direction,
                 FunctionalState NewState)
```

引数:

FUARTx: Full UART チャネルを指定します。

Direction: 送信または受信のどちらかを選択します。
- **FUART_RX**: 受信 FIFO または受信保持レジスタ
- **FUART_TX**: 送信 FIFO または送信保持レジスタ

NewState: 送信 DMA または受信 DMA の許可/禁止を選択します。
- **ENABLE**: 許可。
- **DISABLE**: 禁止。

機能:

送信 DMA または受信 DMA の許可/禁止を選択します。

DMAC を用いた送信/受信 FIFO のデータ転送の場合、バス幅を 8bit に設定してください。

戻り値:

なし

8.2.3.25 FUART_GetModemStatus

モデム状態の取得

関数のプロトタイプ宣言:

FUART_AllModemStatus
FUART_GetModemStatus(TSB_FUART_TypeDef * ***FUARTx***)

引数:

FUARTx: Full UART チャネルを指定します。

機能:

CTS, DSR, DCD, RIN, DTR, RTS の各モデム状態を取得します。

戻り値:

FUART_AllModemStatus: 各モデム状態を格納した変数です。(詳細は“データ構成説明”を参照)

8.2.3.26 FUART_SetRTSStatus

フル UART の RTS(送信要求)モデム状態の設定

関数のプロトタイプ宣言:

void
FUART_SetRTSStatus(TSB_FUART_TypeDef * ***FUARTx***,
FUART_ModemStatus ***Status***)

引数:

FUARTx: Full UART チャネルを指定します。

Status: 送信要求(RTS)のモデムステータス出力を選択します。
- **FUART_MODEM_STATUS_0**: モデムステータス出力を 0 にします。
- **FUART_MODEM_STATUS_1**: モデムステータス出力を 1 にします。

機能:

フル UART の RTS(送信要求)モデム状態を設定します。

戻り値:

なし

8.2.3.27 FUART_SetDTRStatus

フル UART DTR(データ送信準備完了)状態の設定

関数のプロトタイプ宣言:

void
FUART_SetDTRStatus(TSB_FUART_TypeDef * ***FUARTx***,
FUART_ModemStatus ***Status***)

引数:

FUARTx: Full UART チャネルを指定します。

Status: データ送信準備完了(DTR)のモデムステータス出力を選択します。
- **FUART_MODEM_STATUS_0**: モデムステータス出力を 0 にします。
- **FUART_MODEM_STATUS_1**: モデムステータス出力を 1 にします。

機能:

フル UART DTR(データ送信準備完了)状態を設定します。

戻り値:

なし

8.2.4 データ構造

8.2.4.1 FUART_InitTypeDef

メンバ:

uint32_t
BaudRate :ボーレートを設定します。0(bsp)は設定できません。また、2950000(bps)より小さい値を設定してください。

uint32_t
DataBits : フレームで送受信されたデータビットの数を設定します。
- **UART_DATA_BITS_5** : 5bit
- **UART_DATA_BITS_6** : 6bit
- **UART_DATA_BITS_7** : 7bit
- **UART_DATA_BITS_8** : 8bit

uint32_t
StopBits : 送信ストップビット長を設定します。
- **UART_STOP_BITS_1 :** 1bit
- **UART_STOP_BITS_2 :** 2bit

uint32_t
Parity : パリティ状態を設定します。
- **UART_NO_PARITY**:パリティの送信およびチェックなし
- **UART_0_PARITY**: パリティビットとして"0"を送信または受信
- **UART_1_PARITY**: パリティビットとして"1"を送信または受信
- **UART_EVEN_PARITY**: パリティビットとして偶数パリティを送信または受信
- **UART_ODD_PARITY**: パリティビットとして奇数パリティを送信または受信

uint32_t
Mode: 受信、送信あるいは両方の許可/禁止を設定します。
- **UART_ENABLE_TX** : 送信許可
- **UART_ENABLE_RX** : 受信許可
- **UART_ENABLE_TX | UART_ENABLE_RX** : 送受信許可

uint32_t
FlowCtrl: ハードウェアフロー制御を設定します。
- **UART_NONE_FLOW_CTRL:** フロー制御なし
- **UART_CTS_FLOW_CTRL:** CTS フロー制御許可
- **UART_RTS_FLOW_CTRL:** RTS フロー制御許可
- **UART_CTS_FLOW_CTRL | UART_RTS_FLOW_CTRL:** CTS/RTS フロー制御許可

8.2.4.2 FUART_INTStatus

メンバ:

uint32_t

All: Full UART 割り込みステータス、または割り込み制御

ビットフィールド:

uint32_t
RIN: 1 RIN モデム割り込み

uint32_t
CTS: 1 CTS モデム割り込み

uint32_t
DCD: 1 DCD モデム割り込み

uint32_t
DSR: 1 DSR モデム割り込み

uint32_t
RxFIFO: 1 受信 FIFO 割り込み

uint32_t
TxFIFO: 1 送信 FIFO 割り込み

uint32_t
RxTimeout: 1 受信タイムアウト割り込み

uint32_t
FramingErr: 1 フレーミングエラー割り込み

uint32_t
ParityErr: 1 パリティエラー割り込み

uint32_t
BreakErr: 1 ブレークエラー割り込み

uint32_t
OverrunErr: 1 オーバーランエラー割り込み

uint32_t
Reserved: 21 未使用

8.2.4.3 FUART_AllModemStatus

メンバ:

uint32_t
All: Full UART の全モデムステータス

ビットフィールド:

uint32_t
CTS: 1 CTS モデムステータス

uint32_t
DSR: 1　　DSR モデムステータス

uint32_t
DCD: 1　　DCD モデムステータス

uint32_t
Reserved1: 5　　未使用

uint32_t
RI: 1　　RIN モデムステータス

uint32_t
Reserved2: 1　　未使用

uint32_t
DTR: 1　　DTR モデムステータス

uint32_t
RTS: 1　　RTS モデムステータス

uint32_t
Reserved3: 20　　未使用

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/TX04_Periph_Driver/src/ tmpm461 _gpio.c
/Libraries/TX04_Periph_Driver/inc/tmpm461 _gpio.h

9.2 API 関数

9.2.1 関数一覧

- uint16_t GPIO_ReadData(GPIO_Port ***GPIO_x***)
- uint16_t GPIO_ReadDataBit(GPIO_Port ***GPIO_x***, uint16_t ***Bit_x***)
- void GPIO_WriteData(GPIO_Port ***GPIO_x***, uint16_t ***Data***)
- void GPIO_WriteDataBit(GPIO_Port ***GPIO_x***, uint16_t ***Bit_x***, uint16_t ***BitValue***)
- void GPIO_Init(GPIO_Port ***GPIO_x***, uint16_t ***Bit_x***,
  GPIO_InitTypeDef **** GPIO_InitStruct***)
- void GPIO_SetOutput(GPIO_Port ***GPIO_x***, uint16_t ***Bit_x***)
- void GPIO_SetInput(GPIO_Port ***GPIO_x***, uint16_t ***Bit_x***);
- void GPIO_SetOutputEnableReg(GPIO_Port ***GPIO_x***, uint16_t ***Bit_x***,
  FunctionalState ***NewState***)
- void GPIO_SetInputEnableReg(GPIO_Port **GPIO_x**, uint16_t **Bit_x**,
  FunctionalState ***NewState***)
- void GPIO_SetPullUp(GPIO_Port ***GPIO_x***, uint16_t ***Bit_x***,
  FunctionalState ***NewState***)
- void GPIO_SetPullDown(GPIO_Port ***GPIO_x***, uint16_t ***Bit_x***,
  FunctionalState ***NewState***)
- void GPIO_SetOpenDrain(GPIO_Port ***GPIO_x***, uint16_t ***Bit_x***,
  FunctionalState ***NewState***)
- void GPIO_EnableFuncReg(GPIO_Port ***GPIO_x***, uint16_t ***FuncReg_x***, uint16_t ***Bit_x***)
- void GPIO_DisableFuncReg(GPIO_Port ***GPIO_x***, uint16_t ***FuncReg_x***, uint16_t ***Bit_x***)

9.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています:

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

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

3) GPIO_EnableFuncReg(), GPIO_DisableFuncReg()

9.2.3 関数仕様

9.2.3.1 GPIO_ReadData

DATA データレジスタの読み込み

関数のプロトタイプ宣言:

uint16_t
GPIO_ReadData(GPIO_Port ***GPIO_x***)

引数:

GPIO_x: GPIO ポートを選択します。

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

機能:

DATA レジスタを読み込みます。

戻り値:

DATA レジスタの値

9.2.3.2 GPIO_ReadDataBit

ビット単位での DATA レジスタの読み込み

関数のプロトタイプ宣言:

uint16_t
GPIO_ReadDataBit(GPIO_Port ***GPIO_x***,
uint16_t ***Bit_x***)

引数:

GPIO_x: GPIO ポートを選択します。

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

Bit_x: GPIO 端子を選択します。
- **GPIO_BIT_0:** GPIO pin 0,
- **GPIO_BIT_1:** GPIO pin 1,
- **GPIO_BIT_2:** GPIO pin 2,
- **GPIO_BIT_3:** GPIO pin 3,
- **GPIO_BIT_4:** GPIO pin 4,
- **GPIO_BIT_5:** GPIO pin 5,
- **GPIO_BIT_6:** GPIO pin 6,
- **GPIO_BIT_7:** GPIO pin 7,
- **GPIO_BIT_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15.

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

9.2.3.3 GPIO_WriteData

DATA レジスタへの書き込み

関数のプロトタイプ宣言:

```
void
GPIO_WriteData(GPIO_Port  GPIO_x,
               uint16_t Data)
```

引数:

GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A.

- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PJ:** GPIO port J.
- **GPIO_PK:** GPIO port K.

Data: DATA レジスタへのライトデータを指定します。

機能:

DATA レジスタへ指定された値を書き込みます。

戻り値:

なし

9.2.3.4 GPIO_WriteDataBit

ビット単位での DATA レジスタの書き込み

関数のプロトタイプ宣言:

```
void
GPIO_WriteDataBit(GPIO_Port GPIO_x,
                  uint16_t Bit_x,
                  uint16_t BitValue)
```

引数:

GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PJ :** GPIO port J.
- **GPIO_PK :** GPIO port K.

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_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15,
- **GPIO_BIT_ALL:** GPIO pin[0:15],

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

9.2.3.5 GPIO_Init

GPIO ポートの初期設定

関数のプロトタイプ宣言:

```
void
GPIO_Init(GPIO_Port  GPIO_x,
          uint16_t Bit_x,
          GPIO_InitTypeDef * GPIO_InitStruct)
```

引数:

GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PJ:** GPIO port J.
- **GPIO_PK :** GPIO port K.

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_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15,
- **GPIO_BIT_ALL:** GPIO pin[0:15],

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

機能:

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

戻り値:

なし

9.2.3.6 GPIO_SetOutput

出力ポートの設定

関数のプロトタイプ宣言:

```
void
GPIO_SetOutput(GPIO_Port  GPIO_x,
                          uint16_t Bit_x);
```

引数:

GPIO_x: GPIO ポートを選択します。

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

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_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15,
- **GPIO_BIT_ALL:** GPIO pin[0:15],

機能:

出力ポートに設定します。

戻り値:

なし

9.2.3.7 GPIO_SetInput

入力ポートの設定

関数のプロトタイプ宣言:

```
void
GPIO_SetInput(GPIO_Port GPIO_x,
              uint16_t Bit_x)
```

引数:

GPIO_x: GPIO ポートを選択します。

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

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_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15,
- **GPIO_BIT_ALL:** GPIO pin[0:15],

機能:

入力ポートに設定します。

補足: AD 変換のアナログ入力として Port H を使用する場合、PHIE と PHUP は無効にしてください。

戻り値:

なし

9.2.3.8 GPIO_SetOutputEnableReg

出力ポートの許可/禁止設定

関数のプロトタイプ宣言:

```
void
GPIO_SetOutputEnableReg(GPIO_Port GPIO_x,
                        uint16_t Bit_x,
                        FunctionalState NewState)
```

引数:

GPIO_x: GPIO ポートを選択します。

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

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_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15,
- **GPIO_BIT_ALL:** GPIO pin[0:15],

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

9.2.3.9 GPIO_SetInputEnableReg

入力ポートの許可/禁止設定

関数のプロトタイプ宣言:

```
void
GPIO_SetInputEnableReg(GPIO_Port GPIO_x,
                       uint16_t Bit_x,
                       FunctionalState NewState)
```

引数:

GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A.
- **GPIO_PB:** GPIO port B.
- **GPIO_PC:** GPIO port C.
- **GPIO_PD:** GPIO port D.
- **GPIO_PE:** GPIO port E.
- **GPIO_PF:** GPIO port F.
- **GPIO_PG:** GPIO port G.
- **GPIO_PH:** GPIO port H.
- **GPIO_PJ:** GPIO port J.
- **GPIO_PK :** GPIO port K.

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_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15,
- **GPIO_BIT_ALL:** GPIO pin[0:15],

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

9.2.3.10 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,
               uint16_t Bit_x,
               FunctionalState NewState)
```

引数:

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

- **GPIO_PK :** GPIO port K.

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_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15,
- **GPIO_BIT_ALL:** GPIO pin[0:15],

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

9.2.3.11 GPIO_SetPullDown

内蔵プルダウンの設定

関数のプロトタイプ宣言:

```
void
GPIO_SetPullDown(GPIO_Port GPIO_x,
                 uint16_t Bit_x,
                 FunctionalState NewState)
```

引数:

GPIO_x: GPIO ポートを選択します。
- **GPIO_PE:** GPIO port E.

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_4:** GPIO pin 4,
- **GPIO_BIT_ALL:** GPIO pin[0:15],

NewState:

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

機能:

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

戻り値:

なし

9.2.3.12 GPIO_SetOpenDrain

CMOS/オープンドレインポートの設定

関数のプロトタイプ宣言:

```
void
GPIO_SetOpenDrain(GPIO_Port ***GPIO_x***,
                  uint16_t ***Bit_x***,
                  FunctionalState ***NewState***)
```

引数:

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

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_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15,
- **GPIO_BIT_ALL:** GPIO pin[0:15],

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

9.2.3.13 GPIO_EnableFuncReg

機能ポートの有効設定

関数のプロトタイプ宣言:

```
void
GPIO_EnableFuncReg(GPIO_Port GPIO_x,
                   uint16_t FuncReg_x,
                   uint16_t Bit_x);
```

引数:

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

FuncReg_x: GPIO 機能レジスタの番号を選択します。
- **GPIO_FUNC_REG_1** GPIO 機能レジスタ 1
- **GPIO_FUNC_REG_2** GPIO 機能レジスタ 2
- **GPIO_FUNC_REG_3** GPIO 機能レジスタ 3
- **GPIO_FUNC_REG_4** GPIO 機能レジスタ 4
- **GPIO_FUNC_REG_5** GPIO 機能レジスタ 5

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_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15,
- **GPIO_BIT_ALL:** GPIO pin[0:15],

機能:

GPIO 端子の機能を有効に設定します。

戻り値:

なし

9.2.3.14 GPIO_DisableFuncReg

機能ポートの無効設定

関数のプロトタイプ宣言:

```
void
GPIO_DisableFuncReg(GPIO_Port GPIO_x,
                    uint16_t FuncReg_x,
                    uint16_t Bit_x)
```

引数:

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

FuncReg_x: GPIO 機能レジスタの番号を選択します。
- **GPIO_FUNC_REG_1** GPIO 機能レジスタ 1
- **GPIO_FUNC_REG_2** GPIO 機能レジスタ 2
- **GPIO_FUNC_REG_3** GPIO 機能レジスタ 3
- **GPIO_FUNC_REG_4** GPIO 機能レジスタ 4
- **GPIO_FUNC_REG_5** GPIO 機能レジスタ 5

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_8:** GPIO pin 8,
- **GPIO_BIT_9:** GPIO pin 9,
- **GPIO_BIT_10:** GPIO pin 10,
- **GPIO_BIT_11:** GPIO pin 11,
- **GPIO_BIT_12:** GPIO pin 12,
- **GPIO_BIT_13:** GPIO pin 13,
- **GPIO_BIT_14:** GPIO pin 14,
- **GPIO_BIT_15:** GPIO pin 15,
- **GPIO_BIT_ALL:** GPIO pin[0:15],

機能:

GPIO 端子の機能を無効に設定します。

戻り値:

なし

9.2.4 データ構造

9.2.4.1 GPIO_InitTypeDef

メンバ:

uint16_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。

uint16_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。

uint16_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。

uint16_t
PullDown 内蔵プルダウンの有効/無効を選択します。
- **GPIO_PULLDOWN_ENABLE**: 内蔵プルダウンを有効にします。
- **GPIO_PULLDOWN_DISABLE**: 内蔵プルダウンを無効にします。

- **GPIO_PULLDOWN_NONE**: 内蔵プルダウンがない、または設定変更しません。

9.2.4.2 GPIO_RegTypeDef

メンバ:

uint16_t
PinDATA DATA レジスタのマスク値

uint16_t
PinCR CR レジスタのマスク値

uint16_t
PinFR[FRMAX] FR レジスタのマスク値

uint16_t
PinOD OD レジスタのマスク値

uint16_t
PinPUP PUP レジスタのマスク値

uint16_t
PinPDN PDN レジスタのマスク値

uint16_t
PinPIE IE レジスタのマスク値

9.2.4.3 TSB_Port_TypeDef

メンバ:

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

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

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

uint32_t
RESERVED0[RESER] 未使用

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

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

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

uint32_t
RESERVED1[RESER]　未使用

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

10. I2C

10.1 概要

本デバイスは I2C バスを 5 チャンネル (I2C0~4) 内蔵しています。

I2C バスは SDA と SCL を通して、外部デバイスがバスに接続されるバスで、複数のデバイスと通信が可能です。

また独自フォーマットのフリーデータフォーマットに対応しています。フリーデータフォーマットにおいて、データはマスタ側がデータ送信を行い、スレーブ側がデータ受信を行います。

I2C ドライバ API は、スレーブアドレスの設定、クロックの周波数選択、ACK のためのクロック発生、スタート/ストップ状態の発生、データの送受信、状態表示各種ステータスの取得を行う関数セットです。

本ドライバ API は、アプリで使用する API 定義を格納する以下のファイルで構成されています。
/Libraries/TX04_Periph_Driver/src/tmpm461_i2c.c
/Libraries/TX04_Periph_Driver/inc/tmpm461_i2c.h

10.2 API 関数

10.2.1 関数一覧

- void I2C_SetACK(TSB_I2C_TypeDef* ***I2Cx***, FunctionalState ***NewState***);
- void I2C_Init(TSB_I2C_TypeDef* ***I2Cx***, I2C_InitTypeDef* ***InitI2CStruct***);
- void I2C_SetBitNum(TSB_I2C_TypeDef* ***I2Cx***, uint32_t ***I2CBitNum***);
- void I2C_SWReset(TSB_I2C_TypeDef* ***I2Cx***);
- void I2C_ClearINTReq(TSB_I2C_TypeDef* ***I2Cx***);
- void I2C_GenerateStart(TSB_I2C_TypeDef* ***I2Cx***);
- void I2C_GenerateStop(TSB_I2C_TypeDef* ***I2Cx***);
- I2C_State I2C_GetState(TSB_I2C_TypeDef* ***I2Cx***);
- void I2C_SetSendData(TSB_I2C_TypeDef* ***I2Cx***, uint32_t ***Data***);
- uint32_t I2C_GetReceiveData(TSB_I2C_TypeDef* ***I2Cx***);
- void I2C_SetFreeDataMode(TSB_I2C_TypeDef* ***I2Cx,*** FunctionalState ***NewState***);
- FunctionalState I2C_GetSlaveAddrMatchState(TSB_I2C_TypeDef * ***I2Cx***);
- void I2C_SetPrescalerClock(TSB_I2C_TypeDef * ***I2Cx***, uint32_t ***PrescalerClock***);
- void I2C_SetINTReq(TSB_I2C_TypeDef * ***I2Cx***,FunctionalState ***NewState***);
- FunctionalState I2C_GetINTStatus(TSB_I2C_TypeDef * ***I2Cx***);
- void I2C_ClearINTOutput(TSB_I2C_TypeDef * ***I2Cx***);

10.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています:

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

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

10.2.3 関数仕様

補足:** 下記の全 API において、パラメータ “TSB_I2C_TypeDef* ***I2Cx” は、以下のいずれかを指定してください。

TSB_I2C0, TSB_I2C1, TSB_I2C2, TSB_I2C3, TSB_I2C4

10.2.3.1 I2C_SetACK

ACK クロックの発生/停止

関数のプロトタイプ宣言:

void
I2C_SetACK(TSB_I2C_TypeDef* ***I2Cx***,
FunctionalState ***NewState***)

引数:

I2Cx: I2C チャネルを指定します。

NewState ACK のためのクロックを発生する/発生しないを選択します。

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

機能:

ACK のためのクロックを発生する／発生しないを選択します。

戻り値:

なし

10.2.3.2 I2C_Init

I2C の初期化

関数のプロトタイプ宣言:

void
I2C_Init(TSB_I2C_TypeDef* ***I2Cx***,
I2C_InitTypeDef* ***InitI2CStruct***)

引数:

I2Cx: I2C チャネルを指定します。

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

機能:

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

戻り値:

なし

10.2.3.3 I2C_SetBitNum

転送ビット数の設定

関数のプロトタイプ宣言:

```
void
I2C_SetBitNum(TSB_I2C_TypeDef* I2Cx,
              uint32_t I2CBitNum)
```

引数:

I2Cx: I2C チャネルを指定します。

I2CBitNum: 転送ビット数を選択します。

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

機能:

転送ビット数を選択します。

戻り値:

なし

10.2.3.4 I2C_SWReset

ソフトウェアリセットの発生

関数のプロトタイプ宣言:

```
void
I2C_SWReset(TSB_I2C_TypeDef* I2Cx)
```

引数:

I2Cx: I2C チャネルを指定します。

機能:

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

戻り値:

なし

10.2.3.5 I2C_ClearINTReq

INTI2C 割込み要求の解除

関数のプロトタイプ宣言:

void
I2C_ClearINTReq(TSB_I2C_TypeDef* ***I2Cx***)

引数:

I2Cx: I2C チャネルを指定します。

機能:

INTI2C 割込み要求を解除します。

戻り値:

なし

10.2.3.6 I2C_GenerateStart

マスタモードの選択とスタートコンディションの発生

関数のプロトタイプ宣言:

void
I2C_GenerateStart(TSB_I2C_TypeDef* ***I2Cx***)

引数:

I2Cx: I2C チャネルを指定します。

機能:

マスタモードを選択し、スタートコンディションを発生します。

戻り値:

なし

10.2.3.7 I2C_GenerateStop

マスタモードの選択とストップコンディションの発生

関数のプロトタイプ宣言:

void
I2C_GenerateStop(TSB_I2C_TypeDef* ***I2Cx***)

引数:

I2Cx: I2C チャネルを指定します。

機能:

マスタモードを選択し、ストップコンディションを発生します。

戻り値:

なし

10.2.3.8 I2C_GetState

I2C バスステータスの取得

関数のプロトタイプ宣言:

I2C_State
I2C_GetState(TSB_I2C_TypeDef* ***I2Cx***)

引数:

I2Cx: I2C チャネルを指定します。

機能:

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

戻り値:

I2C バスステータス

10.2.3.9 I2C_SetSendData

送信データの設定と送信開始

関数のプロトタイプ宣言:

```
void
I2C_SetSendData(TSB_I2C_TypeDef* I2Cx,
                uint32_t Data)
```

引数:

I2Cx: I2C チャネルを指定します。

Data: 送信データを設定します。送信データの最大値は 0xFF です。

機能:

送信データの設定と送信を開始します。

戻り値:

なし

10.2.3.10 I2C_GetReceiveData

受信データの取得

関数のプロトタイプ宣言:

```
uint32_t
I2C_GetReceiveData(TSB_I2C_TypeDef* I2Cx)
```

引数:

I2Cx: I2C 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

10.2.3.11 I2C_SetFreeDataMode

I2C フリーデータフォーマットの設定

関数のプロトタイプ宣言:

void
I2C_SetFreeDataMode(TSB_I2C_TypeDef* ***I2Cx***,
FunctionalState ***NewState***)

引数:

I2Cx: I2C チャネルを指定します。

NewState: システムが IDLE モードの場合に以下の状態を選択します。
- **ENABLE:** スレーブアドレスを認識しない(フリーデータフォーマット)。
- **DISABLE:** スレーブアドレスを認識する。

機能:

I2C フリーデータフォーマットを設定します。フリーデータフォーマット時、マスタ時は送信に、スレーブ時は受信に転送方向が固定されます。フリーデータフォーマットを解除するには **I2C_Init()** をコールしてください。

戻り値:

なし

10.2.3.12 I2C_GetSlaveAddrMatchState

スレーブアドレス一致検出およびゼネラルコール検出選択状態の取得

関数のプロトタイプ宣言:

FunctionalState
I2C_ GetSlaveAddrMatchState(TSB_I2C_TypeDef* ***I2Cx***)

引数:

I2Cx: I2C チャネルを指定します。

機能:

スレーブアドレス一致検出およびゼネラルコール検出選択状態を取得します。

戻り値:

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

DISABLE:: スレーブ動作時、スレーブアドレス一致及びゼネラルコールを検出しません。

10.2.3.13 I2C_SetPrescalerClock

内部 SCL 出力クロックの周波数選択

関数のプロトタイプ宣言:

```
void
I2C_SetPrescalerClock(TSB_I2C_TypeDef* I2Cx,
                      uint32_t PrescalerClock)
```

引数:

I2Cx: I2C チャネルを指定します。

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

機能:

内部 SCL 出力クロックの周波数を選択します。

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

戻り値:

なし

10.2.3.14 I2C_SetINTReq

I2C 割込み出力の許可/禁止の設定

関数のプロトタイプ宣言:

```
void
I2C _SetINTReq(TSB_I2C_TypeDef* I2Cx,
               FunctionalState NewState)
```

引数:

I2Cx: I2C チャネルを指定します。

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

機能:

I2C 割込み出力の許可/禁止を選択します。

戻り値:

なし

10.2.3.15 I2C_GetINTStatus

I2C 割込み状態の取得

関数のプロトタイプ宣言:

FunctionalState
I2C_GetINTStatus(TSB_I2C_TypeDef* ***I2Cx***)

引数:

I2Cx: I2C チャネルを指定します。

機能:

I2C 割込み状態を取得します。

戻り値:

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

10.2.3.16 I2C_ClearINTOutput

I2C 割込みのクリア

関数のプロトタイプ宣言:

void
I2C_ClearINTOutput(TSB_I2C_TypeDef* ***I2Cx***)

引数:

I2Cx: I2C チャネルを指定します。

機能:

I2C 割込み出力(INTI2C)をクリアします。

戻り値:

なし

10.2.4 データ構造

10.2.4.1 I2C_InitTypeDef

メンバ:

uint32_t
I2CSelfAddr スレーブアドレスを設定します。ビット 0 の指定はできません。

uint32_t
I2CDataLen 送信ビット数を選択します。

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

uint32_t
I2CClkDiv: プリスケーラクロックの分周値を選択します。

- **I2C_SCK_CLK_DIV_20**: シリアルクロックは fprsck を 20 で割った商の値です。
- **I2C_SCK_CLK_DIV_24**: シリアルクロックは fprsck を 24 で割った商の値です。
- **I2C_SCK_CLK_DIV_32**: シリアルクロックは fprsck を 32 で割った商の値です。
- **I2C_SCK_CLK_DIV_48**: シリアルクロックは fprsck を 48 で割った商の値です。
- **I2C_SCK_CLK_DIV_80**: シリアルクロックは fprsck を 80 で割った商の値です。
- **I2C_SCK_CLK_DIV_144**: シリアルクロックは fprsck を 144 で割った商の値です。
- **I2C_SCK_CLK_DIV_272**: シリアルクロックは fprsck を 272 で割った商の値です。
- **I2C_SCK_CLK_DIV_528**: シリアルクロックは fprsck を 528 で割った商の値です。

uint32_t
PrescalerClkDiv: fprsck を出力するシステムクロックの分周値です。

- **I2C_PRESCALER_DIV_1**: fprsck は、fsys を 1 で割った商の値です。
- **I2C_PRESCALER_DIV_2**: fprsck は、fsys を 2 で割った商の値です。
- **I2C_PRESCALER_DIV_3**: fprsck は、fsys を 3 で割った商の値です。
- **I2C_PRESCALER_DIV_4**: fprsck は、fsys を 4 で割った商の値です。
- **I2C_PRESCALER_DIV_5**: fprsck は、fsys を 5 で割った商の値です。
- **I2C_PRESCALER_DIV_6**: fprsck は、fsys を 6 で割った商の値です。
- **I2C_PRESCALER_DIV_7**: fprsck は、fsys を 7 で割った商の値です。
- **I2C_PRESCALER_DIV_8**: fprsck は、fsys を 8 で割った商の値です。
- **I2C_PRESCALER_DIV_9**: fprsck は、fsys を 9 で割った商の値です。
- **I2C_PRESCALER_DIV_10**: fprsck は、fsys を 10 で割った商の値です。
- **I2C_PRESCALER_DIV_11**: fprsck は、fsys を 11 で割った商の値です。

- **I2C_PRESCALER_DIV_12**: fprsck は、fsys を 12 で割った商の値です。
- **I2C_PRESCALER_DIV_13**: fprsck は、fsys を 13 で割った商の値です。
- **I2C_PRESCALER_DIV_14**: fprsck は、fsys を 14 で割った商の値です。
- **I2C_PRESCALER_DIV_15**: fprsck は、fsys を 15 で割った商の値です。
- **I2C_PRESCALER_DIV_16**: fprsck は、fsys を 16 で割った商の値です。
- **I2C_PRESCALER_DIV_17**: fprsck は、fsys を 17 で割った商の値です。
- **I2C_PRESCALER_DIV_18**: fprsck は、fsys を 18 で割った商の値です。
- **I2C_PRESCALER_DIV_19**: fprsck は、fsys を 19 で割った商の値です。
- **I2C_PRESCALER_DIV_20**: fprsck は、fsys を 20 で割った商の値です。
- **I2C_PRESCALER_DIV_21**: fprsck は、fsys を 21 で割った商の値です。
- **I2C_PRESCALER_DIV_22**: fprsck は、fsys を 22 で割った商の値です。
- **I2C_PRESCALER_DIV_23**: fprsck は、fsys を 23 で割った商の値です。
- **I2C_PRESCALER_DIV_24**: fprsck は、fsys を 24 で割った商の値です。
- **I2C_PRESCALER_DIV_25**: fprsck は、fsys を 25 で割った商の値です。
- **I2C_PRESCALER_DIV_26**: fprsck は、fsys を 26 で割った商の値です。
- **I2C_PRESCALER_DIV_27**: fprsck は、fsys を 27 で割った商の値です。
- **I2C_PRESCALER_DIV_28**: fprsck は、fsys を 28 で割った商の値です。
- **I2C_PRESCALER_DIV_29**: fprsck は、fsys を 29 で割った商の値です。
- **I2C_PRESCALER_DIV_30**: fprsck は、fsys を 30 で割った商の値です。
- **I2C_PRESCALER_DIV_31**: fprsck は、fsys を 31 で割った商の値です。
- **I2C_PRESCALER_DIV_32**: fprsck は、fsys を 32 で割った商の値です。

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

FunctionalState
I2CACKState: ACK のためのクロックを発生する／発生しないを選択します。

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

10.2.4.2 I2C_State

メンバ:

uint32_t
All: すべての状態です。

Bit Fields:

uint32_t
LastRxBit: 最終受信ビットモニタ

uint32_t
GeneralCall: ゼネラルコール検出モニタ

uint32_t
SlaveAddrMatch: スレーブアドレス一致検出モニタ

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

uint32_t
INTReq: INTI2C 割込み要求状態モニタ

uint32_t
BusState: I2C バス状態モニタ

uint32_t
TRx: トランスミッタ/レシーバ選択状態モニタ

uint32_t
MasterSlave: マスタ/スレーブ選択状態モニタ

11. IGBT

11.1 概要

本製品は、は、2 チャネルの多目的タイマ (MPT)を内蔵しています。MPT は IGBT モードで動作します。

IGBT モードには、次のような機能があります。
1) 16 ビットプログラマブル矩形波出力モード(PPG、2 相)
2) 外部トリガスタート
3) 周期一致検出機能
4) 緊急停止機能
5) 同期スタートモード

IGBT ドライバの API では、IGBT モジュールを制御するため、スタートモード設定、動作モード、カウンタ状態、ソースクロック分周、初期出力レベル、トリガ/EMG ノイズ除去時間分周、アクティブ/インアクティブタイミング出力変化、波形周期出力、EMG 出力などの機能セットが提供されています。

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX04_Periph_Driver/src/tmpm461_igbt.c
/Libraries/TX04_Periph_Driver/inc/ tmpm461 _igbt.h

11.2 API 関数

11.2.1 関数一覧

- void IGBT_Enable(TSB_MT_TypeDef* ***IGBTx***)
- void IGBT_Disable(TSB_MT_TypeDef* ***IGBTx***)
- void IGBT_SetClkInCoreHalt(TSB_MT_TypeDef* ***IGBTx***, uint8_t ***ClkState***)
- void IGBT_SetSWRunState(TSB_MT_TypeDef* ***IGBTx***, uint8_t ***Cmd***)
- uint16_t IGBT_GetCaptureValue(TSB_MT_TypeDef* ***IGBTx***, uint8_t ***CapReg***)
- void IGBT_Init(TSB_MT_TypeDef* ***IGBTx***, IGBT_InitTypeDef* ***InitStruct***)
- void IGBT_Recount(TSB_MT_TypeDef* ***IGBTx***)
- void IGBT_ChangeOutputActiveTiming(TSB_MT_TypeDef* ***IGBTx***, uint8_t ***Output,*** uint16_t ***Timing***)
- void IGBT_ChangeOutputInactiveTiming(TSB_MT_TypeDef* ***IGBTx***, uint8_t ***Output,*** uint16_t ***Timing***)
- void IGBT_ChangePeriod(TSB_MT_TypeDef* ***IGBTx***, uint16_t ***Period***)
- WorkState IGBT_GetCntState(TSB_MT_TypeDef* ***IGBTx***)
- Result IGBT_CancelEMGState(TSB_MT_TypeDef* ***IGBTx***)
- IGBT_EMGStateTypeDef IGBT_GetEMGState(TSB_MT_TypeDef * ***IGBTx***)
- void IGBT_ChangeTrgValue(TSB_MT_TypeDef*___IGBTx___,uint16_t ***uTrgCnt***)
- void IGBT_ClSynSlaveChCounter(TSB_MT_TypeDef * ***IGBTx***)

11.2.2 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) IGBT の初期化:
   IGBT_Enable(), IGBT_Disable(), IGBT_Init()
2) カウンタ状態の取得と IGBT の設定:
   IGBT_SetClkInCoreHalt(), IGBT_SetSWRunState(), IGBT_Recount(),
   IGBT_GetCntState()
3) 動作パラメータの変更と IGBT 値の取得:
   IGBT_GetCaptureValue(), IGBT_ChangeOutputActiveTiming(),
   IGBT_ChangeOutputInactiveTiming(), IGBT_ChangePeriod()
4) EMG 保護状態の取得とキャンセル:
   IGBT_GetEMGState(), IGBT_CancelEMGState()
5) トリガ値の変更と同期クリア:
   IGBT_ChangeTrgValue(), IGBT_ClSynSlaveChCounter()

11.2.3 関数仕様

補足:** 下記の全 API において、パラメータ"TSB_MT_TypeDef* ***IGBTx"は、以下のいずれかの値となります。:

IGBT0, IGBT1.

11.2.3.1 IGBT_Enable

IGBT モードの許可

関数のプロトタイプ宣言:

void
IGBT_Enable(TSB_MT_TypeDef* ***IGBTx***)

引数:

IGBTx: IGBT モードでの MPT チャネルを指定します。

機能:

IGBTx により選択された指定 MPT チャネルをイネーブルにします。本 API を呼び出すと、MPT は IGBT モードにて動作します。指定チャネルは **IGBT_Init()**により初期化、設定する必要があります。

戻り値:

なし

11.2.3.2 IGBT_Disable

IGBT モードの禁止

関数のプロトタイプ宣言:

void
IGBT_Disable(TSB_MT_TypeDef* ***IGBTx***)

引数:

IGBTx: IGBT モードでの MPT チャネルを指定します。

機能:

IGBTx により選択された指定 MPT チャネルをディセーブルにします。

戻り値:

なし

11.2.3.3 IGBT_SetClkInCoreHalt

IGBT モードにおけるコア Halt 時の動作指定

関数のプロトタイプ宣言:

void
IGBT_SetClkInCoreHalt(TSB_MT_TypeDef* ***IGBTx***,
uint8_t ***ClkState***)

引数:

IGBTx: IGBT モードでの MPT チャネルを指定します。

ClkState: デバッグモードにおけるコア Halt 時の制御を指定します。

- **IGBT_RUNNING_IN_CORE_HALT**: クロック停止動作および MTOUT0x/MTOUT1x 出力の制御を行いません。
- **IGBT_STOP_IN_CORE_HALT**: コア Halt 中はクロックの動作が停止します。また、MTxIGEMGCR<IGEMGOC>の設定に従い、MTOUT0x/MTOUT1x 出力の制御を行います。

機能:

デバッグモードにおけるコア Halt 時、IGBT モードのクロックは、***ClkState*** に応じて停止または動作の継続が可能です。***ClkState*** は、**IGBT_STOP_IN_CORE_HALT** を選択することを強く推奨します。

戻り値:

なし

11.2.3.4 IGBT_SetSWRunState

IGBT モードにおけるカウント動作制御

関数のプロトタイプ宣言:

void
IGBT_SetSWRunState(TSB_MT_TypeDef* ***IGBTx,***
uint8_t ***Cmd***)

引数:

IGBTx: IGBT モードでの MPT チャネルを指定します。

***Cmd*:** カウントの開始/停止を選択します。

- **IGBT_RUN:** カウントを開始します。
- **IGBT_STOP:** カウントを停止します。カウンタは 0 クリアされます。

機能:

カウント動作の開始/停止を選択します。

戻り値:

なし

補足:

1) 実際のカウンタ開始または停止のタイミングは、IGBT モードの設定により異なります。**IGBT_CMD_START** または **IGBT_CMD_START_NO_START_INT** が ***StartMode***(詳細はデータ構成説明を参照)として選択されている場合、本 API は、カウンタを完全に制御することが可能です。***StartMode*** として他の値が設定されている場合、トリガもカウンタを制御することができます。

2) **IGBT_FALLING_TRG_START** または **IGBT_RISING_TRG_START** が ***StartMode***(詳細はデータ構成説明を参照)として選択されている場合、初期化と設定が完了すると、ソフトウェアはコマンドを発行します。**IGBT_SetSWRunState(*IGBTx*, IGBT_RUN)**は、トリガにより開始される前に発行してください。その後トリガによるカウンタの開始が可能になります。

3) EMG 入力が Low レベルで、EMG 割り込みが発生した場合、カウンタの停止には **IGBT_SetSWRunState(*IGBTx*, IGBT_STOP)** を使用してください。

11.2.3.5 IGBT_GetCaptureValue

IGBT モードにおけるキャプチャカウンタの取得

関数のプロトタイプ宣言:

uint16_t

IGBT_GetCaptureValue(TSB_MT_TypeDef* ***IGBTx,***
uint8_t ***CapReg***)

引数:

IGBTx: IGBT モードでの MPT 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。
- **IGBT_CAPTURE_0**: キャプチャレジスタ 0
- **IGBT_CAPTURE_1:** キャプチャレジスタ 1

機能:

現在のアップカウンタの値をキャプチャすることができます。

戻り値:

アップカウンタの値

補足:

カウンタ値は、***StartMode***(詳細はデータ構成説明を参照)として **IGBT_CMD_START** または **IGBT_CMD_START_NO_START_INT** が選択されたときのみ、本関数の呼び出しにより、取得されます。最初の入力エッジのタイミングで取得された値は、キャプチャレジスタ 0 に格納されます。2 番目の入力エッジで取得された値は、キャプチャレジスタ 1 に格納されます。

11.2.3.6 IGBT_Init

IGBT モードにおける MPT チャネルの初期化と設定

関数のプロトタイプ宣言:

void
IGBT_Init(TSB_MT_TypeDef* ***IGBTx***,
IGBT_InitTypeDef* ***InitStruct***)

引数:

IGBTx: IGBT モードでの MPT チャネルを指定します。

InitStruct: スタートモード、動作モード、停止状態での出力、スタートトリガ受付モード、割り込み周期、ソースクロック分周、出力 0/1 の初期化、トリガ入力、EMG 入力用ノイズ除去時間分周、出力 0/1 のアクティブ/インアクティブタイミング、IGBT 出力波形周期と EMG 機能設定 (詳細は、データ構成説明を参照)などの IBGT 設定を含む構成です。

機能:

IGBT_Enable()の呼び出しで IGBT モードをイネーブルにすると、本関数を IGBT モードでの指定 MPT の初期化、設定に使用できます。

戻り値:

なし

補足:

MPT が IGBT モードで動作している場合、対応している I/O ポートは EMG 入力端子として設定してください。

IGBT_CancelEMGState()が **SUCCESS** を返した場合のみ、本関数を呼び出してください。それ以外の場合、初期化と設定は無効です。

11.2.3.7 IGBT_Recount

カウントリスタート

関数のプロトタイプ宣言:

void
IGBT_Recount(TSB_MT_TypeDef* ***IGBTx***)

引数:

IGBTx: IGBT モードでの MPT チャネルを指定します。

機能:

カウンタのクリアとリスタートを行います。

戻り値:

なし

11.2.3.8 IGBT_ChangeOutputActiveTiming

IGBT 出力 0/1 のアクティブタイミングの変更

関数のプロトタイプ宣言:

void
IGBT_ChangeOutputActiveTiming(TSB_MT_TypeDef* ***IGBTx***,
uint8_t ***Output,***
uint16_t ***Timing***)

引数:

IGBTx: IGBT モードでの MPT チャネルを指定します。

Output: IGBT 出力ポートを選択します。
- **IGBT_OUTPUT_0:** IGBT 出力ポート 0
- **IGBT_OUTPUT_1:** IGBT 出力ポート 1

Timing: 新規の出力アクティブタイミングを指定します。本値は 0 から出力インアクティブのタイミングの間で設定してください。

機能:

出力アクティブタイミングの変更に使用されますが、新規のアクティブタイミングは、カウンタが周期の値と一致した後から有効となります。

戻り値:

なし

11.2.3.9 IGBT_ChangeOutputInactiveTiming

IGBT 出力 0/1 のインアクティブタイミングの変更。

関数のプロトタイプ宣言:

```
void
IGBT_ChangeOutputInactiveTiming(TSB_MT_TypeDef* IGBTx,
                                uint8_t Output,
                                uint16_t Timing)
```

引数:

IGBTx: IGBT モードでの MPT チャネルを指定します。

Output: IGBT 出力ポートを選択します。
- **IGBT_OUTPUT_0:** IGBT 出力ポート 0
- **IGBT_OUTPUT_1:** IGBT 出力ポート 1

Timing: 新規の出力インアクティブタイミングを指定します。本値は出力アクティブのタイミングから ***Period*** の間で設定してください。

機能:

出力インアクティブタイミングの変更に使用されますが、新規のインアクティブタイミングは、カウンタが周期の値と一致した後から有効となります。

戻り値:

なし

11.2.3.10 IGBT_ChangePeriod

IGBT 出力周期の変更

関数のプロトタイプ宣言:

void
IGBT_ChangePeriod(TSB_MT_TypeDef* ***IGBTx***,
uint16_t ***Period***)

引数:

IGBTx: IGBT モードでの MPT チャネルを指定します。

Period は、新規出力周期を指定します。本値は、出力インアクティブのタイミングから 0xFFFF の間に設定してください。

機能:

出力の周期変更に使用されますが、新規周期は、カウンタが前の周期の値と一致した後から有効となります。

戻り値:

なし

11.2.3.11 IGBT_GetCntState

カウンタ状態の取得

関数のプロトタイプ宣言:

WorkState
IGBT_GetCntState(TSB_MT_TypeDef* ***IGBTx***)

引数:

IGBTx: IGBT モードでの MPT チャネルを指定します。

機能:

カウンタ状態を取得します。

戻り値:

カウンタ状態は、以下のようになります。
BUSY: カウンタ動作中
DONE: カウンタ停止中

11.2.3.12 IGBT_CancelEMGState

IGBT モードにおける EMG 状態のキャンセル

関数のプロトタイプ宣言:

Result
IGBT_CancelEMGState(TSB_MT_TypeDef* ***IGBTx***)

引数:

IGBTx: IGBT モードでの MPT チャネルを指定します。

機能:

IGBT の EMG 状態をキャンセルするのに使用されます。EMG 状態をキャンセルする前に カウンタの状態の確認のため **IGBT_GetCntState()**を呼び出し、EMG 入力のレベルが High であることを確認してください。

カウンタが動作中の場合、**IGBT_GetCntState()** は **BUSY** を返します。あるいは、EMG 入力が Low の場合、**IGBT_GetCntState()**は、**ERROR** を返します。EMG 状態はキャンセルできていません。

カウンタが停止している場合、**IGBT_GetCntState()** は **DONE** を返します。また、EMG 入力が High の場合、**IGBT_GetCntState()**は、EMG 状態をキャンセルし、**SUCCESS** を返します。

戻り値:

EMG 状態キャンセルの結果を返却します。
SUCCESS:キャンセルの成功
ERROR: キャンセルの失敗

11.2.3.13 IGBT_GetEMGState

IGBT の EMG 状態の取得

関数のプロトタイプ宣言:

IGBT_EMGStateTypeDef
IGBT_GetEMGState(TSB_MT_TypeDef * ***IGBTx***)

引数:

IGBTx: IGBT モードでの MPT チャネルを指定します。

機能:

IGBT の EMG 状態を取得します。EMG 状態には、ノイズ除去後の EMG 入力端子ステータスおよび EMG 保護ステータスが含まれます。

戻り値:

IGBT_EMGStateTypeDef: EMG ステータス (詳細は、データ構造説明を参照)

11.2.3.14 IGBT_ChangeTrgValue

IGBT 出力のタイマカウント値設定

関数のプロトタイプ宣言:

void
IGBT_ChangeTrgValue(TSB_MT_TypeDef*_**IGBTx**_,uint16_t _**uTrgCnt**_)

引数:

IGBTx: IGBT モードでの MPT チャネルを指定します。

uTrgCnt: タイマカウント値を設定します。

機能:

IGBT 出力のタイマカウント値を設定します。

戻り値:

なし

11.2.3.15 IGBT_ClSynSlaveChCounter

同期動作のアップカウンタクリア

関数のプロトタイプ宣言:

void
IGBT_ClSynSlaveChCounter(TSB_MT_TypeDef * _**IGBTx**_)

引数:

IGBTx: IGBT モードでの MPT チャネルを指定します。

機能:

同期動作のアップカウンタをクリアします。

戻り値:

なし

11.2.4 データ構造

11.2.4.1 IGBT_InitTypeDef

メンバ:

uint8_t
***StartMode*:** カウンタのスタートモードを選択します。

- **IGBT_CMD_START:** カウンタはソフトウェアコマンドにより制御されます。入力エッジのタイミングで取得されます。
- **IGBT_CMD_START_NO_START_INT:** カウンタはソフトウェアコマンドにより制御されます。入力エッジのタイミングで取得されます。カウンタ開始の際、割り込みは発生しません。
- **IGBT_CMD_FALLING_TRG_START:** カウンタの開始には 2 つの方法があります。1つは、トリガが Low レベルになっている間に、ソフトウェア開始コマンドを発行する方法です。もう 1 つは、ソフトウェア開始コマンドが発行された後、立下りエッジをトリガへ入力する方法です。
- **IGBT_CMD_FALLING_TRG_START_NO_START_INT:** カウンタの開始方法は、**IGBT_CMD_FALLING_TRG_START** と同一ですが、カウンタがソフトウェアコマンドにより開始された場合、割り込みは発生しません。
- **IGBT_CMD_RISING_TRG_START:** カウンタの開始には 2 つの方法があります。1つは、トリガが High レベルになっている間に、ソフトウェア開始コマンドを発行する方法です。 もう 1 つは、ソフトウェア開始コマンドが発行された後、立ち上がりエッジをトリガへ入力する方法です。
- **IGBT_CMD_RISING_TRG_START_NO_START_INT:** カウンタの開始方法は、**IGBT_CMD_RISING_TRG_START** と同一ですが、カウンタがソフトウェアコマンドにより開始された場合、割り込みは発生しません。
- **IGBT_FALLING_TRG_START:** 立下りトリガエッジでのみカウンタの開始が可能です。立ち上がりトリガエッジではカウンタの停止が可能です。([補足]参照)
- **IGBT_RISING_TRG_START:** 立ち上がりエッジでのみカウンタの開始が可能です。立下りエッジではカウンタを停止が可能です。([補足]参照)
- **IGBT_SYNSLAVE_CHNL_START:** 同期スタート。([補足]参照)

uint8_t
OperationMode: IGBT 動作モードを選択します。

- **IGBT_CONTINUOUS_OUTPUT**: 連続動作
- **IGBT_ONE_TIME_OUTPUT**: 単発動作

uint8_t
CntStopState: カウンタ停止時の出力状態を指定します。

- **IGBT_OUTPUT_INACTIVE**: IGBT 出力インアクティブレベル
- **IGBT_OUTPUT_MAINTAINED**: IGBT 出力は変わりません
- **IGBT_OUTPUT_NORMAL**: カウンタは、停止トリガを除き、その周期が終わるまで停止しません。カウンタが停止する場合、出力はインアクティブレベルにシフトします。

FunctionalState
ActiveAcceptTrg: 出力がアクティブレベルの場合に、開始トリガが受付可能かを選択します。

- **ENABLE**: 常時受け付け
- **DISABLE**: アクティブレベル出力中受付禁止

uint8_t
INTPeriod: 割り込み周期を選択します。
- **IGBT_INT_PERIOD_1**: 1 周期毎
- **IGBT_INT_PERIOD_2**: 2 周期毎
- **IGBT_INT_PERIOD_4**: 3 周期毎

uint8_t
ClkDiv: IGBT のソースクロックを選択します。
- **IGBT_CLK_DIV_1**: φT0(φT0/1)
- **IGBT_CLK_DIV_2**: φT1(φT0/2)
- **IGBT_CLK_DIV_4**: φT2(φT0/4)
- **IGBT_CLK_DIV_8**: φT4(φT0/8)

uint8_t
Output0Init: IGBT 出力 0 の初期化をします。
- **IGBT_OUTPUT_DISABLE**: IGBT 出力をディセーブルにします
- **IGBT_OUTPUT_HIGH_ACTIVE**: 初期出力は Low レベルです。High レベルはアクティブ出力です
- **IGBT_OUTPUT_LOW_ACTIVE**: 初期出力は High レベルです。Low レベルはアクティブ出力です

uint8_t
Output1Init: IGBT 出力 1 の初期化をします。
- **IGBT_OUTPUT_DISABLE**: IGBT 出力をディセーブルにします
- **IGBT_OUTPUT_HIGH_ACTIVE**: 初期出力は Low レベルです。High レベルはアクティブ出力です
- **IGBT_OUTPUT_LOW_ACTIVE**: 初期出力は High レベルです。Low レベルはアクティブ出力です

uint8_t
TrgDenoiseDiv: IGBT モードでのトリガ入力ノイズ除去時間を選択します。
- **IGBT_NO_DENOISE**: ノイズフィルタを経由しません
- **IGBT_DENOISE_DIV_16**: ノイズ除去時間 16 / fsys[s]
- **IGBT_DENOISE_DIV_32**: ノイズ除去時間 32 / fsys[s]
- **IGBT_DENOISE_DIV_48**: ノイズ除去時間 48 / fsys[s]
- **IGBT_DENOISE_DIV_64**: ノイズ除去時間 64 / fsys[s]
- **IGBT_DENOISE_DIV_80**: ノイズ除去時間 80 / fsys[s]
- **IGBT_DENOISE_DIV_96**: ノイズ除去時間 96 / fsys[s]
- **IGBT_DENOISE_DIV_112**: ノイズ除去時間 112 / fsys[s]
- **IGBT_DENOISE_DIV_128**: ノイズ除去時間 128 / fsys[s]
- **IGBT_DENOISE_DIV_144**: ノイズ除去時間 144 / fsys[s]
- **IGBT_DENOISE_DIV_160**: ノイズ除去時間 160 / fsys[s]
- **IGBT_DENOISE_DIV_176**: ノイズ除去時間 176 / fsys[s]
- **IGBT_DENOISE_DIV_192**: ノイズ除去時間 192 / fsys[s]
- **IGBT_DENOISE_DIV_208**: ノイズ除去時間 208 / fsys[s]
- **IGBT_DENOISE_DIV_224**: ノイズ除去時間 224 / fsys[s]
- **IGBT_DENOISE_DIV_240**: ノイズ除去時間 240 / fsys[s]

uint16_t
Output0ActiveTiming: 出力 0 のアクティブタイミングを指定します。本値は、0 から Output0InactiveTiming の間に設定してください。

uint16_t
Output0InactiveTiming: 出力 0 のインアクティブタイミングを指定します。本値は、Output0ActiveTiming から Period の間に設定してください。

uint16_t
Output1ActiveTiming: 出力 1 のアクティブタイミングを指定します。本値は、0 から Output1InactiveTiming の間に設定してください。

uint16_t
Output1InactiveTiming: 出力 1 のインアクティブタイミングを指定します。本値は、Output1ActiveTiming から Period の間に設定してください。

uint16_t
Period: IGBT 出力期間を指定します。最大値は 0xFFFF です。設定値は 0<MTxIGTRG≤MTxIGRG4≤0xFFFF となるように設定してください。

uint16_t
TrgCng: トリガのタイマカウント値を指定します。IGBT 出力期間を指定します。最大値は 0xFFFF です。

uint8_t
EMGFunction: EMG 停止機能を指定します。
- **IGBT_DISABLE_EMG**: IGBT の EMG 停止機能をディセーブルにします。
- **IGBT_EMG_OUTPUT_INACTIVE**: EMG 状態中 IGBT 出力をインアクティブレベルにします。
- **IGBT_EMG_OUTPUT_HIZ**: EMG 状態中の IGBT 出力を Hi-z にします。

uint8_t
EMGDenoiseDiv: IGBT モードでの EMG 入力用ノイズ除去分周時間を選択します。
- **IGBT_NO_DENOISE**: ノイズ除去時間なし
- **IGBT_DENOISE_DIV_16**: ノイズ除去時間 16 / fsys[s]
- **IGBT_DENOISE_DIV_32**: ノイズ除去時間 32 / fsys[s]
- **IGBT_DENOISE_DIV_48**: ノイズ除去時間 48 / fsys[s]
- **IGBT_DENOISE_DIV_64**: ノイズ除去時間 64 / fsys[s]
- **IGBT_DENOISE_DIV_80**: ノイズ除去時間 80 / fsys[s]
- **IGBT_DENOISE_DIV_96**: ノイズ除去時間 96 / fsys[s]
- **IGBT_DENOISE_DIV_112**: ノイズ除去時間 112 / fsys[s]
- **IGBT_DENOISE_DIV_128**: ノイズ除去時間 128 / fsys[s]
- **IGBT_DENOISE_DIV_144**: ノイズ除去時間 144 / fsys[s]
- **IGBT_DENOISE_DIV_160**: ノイズ除去時間 160 / fsys[s]
- **IGBT_DENOISE_DIV_176**: ノイズ除去時間 176 / fsys[s]
- **IGBT_DENOISE_DIV_192**: ノイズ除去時間 192 / fsys[s]
- **IGBT_DENOISE_DIV_208**: ノイズ除去時間 208 / fsys[s]
- **IGBT_DENOISE_DIV_224**: ノイズ除去時間 224 / fsys[s]
- **IGBT_DENOISE_DIV_240**: ノイズ除去時間 240 / fsys[s]

***補足:**

カウンタ開始にトリガを使用するには、最初にソフトウェア開始コマンドを発行してください。

同期スタートモードを使用するには、スレーブチャネルの MTXIGCR<IGSTA[1:0]>に"11"を設定します。マスタチャネルは"11"以外のモードを指定します。

11.2.4.2 IGBT_EMGStateTypeDef

メンバ:

enum
IGBT_EMGInputState: ノイズ除去後の EMG 入力端子のステータスを表示します。
- **IGBT_EMG_INPUT_LOW**: ノイズ除去後 EMG 入力端子 が Low。
- **IGBT_EMG_INPUT_HIGH**: ノイズ除去後 EMG 入力端子が High。

enum
IGBT_EMGProtectState: EMG 保護ステータスを表示します。
- **IGBT_EMG_NORMAL**: EMG 保護ステータスは、ノーマル動作。
- **IGBT_EMG_PROTECT**: EMG 保護ステータスは、保護動作中。

12. LVD

12.1 概要

本製品は、電圧検出回路 (LVD)を内蔵しています。電圧検出回路は、電圧の低下/上昇を検出することにより、リセット信号または割り込みを発生させます。

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

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX04_Periph_Driver/src/tmpm461_lvd.c
/Libraries/TX04_Periph_Driver/inc/tmpm461_lvd.h

12.2 API 関数

12.2.1 関数一覧

- void LVD_EnableVD1(void)
- void LVD_DisableVD1(void)
- void LVD_SetVD1Level(uint32_t ***VDLevel***)
- LVD_VDStatus LVD_GetVD1Status(void)
- void LVD_EnableVD2(void)
- void LVD_DisableVD2(void)
- void LVD_SetVD2Level(uint32_t ***VDLevel***)
- LVD_VDStatus LVD_GetVD2Status(void)
- void LVD_SetVD1ResetOutput(FunctionalState ***NewState***)
- void LVD_SetVD1INTOutput(FunctionalState ***NewState***)
- void LVD_SetVD2ResetOutput(FunctionalState ***NewState***)
- void LVD_SetVD2INTOutput(FunctionalState ***NewState***)

12.2.2 関数の種類

関数は、主に以下の 2 種類に分かれています:

1) LVD 機能の設定:
   LVD_EnableVD1(), LVD_DisableVD1(), LVD_SetVD1Level(), LVD_EnableVD2(), LVD_DisableVD2(), LVD_SetVD2Level(), LVD_SetVD1ResetOutput(), LVD_SetVD1INTOutput(), LVD_SetVD2ResetOutput(), LVD_SetVD2INTOutput()
2) 電圧検出状態の確認:
   LVD_GetVD1Status(), LVD_GetVD2Status()

12.2.3 関数仕様

12.2.3.1 LVD_EnableVD1

LVDLVL1 の許可

関数のプロトタイプ宣言:

void
LVD_EnableVD1(void)

引数:

なし。

機能:

LVDLVL1 を許可します。

戻り値:

なし

12.2.3.2 LVD_DisableVD1

LVDLVL1 の禁止

関数のプロトタイプ宣言:

void
LVD_DisableVD1(void)

引数:

なし。

機能:

LVDLVL1 を禁止します。

戻り値:

なし

12.2.3.3 LVD_SetVD1Level

LVDLVL1 用電圧レベルの選択

関数のプロトタイプ宣言:

void
LVD_SetVD1Level(uint32_t ***VDLevel***)

引数:

VDLevel: LVDLVL1 用の電圧レベルです。
- **LVD_VDLVL1_240:** 2.40 ± 0.1V
- **LVD_VDLVL1_250:** 2.50 ± 0.1V
- **LVD_VDLVL1_260:** 2.60 ± 0.1V
- **LVD_VDLVL1_270:** 2.70 ± 0.1V
- **LVD_VDLVL1_280:** 2.80 ± 0.1V
- **LVD_VDLVL1_290:** 2.90 ± 0.1V

機能:

LVDLVL1 用電圧レベルを設定します。

戻り値:

なし

12.2.3.4 LVD_GetVD1Status

LVDLVL1 状態の取得

関数のプロトタイプ宣言:

```
LVD_VDStatus
LVD_GetVD1Status(void)
```

引数:

なし。

機能:

LVDLVL1 のステータスを取得します。

戻り値:

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

12.2.3.5 LVD_EnableVD2

LVDLVL2 の許可

関数のプロトタイプ宣言:

```
void
LVD_EnableVD2(void)
```

引数:

なし。

機能:

LVDLVL2 を許可します。

戻り値:

なし

12.2.3.6 LVD_DisableVD2

LVDLVL2 の禁止

関数のプロトタイプ宣言:

void
LVD_DisableVD2(void)

引数:

なし。

機能:

LVDLVL2 を禁止します。

戻り値:

なし

12.2.3.7 LVD_SetVD2Level

LVDLVL2 用電圧レベルの選択

関数のプロトタイプ宣言:

void
LVD_SetVD2Level(uint32_t ***VDLevel***)

引数:

VDLevel: LVDLVL2 用の電圧レベルです。

- **LVD_VDLVL_280:** 2.80 ± 0.1V
- **LVD_VDLVL_285:** 2.85 ± 0.1V
- **LVD_VDLVL_290:** 2.90 ± 0.1V

- **LVD_VDLVL_295:** 2.95 ± 0.1V
- **LVD_VDLVL_300:** 3.00 ± 0.1V
- **LVD_VDLVL_305:** 3.05 ± 0.1V
- **LVD_VDLVL_310:** 3.10 ± 0.1V
- **LVD_VDLVL_315:** 3.15 ± 0.1V

機能:

LVDLVL2 用電圧レベルを設定します。

戻り値:

なし

12.2.3.8 LVD_GetVD2Status

LVDLVL2 状態の取得。

関数のプロトタイプ宣言:

LVD_VDStatus
LVD_GetVD2Status(void)

引数:

なし。

機能:

LVDLVL1 のステータスを取得します。

戻り値:

LVD_VDStatus: LVDLVL2 のステータス。

- ***LVD_VD_UPPER***: 電源電圧は検出電圧以上。
- ***LVD_VD_LOWER***: 電源電圧は検出電圧以下。

12.2.3.9 LVD_SetVD1ResetOutput

LVD1 の RESET 信号の出力

関数のプロトタイプ宣言:

void
LVD_SetVD1ResetOutput(FunctionalState ***NewState***)

引数:

NewState: LVDRST 信号の出力状態を選択します。

- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:

LVDRST 信号の出力状態を選択します。

戻り値:

なし

12.2.3.10 LVD_SetVD1INTOutput

LVD1 の INTLVD 信号の出力

関数のプロトタイプ宣言:

void
LVD_SetVD1INTOutput(FunctionalState ***NewState***)

引数:

NewState: INTLVD 信号の出力状態を選択します。

- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:

INTLVD 信号の出力状態を選択します。

戻り値:

なし

12.2.3.11 LVD_SetVD2ResetOutput

LVD2 の RESET 信号の出力

関数のプロトタイプ宣言:

void
LVD_SetVD2ResetOutput(FunctionalState ***NewState***)

引数:

NewState: LVDRST 信号の出力状態を選択します。

- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:

LVDRST 信号の出力状態を選択します。

戻り値:

なし

12.2.3.12 LVD_SetVD2INTOutput

LVD2 の INTLVD 信号の出力

関数のプロトタイプ宣言:

void

LVD_SetVD2INTOutput(FunctionalState ***NewState***)

引数:

NewState: INTLVD 信号の出力状態を選択します。

- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:

INTLVD 信号の出力状態を選択します。

戻り値:

なし

12.2.4 データ構造

なし

13. OFD

13.1 概要

本製品は、周波数検知回路（OFD）を内蔵し、高調波、低調波のよなクロックの異常状態あるい停止を検出すると、リセット信号を発生させることができます。

OFDドライバの API では、OFD 機能の有効/無効、検知周波数の設定、OFD 状態の取得などの機能セットが提供されています。

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX04_Periph_Driver/src/tmpm461_ofd.c
/Libraries/TX04_Periph_Driver/inc/tmpm461_ofd.h

13.2 API 関数

13.2.1 関数一覧

- void OFD_SetRegWriteMode(FunctionalState ***NewState***);
- void OFD_Enable(void);
- void OFD_Disable(void);
- void OFD_SetDetectionFrequency(OFD_OSCSource ***Source***,
  uint32_t ***HigherDetectionCount***,
  uint32_t ***LowerDetectionCount***);
- void OFD_Reset(FunctionalState ***NewState***);
- OFD_Status OFD_GetStatus(void);
- void OFD_SetDetectionMonitor(OFD_MonitorMode ***Mode***);

13.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています:

1) OFD 機能の設定:
   OFD_SetRegWriteMode(), OFD_SetDetectionFrequency(),
   OFD_SetDetectionMonitor(), OFD_Enable (), OFD_Disable()
2) OFD 状態の取得:
   OFD_GetStatus()
3) その他:
   OFD_Reset()

13.2.3 関数仕様

13.2.3.1 OFD_SetRegWriteMode

レジスタ書き込み制御。

関数のプロトタイプ宣言:

void
OFD_SetRegWriteMode(FunctionalState ***NewState***)

引数:

NewState: レジスタ書き込みを制御します。

- **ENABLE :** 許可。
- **DISABLE:** 禁止。

機能:

NewState が **ENABLE** の場合、OFDCR1 を除く全 OFD レジスタの書き込みを可能にします。***NewState*** が **DISABLE** の場合、OFDCR1 を除く全 OFD レジスタの書き込みを不可とします。

戻り値:

なし

13.2.3.2 OFD_Enable

OFD 機能の許可

関数のプロトタイプ宣言:

void
OFD_Enable(void)

引数:

なし。

機能:

OFD 機能を許可します。

戻り値:

なし

13.2.3.3 OFD_Disable

OFD 機能の禁止

関数のプロトタイプ宣言:

void
OFD_Disable(void)

引数:

なし。

機能:

OFD 機能を禁止します。

戻り値:

なし

13.2.3.4 OFD_SetDetectionFrequency

検知周波数の設定

関数のプロトタイプ宣言:

void
OFD_SetDetectionFrequency(OFD_OSCSource ***Source***,
uint32_t ***HigherDetectionCount***,
uint32_t ***LowerDetectionCount***);

引数:

Source: 検知する周波数の発振ソースを選択します。

- **OFD_IHOSC:** 内部高速発信器
- **OFD_EHOSC:** 外部高速発信器

HigherDetectionCount: 検知周波数上限値のカウント値。最大値は 0x1FFU。

LowerDetectionCount: 検知周波数下限値のカウント値。最大値は 0x1FFU。

機能:

外部または内部発振検知用のカウント値を設定します。

戻り値:

なし

13.2.3.5 OFD_Reset

OFD リセット発生制御

関数のプロトタイプ宣言:

void
OFD_Reset(FunctionalState ***NewState***)

引数:

NewState: OFD リセット発生の許可/禁止を選択します。

- **ENABLE :** 許可。
- **DISABLE:** 禁止。

機能:

OFD リセットの許可/禁止を選択します。

戻り値:

なし

13.2.3.6 OFD_GetStatus

OFD 動作状態、異常検知状態の取得

関数のプロトタイプ宣言:

OFD_Status
OFD_GetStatus(void)

引数:

なし。

機能:

OFD 動作状態と異常検知状態を取得します。

戻り値:

OFD_Status: OFD 状態を格納した構造体(詳細は“データ構造説明”を参照)

13.2.3.7 OFD_SetDetectionMonitor

検出対象クロックの選択

関数のプロトタイプ宣言:

void
OFD_SetDetectionMonitor(OFD_MonitorMode ***Mode***)

引数:

Mode: 検出対象クロックを選択します。

- **OFD_NORMAL :** 通常監視モード。
- **OFD_MONITOR:** モニタモード。

機能:

検出対象のクロックを選択します。

戻り値:

なし

13.2.4 データ構造

13.2.4.1 OFD_Status

メンバ:

uint32_t
All: すべてのステータス

ビットフィールド:

uint32_t
FrequencyError: 1 異常検知

uint32_t
OFDBusy: 1 OFD 動作中

14. RMC

14.1 概要

本デバイスは搬送波が取り除かれたリモコン信号の受信を行います。

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/TX04_Periph_Driver/src/tmpm461_rmc.c
/Libraries/TX04_Periph_Driver/inc/tmpm461_rmc.h

14.2 API 関数

14.2.1 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***)

14.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています:

1) 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()

14.2.3 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

14.2.3.1 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

14.2.3.2 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

14.2.3.3 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

14.2.3.4 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

14.2.3.5 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)

14.2.3.6 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

14.2.3.7 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

14.2.3.8 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

14.2.3.9 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

14.2.3.10 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

14.2.3.11 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

14.2.3.12 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

14.2.3.13 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)

14.2.3.14 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

14.2.3.15 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

14.2.3.16 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

14.2.4 データ構造

14.2.4.1 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)**

14.2.4.2 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。

14.2.4.3 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。

14.2.4.4 RMC_INTFactor

メンバ:

uint32_t

All: 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

15. RTC

15.1 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/TX04_Periph_Driver/src/tmpm461_rtc.c
/Libraries/ TX04_Periph_Driver/inc/tmpm461_rtc.h

15.2 API 関数

15.2.1 関数一覧

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

- void RTC_DisableAlarm(void);
- void RTC_SetRTCINT(FunctionalState ***NewState***);
- void RTC_SetAlarmOutput(uint8_t ***Output***);
- void RTC_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***);
- void RTC_SetProtectCtrl(FunctionalState ***NewState***)
- void RTC_EnableCorrection(void)
- void RTC_DisableCorrection(void)
- void RTC_SetCorrectionTime(uint8_t ***Time***)
- void RTC_SetCorrectionValue(RTC_CorrectionMode ***Mode***,uint16_t ***Cnt***)

15.2.2 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) 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) RTC 補正基準時間の設定:
   RTC_EnableCorrection(), RTC_DisableCorrection(), RTC_SetCorrectionTime(), RTC_SetCorrectionValue()
6) その他:
   RTC_SetAlarmOutput(), RTC_SetProtectCtrl(), RTC_SetRTCINT()

15.2.3 関数仕様

15.2.3.1 RTC_SetSec

時計の秒桁設定

関数のプロトタイプ宣言:

void
RTC_SetSec(uint8_t ***Sec***);

引数:

Sec:最大 59 までの秒桁設定の値。

機能:

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

戻り値:

なし

15.2.3.2 RTC_GetSec

時計の秒桁設定

関数のプロトタイプ宣言:

uint8_t
RTC_GetSec(void);

引数:

なし。

機能:

時計の秒桁の値を返します。

戻り値:

時計の秒桁:
- 0 ~ 59

15.2.3.3 RTC_SetMin

時計/アラームの分桁設定

関数のプロトタイプ宣言:

void
RTC_SetMin(RTC_FuncMode ***NewMode***,

uint8_t ***Min***);

引数:

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

Min: 最大 59 までの分析を設定します。

機能:

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

戻り値:

なし

15.2.3.4 RTC_GetMin

時計/アラームの分析読み込み

関数のプロトタイプ宣言:

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

引数:

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

機能:

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

戻り値:

分析:
- 0 ~ 59

15.2.3.5 RTC_GetAMPM

12 時間モードの AM/PM 読み込み

関数のプロトタイプ宣言:

uint8_t
RTC_GetAMPM(RTC_FuncMode ***NewMode***);

引数:

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

機能:

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

戻り値:

時計モード:
RTC_AM_MODE: AM
RTC_PM_MODE: PM

15.2.3.6 RTC_SetHour24

24 時間モードの時計/アラーム時桁設定

関数のプロトタイプ宣言:

void
RTC_SetHour24(RTC_FuncMode ***NewMode***,
uint8_t ***Hour***);

引数:

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

Hour: 最大 23 までの時桁を設定します。

機能:

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

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

戻り値:

なし

15.2.3.7 RTC_SetHour12

12 時間モードの時計/アラーム時桁設定

関数のプロトタイプ宣言:

```
void
RTC_SetHour12(RTC_FuncMode NewMode,
              uint8_t Hour,
              uint8_t AmPm);
```

引数:

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

Hour: 最大 11 までの時桁を設定します。

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

機能:

12 時間モードの時計/アラームの時桁を設定します。
NewMode が **RTC_CLOCK_MODE** の場合、時計機能の時桁を設定し、
NewMode が **RTC_ALARM_MODE** の場合、アラーム機能の時桁を設定します。
RTC レジスタは、INTRTC のタイミングに同期して書換えられます。この関数の実行後、1Hz 割り込みが発生するのを待つ必要があります。
*24 時間モードから 12 時間モードに変更する場合、本関数 **RTC_SetHour12()** によって HOURR レジスタを再度設定してください。

戻り値:

なし

15.2.3.8 RTC_GetHour

時計/アラームの時桁読み込み

関数のプロトタイプ宣言:

uint8_t
RTC_GetHour(RTC_FuncMode ***NewMode***);

引数:

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

機能:

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

戻り値:

24 時間モードでの時桁:
- 0 ~ 23

12H 時間モードでの時桁:
- 0 ~ 11

15.2.3.9 RTC_SetDay

時計/アラームの曜日設定

関数のプロトタイプ宣言:

void
RTC_SetDay(RTC_FuncMode ***NewMode***,
uint8_t ***Day***);

引数:

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

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

機能:

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

戻り値:

なし

15.2.3.10 RTC_GetDay

時計/アラームの曜日の読み込み

関数のプロトタイプ宣言:

```
uint8_t
RTC_GetDay(RTC_FuncMode NewMode);
```

引数:

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

機能:

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

戻り値:

曜日の値:
- 0 ~ 6

15.2.3.11 RTC_SetDate

時計/アラームの日桁設定

関数のプロトタイプ宣言:

```
void
RTC_SetDate(RTC_FuncMode NewMode,
            uint8_t Date);
```

引数:

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

Date: 1 から 31 の日桁を設定します。

機能:

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

戻り値:

なし

15.2.3.12 RTC_GetDate

時計/アラームの日桁読み込み

関数のプロトタイプ宣言:

uint8_t
RTC_GetDate(RTC_FuncMode ***NewMode***);

引数:

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

機能:

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

戻り値:

日桁:
- 1 ~ 31

15.2.3.13 RTC_SetMonth

時計の月桁設定

関数のプロトタイプ宣言:

void
RTC_SetMonth(uint8_t ***Month***);

引数:

Month: 1 から 12 の月桁を設定します。

機能:

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

戻り値:

なし

15.2.3.14 RTC_GetMonth

時計の月桁読み込み

関数のプロトタイプ宣言:

uint8_t
RTC_GetMonth(void);

引数:

なし。

機能:

時計の月桁の値を返します。

戻り値:

月桁:
- 1 ~ 12

15.2.3.15 RTC_SetYear

時計の年桁設定

関数のプロトタイプ宣言:

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

引数:

Year: 最大 99 までの年の値

機能:

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

戻り値:

なし

15.2.3.16 RTC_GetYear

時計の年桁の読み込み

関数のプロトタイプ宣言:

```
uint8_t
RTC_GetYear(void);
```

引数:

なし。

機能:

時計の年桁の値を返します。

戻り値:

年桁:
- 0 ~ 99

15.2.3.17 RTC_SetHourMode

24 時間時計/12 時間時計の選択

関数のプロトタイプ宣言:

```
void
RTC_SetHourMode(uint8_t HourMode);
```

引数:

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

機能:

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

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

15.2.3.18 RTC_GetHourMode

時計モードの読み込み

関数のプロトタイプ宣言:

```
uint8_t
RTC_GetHourMode(void);
```

引数:

なし。

機能:

時計モードを読み込みます。

戻り値:

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

15.2.3.19 RTC_SetLeapYear

うるう年の設定

関数のプロトタイプ宣言:

void
RTC_SetLeapYear(uint8_t ***LeapYear***);

引数:

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

機能:

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

戻り値:

なし

15.2.3.20 RTC_GetLeapYear

うるう年の読み込み

関数のプロトタイプ宣言:

uint8_t
RTC_GetLeapYear(void);

引数:

なし。

機能:

うるう年の状態を返します。

戻り値:

うるう年の状態を表す値

15.2.3.21 RTC_SetTimeAdjustReq

+/– 30 秒の補正

関数のプロトタイプ宣言:

void
RTC_SetTimeAdjustReq(void);

引数:

なし。

機能:

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

戻り値:

なし

15.2.3.22 RTC_GetTimeAdjustReq

ADJUST 要求状態の読み込み

関数のプロトタイプ宣言:

RTC_ReqState
RTC_GetTimeAdjustReq(void);

引数:

なし。

機能:

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

戻り値:

ADJUST 要求状態を読み込みます。

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

15.2.3.23 RTC_EnableClock

時計機能の起動

関数のプロトタイプ宣言:

void
RTC_EnableClock(void);

引数:

なし。

機能:

時計機能を有効にします。

戻り値:

なし

15.2.3.24 RTC_DisableClock

時計機能の終了

関数のプロトタイプ宣言:

void
RTC_DisableClock(void);

引数:

なし。

機能:

時計機能を無効にします。

戻り値:

なし

15.2.3.25 RTC_EnableAlarm

アラーム機能の起動

関数のプロトタイプ宣言:

void
RTC_EnableAlarm(void);

引数:

なし。

機能:

アラーム機能を有効にします。

戻り値:

なし

15.2.3.26 RTC_DisableAlarm

アラーム機能の終了

関数のプロトタイプ宣言:

void
RTC_DisableAlarm(void);

引数:

なし。

機能:

アラーム機能を無効にします。

戻り値:

なし

15.2.3.27 RTC_SetRTCINT

INTRTC 割り込みの有効/無効設定

関数のプロトタイプ宣言:

void
RTC_SetRTCINT(FunctionalState ***NewState***);

引数:

NewState*:* 以下から *INT RTC* の有効/無効を選択します。

- ***ENABLE****:* INTRTC 割り込み有効
- ***DISABLE****:* INTRTC 割り込み無効

機能:

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

戻り値:

なし

15.2.3.28 RTC_SetAlarmOutput

ALARM 端子の出力設定

関数のプロトタイプ宣言:

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

引数:

Output: 以下から、アラーム端子の出力を選択します。

- **RTC_LOW_LEVEL**: “0” パルス
- **RTC_PULSE_1_HZ**: 1Hz 周期の “0” パルス
- **RTC_PULSE_16_HZ**: 16Hz 周期の “0” パルス
- **RTC_PULSE_2_HZ**: 2Hz 周期の “0” パルス
- **RTC_PULSE_4_HZ**: 4Hz 周期の “0” パルス
- **RTC_PULSE_8_HZ**: 8Hz 周期の “0” パルス

機能:

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

戻り値:

なし

15.2.3.29 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

15.2.3.30 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

15.2.3.31 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

15.2.3.32 RTC_SetDateValue

時計の日付設定

関数のプロトタイプ宣言:

void
RTC_SetDateValue(RTC_DateTypeDef * ***DateStruct***);

引数:

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

機能:

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

戻り値:

なし

15.2.3.33 RTC_GetDateValue

時計の日付の読み込み

関数のプロトタイプ宣言:

void
RTC_GetDateValue(RTC_DateTypeDef * ***DateStruct***);

引数:

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

機能:

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

戻り値:

なし

15.2.3.34 RTC_SetTimeValue

時計の時刻設定

関数のプロトタイプ宣言:

void
RTC_SetTimeValue(RTC_TimeTypeDef * ***TimeStruct***);

引数:

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

機能:

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

戻り値:

なし

15.2.3.35 RTC_GetTimeValue

時計の時刻の読み込み

関数のプロトタイプ宣言:

void
RTC_GetTimeValue(RTC_TimeTypeDef * ***TimeStruct***);

引数:

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

機能:

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

戻り値:

なし

15.2.3.36 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

15.2.3.37 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

15.2.3.38 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

15.2.3.39 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

15.2.3.40 RTC_SetProtectCtrl

補正機能レジスタ書き込み制御

関数のプロトタイプ宣言:

void
RTC_SetProtectCtrl(FunctionalState **NewState**);

引数:

NewState: 補正機能レジスタへの書き込み許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:

補正機能レジスタ(RTCADJCTL、RTCADJDAT)への書き込み許可/禁止を選択します。

戻り値:

なし

15.2.3.41 RTC_EnableCorrection

補正機能の許可

関数のプロトタイプ宣言:

void
RTC_EnableCorrection(void);

引数:

なし。

機能:

補正機能を許可します。

戻り値:

なし

15.2.3.42 RTC_DisableCorrection

補正機能の禁止

関数のプロトタイプ宣言:

void
RTC_DisableCorrection(void);

引数:

なし。

機能:

補正機能を禁止します。

戻り値:

なし

15.2.3.43 RTC_SetCorrectionTime

補正基準時間の設定

関数のプロトタイプ宣言:

void
RTC_SetCorrectionTime(uint8_t **Time**);

引数:

Time:補正基準時間を選択します。

- **RTC_ADJ_TIME_1_SEC** : 1 秒
- **RTC_ADJ_TIME_10_SEC**: 10 秒
- **RTC_ADJ_TIME_20_SEC**: 20 秒
- **RTC_ADJ_TIME_30_SEC**: 30 秒
- **RTC_ADJ_TIME_1_MIN**: 1 分

機能:

補正基準時間を設定します。

戻り値:

なし

15.2.3.44 RTC_SetCorrectionValue

補正値の設定

関数のプロトタイプ宣言:

void
RTC_SetCorrectionValue(RTC_CorrectionMode **Mode**, uint16_t **Cnt**);

引数:

Mode: 補正符号を選択します。

- **RTC_CORRECTION_PLUS** : プラス補正
- **RTC_CORRECTION_MINUS**: マイナス補正

Cnt: 1 秒に対する補正値を選択します。

- **RTC_CORRECTION_PLUS** の場合、0~255 を選択できます。
- **RTC_CORRECTION_ MINUS** の場合、1~256 を選択できます。

機能:

補正値を選択します。

戻り値:

なし

15.2.4 データ構造

15.2.4.1 RTC_DateTypeDef

メンバ:

uint8_t

***LeapYear* :** うるう年を設定します:

- **RTC_LEAP_YEAR_0:** 現在の年(今年)がうるう年

- **RTC_LEAP_YEAR_1:** 現在がうるう年から 1 年目
- **RTC_LEAP_YEAR_2:** 現在がうるう年から 2 年目
- **RTC_LEAP_YEAR_3:** 現在がうるう年から 3 年目

uint8_t
Year 年桁の値(0～99)。

uint8_t
Month 月桁の値(1～12)。

uint8_t
Date 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

15.2.4.2 RTC_TimeTypeDef

メンバ:

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

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

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

uint8_t
Min 0～59 までの分桁の値。

uint8_t
Sec 0～59 までの秒桁の値。

15.2.4.3 RTC_AlarmTypeDef

メンバ:

uint8_t
Date アラーム機能有効時の日桁の値(1～31)。

uint8_t
Day アラーム機能有効時の週桁の値。

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

uint8_t
Hour アラーム機能有効時の時間桁の値。

uint8_t
AmPm アラーム機能有効時の AM/PM 選択の値:

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

uint8_t
Min アラーム機能有効時の分桁の値(0～59)。

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 3 チャネル内蔵しています。(SSP0, SSP1, SSP2)

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/TX04_Periph_Driver/src/tmpm461_ssp.c
/Libraries/TX04_Periph_Driver/inc/tmpm461_ssp.h

16.2 API 関数

16.2.1 関数一覧

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

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

16.2.2 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()

16.2.3 関数仕様

補足:** 下記の全 API において、パラメータ"TSB_SSP_TypeDef* ***SSPx"は、以下のいずれかの値となります。
SSP0, **SSP1**, **SSP2**

16.2.3.1 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(TSB_SSP_TypeDef * ***SSPx***)

引数:

SSPx*:* SSP チャネルを指定します。

機能:

SSP 動作を有効にします。

戻り値:

なし

16.2.3.2 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(TSB_SSP_TypeDef * ***SSPx***)

引数:

SSPx*:* SSP チャネルを指定します。

機能:

SSP 動作を無効にします。

戻り値:

なし

16.2.3.3 SSP_Init

SSP 通信の初期化

関数のプロトタイプ宣言:

void
SSP_Init(TSB_SSP_TypeDef * ***SSPx***,
SSP_InitTypeDef* ***InitStruct***)

引数:

SSPx*:* SSP チャネルを指定します。

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

16.2.3.4 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(TSB_SSP_TypeDef * SSPx,
                   uint8_t PreScale,
                   uint8_t ClkRate)
```

引数:

***SSPx**:* SSP チャネルを指定します。

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

16.2.3.5 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(TSB_SSP_TypeDef * SSPx,
                   SSP_FrameFormat FrameFormat)
```

引数:

***SSPx**:* SSP チャネルを指定します。

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

16.2.3.6 SSP_SetClkPolarity

SPxCLK 極性の選択

関数のプロトタイプ宣言:

void
SSP_SetClkPolarity(TSB_SSP_TypeDef * ***SSPx***,
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

16.2.3.7 SSP_SetClkPhase

SPxCLK フェーズの選択

関数のプロトタイプ宣言:

void
SSP_SetClkPhase(TSB_SSP_TypeDef * ***SSPx***,
SSP_ClkPhase ***ClkPhase***)

引数:

SSPx*:* SSP チャネルを指定します。

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

16.2.3.8 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(TSB_SSP_TypeDef * SSPx,
                uint8_t DataSize)
```

引数:

***SSPx**:* SSP チャネルを指定します。

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

16.2.3.9 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(TSB_SSP_TypeDef * SSPx,
                       FunctionalState NewState)
```

引数:

***SSPx**:* SSP チャネルを指定します。

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

16.2.3.10 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(TSB_SSP_TypeDef * SSPx,
                    SSP_MS_Mode Mode)
```

引数:

SSPx: SSP チャネルを指定します。

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

16.2.3.11 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(TSB_SSP_TypeDef * SSPx,
                          FunctionalState NewState)
```

引数:

***SSPx**:* SSP チャネルを指定します。

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

16.2.3.12 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(TSB_SSP_TypeDef * SSPx,
              uint16_t Data)
```

引数:

***SSPx**:* SSP チャネルを指定します。

Data: 送信データを 0～16 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

16.2.3.13 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(TSB_SSP_TypeDef * SSPx)
```

引数:

***SSPx**:* SSP チャネルを指定します。

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

16.2.3.14 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(TSB_SSP_TypeDef * ***SSPx***)

引数:

***SSPx**:* SSP チャネルを指定します。

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

16.2.3.15 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(TSB_SSP_TypeDef * ***SSPx***
SSP_Direction ***Direction***)

引数:

***SSPx**:* SSP チャネルを指定します。

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。

16.2.3.16 SSP_SetINTConfig

割り込みの制御

関数のプロトタイプ宣言:

```
void
SSP_SetINTConfig(TSB_SSP_TypeDef * SSPx,
                 uint32_t IntSrc)
```

引数:

SSPx*:* SSP チャネルを指定します。

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(SSP0, SSP_INTCFG_RX | SSP_INTCFG_TX)

戻り値:

なし

16.2.3.17 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(TSB_SSP_TypeDef * ***SSPx***)

引数:

SSPx*:* SSP チャネルを指定します。

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。

16.2.3.18 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(TSB_SSP_TypeDef * ***SSPx***)

引数:

SSPx*:* SSP チャネルを指定します。

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。

16.2.3.19 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(TSB_SSP_TypeDef * ***SSPx***)

引数:

SSPx*:* SSP チャネルを指定します。

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。

16.2.3.20 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(TSB_SSP_TypeDef * ***SSPx***,
uint32_t ***IntSrc***)

引数:

SSPx*:* SSP チャネルを指定します。

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

16.2.3.21 SSP_SetDMACtrl

送受信 FIFO の DMA 制御

関数のプロトタイプ宣言:

void
SSP_SetDMACtrl(TSB_SSP_TypeDef * ***SSPx***,
SSP_Direction ***Direction***,
FunctionalState ***NewState***)

引数:

SSPx: SSP チャネルを指定します。

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

16.2.4 データ構造

16.2.4.1 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 0～16 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

16.2.4.2 SSP_INTState

メンバ:

uint32_t
All: 割り込み要因

Bit

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***:	28	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。

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

本デバイスは、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/TX04_Periph_Driver/src/tmpm461_tmrb.c
/Libraries/TX04_Periph_Driver/inc/tmpm461_tmrb.h

17.2 API 関数

17.2.1 関数一覧

- void TMRB_Enable(TSB_TB_TypeDef * ***TBx***)
- void TMRB_Disable(TSB_TB_TypeDef * ***TBx***)
- void TMRB_SetRunState(TSB_TB_TypeDef * ***TBx***, uint32_t ***Cmd***)
- void TMRB_Init(TSB_TB_TypeDef * ***TBx,*** TMRB_InitTypeDef * ***InitStruct***)
- void TMRB_SetCaptureTiming(TSB_TB_TypeDef * ***TBx***, uint32_t ***CaptureTiming***)
- void TMRB_SetFlipFlop(TSB_TB_TypeDef * ***TBx***,
  TMRB_FFOutputTypeDef * ***FFStruct***)
- TMRB_INTFactor TMRB_GetINTFactor(TSB_TB_TypeDef * ***TBx***)
- void TMRB_SetINTMask(TSB_TB_TypeDef * ***TBx***, uint32_t ***INTMask***)
- void TMRB_ChangeLeadingTiming(TSB_TB_TypeDef * ***TBx***,
  uint32_t ***LeadingTiming***)
- void TMRB_ChangeTrailingTiming(TSB_TB_TypeDef * ***TBx***,
  uint32_t ***TrailingTiming***)
- uint16_t TMRB_GetUpCntValue(TSB_TB_TypeDef * ***TBx***)

- uint16_t TMRB_GetCaptureValue(TSB_TB_TypeDef * ***TBx***, uint8_t ***CapReg***)
- void TMRB_ExecuteSWCapture(TSB_TB_TypeDef * ***TBx***)
- void TMRB_SetIdleMode(TSB_TB_TypeDef * ***TBx***, FunctionalState ***NewState***)
- void TMRB_SetSyncMode(TSB_TB_TypeDef * ***TBx***, FunctionalState ***NewState***)
- void TMRB_SetDoubleBuf(TSB_TB_TypeDef * ***TBx***, FunctionalState ***NewState***, uint8_t ***WriteRegMode***)
- void TMRB_SetExtStartTrg(TSB_TB_TypeDef * ***TBx***, FunctionalState ***NewState***, uint8_t ***TrgMode***)
- void TMRB_SetClkInCoreHalt(TSB_TB_TypeDef * ***TBx***, uint8_t ***ClkState***)

17.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています:

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

17.2.3 関数仕様

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

TSB_TB0, TSB_TB1, 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, TSB_TB_MPT0, TSB_TB_MPT1.

17.2.3.1 TMRB_Enable

TMRB 動作の許可

関数のプロトタイプ宣言:

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

引数:

***TBx*:** TMRB チャネルを指定します。

機能:

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

戻り値:

なし

17.2.3.2 TMRB_Disable

TMRB 動作の禁止

関数のプロトタイプ宣言:

void
TMRB_Disable(TSB_TB_TypeDef* ***TBx***)

引数:

***TBx*:** TMRB チャネルを指定します。

機能:

TMRB 動作を無効にします。

戻り値:

なし

17.2.3.3 TMRB_SetRunState

カウンタ動作の設定

関数のプロトタイプ宣言:

void
TMRB_SetRunState(TSB_TB_TypeDef* ***TBx***,
uint32_t ***Cmd***)

引数:

***TBx*:** TMRB チャネルを指定します。

Cmd: カウンタ動作を選択します。

- **TMRB_RUN:** カウント
- **TMRB_STOP:** 停止&クリア

機能:

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

戻り値:

なし

17.2.3.4 TMRB_Init

TMRB チャネルの初期化

関数のプロトタイプ宣言:

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

引数:

***TBx*:** TMRB チャネルを指定します。

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

機能:

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

戻り値:

なし

17.2.3.5 TMRB_SetCaptureTiming

キャプチャタイミングの設定

関数のプロトタイプ宣言:

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

引数:

***TBx*:** TMRB チャネルを指定します。

CaptureTiming: キャプチャタイミングを選択します。
TBx = TSB_TB_MPT0 または TSB_TB_MPT1 の場合:

- **MPT_DISABLE_CAPTURE**: キャプチャ禁止
- **MPT_CAPTURE_IN_RISING**: MTxTBIN 端子入力の立ち上がりでキャプチャレジスタ 0（MTxCP0）にカウント値を取り込みます

- **MPT_CAPTURE_IN_RISING_FALLING**: MTxTBIN 端子入力の立ち上がりでキャプチャレジスタ 0（MTxCP0）にカウント値を取り込み、MTxTBIN 端子入力の立ち下がりでキャプチャレジスタ 1（MTxCP1）にカウント値を取り込みます。

TBx = TSB_TB0 ~ TSB_TBF の場合:

- **TMRB_DISABLE_CAPTURE**: キャプチャ禁止
- **TMRB_CAPTURE_TBIN0_TBIN1_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。(**TSB_TB4 ~ TSB_TBF のみ TMRB_CAPTURE_TBIN0_TBIN1_RISING を選択可能です**)
- **TMRB_CAPTURE_TBIN0_RISING_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_TBFF0_EDGE**: TBxFF0 の立ち上がりでキャプチャレジスタ 0 (TBxCP0)にカウント値を取り込み、TBxFF0 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_CLEAR_TBIN1_RISING**: TBxIN1↑でアップカウンタをクリアします。(**TSB_TB4 ~ TSB_TBF のみ TMRB_CLEAR_TBIN1_RISING を選択可能です**)

TMRB_CAPTURE_TBIN0_RISING_CLEAR_TBIN1_RISING: TBxIN0↑でキャプチャレジスタ 0(TBxCP0)にカウント値を取り込み、TBxIN1↑でアップカウンタをクリアします。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が同時だった場合、キャプチャが実行された後にアップカウンタのクリアが実行されます。(**TSB_TB4~TSB_TBF のみ TMRB_CAPTURE_TBIN0_RISING_CLEAR_TBIN1_RISING を選択可能です**)

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

17.2.3.6 TMRB_SetFlipFlop

フリップフロップ機能の設定

関数のプロトタイプ宣言:

```
void
TMRB_SetFlipFlop(TSB_TB_TypeDef* TBx,
                 TMRB_FFOutputTypeDef* FFStruct)
```

引数:

***TBx*:** TMRB チャネルを指定します。

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

機能:

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

戻り値:

なし

17.2.3.7 TMRB_GetINTFactor

割り込み要因の取得

関数のプロトタイプ宣言:

TMRB_INTFactor
TMRB_GetINTFactor(TSB_TB_TypeDef* ***TBx***)

引数:

***TBx*:** TMRB チャネルを指定します。

機能:

割り込み要因を取得します。

戻り値:

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

補足:

異なる割り込み要因を処理する場合は、以下のように記述してください。

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

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

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

17.2.3.8 TMRB_SetINTMask

割り込みマスクの設定

関数のプロトタイプ宣言:

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

引数:

***TBx*:** TMRB チャネルを指定します。

***INTMask*:** マスクする割り込みを選択します。

- **TMRB_MASK_MATCH_TRAILING_INT**: 一致 (TBxRG0) 割り込み
- **TMRB_MASK_MATCH_LEADING_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
- **TMRB_MASK_MATCH_LEADING_INT | TMRB_MASK_MATCH_TRAILING_INT**: 一致 (TBxRG0) 割り込み、または一致 (TBxRG1) 割り込み
- **TMRB_MASK_MATCH_LEADING_INT | TMRB_MASK_OVERFLOW_INT**: 一致 (TBxRG1) 割り込み、またはオーバーフロー割り込み
- **TMRB_MASK_MATCH_TRAILING_INT | TMRB_MASK_OVERFLOW_INT**: 一致 (TBxRG0) 割り込み、またはオーバーフロー割り込み
- **TMRB_MASK_MATCH_LEADING_INT | TMRB_MASK_MATCH_TRAILING_INT | TMRB_MASK_OVERFLOW_INT**: 一致 (TBxRG1) 割り込み、または一致 (TBxRG0) 割り込み、またはオーバーフロー割り込み

機能:

TMRB_MASK_MATCH_TRAILING_INT 選択時、アップカウンタ値と TBxRG1 が一致した場合、割り込みは発生しません。
TMRB_MASK_MATCH_LEADING_INT 選択時、アップカウンタ値と TBxRG0 が一致した場合、割り込みは発生しません。
TMRB_MASK_OVERFLOW_INT 選択時、オーバ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。
TMRB_MASK_MATCH_TRAILING_INT と **TMRB_MASK_MATCH_LEADING_INT** と **TMRB_MASK_OVERFLOW_INT** 選択時、どの条件が成立しても割り込みは発生しません。

戻り値:

なし

17.2.3.9 TMRB_ChangeLeadingTiming

デューティの設定

関数のプロトタイプ宣言:

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

引数:

***TBx*:** TMRB チャネルを指定します。

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

機能:

デューティを設定します。実際のデューティのインターバルは、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。

17.2.3.10 TMRB_ChangeTrailingTiming

周期の設定

関数のプロトタイプ宣言:

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

引数:

***TBx*:** TMRB チャネルを指定します。

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

機能:

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

戻り値:

なし

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。

17.2.3.11 TMRB_GetUpCntValue

アップカウンタ値の読み込み

関数のプロトタイプ宣言:

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

引数:

***TBx*:** TMRB チャネルを指定します。

機能:

アップカウンタ値の読み込みを行います。

戻り値:

アップカウンタ値

17.2.3.12 TMRB_GetCaptureValue

キャプチャレジスタの読み込み

関数のプロトタイプ宣言:

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

引数:

***TBx*:** TMRB チャネルを指定します。

CapReg: キャプチャレジスタを選択します。

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

機能:

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

戻り値:

キャプチャレジスタの値

17.2.3.13 TMRB_ExecuteSWCapture

ソフトウェアキャプチャの実行

関数のプロトタイプ宣言:

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

引数:

***TBx*:** TMRB チャネルを指定します。

機能:

キャプチャレジスタ 0 (TBxCP0)にカウント値を取り込みます。

戻り値:

なし

17.2.3.14 TMRB_SetIdleMode

IDLE 時の動作設定

関数のプロトタイプ宣言:

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

引数:

***TBx*:** TMRB チャネルを指定します。

NewState: IDLE 時の動作を指定します。

- **ENABLE:** 動作
- **DISABLE:** 停止

機能:

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

戻り値:

なし

17.2.3.15 TMRB_SetSyncMode

同期モードの切り替え

関数のプロトタイプ宣言:

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

引数:

***TBx*:** TMRB チャネルを以下から選択します。

TSB_TB1, TSB_TB2, TSB_TB3, TSB_TB5, TSB_TB6, TSB_TB7, TSB_TB9, TSB_TBA, TSB_TBB

***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 のカウントを開始する前に、**TMRB_SetRunState()** によって TMRB1 ～ TMRB3、TMRB5 ～ TMRB7、TMRB9 ～ TMRBB をスタートしてください。

17.2.3.16 TMRB_SetDoubleBuf

ダブルバッファ動作の制御

関数のプロトタイプ宣言:

void
TMRB_SetDoubleBuf(TSB_TB_TypeDef* ***TBx***,
FunctionalState ***NewState***,
uint8_t ***WriteRegMode***)

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。

***WriteRegMode*:** ダブルバッファがイネーブルの場合のタイマレジスタ 0 および 1 への書き込みタイミングを指定します。
- **TMRB_WRITE_REG_SEPARATE:** タイマレジスタ 0 および 1 は個別に書き込みが可能です。一方のレジスタのみ書き込み準備が完了した場合も同様です。
- **TMRB_WRITE_REG_SIMULTANEOUS:** 両方のレジスタの書き込み準備が完了していない場合、タイマレジスタ 0 および 1 への書き込みはできません。

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

補足:

WriteRegMode は、TMRB0~TMRB7 に対し無効です。そのため、本 API がこれらのチャネルに使用される場合は、***WriteRegMode*** を 0 に設定することを推奨します。

17.2.3.17 TMRB_SetExtStartTrg

外部トリガの設定

関数のプロトタイプ宣言:

```
void
TMRB_SetExtStartTrg (TSB_TB_TypeDef* TBx,
                     FunctionalState NewState,
                     uint8_t TrgMode)
```

引数:

***TBx*:** TMRB チャネルを指定します。

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

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

機能:

外部トリガによる変換開始の有無とアクティブエッジの設定を行います。

戻り値:

なし

17.2.3.18 TMRB_SetClkInCoreHalt

デバッグ HALT 中のクロック動作

関数のプロトタイプ宣言:

void
TMRB_SetClkInCoreHalt (TSB_TB_TypeDef* ***TBx***, uint8_t ***ClkState***)

引数:

***TBx*:** TMRB チャネルを指定します。

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

機能:

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

戻り値:

なし

17.2.4 データ構造

17.2.4.1 TMRB_InitTypeDef

メンバ:

uint32_t
Mode s 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
- **TMRB_CLK_DIV_64**: fperiph / 64 (TMRB0~TMRB7 のみ)
- **TMRB_CLK_DIV_128**: fperiph / 128 (TMRB0~TMRB7 のみ)
- **TMRB_CLK_DIV_256**: fperiph / 256 (TMRB0~TMRB7 のみ)
- **TMRB_CLK_DIV_512**: fperiph / 512 (TMRB0~TMRB7 のみ)

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。(TMRB0~TMRBF のみ)
- **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。(TMRB0~TMRBF のみ)
- **MPT_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。(MPT0~MPT1 のみ)
- **MPT_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。(MPT0~MPT1 のみ)

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。

17.2.4.2 TMRB_FFOutputTypeDef

メンバ:

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

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

17.2.4.3 TMRB_INTFactor

メンバ:

uint32_t
All: TMRB 割り込み要因

Bit

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

uint32_t
MatchTrailingTiming : 1周期との一致検出

uint32_t
OverFlow : 1 オーバーフロー

uint32_t
Reserverd : 29 -

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/TX04_Periph_Driver/src/tmpm461_uart.c
/Libraries/TX04_Periph_Driver/inc/tmpm461_uart.h

18.2 API 関数

18.2.1 関数一覧

- void UART_Enable(TSB_SC_TypeDef* ***UARTx***)
- void UART_Disable(TSB_SC_TypeDef* ***UARTx***)
- WorkState UART_GetBufState(TSB_SC_TypeDef* ***UARTx***, uint8_t ***Direction***)
- void UART_SWReset(TSB_SC_TypeDef* ***UARTx***)
- void UART_Init(TSB_SC_TypeDef* ***UARTx***, UART_InitTypeDef* ***InitStruct***)
- uint32_t UART_GetRxData(TSB_SC_TypeDef* ***UARTx***)
- void UART_SetTxData(TSB_SC_TypeDef* ***UARTx***, uint32_t ***Data***)
- void UART_DefaultConfig(TSB_SC_TypeDef* ***UARTx***)
- UART_Err UART_GetErrState(TSB_SC_TypeDef* ***UARTx***)
- void UART_SetWakeUpFunc(TSB_SC_TypeDef* ***UARTx***,
  FunctionalState ***NewState***)
- void UART_SetIdleMode(TSB_SC_TypeDef* ***UARTx***, FunctionalState ***NewState***)
- void UART_FIFOConfig(TSB_SC_TypeDef * ***UARTx***, FunctionalState NewState);
- void UART_SetFIFOTransferMode(TSB_SC_TypeDef * ***UARTx,***
  uint32_t ***TransferMode***);
- void UART_TRxAutoDisable(TSB_SC_TypeDef * ***UARTx***,
  UART_TRxAutoDisable ***TRxAutoDisable***);
- void UART_RxFIFOINTCtrl(TSB_SC_TypeDef * ***UARTx***, FunctionalState ***NewState***);
- void UART_TxFIFOINTCtrl(TSB_SC_TypeDef * ***UARTx***, FunctionalState ***NewState***);
- void UART_RxFIFOByteSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***BytesUsed***);
- void UART_RxFIFOFillLevel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***RxFIFOLevel***);
- void UART_RxFIFOINTSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***RxINTCondition***);
- void UART_RxFIFOClear(TSB_SC_TypeDef * ***UARTx***);
- void UART_TxFIFOFillLevel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***TxFIFOLevel***);
- void UART_TxFIFOINTSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***TxINTCondition***);
- void UART_TxFIFOClear(TSB_SC_TypeDef * ***UARTx***);
- void UART_TxBufferClear(TSB_SC_TypeDef * ***UARTx***);
- uint32_t UART_GetRxFIFOFillLevelStatus(TSB_SC_TypeDef * ***UARTx***);
- uint32_t UART_GetRxFIFOOverRunStatus(TSB_SC_TypeDef * ***UARTx***);
- uint32_t UART_GetTxFIFOFillLevelStatus(TSB_SC_TypeDef * ***UARTx***);

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

18.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています:

1) 初期化と設定:
   UART_Enable(), UART_Disable(),UART_SetInputClock(),UART_Init(),
   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() and
   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_TxBufferClear(), UART_GetRxFIFOFillLevelStatus(),
   UART_GetRxFIFOOverRunStatus(), UART_GetTxFIFOFillLevelStatus(),
   UART_GetTxFIFOUnderRunStatus()

18.2.3 関数仕様

補足: 引数に記述している“TSB_SC_TypeDef* ***UARTx***” は、以下から選択してください。
UART0, UART1, UART2, UART3, UART4, UART5

引数に記述している“TSB_SC_TypeDef* ***SIOx***” は、以下から選択してください。
SIO0, SIO1, SIO2, SIO3, SIO4, SIO5

18.2.3.1 UART_Enable

UART 動作の許可

関数のプロトタイプ宣言:

void
UART_Enable(TSB_SC_TypeDef* ***UARTx***)

引数:

UARTx: UART チャネルを指定します。

機能:

UART 動作を許可します。

戻り値:

なし

18.2.3.2 UART_Disable

UART 動作の禁止

関数のプロトタイプ宣言:

void
UART_Disable(TSB_SC_TypeDef* ***UARTx***)

引数:

UARTx: UART チャネルを指定します。

機能:

UART 動作を禁止します。

戻り値:

なし

18.2.3.3 UART_GetBufState

送受信バッファ状態の読み込み

関数のプロトタイプ宣言:

WorkState
UART_GetBufState(TSB_SC_TypeDef* ***UARTx***,
uint8_t ***Direction***)

引数:

UARTx: UART チャネルを指定します。

Direction: 送信/受信を選択します。

- **UART_RX**: 受信
- **UART_TX**: 送信

機能:

Direction が **UART_RX** の場合、以下の受信バッファの状態を返します。
DONE:受信データはバッファに保存済み
BUSY:データ受信中

Direction が **UART_TX** の場合、以下の送信バッファの状態を返します。
DONE:バッファ中のデータは送信済み
BUSY:データ送信中

戻り値:

- **DONE**: バッファリード/ライト可能状態
- **BUSY**: 送受信中

18.2.3.4 UART_SWReset

ソフトウェアリセット

関数のプロトタイプ宣言:

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

引数:

UARTx: UART チャネルを指定します。

機能:

ソフトウェアリセットが発生します。

戻り値:

なし

18.2.3.5 UART_Init

UART チャネルの初期化

関数のプロトタイプ宣言:

void
UART_Init(TSB_SC_TypeDef* ***UARTx***,
UART_InitTypeDef* ***InitStruct***)

引数:

UARTx: UART チャネルを指定します。

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

機能:

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

戻り値:

なし

18.2.3.6 UART_GetRxData

受信データの読み込み

関数のプロトタイプ宣言:

```
uint32_t
UART_GetRxData(TSB_SC_TypeDef* UARTx)
```

引数:

UARTx: UART チャネルを指定します。

機能:

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

戻り値:

受信データです。データ範囲は 0x00～0x1FF です

18.2.3.7 UART_SetTxData

送信データの設定

関数のプロトタイプ宣言:

```
void
UART_SetTxData(TSB_SC_TypeDef* UARTx,
               uint32_t Data)
```

引数:

UARTx: UART チャネルを指定します。

Data: 送信データ(7 ビット、8 ビット、9 ビット)

機能:

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

戻り値:

なし

18.2.3.8 UART_DefaultConfig

デフォルト構成での初期化

関数のプロトタイプ宣言:

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

引数:

UARTx: UART チャネルを指定します。

機能:

以下の構成で初期化します:

ボーレート:	115200 bps
データ長:	8 ビット
ストップビット:	1 ビット
パリティ:	なし
フローコントロール:	なし

送受信有効。ボーレートジェネレータはソースクロックとして使用。

戻り値:

なし

18.2.3.9 UART_GetErrState

転送エラーフラグの読み出し

関数のプロトタイプ宣言:

UART_Err
UART_GetErrState(TSB_SC_TypeDef* ***UARTx***)

引数:

UARTx: UART チャネルを指定します。

機能:

転送エラーフラグを読み出します。

戻り値:

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

18.2.3.10 UART_SetWakeUpFunc

9 ビットモード時のウェイクアップ機能の設定

関数のプロトタイプ宣言:

```
void
UART_SetWakeUpFunc(TSB_SC_TypeDef* UARTx,
                   FunctionalState NewState)
```

引数:

UARTx: UART チャネルを指定します。

NewState:ウェイクアップ機能の有効/無効を選択します。
- **ENABLE**: 有効
- **DISABLE**: 無効

機能:

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

戻り値:

なし

18.2.3.11 UART_SetInputClock

プリスケーラの入力クロック選択

関数のプロトタイプ宣言:

```
void
UART_SetInputClock (TSB_SC_TypeDef * UARTx,
                    uint32_t clock)
```

引数:

UARTx: UART チャネルを指定します。

Clock: プリスケーラの入力クロックを選択します。
- **0 :**PhiT0/2
- **1 :**PhiT0

機能:

プリスケーラの入力クロックを選択します。

戻り値:

なし

18.2.3.12 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

18.2.3.13 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

18.2.3.14 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

18.2.3.15 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

18.2.3.16 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

18.2.3.17 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

18.2.3.18 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

18.2.3.19 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

18.2.3.20 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

18.2.3.21 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

18.2.3.22 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

18.2.3.23 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

18.2.3.24 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

18.2.3.25 UART_TxBufferClear

送信バッファクリア

関数のプロトタイプ宣言:

void
UART_TxBufferClear (TSB_SC_TypeDef* ***UARTx***);

引数:

UARTx: UART チャネルを指定します。

機能:

送信バッファをクリアします。

戻り値:

なし

18.2.3.26 UART_GetRxFIFOFillLevelStatus

受信 FIFO の fill レベルの取得

関数のプロトタイプ宣言:

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

引数:

UARTx: UART チャネルを指定します。

機能:

受信 FIFO の fill レベルを取得します。

戻り値:

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

18.2.3.27 UART_GetRxFIFOOverRunStatus

受信 FIFO オーバーラン状態の取得

関数のプロトタイプ宣言:

uint32_t
UART_ GetRxFIFOOverRunStatus (TSB_SC_TypeDef* ***UARTx***);

引数:

UARTx: UART チャネルを指定します。

機能:

受信 FIFO オーバーラン状態を取得します。

戻り値:

- **UART_RXFIFO_OVERRUN:** オーバーラン発生
- **0:** オーバーランは発生していない

18.2.3.28 UART_GetTxFIFOFillLevelStatus

送信 FIFO の fill レベルの取得

関数のプロトタイプ宣言:

uint32_t
UART_GetTxFIFOFillLevelStatus (TSB_SC_TypeDef* ***UARTx***);

引数:

UARTx: UART チャネルを指定します。

機能:

送信 FIFO の fill レベルの取得

戻り値:

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

18.2.3.29 UART_GetTxFIFOUnderRunStatus

送信 FIFO アンダーラン状態の取得

関数のプロトタイプ宣言:

```
uint32_t
UART_ GetTxFIFOUnderRunStatus (TSB_SC_TypeDef* UARTx);
```

引数:

UARTx: UART チャネルを指定します。

機能:

送信 FIFO アンダーラン状態を取得します。

戻り値:

- **UART_TXFIFO_UNDERRUN**: アンダーラン発生
- **0**: アンダーランは発生していない

18.2.3.30 SIO_SetInputClock

プリスケーラの入力クロック選択

関数のプロトタイプ宣言:

```
void
SIO_SetInputClock (TSB_SC_TypeDef * SIOx,
                   uint32_t Clock)
```

引数:

SIOx: SIO チャネルを指定します。

Clock: プリスケーラの入力クロックを選択します。

- **SIO_CLOCK_T0_HALF:** PhiT0/2
- **SIO_CLOCK_T0:** PhiT0

機能:

プリスケーラの入力クロックを選択します。

戻り値:

なし

18.2.3.31 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

18.2.3.32 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

18.2.3.33 SIO_GetRxData

受信用バッファ

関数のプロトタイプ宣言:

uint32_t
SIO_GetRxData(TSB_SC_TypeDef* ***SIOx***)

引数:

SIOx: SIO チャネルを指定します。

機能:

受信用バッファを取得します。

戻り値:

受信用バッファ(値の範囲は 0x00 ~ 0xFF です)

18.2.3.34 SIO_SetTxData

送信用バッファ

関数のプロトタイプ宣言:

void
SIO_SetTxData(TSB_SC_TypeDef* ***SIOx***,
uint8_t ***Data***)

引数:

SIOx: SIO チャネルを指定します。

Data: 送信用バッファ

機能:

送信用バッファを指定します。

戻り値:

なし

18.2.3.35 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

18.2.4 データ構造

18.2.4.1 UART_InitTypeDef

メンバ:

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

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

uint32_t

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

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

uint32_t
Mode: 転送モードを選択します。送受信の場合は、送信と受信を OR 演算子によって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

18.2.4.2 SIO_InitTypeDef

メンバ:

uint32_t
InputClkEdge: 入力クロックエッジを選択します。
- **SIO_SCLKS_TXDF_RXDR**: 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
TIDLE: 最終ビット出力後の SCxTXD 端子の状態を選択します。
- **SIO_TIDLE_LOW**: "Low"出力保持
- **SIO_TIDLE_HIGH**: "High"出力保持
- **SIO_TIDLE_LAST**: 最終ビット保持

uint32_t
TXDEMP: クロック入力モード時、アンダーランエラーが発生したときの SCxTXD 端子の状態を選択します。
- **SIO_TXDEMP_LOW**: "Low"出力
- **SIO_TXDEMP_HIGH**: "High"出力

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

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

uint32_t

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

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

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

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

uint32_t
BaudRateClock: ボーレートジェネレータ入力クロックを選択します。
- **SIO_BR_CLOCK_TS0**: φTS0
- **SIO_BR_CLOCK_TS2**: φTS2
- **SIO_BR_CLOCK_TS8**: φTS8
- **SIO_BR_CLOCK_TS32**: φTS32

uint32_t
Divider: 分周値"N"を選択します。
- **SIO_BR_DIVIDER_16**: 16 分周
- **SIO_BR_DIVIDER_1**: 1 分周
- **SIO_BR_DIVIDER_2**: 2 分周
- **SIO_BR_DIVIDER_3**: 3 分周
- **SIO_BR_DIVIDER_4**: 4 分周
- **SIO_BR_DIVIDER_5**: 5 分周
- **SIO_BR_DIVIDER_6**: 6 分周
- **SIO_BR_DIVIDER_7**: 7 分周
- **SIO_BR_DIVIDER_8**: 8 分周
- **SIO_BR_DIVIDER_9**: 9 分周
- **SIO_BR_DIVIDER_10**: 10 分周
- **SIO_BR_DIVIDER_11**: 11 分周
- **SIO_BR_DIVIDER_12**: 12 分周
- **SIO_BR_DIVIDER_13**: 13 分周
- **SIO_BR_DIVIDER_14**: 14 分周
- **SIO_BR_DIVIDER_15**: 15 分周

*: 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。

19. uDMAC

19.1 概要

本デバイスには、マイクロ DMAC (uDMAC)モジュールを内蔵しています。

主な機能は以下の通りです。

Functions	Features		Descriptions
Channels	32 channels		-
Start trigger	Start by Hardware		DMA requests from peripheral functions
	Start by Software		Specified by DMAxChnlSwRequest register
Priority	Between channels	ch0 (high priority) > … > ch31 (high priority) > ch0 (Normal priority) > … > ch31 (Normal priority)	High-priority can be configured by DMAxChnlPriority-Set register
Transfer data size	8/16/32bit		Can be specified source and destination independently
The number of transfer	1 to 4095 times		-
Address	Transfer source address	Increment / fixed	Transfer source address and destination address can be selected to increment or fixed.
	transfer destination address	Increment / fixed	
Endian	Little Endian		-
Transfer type	Peripheral (register) → memory Memory → peripheral (register) Memory → memory		If you select memory to memory, hardware start for DMA start up is not supported. Refer to the DMACxConfiguration register for more information.
Interrupt function	Transfer end interrupt Error interrupt		Output for each unit
Transfer mode	Basic mode Automatic request mode Ping-pong mode Memory scatter / gather mode Peripheral scatter / gather mode		-

µDMAC API は、MCU の µDMAC モジュールを使用するための機能セットを提供します。本 API には、µDMAC 転送タイプセット、チャネルセット、マスクセット、初期/代替データエリアセット、チャネル優先、初期化データ設定などが含まれます。

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/TX04_Periph_Driver\src\tmpm461_udmac.c
/Libraries/TX04_Periph_Driver\inc\tmpm461_udmac.h

***補足:** 本ドキュメントでは、DMAC は µDMAC を意味します。

19.2 API 関数

19.2.1 関数一覧

◆ FunctionalState DMAC_GetDMACState(TSB_DMA_TypeDef * ***DMACx***)

- void DMAC_Enable(TSB_DMA_TypeDef * ***DMACx***)
- void DMAC_Disable(TSB_DMA_TypeDef * ***DMACx***)
- void DMAC_SetPrimaryBaseAddr(TSB_DMA_TypeDef * ***DMACx***, uint32_t ***Addr***)
- uint32_t DMAC_GetBaseAddr(TSB_DMA_TypeDef * ***DMACx***,
  DMAC_PrimaryAlt ***PriAlt***)
- void DMAC_SetSWReq(TSB_DMA_TypeDef * ***DMACx*** ,
  uint8_t ***Channel***)
- void DMACA_SetTransferType(DMACA_Channel ***Channel***,
  DMAC_TransferType ***Type***)
- DMAC_TransferType DMACA_GetTransferType(DMACA_Channel ***Channel***)
- void DMAC_SetMask(TSB_DMA_TypeDef * ***DMACx*** ,
  uint8_t ***Channel*** ,
  FunctionalState ***NewState***)
- FunctionalState DMAC_GetMask(TSB_DMA_TypeDef * ***DMACx*** ,
  uint8_t ***Channel***)
- void DMAC_SetChannel(TSB_DMA_TypeDef * ***DMACx*** ,
  uint8_t ***Channel*** ,
  FunctionalState ***NewState***)
- FunctionalState DMAC_GetChannelState(TSB_DMA_TypeDef * ***DMACx*** ,
  uint8_t ***Channel***)
- void DMAC_SetPrimaryAlt(TSB_DMA_TypeDef * ***DMACx*** ,
  uint8_t ***Channel***
  DMAC_PrimaryAlt ***PriAlt***)
- DMAC_PrimaryAlt DMAC_GetPrimaryAlt(TSB_DMA_TypeDef * ***DMACx*** ,
  uint8_t ***Channel***)
- void DMAC_SetChannelPriority(TSB_DMA_TypeDef * ***DMACx*** ,
  uint8_t ***Channel*** ,
  DMAC_Priority ***Priority***)
- DMAC_Priority DMAC_GetChannelPriority(TSB_DMA_TypeDef * ***DMACx*** ,
  uint8_t ***Channel***)
- void DMAC_ClearBusErr(TSB_DMA_TypeDef * ***DMACx***)
- Result DMAC_GetBusErrState(TSB_DMA_TypeDef * ***DMACx***)
- void DMAC_FillInitData(TSB_DMA_TypeDef * ***DMACx*** ,
  uint8_t ***Channel*** ,
  DMAC_InitTypeDef * ***InitStruct***)
- DMACA_Flag DMACA_GetINTFlag(void)
- DMACB_Flag DMACB_GetINTFlag(void)
- DMACC_Flag DMACC_GetINTFlag(void)

19.2.2 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) 設定:
   DMACA_SetTransferType(), DMACA_GetTransferType(), DMAC_SetMask(), DMAC_GetMask(), DMAC_SetChannel(), DMAC_GetChannelState(), DMAC_SetPrimaryAlt(), DMAC_GetPrimaryAlt(), DMAC_SetChannelPriority(), DMAC_GetChannelPriority()

2) 機能の許可/禁止::
DMAC_GetDMACState(), DMAC_Enable(), DMAC_Disable()
3) ソフトウェアトリガ制御:
DMAC_SetSWReq()
4) バスエラー監視:
DMAC_ClearBusErr(), DMAC_GetBusErrState()
5) 制御データエリア設定:
DMAC_FillInitData(), DMAC_SetPrimaryBaseAddr(), DMAC_GetBaseAddr()
6) DMA 要因の取得:
DMACA_GetINTFlag(), DMACB_GetINTFlag(), DMACC_GetINTFlag()

19.2.3 関数仕様

補足: 引数に記述している“DMACx“および “Channel“は、特に断りのない限り、以下から選択してください。

DMACx:ユニット選択です。

- **DMAC_UNIT_A:** DMAC ユニット A
- **DMAC_UNIT_B:** DMAC ユニット B
- **DMAC_UNIT_C:** DMAC ユニット C

Channel: チャネル選択です。

[DMAC_UNIT_A の場合]
- **DMACA_ADC_COMPLETION :** ADC 変換終了
- **DMACA_SSP0_RX :** SSP0 受信
- **DMACA_SSP0_TX :** SSP0 送信
- **DMACA_SSP1_RX :** SSP1 受信
- **DMACA_SSP1_TX :** SSP1 送信
- **DMACA_SSP2_RX :** SSP2 受信
- **DMACA_SSP2_TX :** SSP2 送信
- **DMACA_UART0_RX :** UART0 受信
- **DMACA_UART0_TX :** UART0 送信
- **DMACA_UART1_RX :** UART1 受信
- **DMACA_UART1_TX :** UART1 送信
- **DMACA_I2C0_TX_RX :** I2C0 送受信
- **DMACA_I2C1_TX_RX:** I2C1 送受信
- **DMACA_I2C2_TX_RX :** I2C2 送受信
- **DMACA_I2C3_TX_RX :** I2C3 送受信
- **DMACA_I2C4_TX_RX :** I2C4 送受信
- **DMACA_DMAREQA :** DMA ユニット A リクエストピン MAREQA

[DMAC_UNIT_B の場合]
- **DMACB_SIO0_UART0_RX :** SIO/UART0 受信
- **DMACB_SIO0_UART0_TX :** SIO/UART0 送信
- **DMACB_SIO1_UART1_RX :** SIO/UART1 受信
- **DMACB_SIO1_UART1_TX :** SIO/UART1 送信
- **DMACB_SIO2_UART2_RX :** SIO/UART2 受信
- **DMACB_SIO2_UART2_TX :** SIO/UART2 送信
- **DMACB_TMRB0_CMP_MATCH :** TMRB0 コンペアー致

- **DMACB_TMRB1_CMP_MATCH :** TMRB1 コンペアー致
- **DMACB_TMRB2_CMP_MATCH :** TMRB2 コンペアー致
- **DMACB_TMRB3_CMP_MATCH :** TMRB3 コンペアー致
- **DMACB_TMRB4_CMP_MATCH :** TMRB4 コンペアー致
- **DMACB_TMRB5_CMP_MATCH :** TMRB5 コンペアー致
- **DMACB_TMRB6_CMP_MATCH :** TMRB6 コンペアー致
- **DMACB_TMRB7_CMP_MATCH :** TMRB7 コンペアー致
- **DMACB_TMRBF_CMP_MATCH :** TMRBF コンペアー致
- **DMACB_TMRB0_INPUT_CAP0 :** TMRB0 インプットキャプチャ 0
- **DMACB_TMRB0_INPUT_CAP1 :** TMRB0 インプットキャプチャ 1
- **DMACB_TMRB1_INPUT_CAP0 :** TMRB1 インプットキャプチャ 0
- **DMACB_TMRB1_INPUT_CAP1 :** TMRB1 インプットキャプチャ 1
- **DMACB_TMRB2_INPUT_CAP0 :** TMRB2 インプットキャプチャ 0
- **DMACB_TMRB2_INPUT_CAP1 :** TMRB2 インプットキャプチャ 1
- **DMACB_TMRB3_INPUT_CAP0 :** TMRB3 インプットキャプチャ 0
- **DMACB_TMRB3_INPUT_CAP1 :** TMRB3 インプットキャプチャ 1
- **DMACB_TMRB4_INPUT_CAP0 :** TMRB4 インプットキャプチャ 0
- **DMACB_TMRB4_INPUT_CAP1 :** TMRB4 インプットキャプチャ 1
- **DMACB_TMRB5_INPUT_CAP0 :** TMRB5 インプットキャプチャ 0
- **DMACB_TMRB5_INPUT_CAP1 :** TMRB5 インプットキャプチャ 1
- **DMACB_DMAREQB :** DMA ユニット B リクエストピン DMAREQB

[DMAC_UNIT_C の場合]

- **DMACC_SIO3_UART3_RX :** SIO/UART3 受信
- **DMACC_SIO3_UART3_TX :** SIO/UART3 送信
- **DMACC_SIO4_UART4_RX :** SIO/UART4 受信
- **DMACC_SIO4_UART4_TX :** SIO/UART4 送信
- **DMACC_SIO5_UART5_RX :** SIO/UART5 受信
- **DMACC_SIO5_UART5_TX :** SIO/UART5 送信
- **DMACC_TMRB8_CMP_MATCH :** TMRB8 コンペアー致
- **DMACC_TMRB9_CMP_MATCH :** TMRB9 コンペアー致
- **DMACC_TMRBA_CMP_MATCH :** TMRBA コンペアー致
- **DMACC_TMRBB_CMP_MATCH :** TMRBB コンペアー致
- **DMACC_TMRBC_CMP_MATCH :** TMRBC コンペアー致
- **DMACC_TMRBD_CMP_MATCH :** TMRBD コンペアー致
- **DMACC_TMRBE_CMP_MATCH :** TMRBE コンペアー致
- **DMACC_TMRB7_CMP_MATCH :** TMRB7 コンペアー致
- **DMACC_TMRBF_CMP_MATCH :** TMRBF コンペアー致
- **DMACC_TMRB8_INPUT_CAP0 :** TMRB8 インプットキャプチャ 0
- **DMACC_TMRB8_INPUT_CAP1 :** TMRB8 インプットキャプチャ 1
- **DMACC_TMRB9_INPUT_CAP0 :** TMRB9 インプットキャプチャ 0
- **DMACC_TMRB9_INPUT_CAP1 :** TMRB9 インプットキャプチャ 1
- **DMACC_TMRBA_INPUT_CAP0 :** TMRBA インプットキャプチャ 0
- **DMACC_TMRBA_INPUT_CAP1 :** TMRBA インプットキャプチャ 1
- **DMACC_TMRBB_INPUT_CAP0 :** TMRBB インプットキャプチャ 0
- **DMACC_TMRBB_INPUT_CAP1 :** TMRBB インプットキャプチャ 1
- **DMACC_TMRBC_INPUT_CAP0 :** TMRBC インプットキャプチャ 0
- **DMACC_TMRBC_INPUT_CAP1 :** TMRBC インプットキャプチャ 1
- **DMACC_TMRBD_INPUT_CAP0 :** TMRBD インプットキャプチャ 0
- **DMACC_TMRBD_INPUT_CAP1 :** TMRBD インプットキャプチャ 1

- **DMACC_DMAREQC :** DMA ユニット C リクエストピン DMAREQC

19.2.3.1 DMAC_GetDMACState

DMAC ユニットの許可/禁止状態の読み出し

関数のプロトタイプ宣言:

FunctionalState
DMAC_GetDMACState(TSB_DMA_TypeDef * ***DMACx***)

引数:

DMACx: DMAC ユニットを選択します。

機能:

DMAC ユニットの許可/禁止状態を読み出します。

戻り値:

- **DISABLE:** 禁止状態
- **ENABLE:** 許可状態

19.2.3.2 DMAC_Enable

DMAC ユニット動作の許可

関数のプロトタイプ宣言:

void
DMAC_Enable(TSB_DMA_TypeDef * ***DMACx***)

引数:

DMACx: DMAC ユニットを選択します。

機能:

DMAC ユニット動作を許可します。

戻り値:

なし

19.2.3.3 DMAC_Disable

DMAC ユニット動作の禁止

関数のプロトタイプ宣言:

void
DMAC_Disable(TSB_DMA_TypeDef * ***DMACx***)

引数:

DMACx: DMAC ユニットを選択します。

機能:

DMAC ユニット動作を禁止します。

戻り値:

なし

19.2.3.4 DMAC_SetPrimaryBaseAddr

DMAC ユニットの一次データのベースアドレスの設定

関数のプロトタイプ宣言:

void
DMAC_SetPrimaryBaseAddr(TSB_DMA_TypeDef * ***DMACx***,
uint32_t ***Addr***)

引数:

DMACx: DMAC ユニットを選択します。

Addr: 一次データのベースアドレスを指定します。ビット 0 ～9 は 0 に設定してください。

機能:

DMAC ユニットの一次データのベースアドレスを設定します。

戻り値:

なし

19.2.3.5 DMAC_GetBaseAddr

DMAC ユニットの一次/代替ベースアドレスの取得

関数のプロトタイプ宣言:

uint32_t
DMAC_GetBaseAddr(TSB_DMA_TypeDef * ***DMACx***,
DMAC_PrimaryAlt ***PriAlt***)

引数:

DMACx: DMAC ユニットを選択します。

PriAlt: ベースアドレスタイプを選択します。
- **DMAC_PRIMARY:** 一次ベースアドレス
- **DMAC_ALTERNATE:** 代替ベースアドレス

機能:

DMAC ユニットの初期/代替ベースアドレスを取得します。

戻り値:

初期/代替データのベースアドレス

19.2.3.6 DMAC_SetSWReq

ソフトウェア転送要求の設定

関数のプロトタイプ宣言:

void
DMAC_SetSWReq(TSB_DMA_TypeDef * ***DMACx***,
DMAC_Channel ***Channel***)

引数:

DMACx: DMAC ユニットを選択します。

Channel: チャネルを選択します。

機能:

ソフトウェア転送要求を設定します。

戻り値:

なし

19.2.3.7 DMACA_SetTransferType

DMAC ユニット A の転送タイプの設定

関数のプロトタイプ宣言:

void
DMACA_SetTransferType(uint8_t ***Channel***,
DMAC_TransferType ***Type***)

引数:

Channel: ユニット A のチャネルを選択します。
Type が **DMAC_BURST** の場合:

- **DMACA_ADC_COMPLETION :** ADC 変換終了
- **DMACA_SSP0_RX :** SSP0 受信
- **DMACA_SSP0_TX :** SSP0 送信
- **DMACA_SSP1_RX :** SSP1 受信
- **DMACA_SSP1_TX :** SSP1 送信
- **DMACA_SSP2_RX :** SSP2 受信
- **DMACA_SSP2_TX :** SSP2 送信
- **DMACA_UART0_RX :** UART0 受信
- **DMACA_UART0_TX :** UART0 送信
- **DMACA_UART1_RX :** UART1 受信
- **DMACA_UART1_TX :** UART1 送信
- **DMACA_I2C0_TX_RX :** I2C0 送受信
- **DMACA_I2C1_TX_RX:** I2C1 送受信
- **DMACA_I2C2_TX_RX :** I2C2 送受信
- **DMACA_I2C3_TX_RX :** I2C3 送受信
- **DMACA_I2C4_TX_RX :** I2C4 送受信
- **DMACA_DMAREQA :** DMA ユニット A リクエストピン DMAREQA

Type が **DMAC_SINGLE** の場合**:**

- **DMACA_SSP0_RX :** SSP0 受信
- **DMACA_SSP0_TX :** SSP0 送信
- **DMACA_SSP1_RX :** SSP1 受信
- **DMACA_SSP1_TX :** SSP1 送信
- **DMACA_SSP2_RX :** SSP2 受信
- **DMACA_SSP2_TX :** SSP2 送信
- **DMACA_UART0_RX :** UART0 受信
- **DMACA_UART0_TX :** UART0 送信
- **DMACA_UART1_RX :** UART1 受信
- **DMACA_UART1_TX :** UART1 送信

Type: 転送タイプを選択します。

- **DMAC_BURST :** シングル転送が禁止され、バースト転送要求のみが有効になります。
- **DMAC_SINGLE :** シングル転送。

機能:

転送タイプを設定します。

戻り値:

なし

19.2.3.8 DMACA_GetTransferType

ユニット A の転送タイプの読み出し

関数のプロトタイプ宣言:

DMAC_TransferType
DMACA_GetTransferType(uint8_t ***Channel***)

引数:

Channel: ユニット A のチャネルを選択します。

- **DMACA_ADC_COMPLETION :** ADC 変換終了
- **DMACA_SSP0_RX :** SSP0 受信
- **DMACA_SSP0_TX :** SSP0 送信
- **DMACA_SSP1_RX :** SSP1 受信
- **DMACA_SSP1_TX :** SSP1 送信
- **DMACA_SSP2_RX :** SSP2 受信
- **DMACA_SSP2_TX :** SSP2 送信
- **DMACA_UART0_RX :** UART0 受信
- **DMACA_UART0_TX :** UART0 送信
- **DMACA_UART1_RX :** UART1 受信
- **DMACA_UART1_TX :** UART1 送信
- **DMACA_I2C0_TX_RX :** I2C0 送受信
- **DMACA_I2C1_TX_RX:** I2C1 送受信
- **DMACA_I2C2_TX_RX :** I2C2 送受信
- **DMACA_I2C3_TX_RX :** I2C3 送受信
- **DMACA_I2C4_TX_RX :** I2C4 送受信
- **DMACA_DMAREQA :** DMA ユニット A リクエストピン DMAREQA

機能:

転送タイプを読み出します。

戻り値:

転送タイプ:

- **DMAC_BURST :** シングル転送が禁止され、バースト転送要求のみが有効
- **DMAC_SINGLE :** シングル転送

19.2.3.9 DMAC_SetMask

DMA 要求のマスク設定/クリア制御

関数のプロトタイプ宣言:

void
DMAC_SetMask(TSB_DMA_TypeDef * ***DMACx***,
DMAC_Channel ***Channel***,
FunctionalState ***NewState***)

引数:

DMACx: DMAC ユニットを選択します。

Channel: チャネルを選択します。

NewState: DMA 要求のマスク設定/クリアを選択します。
- **ENABLE:** DMA 要求マスクのクリア。
- **DISABLE:** DMA 要求のマスク設定

機能:

DMA 要求のマスク設定/クリアを選択します。

戻り値:

なし

19.2.3.10 DMAC_GetMask

DMA 要求のマスク状態の読み出し

関数のプロトタイプ宣言:

```
FunctionalState
DMAC_GetMask(TSB_DMA_TypeDef * DMACx,
             DMAC_Channel Channel)
```

引数:

DMACx: DMAC ユニットを選択します。

Channel: チャネルを選択します。

機能:

DMA 要求のマスク状態を読み出します。

戻り値:

DMA 要求のマスク状態:
- **ENABLE:** DMA 要求のマスク設定なし
- **DISABLE:** DMA 要求のマスク設定あり

19.2.3.11 DMAC_SetChannel

DMA チャネルの有効/無効設定

関数のプロトタイプ宣言:

void
DMAC_SetChannel(TSB_DMA_TypeDef * ***DMACx***,
DMAC_Channel ***Channel***,
FunctionalState ***NewState***)

引数:

DMACx: DMAC ユニットを選択します。

Channel: チャネルを選択します。

NewState: DMA チャネルの有効/無効を選択します。

- **ENABLE:** 有効
- **DISABLE:** 無効

機能:

DMA チャネルの有効/無効を設定します。

戻り値:

なし

19.2.3.12 DMAC_GetChannelState

DMA チャネルの有効/無効状態の取得

関数のプロトタイプ宣言:

FunctionalState
DMAC_GetChannelState(TSB_DMA_TypeDef * ***DMACx***,
DMAC_Channel ***Channel***)

引数:

DMACx: DMAC ユニットを選択します。

Channel: チャネルを選択します。

機能:

DMA チャネルの有効/無効状態を取得します。

戻り値:

DMA チャネルの有効/無効状態:

- **ENABLE:** 有効
- **DISABLE:** 無効

19.2.3.13 DMAC_SetPrimaryAlt

一次データあるは代替データの選択

関数のプロトタイプ宣言:

```
void
DMAC_SetPrimaryAlt(TSB_DMA_TypeDef * DMACx,
                   DMAC_Channel Channel,
                   DMAC_PrimaryAlt PriAlt)
```

引数:

DMACx: DMAC ユニットを選択します。

Channel: チャネルを選択します。

PriAlt: 一次データあるいは代替データを選択します。

- **DMAC_PRIMARY:** 一次データ使用
- **DMAC_ALTERNATE:** 代替データ使用

機能:

一次データあるは代替データの使用有無を設定します。

戻り値:

なし

19.2.3.14 DMAC_GetPrimaryAlt

一次データあるは代替データの選択状態の読み出し

関数のプロトタイプ宣言:

```
DMAC_PrimaryAlt
DMAC_GetPrimaryAlt(TSB_DMA_TypeDef * DMACx,
                   DMAC_Channel Channel)
```

引数:

DMACx: DMAC ユニットを選択します。

Channel: チャネルを選択します。

機能:

一次データあるは代替データの選択状態を読み出します。

戻り値:

一次データあるは代替データの選択状態:

- **DMAC_PRIMARY:** 一次データ
- **DMAC_ALTERNATE:** 代替データ

19.2.3.15 DMAC_SetChannelPriority

優先度の設定。

関数のプロトタイプ宣言:

```
void
DMAC_SetChannelPriority(TSB_DMA_TypeDef * DMACx,
                        DMAC_Channel Channel,
                        DMAC_Priority Priority)
```

引数:

DMACx: DMAC ユニットを選択します。

Channel: チャネルを選択します。

Priority: 優先順位を選択します。

- **DMAC_PRIOTIRY_NORMAL:** 通常優先度
- **DMAC_PRIOTIRY_HIGH:** 高優先度

機能:

優先度を設定します。

戻り値:

なし

19.2.3.16 DMAC_GetChannelPriority

優先度の読み出し

関数のプロトタイプ宣言:

```
DMAC_Priority
DMAC_GetChannelPriority(TSB_DMA_TypeDef * DMACx,
                        DMAC_Channel Channel)
```

引数:

DMACx: DMAC ユニットを選択します。

Channel: チャネルを選択します。

機能:

優先度の設定状態を読み出します。

戻り値:

優先度の設定状態:
- **DMAC_PRIOTIRY_NORMAL:** 通常優先度
- **DMAC_PRIOTIRY_HIGH:** 高優先度

19.2.3.17 DMAC_ClearBusErr

バスエラー解除

関数のプロトタイプ宣言:

void
DMAC_ClearBusErr(TSB_DMA_TypeDef * ***DMACx***)

引数:

DMACx: DMAC ユニットを選択します。

機能:

バスエラーを解除します。

戻り値:

なし

19.2.3.18 DMAC_GetBusErrState

バスエラー状態の読み出し

関数のプロトタイプ宣言:

Result
DMAC_GetBusErrState(TSB_DMA_TypeDef * ***DMACx***)

引数:

DMACx: DMAC ユニットを選択します。

機能:

バスエラー状態を読み出します。

戻り値:

バスエラー状態:

- **SUCCESS:** バスエラーなし
- **ERROR:** バスエラー状態

19.2.3.19 DMAC_FillInitData

DMA 設定状態の読み出し

関数のプロトタイプ宣言:

```
void
DMAC_FillInitData(TSB_DMA_TypeDef * DMACx,
                  DMAC_Channel Channel,
                  DMAC_InitTypeDef * InitStruct)
```

引数:

DMACx: DMAC ユニットを選択します。

Channel: チャネルを選択します。

InitStruct: DMA 設定に関する構造体です。

機能:

DMA 設定状態を読み出します。

戻り値:

なし

19.2.3.20 DMACA_GetINTFlag

ユニット A の DMA 要因の取得

関数のプロトタイプ宣言:

```
DMACA_Flag
DMACA_GetINTFlag(void)
```

引数:

なし

機能:

ユニット A の DMA 要因を取得します。

戻り値:

ユニット A の DMA 要因の構造体 (詳細は、"データ構造"を参照してください)

19.2.3.21 DMACB_GetINTFlag

ユニット B の DMA 要因の取得

関数のプロトタイプ宣言:

```
DMACB_Flag
DMACB_GetINTFlag(void)
```

引数:

なし

機能:

ユニット B の DMA 要因を取得します。

戻り値:

ユニット B の DMA 要因の構造体 (詳細は、"データ構造"を参照してください)

19.2.3.22 DMACC_GetINTFlag

ユニット C の DMA 要因の取得

関数のプロトタイプ宣言:

```
DMACC_Flag
DMACC_GetINTFlag(void)
```

引数:

なし

機能:

ユニット C の DMA 要因を取得します。

戻り値:

ユニット C の DMA 要因の構造体 (詳細は、"データ構造"を参照してください)

19.2.4 データ構造

19.2.4.1 DMAC_InitTypeDef

メンバ:

uint32_t
SrcEndPointer: データ送信元最終アドレス

uint32_t
DstEndPointer: データ送信先最終アドレス

DMAC_CycleCtrl
Mode: 動作モード

- **DMAC_INVALID:** 無効。DMA は動作を停止します。
- **DMAC_BASIC:** 基本モード
- **DMAC_AUTOMATIC:** 自動要求モード
- **DMAC_PINGPONG:** ピンポンモード
- **DMAC_MEM_SCATTER_GATHER_PRI:** メモリスキャッターギャザーモード(一次データ)
- **DMAC_MEM_SCATTER_GATHER_ALT:** メモリスキャッターギャザーモード(代替データ)
- **DMAC_PERI_SCATTER_GATHER_PRI:** 周辺スキャッターギャザーモード(一次データ)
- **DMAC_PERI_SCATTER_GATHER_ALT:** 周辺スキャッターギャザーモード(代替データ)

DMAC_Next_UseBurst
NextUseBurst: 周辺スキャッターギャザーモードで代替データを用いた DMA 転送終了時に<chnl_useburst_set>ビットに"1"を設定するかどうかを指定します。

- **DMAC_NEXT_NOT_USE_BURST:** <chnl_useburst_set>の値を変更しない。
- **DMAC_NEXT_USE_BURST:** <chnl_useburst_set> に"1"を設定する。

uint32_t
TxNum: 転送数(最大値は 1024 回)

DMAC_Arbitration
ArbitrationMoment: アービトレーションを選択します。設定した回数の転送後に転送要求を確認し、優先度の高い要求があれば制御が高優先度のチャネルに切り替わります。

DMAC_BitWidth
SrcWidth: 転送元データサイズ

- **DMAC_BYTE:** 1 バイト
- **DMAC_HALF_WORD:** 2 バイト
- **DMAC_WORD:** 4 バイト

DMAC_IncWidth
SrcInc: 転送元アドレスのインクリメント
- **DMAC_INC_1B:** 1 バイト
- **DMAC_INC_2B:** 2 バイト
- **DMAC_INC_4B:** 4 バイト
- **DMAC_INC_0B:** インクリメントなし

DMAC_BitWidth
DstWidth: 転送先データサイズ
- **DMAC_BYTE:** 1 バイト
- **DMAC_HALF_WORD:** 2 バイト
- **DMAC_WORD:** 4 バイト

DMAC_IncWidth
DstInc: 転送先アドレスのインクリメント
- **DMAC_INC_1B:** 1 バイト
- **DMAC_INC_2B:** 2 バイト
- **DMAC_INC_4B:** 4 バイト
- **DMAC_INC_0B:** インクリメントなし

19.2.4.2 DMACA_Flag

メンバ:

uint32_t

All ユニット A のすべての DMA 要因

ビットフィールド: 各ビットの値の意味は以下の通りです。
‘0’: 終了割り込みは発生していない
‘1’: 終了割り込みが発生

uint32_t
ADCCompletion (Bit 0) AD 変換終了による終了割り込み

uint32_t
SSP0Reception (Bit 1) SSP0 受信による終了割り込み

uint32_t
SSP0Transmission (Bit 2) SSP0 送信による終了割り込み

uint32_
SSP1Reception (Bit 3) SSP1 受信による終了割り込み

uint32_t
SSP1Transmission (Bit 4) SSP1 送信による終了割り込み

uint32_t
SSP2Reception (Bit 5) SSP2 受信による終了割り込み

uint32_t
SSP2Transmission (Bit 6) SSP2 送信による終了割り込み

uint32_t

UART0Reception (Bit 7)　　UART0 受信による終了割り込み

uint32_t
UART0Transmission (Bit 8)　　UART0 送信による終了割り込み

uint32_t
UART1Reception (Bit 9)　　UART1 受信による終了割り込み

uint32_t
UART1Transmission (Bit 10)　UART1 送信による終了割り込み

uint32_t
I2C0RxorTx (Bit 11)　　I2C0 送受信による終了割り込み

uint32_t
I2C1RxorTx (Bit 12)　　I2C1 送受信による終了割り込み

uint32_t
I2C2RxorTx (Bit 13)　　I2C2 送受信による終了割り込み

uint32_t
I2C3RxorTx (Bit 14)　　I2C3 送受信による終了割り込み

uint32_t
I2C4RxorTx (Bit 15)　　I2C4 送受信による終了割り込み

uint32_t
Reserved (Bit 16 ~ Bit 30)　未使用

uint32_t
DMAREQA (Bit 31)　　DMA ユニット A リクエストピン DMAREQA による終了割り込み

19.2.4.3 DMACB_Flag

メンバ:

uint32_t
All　　DMA UINTB interrupt.

ビットフィールド: 各ビットの値の意味は以下の通りです。
‘0’: 終了割り込みは発生していない
‘1’: 終了割り込みが発生

uint32_t
SIO_UART0Reception (Bit 0) SIO/UART0 受信による終了割り込み

uint32_t
SIO_UART0Transmission (Bit 1) SIO/UART0 送信による終了割り込み

uint32_t
SIO_UART1Reception (Bit 2) SIO/UART1 受信による終了割り込み

uint32_t
SIO_UART1Transmission (Bit 3) SIO/UART1 送信による終了割り込み

uint32_t
SIO_UART2Reception (Bit 4) SIO/UART2 受信による終了割り込み

uint32_t
SIO_UART2Transmission (Bit 5) SIO/UART2 送信による終了割り込み

uint32_t
Reserved0 (Bit 6 ~ Bit 9)　　未使用

uint32_t
TMRB0CompareMatch (Bit 10)　TMRB0 コンペアー致による終了割り込み

uint32_t
TMRB1CompareMatch (Bit 11)　TMRB1 コンペアー致による終了割り込み

uint32_t
TMRB2CompareMatch (Bit 12)　TMRB2 コンペアー致による終了割り込み

uint32_t
TMRB3CompareMatch (Bit 13)　TMRB3 コンペアー致による終了割り込み

uint32_t
TMRB4CompareMatch (Bit 14)　TMRB4 コンペアー致による終了割り込み

uint32_t
TMRB5CompareMatch (Bit 15)　TMRB5 コンペアー致による終了割り込み

uint32_t
TMRB6CompareMatch (Bit 16)　TMRB6 コンペアー致による終了割り込み

uint32_t
TMRB7CompareMatch (Bit 17)　TMRB7 コンペアー致による終了割り込み

uint32_t
TMRBFCompareMatch (Bit 18)　TMRBF コンペアー致による終了割り込み

uint32_t
TMRB0InputCapture0 (Bit 19)　TMRB0 インプットキャプチャ 0 による終了割り込み

uint32_t
TMRB0InputCapture1 (Bit 20)　TMRB0 インプットキャプチャ 1 による終了割り込み

uint32_t
TMRB1InputCapture0 (Bit 21)　TMRB1 インプットキャプチャ 0 による終了割り込み

uint32_t
TMRB1InputCapture1 (Bit 22)　TMRB1 インプットキャプチャ 1 による終了割り込み

uint32_t
TMRB2InputCapture0 (Bit 23)　TMRB2 インプットキャプチャ 0 による終了割り込み

uint32_t
TMRB2InputCapture1 (Bit 24)　TMRB2 インプットキャプチャ 1 による終了割り込み

uint32_t
TMRB3InputCapture0 (Bit 25)　TMRB3 インプットキャプチャ 0 による終了割り込み

uint32_t
TMRB3InputCapture1 (Bit 26)　TMRB3 インプットキャプチャ 1 による終了割り込み

uint32_t
TMRB4InputCapture0 (Bit 27) TMRB4 インプットキャプチャ 0 による終了割り込み

uint32_t
TMRB4InputCapture1 (Bit 28) TMRB4 インプットキャプチャ 1 による終了割り込み

uint32_t
TMRB5InputCapture0 (Bit 29) TMRB5 インプットキャプチャ 0 による終了割り込み

uint32_t
TMRB5InputCapture1 (Bit 30) TMRB5 インプットキャプチャ 1 による終了割り込み

uint32_t
DMAREQB (Bit 31) DMA ユニット B リクエストピン DMAREQB による終了割り込み

19.2.4.4 DMACC_Flag

メンバ:

uint32_t
All DMA UINTC interrupt.

ビットフィールド: 各ビットの値の意味は以下の通りです。
'0': 終了割り込みは発生していない
'1': 終了割り込みが発生

uint32_t
SIO_UART3Reception (Bit 0) SIO/UART3 受信による終了割り込み

uint32_t
SIO_UART3Transmission (Bit 1) SIO/UART3 送信による終了割り込み

uint32_t
SIO_UART4Reception (Bit 2) SIO/UART4 受信による終了割り込み

uint32_t
SIO_UART4Transmission (Bit 3) SIO/UART4 送信による終了割り込み

uint32_t
SIO_UART5Reception (Bit 4) SIO/UART5 受信による終了割り込み

uint32_t
SIO_UART5Transmission (Bit 5) SIO/UART5 送信による終了割り込み

uint32_t
Reserved0 (Bit 6 ~ Bit 9) 未使用

uint32_t
TMRB8CompareMatch (Bit 10) TMRB8 コンペアー致による終了割り込み

uint32_t
TMRB9CompareMatch (Bit 11) TMRB9 コンペアー致による終了割り込み

uint32_t
TMRBACompareMatch (Bit 12) TMRBA コンペアー致による終了割り込み

uint32_t
TMRBBCompareMatch (Bit 13) TMRBB コンペアー致による終了割り込み

uint32_t
TMRBCCompareMatch (Bit 14) TMRBC コンペアー致による終了割り込み

uint32_t
TMRBDCompareMatch (Bit 15) TMRBD コンペアー致による終了割り込み

uint32_t
TMRBECompareMatch (Bit 16) TMRBE コンペアー致による終了割り込み

uint32_t
TMRB7CompareMatch (Bit 17) TMRB7 コンペアー致による終了割り込み

uint32_t
TMRBFCompareMatch (Bit 18) TMRBF コンペアー致による終了割り込み

uint32_t
TMRB8InputCapture0 (Bit 19) TMRB8 インプットキャプチャ 0 による終了割り込み

uint32_t
TMRB8InputCapture1 (Bit 20) TMRB8 インプットキャプチャ 1 による終了割り込み

uint32_t
TMRB9InputCapture0 (Bit 21) TMRB9 インプットキャプチャ 0 による終了割り込み

uint32_t
TMRB9InputCapture1 (Bit 22) TMRB9 インプットキャプチャ 1 による終了割り込み

uint32_t
TMRBAInputCapture0 (Bit 23) TMRBA インプットキャプチャ 0 による終了割り込み

uint32_t
TMRBAInputCapture1 (Bit 24) TMRBA インプットキャプチャ 1 による終了割り込み

uint32_t
TMRBBInputCapture0 (Bit 25) TMRBB インプットキャプチャ 0 による終了割り込み

uint32_t
TMRBBInputCapture1 (Bit 26) TMRBB インプットキャプチャ 1 による終了割り込み

uint32_t
TMRBCInputCapture0 (Bit 27) TMRBC インプットキャプチャ 0 による終了割り込み

uint32_t
TMRBCInputCapture1 (Bit 28) TMRBC インプットキャプチャ 1 による終了割り込み

uint32_t
TMRBDInputCapture0 (Bit 29) TMRBD インプットキャプチャ 0 による終了割り込み

uint32_t
TMRBDInputCapture1 (Bit 30) TMRBD インプットキャプチャ 1 による終了割り込み

uint32_t
DMAREQC (Bit 31) DMA ユニット C リクエストピン DMAREQC による終了割り込み

20. WDT

20.1 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\TX04_Periph_Driver\src\tmpm461_wdt.c
\Libraries/TX04_Periph_Driver\inc\tmpm461_wdt.h

20.2 API 関数

20.2.1 関数一覧

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

20.2.2 関数の種類

関数は、主に以下の 2 種類に分かれています:

1) ウォッチドッグタイマ設定:
   WDT_SetDetectTime(), WDT_SetOverflowOutput(), WDT_Init(), WDT_Enable(), WDT_Disable(), WDT_WriteClearCode()
2) IDLE モード時の開始・停止など:
   WDT_SetIdleMode().

20.2.3 関数仕様

20.2.3.1 WDT_SetDetectTime

検出時間の設定

関数のプロトタイプ宣言:

void
WDT_SetDetectTime(uint32_t ***DetectTime***)

引数:

***DetectTime*:** 検出時間を選択します。

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

機能:

WDT の検出時間を設定します。

戻り値:

なし

20.2.3.2 WDT_SetIdleMode

IDLE モード時の動作

関数のプロトタイプ宣言:

void
WDT_SetIdleMode(FunctionalState ***NewState***)

引数:

NewState: IDLE 時の動作の有効/無効を選択します。

- **ENABLE**: 動作
- **DISABLE**. 停止

機能:

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

補足:

CPU が IDLE モードに入る前に、設定してください。

戻り値:

なし

20.2.3.3 WDT_SetOverflowOutput

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

関数のプロトタイプ宣言:

void
WDT_SetOverflowOutput(uint32_t ***OverflowOutput***)

引数:

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

機能:

カウンタオーバーフロー時の NMI 割り込み/リセットの設定を行います。***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

20.2.3.4 WDT_Init

WDT の初期化

関数のプロトタイプ宣言:

void
WDT_Init (WDT_InitTypeDef* ***InitStruct***)

引数:

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

機能:

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

戻り値:

なし

20.2.3.5 WDT_Enable

WDT 動作の許可

関数のプロトタイプ宣言:

void
WDT_Enable(void)

引数:

なし。

機能:

WDT 動作を許可します。

戻り値:

なし

20.2.3.6 WDT_Disable

WDT 動作の禁止

関数のプロトタイプ宣言:

void
WDT_Disable(void)

引数:

なし。

機能:

WDT 動作を禁止します。

戻り値:

なし

20.2.3.7 WDT_WriteClearCode

クリアコードの書き込み

関数のプロトタイプ宣言:

void
WDT_WriteClearCode (void)

引数:

なし。

機能:

WDT カウンタにクリアコードを書き込みます。

戻り値:

なし

20.2.4 データ構造

20.2.4.1 WDT_InitTypeDef

メンバ:

uint32_t
DetectTime 検出時間を選択します。

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

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

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