TOSHIBA

TOSHIBA TX03 ペリフェラルドライバ ユーザーガイド (TMPM380)

第一版

2017 年 9 月

東芝デバイス＆ストレージ株式会社

CMDR-M380UG-01J

本製品取り扱い上のお願い

- ソフトウエア使用権許諾契約書の同意無しに使用しないで下さい。

目次

1. はじめに

本製品は、東芝TX03シリーズマイコン用ペリフェラルドライバセットです。TMPM380ペリフェラルドライバは、東芝TX03ペリフェラルドライバのTMPM380シリーズMCU用です。

TX03 ペリフェラルドライバでは、ユーザーアプリケーション内で各ペリフェラルを簡単に使用するためのマクロ、データ構造、関数および使用例を用意しています。

TMPM380 ペリフェラルドライバは以下の仕様に基づいています。
- スタートアップルーチンといくつかの関数を除き、C 言語で記述されています。
- すべての周辺機能をカバーしています。

2. TX03 ペリフェラルドライバの構成

/Libraries
TX03 CMSIS ファイルと TMPM380 ペリフェラルドライバが格納されています。

/Libraries/TX03_CMSIS
このフォルダには TMPM380 CMSIS ファイルのデバイス・ペリフェラル・アクセス・レイヤーが格納されています。

/Libraries/TX03_Periph_Driver
TMPM380 ペリフェラルドライバの全てのソースコードが格納されています。

/Libraries/TX03_Periph_Driver/inc
TMPM380 ペリフェラルドライバのヘッダファイルが格納されています。

/Libraries/TX03_Periph_Driver/src
TMPM380 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
TMPM380 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
TMPM380 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
TMPM380 ペリフェラルドライバの使用例が格納されています。

/Utilities/TMPM380-SK
TMPM380 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、12/10(選択可能)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 18 チャネル内蔵しています。

機能と特徴:
(1)PMD(MPT) やタイマからのトリガ信号に同期して任意のアナログ入力を変換することができます。
(2)ソフトウエア起動、常時起動において任意のアナログ入力を変換する事ができます。
(3)AD 変換値レジスタが12 個あります。
(4)PMD(MPT)やタイマのトリガ起動によるプログラム終了時に割り込みを発生できます。
(5)ソフトウェア起動、常時起動によるプログラム終了時に割り込みを発生できます。
(6)AD 監視機能があります。有効時に比較条件と一致した場合は割り込みを発生します。

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

全ドライバ API は、アプリで使用する API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm380_adc.c
/Libraries/TX03_Periph_Driver/inc/tmpm380_adc.h

3.2 API 関数

3.2.1 関数一覧

- void ADC_SetClk(uint32_t ***Sample_HoldTime***, uint32_t ***Prescaler_Output***);
- void ADC_SetResolution(ADC_Resolution ***ADBits***);
- ADC_Resolution ADC_GetResolution(void);
- void ADC_Enable(void);
- void ADC_Disable(void);
- void ADC_Start(TrgType ***Trg***);
- void ADC_StopConstantTrg(void);
- WorkState ADC_GetConvertState(TrgType ***Trg***);
- void ADC_SetLowPowerMode(FunctionalState ***NewState***);
- void ADC_SetMonitor(ADC_MonitorTypeDef * ***Monitor***);
- void ADC_DisableMonitor(ADC_CMPCRx ***CMPCRx***);
- ADC_ResultUnion ADC_GetConvertResult(ADC_REGx ***ResultREGx***);
- void ADC_SelectPMDTrgProgNum(PMD_TRG_PROGRAM_SELx ***SELx***, uint8_t ***MacroProgNum***);
- void ADC_SetPMDTrgProgINT(PMD_TrgProgINTTypeDef * ***TrgProgINT***);
- void ADC_SetTimerTrg(ADC_REGx ***REGx***, uint8_t ***MacroAINx***);
- void ADC_SetSWTrg(ADC_REGx ***REGx***, uint8_t ***MacroAINx***);
- void ADC_SetConstantTrg(ADC_REGx ***REGx***, uint8_t ***MacroAINx***);

3.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。

1) AD 変換設定:
   ADC_SetClk(), ADC_SetResolution(), ADC_SetMonitor(), ADC_DisableMonitor(), ADC_SelectPMDTrgProgNum(), ADC_SetPMDTrgProgINT(), ADC_SetTimerTrg(), ADC_SetSWTrg(), ADC_SetConstantTrg()
2) AD 機能の有効/無効、変換開始/停止:
   ADC_Enable(), ADC_Disable(), ADC_Start(), ADC_StopConstantTrg()
3) AD 変換ステータス/結果の読み出し:
   ADC_GetResolution(), ADC_GetConvertState(), ADC_GetConvertResult()
4) その他:
   ADC_SetLowPowerMode()

3.2.3 関数仕様

3.2.3.1 ADC_SetClk

AD 変換サンプルホールド時間とプリスケーラ出力の設定。

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

引数:
Sample_HoldTime: 以下から ADC サンプルホールド時間を選択します。
- **ADC_HOLD_FIX**: write “1001b” to TSH<0:3>へ“1001b”をライトします。

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_NONE:** fc

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。

戻り値:
なし

3.2.3.2 ADC_SetResolution

12bit/10bit 分解能モードの選択

関数のプロトタイプ宣言:
void
ADC_SetResolution(ADC_Resolution ***ADBits***)

引数:
ADBits: 12bit/10bit 分解能モードを選択します。
- **ADC_10BITS: 10bit**
- **ADC_12BITS: 12bit**

機能:
12bit/10bit 分解能モードを選択します。初期設定時に本 API をコールしてください。
電源投入後の初期状態は 12bit 分解能です。

戻り値:

なし

3.2.3.3 ADC_GetResolution

12bit/10bit 分解能モードの選択状態の取得

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

引数:
なし

機能:
12bit/10bit 分解能モードの選択状態を取得します。

戻り値:
12bit/10bit 分解能モードの選択状態:
ADC_10BITS: 10bit
ADC_12BITS: 12bit
ADC_RESERVEDBITS: 未使用

3.2.3.4 ADC_Enable

AD 変換の許可

関数のプロトタイプ宣言:
void
ADC_Enable(void)

引数:
なし

機能:
AD 変換を許可します。

戻り値:
なし

3.2.3.5 ADC_Disable

AD 変換の禁止

関数のプロトタイプ宣言:
void
ADC_Disable(void)

引数:
なし

機能:

AD 変換を禁止します。

戻り値:
なし

3.2.3.6 ADC_Start

AD 変換の開始

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

引数:
***Trg*: 以下からトリガタイプを選択します。**
- **ADC_TRG_SW:** ソフトウェア変換
- **ADC_TRG_CONSTANT:** 常時 AD 変換

機能:
選択したトリガタイプで AD 変換を開始します。

戻り値:
なし

3.2.3.7 ADC_StopConstantTrg

通常起動時 の AD 変換停止

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

引数:
なし

機能:
通常起動時の AD 変換を停止します。

戻り値:
なし

3.2.3.8 ADC_GetConvertState

AD 変換状態の確認

関数のプロトタイプ宣言:
WorkState
ADC_GetConvertState(TrgType ***Trg***)

引数:
***Trg*: 以下から起動方法を選択します。**

- **ADC_TRG_SW:** ソフトウェア起動
- **ADC_TRG_CONSTANT:** 常時 AD 変換
- **ADC_TRG_TIMER:** タイマからのトリガ信号
- **ADC_TRG_PMD:** PMD からのトリガ信号

機能:
指定された起動方法にて AD 変換状態を確認します。

戻り値:
AD 変換状態:
BUSY: 変換中
DONE: 停止

3.2.3.9 ADC_SetLowPowerMode

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_SetLowPowerMode(FunctionalState ***NewState***)

引数:
NewState: 以下から、AVREFH-AVREFL 間のリファレンス電流を制御します。
- **DISABLE**: リセット時以外常時通電
- **ENABLE**: 変換中のみ通電

機能:
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_SetMonitor

AD 監視機能の許可

関数のプロトタイプ宣言:
void
ADC_SetMonitor(ADC_MonitorTypeDef * ***Monitor***)

引数:
***Monitor*:** 構造体の詳細は以下です。
```
typedef struct {
   ADC_CMPCRx CMPCRx;
   ADC_REGx ResultREGx;
   uint32_t CmpTimes;
   ADC_CmpCondition Condition;
   uint32_t CmpValue;
} ADC_MonitorTypeDef
```
詳細は後述の“データ構造:”を参照してください。

機能:

ADC_MonitorTypeDef * ***Monitor*** で AD 監視設定を行い、有効にします。

戻り値:
なし

3.2.3.11 ADC_DisableMonitor

AD 監視機能の禁止

関数のプロトタイプ宣言:
void
ADC_DisableMonitor(ADC_CMPCRx ***CMPCRx***)

引数:
CMPCRx: 比較制御レジスタを選択します。
- **ADC_CMPCR_0**: ADCMPCR0
- **ADC_CMPCR_1**: ADCMPCR1

機能:
CMPCRx で無効にする AD 監視機能を選択します。

戻り値:
なし

3.2.3.12 ADC_GetConvertResult

AD 変換結果の読み出し

関数のプロトタイプ宣言:
ADC_ResultUnion
ADC_GetConvertResult(ADC_REGx ***ResultREGx***)

引数:
***ResultREGx*:** 以下から、AD 変換結果レジスタを選択します。
- **ADC_REG0**: ADREG0
- **ADC_REG1**: ADREG1
- **ADC_REG2**: ADREG2
- **ADC_REG3**: ADREG3
- **ADC_REG4**: ADREG4
- **ADC_REG5**: ADREG5
- **ADC_REG6**: ADREG6
- **ADC_REG7**: ADREG7
- **ADC_REG8**: ADREG8
- **ADC_REG9**: ADREG9
- **ADC_REG10**: ADREG10
- **ADC_REG11**: ADREG11

機能:
ResultREGx に設定されている AD 変換結果格納フラグ、オーバーランフラグ、変換結果を読み出します。

戻り値:

AD 変換結果 ADC_ResultUnion type:
--- ADC result value
--- Stored
--- OverRun
詳細は “データ構造” を参照してください。

3.2.3.13 ADC_SelectPMDTrgProgNum

AD 変換ユニットの PMD が発生するトリガ信号(PMD0~PMD3)に対して起動するプログラム(0~5)を選択します

関数のプロトタイプ宣言:
void
ADC_SelectPMDTrgProgNum(PMD_TRG_PROGRAM_SELx ***SELx***,
uint8_t ***MacroProgNum***)

引数:
***SELx*:** PMD トリガ用プログラム選択番号を以下から選択します。
- **PMD_TRG_PROGRAM_SEL0:** ADPSEL0
- **PMD_TRG_PROGRAM_SEL1:** ADPSEL1
- **PMD_TRG_PROGRAM_SEL2:** ADPSEL2
- **PMD_TRG_PROGRAM_SEL3:** ADPSEL3

***MacroProgNum*:** プログラム起動のトリガとなる PMD を選択します。
- **TRG_ENABLE(PROGRAM y)**: 有効にする PMD トリガ用プログラム y **(y = 0~5)**
- **TRG_DISABLE(PROGRAM y)**: 無効にする PMD トリガ用プログラム y **(y = 0~5)**

機能:
SELx で設定される ADC ユニットの PMD トリガ用プログラム選択レジスタを設定し、***MacroProgNum*** でレジスタの有効/無効を選択します。

戻り値:
なし

3.2.3.14 ADC_SetPMDTrgProgINT

PMD トリガ用割り込みプログラムの設定。

関数のプロトタイプ宣言:
void
ADC_SetPMDTrgProgINT(PMD_TrgProgINTTypeDef* ***TrgProgINT***)

引数:
***TrgProgINT*:** PMD トリガ用割り込みプログラムの構造体です。

```
typedef struct {
   PMD_INT_NAME INTProg0;
   PMD_INT_NAME INTProg1;
   PMD_INT_NAME INTProg2;
   PMD_INT_NAME INTProg3;
   PMD_INT_NAME INTProg4;
   PMD_INT_NAME INTProg5;
```

} PMD_TrgProgINTTypeDef
詳細は後述の “データ構造:”を参照してください。

機能:
TrgProgINT により PMD 用トリガ割り込みプログラムを設定します。

戻り値:
なし

3.2.3.15 ADC_SetPMDTrg

PMD トリガ用プログラム選択レジスタの設定

関数のプロトタイプ宣言:
void
ADC_SetPMDTrg(PMD_TrgTypeDef * ***PMDTrg***)

引数:
PMDTrg: PMD トリガ用プログラムの構造体です。
typedef struct {
  PMD_PROGRAMx ProgNum;
  uint8_t Reg0_AINx;
  uint8_t Reg1_AINx;
  uint8_t Reg2_AINx;
  uint8_t Reg3_AINx;
} PMD_TrgTypeDef
詳細は後述の “データ構造:”を参照してください。

機能:
PMD トリガ用プログラム選択レジスタの設定を行います。

戻り値:
なし

3.2.3.16 ADC_SetTimerTrg

タイマトリガ用プログラムレジスタの設定

関数のプロトタイプ宣言:
void
ADC_SetTimerTrg(ADC_REGx ***REGx***, uint8_t ***MacroAINx***)

引数:
***ResultREGx*:** 以下から、タイマトリガ用プログラムレジスタを選択します。

- **ADC_REG0: ADREG0**
- **ADC_REG1: ADREG1**
- **ADC_REG2: ADREG2**
- **ADC_REG3: ADREG3**
- **ADC_REG4: ADREG4**
- **ADC_REG5: ADREG5**
- **ADC_REG6: ADREG6**
- **ADC_REG7: ADREG7**
- **ADC_REG8: ADREG8**

- **ADC_REG9: ADREG9**
- **ADC_REG10: ADREG10**
- **ADC_REG11: ADREG11**

MacroAINx: 以下から、許可/禁止付 AD チャネルを選択します。

- **TRG_ENABLE(y)**: ***ResultREGx*** に対する AD チャネル'y'を許可
- **TRG_DISABLE(y)**: ***ResultREGx*** に対する AD チャネル'y'を禁止
  以下から、'y'を選択します。
  ADC_AIN0~ADC_AIN17

機能:
ResultREGx により AD 変換結果レジスタの設定を行い、タイマトリガ用プログラムレジスタの ***MacroAINx*** により AIN 端子に対するレジスタの許可/禁止を設定します。

戻り値:
なし

3.2.3.17 ADC_SetSWTrg

ソフトウェアトリガ用レジスタの設定

関数のプロトタイプ宣言:
void
ADC_SetSWTrg(ADC_REGx ***REGx***, uint8_t ***MacroAINx***)

引数:
***ResultREGx*:**ソフトウェアトリガ用プログラムによる AD 変換結果レジスタを選択します。

- **ADC_REG0: ADREG0**
- **ADC_REG1: ADREG1**
- **ADC_REG2: ADREG2**
- **ADC_REG3: ADREG3**
- **ADC_REG4: ADREG4**
- **ADC_REG5: ADREG5**
- **ADC_REG6: ADREG6**
- **ADC_REG7: ADREG7**
- **ADC_REG8: ADREG8**
- **ADC_REG9: ADREG9**
- **ADC_REG10: ADREG10**
- **ADC_REG11: ADREG11**

MacroAINx: 以下から、許可/禁止付 AD チャネルを選択します。

- **TRG_ENABLE(y)**: ***ResultREGx*** に対する AD チャネル'y'を許可
- **TRG_DISABLE(y)**: ***ResultREGx*** に対する AD チャネル'y'を禁止
  以下から、'y'を選択します。
  ADC_AIN00~ADC_AIN17

機能:
ResultREGx により AD 変換結果レジスタの設定を行い、ソフトウェアトリガ用プログラムレジスタの ***MacroAINx*** により AIN 端子に対するレジスタの許可/禁止を設定します。

戻り値:

なし

3.2.3.18 ADC_SetConstantTrg

常時トリガ用プログラムレジスタの設定

関数のプロトタイプ宣言:
void
ADC_SetConstantTrg(ADC_REGx ***REGx***, uint8_t ***MacroAINx***)

引数:
***ResultREGx*:** 以下から、常時トリガプログラム用 AD 変換結果レジスタを選択します。

- **ADC_REG0: ADREG0**
- **ADC_REG1: ADREG1**
- **ADC_REG2: ADREG2**
- **ADC_REG3: ADREG3**
- **ADC_REG4: ADREG4**
- **ADC_REG5: ADREG5**
- **ADC_REG6: ADREG6**
- **ADC_REG7: ADREG7**
- **ADC_REG8: ADREG8**
- **ADC_REG9: ADREG9**
- **ADC_REG10: ADREG10**
- **ADC_REG11: ADREG11**

MacroAINx: 以下から、許可/禁止付 AD チャネルを選択します。

- **TRG_ENABLE(y)**: ***ResultREGx*** に対する AD チャネル'y'を許可
- **TRG_DISABLE(y)**: ***ResultREGx*** に対する AD チャネル'y'を禁止
  以下から、'y'を選択します。
  ADC_AIN00~ADC_AIN17

機能:
ResultREGx により AD 変換結果レジスタの設定を行い、常時トリガ用プログラムレジスタの ***MacroAINx*** により AIN 端子に対するレジスタの許可/禁止を設定します。

戻り値:
なし

3.2.4 データ構造

3.2.4.1 ADC_MonitorTypeDef

メンバ:
ADC_CMPCRx
CMPCRx 以下からコンペア制御レジスタを選択します。

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

ADC_REGx
ResultREGx 以下から AD 変換結果レジスタを選択します。

- **ADC_REG0: ADREG0**
- **ADC_REG1: ADREG1**
- **ADC_REG2: ADREG2**
- **ADC_REG3: ADREG3**

- **ADC_REG4: ADREG4**
- **ADC_REG5: ADREG5**
- **ADC_REG6: ADREG6**
- **ADC_REG7: ADREG7**
- **ADC_REG8: ADREG8**
- **ADC_REG9: ADREG9**
- **ADC_REG10: ADREG10**
- **ADC_REG11: ADREG11**

uint32_t
CmpTimes 以下から比較カウント数を選択します。
- **1~16**

ADC_CmpCondition
Condition 以下から ADREGx <ADCMPy>のの比較設定を選択します。(x=0~11, y=0~1)
- **ADC_LARGER_THAN_CMP_REG:** 変換結果レジスタ値が比較レジスタ 0 より大きい場合に割り込みを発生します。
- **ADC_SMALLER_THAN_CMP_REG:** 変換結果レジスタ値が比較レジスタ 0 より小さい場合に割り込みを発生します。

uint32_t
CmpValue 以下から ADxCMP0、または ADxCMP1 に設定する比較値を選択します。
- **12bit モードの場合: 0~4095**
- **10bit モードの場合: 0~1023**

3.2.4.2 ADC_ResultUnion

メンバ:
uint32_t
All: AD 変換結果

ビットフィールド:

uint32_t ***Stored***:	1	AD 変換結果の格納状態
uint32_t ***OverRun***:	1	AD 変換オーバーランフラグ
uint32_t ***Reserved1***:	2	予約
uint32_t ***ADResult***:	12	AD 変換結果
uint32_t ***Reserved2***:	16	予約

3.2.4.3 PMD_IntForProgNumTypeDef

メンバ:

PMD_INT_NAME
INTProg0 以下からプログラム 0 に対する割り込みを選択します。
- **PMD_INTNONE:** 割り込み出力なし
- **PMD_INTADPD0:** INTADPD0 出力
- **PMD_INTADPD1:** INTADPD1 出力

PMD_INT_NAME

INTProg1 プログラム 1 に対する割り込みを選択する以外 ***INTProg0*** と同じです。

PMD_INT_NAME
INTProg2 プログラム 2 に対する割り込みを選択する以外 ***INTProg0*** と同じです。

PMD_INT_NAME
INTProg3 プログラム 3 に対する割り込みを選択する以外 ***INTProg0*** と同じです。

PMD_INT_NAME
INTProg4 プログラム 4 に対する割り込みを選択する以外 ***INTProg0*** と同じです。

PMD_INT_NAME
INTProg5 プログラム 5 に対する割り込みを選択する以外 ***INTProg0*** と同じです。

3.2.4.4 PMD_TrgTypeDef

メンバ:

PMD_PROGRAMx
ProgNum 以下から ADPSETx (x = 0~5)に対するプログラム番号を選択します。
- **PROGRAM0~PROGRAM5**

uint8_t
Reg0_AINx 以下から、ADPSETx の REG0 に対する許可/禁止付 AD 入力チャネルを選択します。
- **TRG_ENABLE(y)**: AD チャネル'y'を許可
- **TRG_DISABLE(y)**: AD チャネル'y'を禁止

以下から、'y'を選択します。
AIN0~AIN17

uint8_t
Reg1_AINx ADxPSETn の REG1 に対する許可/禁止付 AD チャネルを選択します。選択する AD チャネルは ***Reg0_AINx*** と同じです。

uint8_t
Reg2_AINx ADxPSETn の REG2 に対する許可/禁止付 AD チャネルを選択します。選択する AD チャネルは ***Reg0_AINx*** と同じです。

uint8_t
Reg3_AINx ADxPSETn の REG3 に対する許可/禁止付 AD チャネルを選択します。選択する AD チャネルは ***Reg0_AINx*** と同じです。

4. CG

4.1 概要

本 CG API は TMPM380 CG において以下の機能を提供します。
- システムクロックの制御、クロック逓倍回路（PLL）の制御
- プリスケーラクロックの制御
- ウォーミングアップタイマの制御
- 各種低消費電力モードの制御
- 割り込み要求制御

本ドライバは、以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver\src\tmpm380_cg.c
/Libraries/TX03_Periph_Driver\inc\tmpm380_cg.h

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

fosc : X1, X2 端子より入力されるクロック
fs: XT1, XT2 (低速クロック) 端子より入力されるクロック
fPLL : PLL により逓倍(4 逓倍) されたクロック
fc : CGPLLSEL<PLLSEL> で選択されたクロック(高速クロック)
fgear : CGSYSCR<GEAR[2:0]> で選択されたクロック
fsys : fgear と同一クロック(システムクロック)
fperiph : CGSYSCR<FPSEL> で選択されたクロック
φT0 : CGSYSCR<PRCK[2:0]> で選択されたクロック (プリスケーラクロック)

4.2 API 関数

4.2.1 関数一覧

- void CG_SetFgearLevel(CG_DivideLevel ***DivideFgearFromFc***)
- CG_DivideLevel CG_GetFgearLevel(void)
- void CG_SetPhiT0Src(CG_PhiT0Src ***PhiT0Src***)
- CG_PhiT0Src CG_GetPhiT0Src(void)
- Result CG_SetPhiT0Level(CG_DivideLevel ***DividePhiT0FromFc***)
- CG_DivideLevel CG_GetPhiT0Level(void)
- void CG_SetSCOUTSrc(CG_SCOUTSrc ***Source***)
- CG_SCOUTSrc CG_GetSCOUTSrc(void)
- void CG_SetWarmUpTime(CG_WarmUpSrc ***Source***, uint16_t ***Time***)
- void CG_StartWarmUp(void)
- WorkState CG_GetWarmUpState(void)
- Result CG_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***)
- Result CG_SetFs(FunctionalState ***NewState***)
- FunctionalState CG_GetFsState(void)
- void CG_SetPortM(CG_PortMMode ***Mode***)
- void CG_SetSTBYMode(CG_STBYMode ***Mode***)
- CG_STBYMode CG_GetSTBYMode(void)
- void CG_SetExitStopModeFosc(FunctionalState ***NewState***)
- FunctionalState CG_GetExitStopModeFoscState(void)
- void CG_SetExitStopModeFs(FunctionalState ***NewState***)
- FunctionalState CG_GetExitStopModeFsState(void)
- void CG_SetPinStateInStopMode(FunctionalState ***NewState***)
- FunctionalState CG_GetPinStateInStopMode(void)
- Result CG_SetFcSrc(CG_FcSrc ***Source***)
- CG_FcSrc CG_GetFcSrc(void)
- Result CG_SetFsysSrc(CG_FsysSrc ***Source***)
- CG_FsysSrc CG_GetFsysSrc(void)
- void CG_SetSTBYReleaseINTSrc(CG_INTSrc ***INTSource***,
  CG_INTActiveState ***ActiveState***,
  FunctionalState ***NewState***)
- CG_INTActiveState CG_GetSTBYReleaseINTState(CG_INTSrc ***INTSource***)
- void CG_ClearINTReq(CG_INTSrc ***INTSource***)
- CG_NMIFactor CG_GetNMIFlag(void)
- CG_ResetFlag CG_GetResetFlag(void)

4.2.2 関数の種類

CG API は 3 つのグループに分けられます。

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

4.2.3 関数仕様

4.2.3.1 CG_SetFgearLevel

fgear,fc 間の分周レベル設定

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

引数:

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

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

機能：
fgear,fc 間の分周レベルを設定します。

戻り値:
なし

4.2.3.2 CG_GetFgearLevel

fgear,fc 間の分周レベルの取得

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

引数:
なし

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

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

4.2.3.3 CG_SetPhiT0Src

PhiT0(ΦT0) ,fc 間の PhiT0(ΦT0) ソースの設定

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

引数:
PhiT0Src: 以下から PhiT0 ソースを選択します。

- **CG_PHIT0_SRC_FGEAR** : fgear が PhiT0 ソース
- **CG_PHIT0_SRC_FC**: fc が PhiT0 ソース
- **CG_PHIT0_SRC_FS**: fs が PhiT0 ソース

機能:

PhiT0 (ΦT0) ソースを選択します。

戻り値:
なし

4.2.3.4 CG_GetPhiT0Src

PhiT0 (ΦT0) ソースの取得

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

引数:
なし

機能:
PhiT0 (ΦT0) ソースを取得します。

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

4.2.3.5 CG_SetPhiT0Level

PhiT0 (ΦT0) と fc 間の分周レベルの設定

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

引数:
DividePhiT0FromFc: PhiT0 (ΦT0) と fc 間の分周レベルを下記の値から設定します。

- **CG_DIVIDE_1**: ΦT0 = fc or fs
- **CG_DIVIDE_2**: ΦT0 = fc/2 or fs/2
- **CG_DIVIDE_4**: ΦT0 = fc/4 or fs/2
- **CG_DIVIDE_8**: ΦT0 = fc/8 or fs/8
- **CG_DIVIDE_16**: ΦT0 = fc/16 or fs/16
- **CG_DIVIDE_32**: ΦT0 = fc/32 or fs/32
- **CG_DIVIDE_64**: ΦT0 = fc/64
- **CG_DIVIDE_128**: ΦT0 = fc/128
- **CG_DIVIDE_256**: ΦT0 = fc/256
- **CG_DIVIDE_512**: ΦT0 = fc/512

機能:
プリスケーラークロックの分周レベルを設定します。

戻り値:
SUCCESS 設定成功
ERROR エラー

4.2.3.6 CG_GetPhiT0Level

PhiT0(ΦT0) ,fc 間の分周レベルの取得

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

引数:
なし

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

戻り値:
PhiT0(ΦT0) ,fc 間の分周レベルを以下から設定します。
CG_DIVIDE_1: ΦT0 = fc or fs
CG_DIVIDE_2: ΦT0 = fc/2 or fs/2
CG_DIVIDE_4: ΦT0 = fc/4 or fs/4
CG_DIVIDE_8: ΦT0 = fc/8 or fs/8
CG_DIVIDE_16: ΦT0 = fc/16 or fs/16
CG_DIVIDE_32: ΦT0 = fc/32 or fs/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 : invalid data is read

4.2.3.7 CG_SetSCOUTSrc

SCOUT ソースクロックの選択

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

引数:
Source:以下から SCOUT 出力のソースクロックを選択します。
- **CG_SCOUT_SRC_FS**: fs
- **CG_SCOUT_SRC_HALF_FSYS**: fsys/2
- **CG_SCOUT_SRC_FSYS**: fsys
- **CG_SCOUT_SRC_PHIT0**:ΦT0

機能:
SCOUT 出力のソースクロックを選択します。

戻り値:
なし

4.2.3.8 CG_GetSCOUTSrc

SCOUT ソースクロックの取得

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

引数:
なし

機能:
SCOUT ソースクロックを取得します。

戻り値:
SCOUT ソースクロック:
- **CG_SCOUT_SRC_FS**: fs
- **CG_SCOUT_SRC_HALF_FSYS**: fsys/2
- **CG_SCOUT_SRC_FSYS**: fsys
- **CG_SCOUT_SRC_PHIT0**:ΦT0

4.2.3.9 CG_SetWarmUpTime

ウォームアップ時間の設定

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

引数:
Source: 以下から、ウォームアップカウンタを選択します。
- **CG_WARM_UP_SRC_OSC1**: fosc1
- **CG_WARM_UP_SRC_OSC2**: fosc2
- **CG_WARM_UP_SRC_XT1**: fs

Time: ***Source*** が CG_WARM_UP_SRC_XT1 の場合は 0U~0x4000U から、それ以外の場合は 0U~0x1000U から選択します。

機能:
ウォームアップ時間の設定を行います。設定時間の算出方法は以下です。
設定値 = ((ウォームアップ時間) / (入力クロック))/16

以下はウォームアップ時間の設定例です。
/* ウォームアップ時間が 100us で、入力クロックが 8M の場合 */
設定値= 100*10E(-6)/(1/(8*10E(6)))/16=0x0320>>4=0x32

戻り値:
なし

4.2.3.10 CG_StartWarmUp

ウォームアップの開始

関数のプロトタイプ宣言:
void

CG_StartWarmUp (void)

引数:
なし

機能:
ウォームアップを開始します。

戻り値:
なし

4.2.3.11 CG_GetWarmUpState

ウォームアップ動作状態の確認

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

引数:
なし

機能:
ウォーミングアップ動作状態を確認します。

ウォーミングアップタイマの使用例:

```
CG_SetWarmUpTime(CG_WARM_UP_SRC_OSC1, 0x32);
/* start warm up */
CG_StartWarmUp();
/* check warm up is finished or not*/
while(CG_GetWarmUpState()==BUSY);
```

戻り値:
ウォームアップ動作状態:
DONE: 動作終了
BUSY: 動作中

4.2.3.12 CG_SetPLL

PLL 動作の許可/禁止

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

引数:
NewState: 以下から選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
PLL 動作の許可/禁止を選択します。

PLL として fc を選択した場合、無効にすることができません。この場合、戻り値は **ERROR** となります。

戻り値:
SUCCESS: 成功
ERROR: 失敗

4.2.3.13 CG_GetPLLState

PLL 設定状態の確認

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

引数:
なし

機能:
PLL 設定状態を確認します。

戻り値:
PLL 設定状態です
ENABLE: PLL 有効
DISABLE: PLL 無効

4.2.3.14 CG_SetFosc

高速発振器(osc1 or osc2)の有効/無効

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

引数:
Source: 以下から fosc のソースクロックを選択します。
- **CG_FOSC_OSC1**: fosc1
- **CG_FOSC_OSC2**: fosc2

NewState: 以下から、fosc の有効/無効を選択します。
- **ENABLE**: 有効
- **DISABLE**: 無効

機能:
高速発振器の有効/無効を選択します。
fgear と system clock (fsys)が選択されている場合、高速発振器 (fosc) は無効にできません。この場合、戻り値は **ERROR** となります。

戻り値:
SUCCESS: 成功

ERROR: 失敗

4.2.3.15 CG_SetFoscSrc

高速発振器(fosc)のソース設定

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

引数:
Source: fosc のソースを選択します。
- **CG_FOSC_OSC1**: fosc1
- **CG_FOSC_OSC2**: fosc2

機能:
高速発振器(fosc)のソースを設定します。

戻り値:
なし

4.2.3.16 CG_GetFoscSrc

高速発振器(fosc)ソースの取得

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

引数:
なし

機能:
高速発振器ソースを取得します。

戻り値:
fosc のソース
CG_FOSC_OSC1: fosc1
CG_FOSC_OSC2: fosc2

4.2.3.17 CG_GetFoscState

高速発振器(fosc)の状態

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

引数:
Source: 以下から fosc のソースを選択します。
- **CG_FOSC_OSC1**: fosc1
- **CG_FOSC_OSC2**: fosc2

機能:
高速発振器の状態を取得します。

戻り値:
fosc の状態
ENABLE: 有効
DISABLE: 無効

4.2.3.18 CG_SetFs

低速クロック(XT1)の有効/無効

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

引数:
NewState: 以下から、fs の有効/無効を選択します。
- **ENABLE**: 有効
- **DISABLE**: 無効

機能:
低速発振器(XT1)の有効/無効を選択します。
system clock (fsys)として fs が選択されている場合、低速発振器 (XT1) は無効にできません。この場合、戻り値は **ERROR** となります。

戻り値:
SUCCESS: 成功
ERROR: 失敗

4.2.3.19 CG_GetFsState

低速発振器(XT1)の状態

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

引数:
なし

機能:
低速発振器(XT1)の状態を取得します。

戻り値:
XT1 の状態
ENABLE: 有効
DISABLE: 無効

4.2.3.20 CG_SetPortM

ポート M の設定(X1/X2 または汎用ポート)

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

引数:
Mode:
- **CG_PORTM_AS_GPIO** : 汎用ポート
- **CG_PORTM_AS_HOSC**: X1/X2

機能:
ポート M の設定を行います。***Mode*** が **CG_PORTM_AS_GPIO** の時は汎用ポート、***Mode*** が **CG_PORTM_AS_HOSC** の時は X1/X2 に設定します。

戻り値:
なし

4.2.3.21 CG_SetSTBYMode

スタンバイモードの設定

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

引数:
Mode:以下からスタンバイモードを選択します。
- **CG_STBY_MODE_STOP**: STOP モード(内部発振器含めてすべての内部回路が停止します)
- **CG_STBY_MODE_SLEEP**: SLEEP モード(fs と RMC のみ動作します)
- **CG_STBY_MODE_IDLE**: IDLE モード(CPU のみ停止します)

機能:
スタンバイ命令実行後の低消費電力モードを選択します。

戻り値:
なし

4.2.3.22 CG_GetSTBYMode

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:
スタンバイモードの設定状態を取得します。
設定状態が “Reserved”の場合、戻り値は “**CG_STBY_MODE_UNKNOWN**” です。

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

4.2.3.23 CG_SetExitStopModeFosc

STOP モード解除後の自動高速発振設定

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

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

機能:
STOP モード解除後の自動高速発振を設定します。

戻り値:
なし

4.2.3.24 CG_GetExitStopModeFoscState

STOP モード解除後の自動高速発振設定状態の取得

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

引数:
なし

機能:
STOP モード解除後の自動高速発振設定状態を取得します。

戻り値:
ENABLE: 発振
DISABLE: 停止

4.2.3.25 CG_SetExitStopModeFs

STOP モード解除後の自動低速発振設定

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

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

機能:
STOP モード解除後の自動低速発振を設定します。

戻り値:
なし

4.2.3.26 CG_GetExitStopModeFsState

STOP モード解除後の自動低速発振設定状態の取得

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

引数:
なし

機能:
STOP モード解除後の自動低速発振設定状態の取得

戻り値:
ENABLE: 発振
DISABLE: 停止

4.2.3.27 CG_SetPinStateInStopMode

STOP モード中の端子状態の設定

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

引数:
NewState:
- **DISABLE:** STOP モード中端子をドライブしません。(<DRVE>=0)
- **ENABLE:** STOP モード中端子をドライブします(<DRVE>=1)

STOP1 モード中の端子状態制御については、MCU データシートの“低消費電力モード”を参照してくだいさい。

機能:
STOP モード中の端子状態を設定します。

戻り値:
なし

4.2.3.28 CG_GetPinStateInStopMode

STOP モード中の端子状態設定の取得

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

引数:
なし

機能:
STOP モード中の端子状態設定を取得します。

戻り値:
DISABLE: STOP モード中端子をドライブしません。(<DRVE>=0)
ENABLE: STOP モード中端子をドライブします。(<DRVE>=1)

4.2.3.29 CG_SetFcSrc

fc ソースクロック選択

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

引数:
Source: 以下から、fc ソースクロックを選択します。
- **CG_FC_SRC_FOSC** : fosc
- **CG_FC_SRC_FPLL**: fpll

機能:
fc ソースクロックを選択します。
本 API をコールする前に以下の状態になっていることを確認してください。
a) 高速発振器を選択している
b) a)かつ ***Source*** が **CG_FC_SRC_FPLL** の場合、PLL が有効
(“**CG_SetPLL**(ENABLE)”)になっている。
上記状態になっていない場合、戻り値は **ERROR** となります。

戻り値:
SUCCESS: 成功
ERROR: 無効

4.2.3.30 CG_GetFcSrc

fc ソースクロック設定状態の取得

関数のプロトタイプ宣言:

CG_FcSrc
CG_GetFosc (void)

引数:
なし

機能:
fc ソースクロック設定状態を取得します。

戻り値:
CG_FC_SRC_FOSC: fosc
CG_FC_SRC_FPLL: fpll

4.2.3.31 CG_SetFsysSrc

fsys ソースクロック選択

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

引数:
Source: 以下から、fsys ソースクロックを選択します。
- **CG_FC_SRC_FGEAR** : fgear
- **CG_FC_SRC_FS**: fs

機能:
fsys ソースクロックを選択します。
CG_FSYS_SRC_FGEAR を選択する場合、高速発振器(X1)が既に発振していることを、また **CG_FSYS_SRC_FS** を選択する場合、低速発振器(XT1)が既に発振していることを確認してください。
上記状態になっていない場合戻り値は **ERROR** となります。

戻り値:
SUCCESS: 成功
ERROR: 無効

4.2.3.32 CG_GetFsysSrc

fsys ソースクロック設定状態の取得

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

引数:
なし

機能:
fsys ソースクロック設定状態を取得します。

戻り値:

CG_FSYS_SRC_FGEAR : fgear
CG_FSYS_SRC_FS : fs.

4.2.3.33 CG_SetSTBYReleaseINTSrc

スタンバイモードの解除割り込みソースの設定

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

引数:
INTSource: 以下から、スタンバイモードの解除割り込みソースを選択します。
- **CG_INT_SRC_0** : INT0
- **CG_INT_SRC_1** : INT1
- **CG_INT_SRC_2** : INT2
- **CG_INT_SRC_3** : INT3
- **CG_INT_SRC_4** : INT4
- **CG_INT_SRC_5** : INT5
- **CG_INT_SRC_6** : INT6
- **CG_INT_SRC_7** : INT7
- **CG_INT_SRC_8** : INT8
- **CG_INT_SRC_9** : INT9
- **CG_INT_SRC_A** : INT10
- **CG_INT_SRC_B** : INT11
- **CG_INT_SRC_C** : INT12
- **CG_INT_SRC_D** : INT13
- **CG_INT_SRC_E** : INT14
- **CG_INT_SRC_F** : INT15
- **CG_INT_SRC_RTC**: RTC 割り込み
- **CG_INT_SRC_RMC_RX**: RMC 受信割り込み

ActiveState: 以下から、解除トリガのアクティブ状態を選択します。
- **CG_INT_ACTIVE_STATE_L**: Low レベル
- **CG_INT_ACTIVE_STATE_H**: High レベル
- **CG_INT_ACTIVE_STATE_FALLING**: 立下りエッジ
- **CG_INT_ACTIVE_STATE_RISGING**: 立ち上がりエッジ
- **CG_INT_ACTIVE_STATE_BOTH_EDGES**: 両エッジ

NewState: 以下から、解除トリガの許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE:** 禁止

機能:
スタンバイモードの解除割り込みソースを設定します。

戻り値:
なし

4.2.3.34 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: 以下から、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_RTC, CG_INT_SRC_RMC_RX

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:
- **CG_INT_ACTIVE_STATE_FALLING**: 立下りエッジ
- **CG_INT_ACTIVE_STATE_RISGING**: 立ち上がりエッジ
- **CG_INT_ACTIVE_STATE_BOTH_EDGES**: 両エッジ
- **CG_INT_ACTIVE_STATE_INVALID**: 無効

4.2.3.35 CG_ClearINTReq

スタンバイ解除割り込み要求のクリア

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

引数:
INTSource: 以下から、解除割り込みソースを選択します。
CG_INT_SRC_0, 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_RTC, CG_INT_SRC_RMC_RX.

機能:
スタンバイ解除割り込み要求をクリアします。

戻り値:
なし

4.2.3.36 CG_GetNMIFlag

NMI フラグの取得

関数のプロトタイプ宣言:
CG_NMIFactor

CG_ GetNMIFlag (void)

引数:
なし

機能:
NMI フラグを取得します。

戻り値:
WDT (Bit 0) : WDT による NMI 発生
VoltageDetection (Bit 2) : 低電圧検出による NMI 発生

4.2.3.37 CG_GetResetFlag

リセットフラグの取得

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

引数:
なし

機能:
リセットフラグを取得します。

戻り値:
PowerOn (Bit 0) : パワーオンリセット
ResetPin (Bit 1) : 端子リセット
WDTReset (Bit 2) :WDT リセット
DebugReset (Bit 4) :SYSRESETREQ によるリセット
OFDReset (Bit 5) : OFD リセット

4.2.4 データ構造

4.2.4.1 CG_NMIFactor

メンバ:
uint32_t
All NMI 要因フラグです。

ビットフィールド:
uint32_t
WDT(Bit 0) WDT による NMI 発生
uint32_t
Reserved(Bit 1) 予約
uint32_t
VoltageDetection (Bit 2) 低電圧検出による NMI 発生

4.2.4.2 CG_ResetFlag

メンバ:

uint32_t
All リセット要因フラグです。

ビットフィールド:
uint32_t
PowerOn(Bit 0)　Power On Reset フラグ
uint32_t
ResetPin(Bit 1)　RESET 端子フラグ
uint32_t
WDTReset(Bit 2)　WDT リセットフラグ
uint32_t
Reserved(Bit 3)　予約
uint32_t
DebugReset(Bit 4)　デバッグリセットフラグ
uint32_t
OFDReset(Bit 5)　OFD リセットフラグ

5. DMAC

5.1 概要

本デバイスは、DMA 要求選択レジスタにより制御される DMA コントローラを 1 ユニット(2 チャネル)内蔵しています。DMA コントローラは、3 つの転送タイプのいずれかで動作します。3 つの転送タイプは、メモリ-メモリ、メモリ-周辺回路、周辺回路–メモリです。DMA チャネル 0 は DMA チャネル 1 より優先度が高くなります。

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

全ドライバ API は、アプリ使用の API 定義を格納する以下のファイルで構成されています。
\Libraries\TX03_Periph_Driver\src\tmpm380_dmac.c
\Libraries/TX03_Periph_Driver\inc\tmpm380_dmac.h

5.2 API 関数

5.2.1 関数一覧

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

5.2.2 関数の種類

関数は、主に以下の 5 種類に分かれています。:

1) DMAC 基本設定:
   DMAC_Enable(),DMAC_Disable(),DMAC_SetDMAChannel(), DMAC_Init()

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

5.2.3 関数仕様

5.2.3.1 DMAC_Enable

DMA 回路動作の許可

関数のプロトタイプ宣言:
void
DMAC_Enable(void);

引数:
なし

機能:
DMA 回路動作を許可します。

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

戻り値:
なし

5.2.3.2 DMAC_Disable

DMA 回路動作の禁止

関数のプロトタイプ宣言:
void
DMAC_Disable(void);

引数:
なし

機能:
DMA 回路動作を禁止します。

戻り値:
なし

5.2.3.3 DMAC_GetINTReq

DMA チャネル割り込みステータスの取得

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

引数:
なし

機能:
DMA チャネル割り込み要求状態を取得します。

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

5.2.3.4 DMAC_GetTxINTReq

DMA チャネル転送割り込み要求状態の取得

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

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

機能:
DMA チャネル転送割り込み要求状態を取得します。

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

5.2.3.5 DMAC_ClearTxINTReq

転送割り込み要求のクリア

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

引数:
Chx: 以下から DMA チャネルを選択します。

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

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

機能:
転送割り込み要求をクリアします。

戻り値:
なし

5.2.3.6 DMAC_GetRawTxINTReq

DMA チャネルの許可前転送終了割り込み発生状態の取得

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

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

機能:
DMA チャネルの許可前転送終了割り込み発生状態を取得します。

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

5.2.3.7 DMAC_GetChannelTxState

DMA チャネル転送状態の取得

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

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

機能:
本関数は、***Chx*** が **DMAC_CHANNEL_0** の時、DMA チャネル 0 転送状態を取得します。***Chx*** が **DMAC_CHANNEL_1** の時、DMA チャネル 1 転送状態を取得します。戻り

値が **BUSY** の時は、DMA チャネルは有効で、データ送信中であることを示します。戻り値が **DONE** の時は、DMA チャネルは無効で、データ送信は終了していることを示します。

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

5.2.3.8 DMAC_SetSWBurstReq

ソフトウェアによる DMA バースト転送要求の設定

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

引数:
BurstReq: 以下のいずれかのバースト要求番号を選択します。
- **DMAC_SIO_0_RTX** SIO0/UART0 受信
- **DMAC_SIO_1_RTX** SIO1/UART1 受信
- **DMAC_SIO_2_RTX** SIO2/UART2 受信
- **DMAC_SIO_3_RTX** SIO3/UART3 受信
- **DMAC_SIO_4_RTX** SIO4/UART4 受信
- **DMAC_SSP0_TX** SSP0 送信
- **DMAC_SSP0_RX** SSP0 受信
- **DMAC_SSP1_TX** SSP1 送信
- **DMAC_SSP1_RX** SSP1 受信

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

戻り値:
なし

5.2.3.9 DMAC_GetSWBurstReqState

ソフトウェアによる DMA バースト要求状態の取得

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

引数:
なし

機能:
ソフトウェアによる DMA バースト要求状態を取得します。

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

5.2.3.10 DMAC_SetSWSingleReq

ソフトウェアによる DMA シングル転送要求の設定

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

引数:
SingleReq: 以下から、シングル要求番号を選択します。
- **DMAC_SSP0_RX** SSP0 受信
- **DMAC_SSP1_RX** SSP1 受信

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

戻り値:
なし

5.2.3.11 DMAC_GetSWSingleReqState

ソフトウェアによる DMA シングル要求状態の取得

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

引数:
なし

機能:
ソフトウェアによる DMA シングル要求状態を取得します。

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

5.2.3.12 DMAC_SetLinkedList

DMA チャネル・コレクションアイテムレジスタの設定

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

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

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

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

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

戻り値:
なし

5.2.3.13 DMAC_GetFIFOState

FIFO 状態の取得

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

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

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

戻り値:
以下のいずれかの FIFO 状態を返します。
BUSY、または **DONE**

5.2.3.14 DMAC_SetDMAHalt

DMA 要求の設定

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

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

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

機能:
DMA 要求受付制御を設定します。

戻り値:
なし

5.2.3.15 DMAC_SetLockedTx

ロック転送の設定

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

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

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

機能:
ロック転送を設定します。

戻り値:
なし

5.2.3.16 DMAC_SetTxINTConfig

転送割り込みの設定

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

引数:

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

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

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

機能:
転送割り込みを設定します。

戻り値:
なし

5.2.3.17 DMAC_SetDMAChannel

DMA チャネルの許可/禁止設定

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

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

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

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

戻り値:
なし

5.2.3.18 DMAC_Init

DMA チャネルの初期設定

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

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

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

機能:
DMA チャネルの初期設定を行います。

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

戻り値:
なし

5.2.4 データ構造

5.2.4.1 DMAC_InitTypeDef

メンバ:

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

uint32_t
SrcAddr: 転送元アドレスを設定します。

uint32_t
DstAddr: 転送先アドレスを設定します。

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

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

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

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

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

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

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

DMAC_ReqNum
TxPeriph:以下から、転送先の周辺回路を設定します。(*)
- **DMAC_SIO_0_RTX** SIO0 送受信
- **DMAC_SIO_1_RTX** SIO1 送受信
- **DMAC_SIO_2_RTX** SIO2 送受信
- **DMAC_SIO_3_RTX** SIO3 送受信
- **DMAC_SIO_4_RTX** SIO4 送受信
- **DMAC_SSP0_TX** SSP0 送信
- **DMAC_SSP0_RX** SSP0 受信
- **DMAC_SSP1_TX** SSP1 送信
- **DMAC_SSP1_RX** SSP1 受信

FunctionalState

TxINT：以下から、転送割り込みステートを選択します。

- **EANBLE**：転送割り込み許可
- **DISABLE**：転送割り込み無効

(*):
TxPeriph に転送先の周辺回路を指定する場合、***TxDirection*** に **DMAC_MEMORY_TO_PERIPH** を選択してください。また ***TxPeriph*** に転送元の周辺回路を指定する場合、***TxDirection*** に **DMAC_PERIPH_TO_MEMORY** を選択してください。
TxDirection に **DMAC_MEMORY_TO_ MEMORY** を選択する場合は、***TxPeriph*** は何も指定できません。

6. FC

6.1 概要

本デバイスは、フラッシュメモリを内蔵しています。
フラッシュメモリのサイズは、TMPM380FY が 256Kbyte、TMPM380FW が 128Kbyte です。

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

ブロック構成は、デバイスのデータシートを参照してください。

全ドライバ API は、アプリで使用する API 定義、マクロ、データタイプ、構造を格納する以下のファイルで構成されています。
Libraries\TX03_Periph_Driver\src\tmpm380_fc.c
\Libraries\TX03_Periph_Driver \inc\tmpm380_fc.h

6.2 API 関数

6.2.1 関数一覧

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

6.2.2 関数の種類

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

6.2.3 関数仕様

6.2.3.1 FC_SetSecurityBit

セキュリティビットの設定

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

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

機能:
1) 書き込み/消去プロテクト用のすべてのプロテクトビット (PSRA<BLn>, n=0,1)を”1” にします。
2) FCSECBIT<SECBIT>を”1”にします。

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

したがって、この API を使用する場合は、注意して実行してください。

FCSECBIT<SECBIT>はパワーオンリセットで初期化されます。

戻り値:
なし

6.2.3.2 FC_GetSecurityBit

セキュリティビットの設定状態の取得

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

引数:
なし

機能:
セキュリティビットの設定状態を取得します。

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

6.2.3.3 FC_GetBusyState

自動動作状態の取得

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

引数:
なし

機能:
自動動作状態を取得します。

戻り値:
BUSY: 自動動作中
DONE: 自動動作終了

6.2.3.4 FC_GetBlockProtectState

ブロックのプロテクト状態の取得。

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

引数:
BlockNum: ブロック番号を選択します。
- **FC_BLOCK_0** block 0 (M380FY)
- **FC_BLOCK_1** block 1 (M380FY)
- **FC_BLOCK_2** block 2 (M380FY, M380FW)
- **FC_BLOCK_3** block 3 (M380FY, M380FW)
- **FC_BLOCK_4** block 4 (M380FY, M380FW)
- **FC_BLOCK_5** block 5 (M380FY, M380FW)

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:
DISABLE: プロテクト状態ではない。
ENABLE: プロテクト状態

6.2.3.5 FC_ProgramBlockProtectState

ブロックのプロテクト設定

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

引数:
BlockNum: ブロック番号を選択します。
- **FC_BLOCK_0** block 0 (M380FY)
- **FC_BLOCK_1** block 1 (M380FY)
- **FC_BLOCK_2** block 2 (M380FY, M380FW)
- **FC_BLOCK_3** block 3 (M380FY, M380FW)
- **FC_BLOCK_4** block 4 (M380FY, M380FW)
- **FC_BLOCK_5** block 5 (M380FY, M380FW)

機能:

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

戻り値:
FC_SUCCESS: プロテクト設定の成功
FC_ERROR_PROTECTED: プロテクト設定の失敗(すでにプロテクト済の場合は再度プロテクト設定を行いません)
FC_ERROR_OVER_TIME: プロテクト設定の失敗(自動動作のタイムアウト)

6.2.3.6 FC_EraseBlockProtectState

プロテクトの解除

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

引数:
BlockGroup: ブロックグループを指定してください。
- **FC_BLOCK_GROUP_1** block 4, block 5
- **FC_BLOCK_GROUP_0** 上記以外のブロック

機能:
プロテクトビットを"0"にすることでプロテクトを解除します。

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

6.2.3.7 FC_WritePage

ページ単位の書き込み

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

引数:
PageAddr: ページの開始アドレスを指定します。
Data: 書き込むデータバッファへのポインタを指定します。サイズは 256Byte 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することはありません。

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

戻り値:
FC_SUCCESS: 書き込み成功
FC_ERROR_PROTECTED: 書き込み失敗(ブロックにプロテクトが設定されている)

FC_ERROR_OVER_TIME: 書き込みの失敗(自動動作のタイムアウト)

6.2.3.8 FC_EraseBlock

ブロック単位の消去

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

引数:
BlockAddr: ブロック開始アドレスを指定します。

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

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

6.2.3.9 FC_EraseChip

チップ消去

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

引数:
なし

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

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

6.2.4 データ構造

なし.

7. GPIO

7.1 概要

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

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

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm380_gpio.c
/Libraries/TX03_Periph_Driver/inc/tmpm380_gpio.h

7.2 API 関数

7.2.1 関数一覧

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

7.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています。:

1) 入出力ポートへの書き込み/読み出し:
   GPIO_ReadData(), GPIO_ReadDataBit(), GPIO_WriteData(), GPIO_WriteDataBit()
2) 入出力ポートの初期化と設定:
   GPIO_SetOutput(), GPIO_SetInput(),GPIO_SetOutputEnableReg(),
   GPIO_SetInputEnableReg(), GPIO_SetPullUp(), GPIO_SetPullDown(),
   GPIO_SetOpenDrain(), GPIO_Init()
3) その他:
   GPIO_EnableFuncReg(), GPIO_DisableFuncReg()

7.2.3 関数仕様

7.2.3.1 GPIO_ReadData

DATA データレジスタの読み込み

関数のプロトタイプ宣言:

uint8_t
GPIO_ReadData(GPIO_Port ***GPIO_x***)

引数:

GPIO_x: GPIO ポートを選択します。

- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI :** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PP:** GPIO port P

機能:

DATA レジスタを読み込みます。

戻り値:

DATA レジスタの値です。

7.2.3.2 GPIO_ReadDataBit

ビット単位での DATA レジスタの読み込み

関数のプロトタイプ宣言:

uint8_t
GPIO_ReadDataBit(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***)

引数:

GPIO_x: GPIO ポートを選択します。

- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI :** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PL:** GPIO port L

- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PP:** GPIO port P

Bit_x: GPIO 端子を選択します。
- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2
- **GPIO_BIT_3:** GPIO pin 3
- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7

機能:
ビット単位で DATA データレジスタを読み込みます。

戻り値:
- **GPIO_BIT_VALUE_0**: 0
- **GPIO_BIT_VALUE_1**: 1

7.2.3.3 GPIO_WriteData

DATA レジスタへの書き込み

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

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI :** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PP:** GPIO port P

Data: DATA レジスタに書き込む値を設定します。

機能:
DATA レジスタへ指定された値を書き込みます。

戻り値:
なし

7.2.3.4 GPIO_WriteDataBit

ビット単位での DATA レジスタの書き込み

関数のプロトタイプ宣言:

```
void
GPIO_WriteDataBit(GPIO_Port GPIO_x,
                  uint8_t Bit_x,
                  uint8_t BitValue)
```

引数:

GPIO_x: GPIO ポートを選択します。

- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI :** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PP:** GPIO port P

Bit_x: GPIO 端子を選択します。

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

BitValue: GPIO 端子値

- **GPIO_BIT_VALUE_0**: 0
- **GPIO_BIT_VALUE_1**: 1

機能:

ビット単位で DATA データレジスタを書き込みます。

戻り値:

なし

7.2.3.5 GPIO_Init

GPIO ポートの初期設定

関数のプロトタイプ宣言:

```
void
GPIO_Init(GPIO_Port  GPIO_x,
          uint8_t Bit_x,
          GPIO_InitTypeDef * GPIO_InitStruct)
```

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI :** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PP:** GPIO port P

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

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

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

戻り値:
なし

7.2.3.6 GPIO_SetOutput
出力ポートの設定

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

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

- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI :** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PP:** GPIO port P

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

機能:
出力ポートに設定します。

戻り値:
なし

7.2.3.7 GPIO_SetInput
入力ポートの設定

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

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI :** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PP:** GPIO port P

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

- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2
- **GPIO_BIT_3:** GPIO pin 3
- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7
- **GPIO_BIT_ALL:** すべての GPIO 端子

機能:
入力ポートに設定します。

戻り値:
なし

7.2.3.8 GPIO_SetOutputEnableReg

出力ポートの許可/禁止設定

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

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI :** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PP:** GPIO port P

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

NewState:

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

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

戻り値:
なし

7.2.3.9 GPIO_SetInputEnableReg

入力ポートの許可/禁止設定

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

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI :** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PP:** GPIO port P

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

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

機能:

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

戻り値:
なし

7.2.3.10 GPIO_SetPullUp

プルアップポートの設定

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

引数:
GPIO_x: GPIO ポートを選択します。

- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI :** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PP:** GPIO port P

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

- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2
- **GPIO_BIT_3:** GPIO pin 3
- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7
- **GPIO_BIT_ALL:** すべての GPIO 端子

NewState:

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

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

戻り値:
なし

7.2.3.11 GPIO_SetPullDown

プルダウンポートの設定

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

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI :** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PP:** GPIO port P

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

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

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

戻り値:
なし

7.2.3.12 GPIO_SetOpenDrain

CMOS/オープンドレインポートの設定

関数のプロトタイプ宣言:

```
void
GPIO_SetOpenDrain(GPIO_Port GPIO_x,
                  uint8_t Bit_x,
                  FunctionalState NewState)
```

引数:

GPIO_x: GPIO ポートを選択します。

- **GPIO_PA:** GPIO port A
- **GPIO_PB:** GPIO port B
- **GPIO_PC:** GPIO port C
- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI :** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PP:** GPIO port P

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

- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1
- **GPIO_BIT_2:** GPIO pin 2
- **GPIO_BIT_3:** GPIO pin 3
- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7
- **GPIO_BIT_ALL:** すべての GPIO 端子

NewState:

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

機能:

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

戻り値:

なし

7.2.3.13 GPIO_EnableFuncReg

機能ポートの有効設定

関数のプロトタイプ宣言:

void
GPIO_EnableFuncReg(GPIO_Port ***GPIO_x***,
uint8_t ***FuncReg_x***,
uint8_t ***Bit_x***)

引数:

GPIO_x: GPIO ポートを選択します。
- **GPIO_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_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PP:** GPIO port P

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

機能:
GPIO 端子の機能を有効に設定します。

戻り値:
なし

7.2.3.14 GPIO_DisableFuncReg

機能ポートの無効設定

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

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_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_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PL:** GPIO port L
- **GPIO_PM:** GPIO port M
- **GPIO_PN:** GPIO port N
- **GPIO_PP:** GPIO port P

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

機能:
GPIO 端子の機能を無効に設定します。

戻り値:
なし

7.2.4 データ構造

7.2.4.1 GPIO_InitTypeDef

メンバ:

uint8_t
IOMode　ポートの入出力設定

- **GPIO_INPUT:** 入力ポートに設定
- **GPIO_OUTPUT:** 出力ポートに設定
- **GPIO_IO_MODE_NONE:** 入出力モードを変更しない

uint8_t

PullUp　プルアップポートの許可/禁止設定

- **GPIO_PULLUP_ENABLE:** プルアップ許可
- **GPIO_PULLUP_DISABLE:** プルアップ禁止
- **GPIO_PULLUP_NONE:** プルアップ機能が無い、または設定変更しない

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

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

8. OFD

8.1 概要

本デバイスは周波数検知回路(OFD)を内蔵しています。この回路は、クロックの異常状態や停止状態を検出するとリセットを発生する回路です。

OFD ドライバ API は、OFD 動作のの許可/禁止、検知周波数設定、OFD 回路の状態の取得などを行う関数セットです。

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm380_ofd.c
/Libraries/TX03_Periph_Driver/inc/tmpm380_ofd.h

8.2 API 関数

8.2.1 関数一覧

- void OFD_SetRegWriteMode(FunctionalState NewState);
- void OFD_Enable(void);
- void OFD_Disable(void);
- void OFD_SetDetectionFrequency(uint8_t HigherDetectionCount,
  uint8_t LowerDetectionCount);
- void OFD_Reset(FunctionalState NewState);
- OFD_Status OFD_GetStatus(void);

8.2.2 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 回路の初期化と設定:
   OFD_SetRegWriteMode(), OFD_SetDetectionFrequency(), OFD_Enable(), OFD_Disable()
2) OFD 動作状態、周波数異常検知フラグの取得:
   OFD_GetStatus()
3) OFD リセットの許可/禁止:
   OFD_Reset ()

8.2.3 関数仕様

8.2.3.1 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 : 以下から、OFDCR2/OFDMN/OFDMX 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:
本関数は、***NewState*** が **ENABLE** の時に、OFDCR2/OFDMN/OFDMX レジスタの書き込みを許可し、**DISABLE** の時に書き込みを禁止します。

戻り値:
なし

8.2.3.2 OFD_Enable

OFD 動作の許可

関数のプロトタイプ宣言:
void
OFD_Enable(void)

引数:
なし

機能:
OFD 動作を許可します。

戻り値:
なし

8.2.3.3 OFD_Disable

OFD 動作の禁止

関数のプロトタイプ宣言:
void
OFD_Disable(void)

引数:
なし

機能:
OFD 動作を禁止します。

戻り値:
なし

8.2.3.4 OFD_SetDetectionFrequency

検知周波数の上限下限値設定

関数のプロトタイプ宣言:
void

OFD_SetDetectionFrequency(uint8_t ***HigherDetectionCount***,
uint8_t ***LowerDetectionCount***)

引数:
HigherDetectionCount: 検出周波数上限値
LowerDetectionCount: 検出周波数下限値

機能:
本関数は、検出周波数上限値、検出周波数下限値を設定します。

戻り値:
なし

8.2.3.5 OFD_Reset

OFD リセットの許可/禁止

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

引数:
NewState : 以下から、OFD リセットの許可/禁止を選択します。

- **ENABLE:** 許可
- **DISABLE:** 禁止

機能:
OFD リセットの許可/禁止を設定します。

戻り値:
なし

8.2.3.6 OFD_GetStatus

OFD 動作状態、周波数異常検知フラグの取得

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、周波数異常検知フラグを取得します。

戻り値:
OFD_Status: OFD ステータスの構造体です。(詳細は"データ構造"を参照)

8.2.4 データ構造

8.2.4.1 OFD_Status

メンバ:

uint32_t
All: データ
ビットフィールド:
uint32_t
FrequencyError: 1　　周波数異常検知フラグ
uint32_t
OFDBusy: 1　　OFD 動作状態

9. RMC

9.1 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(32.768KHz)とタイマ出力を選択可能。
• ノイズキャンセル時間を調整可能。
• リーダ検出。
• 最大 72bit まで一括受信。

RMC ドライバ API ではチャネル毎の機能セットが提供されています。

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm380_rmc.c
/Libraries/TX03_Periph_Driver/inc/tmpm380_rmc.h

9.2 API 関数

9.2.1 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***)

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) 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) 受信状態の取得、受信データの取得:
   RMC_GetINTFactor(), RMC_GetLeader(), RMC_GetRxData()

9.2.3 関数仕様

補足: 引数"TSB_RMC_TypeDef * ***RMCx***"は **TSB_RMC0** を指定してください。

9.2.3.1 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:
RMCEN<RMCEN>ビットを 1 に設定し、RMC 機能を許可します。

戻り値:
なし

9.2.3.2 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:
RMCEN<RMCEN>ビットを 0 クリアし、RMC 機能を禁止します。

戻り値:
なし

9.2.3.3 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”を参照してください。)

機能:
RMC チャネルの初期化を行います。

戻り値:
なし

9.2.3.4 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

9.2.3.5 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:
RMCRBUF1~RMCRBUF 3 と RMCRSTAT<RMCRNUM>から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)

9.2.3.6 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。
この関数は RMCRCR1、RMCRCR2<RMCLIEN>、RMCRCR2<RMCLD>の設定を行います。
詳細は MCU データシートを参照してください。

戻り値:
なし

9.2.3.7 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

9.2.3.8 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

9.2.3.9 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

9.2.3.10 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

9.2.3.11 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

9.2.3.12 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

9.2.3.13 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)

9.2.3.14 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

9.2.3.15 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

9.2.3.16 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

9.2.4 データ構造

9.2.4.1 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)

9.2.4.2 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。

9.2.4.3 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。

9.2.4.4 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

10. RTC

10.1 概要

RTC の機能概略は以下です。

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

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

本ドライバ は、アプリで使用する API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm380_rtc.c
/Libraries/ TX03_Periph_Driver/inc/tmpm380_rtc.h

10.2 API 関数

10.2.1 関数一覧

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

- void RTC_SetAlarmOutput(uint8_t ***Output***);
- void RTC_ResetClockSec(void);
- RTC_ReqState RTC_GetResetClockSecReq(void);
- void RTC_ResetAlarm(void);
- void RTC_SetDateValue(RTC_DateTypeDef * ***DateStruct***);
- void RTC_GetDateValue(RTC_DateTypeDef * ***DateStruct***);
- void RTC_SetTimeValue(RTC_TimeTypeDef * ***TimeStruct***);
- void RTC_GetTimeValue(RTC_TimeTypeDef * ***TimeStruct***);
- void RTC_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***);

10.2.2 関数の種類

関数は、主に以下の 5 種類に分かれています。:

1) RTC 機能の年月日の設定:
   RTC_SetDay(), RTC_GetDay(), RTC_SetDate(), RTC_GetDate(), RTC_SetMonth(), RTC_GetMonth(), RTC_SetYear(), RTC_GetYear(), RTC_SetLeapYear(), RTC_GetLeapYear(), RTC_SetDateValue(), RTC_GetDateValue()
2) RTC 機能の時間の設定:
   RTC_SetSec(), RTC_GetSec(), RTC_SetMin(),RTC_GetMin(),RTC_SetHour24(), RTC_SetHour12(), RTC_GetHour(), RTC_SetHourMode(), RTC_GetHourMode, RTC_GetAMPM(), RTC_SetTimeValue(), RTC_GetTimeValue()
3) RTC(clock)の設定:
   RTC_EnableClock(), RTC_DisableClock(), RTC_SetTimeAdjustReq(), RTC_GetTimeAdjustReq(), RTC_ResetClockSec(), RTC_GetResetClockSec(), RTC_SetClockValue(), RTC_GetClockValue()
4) RTC(alarm)の設定:
   RTC_EnableAlarm(), RTC_DisableAlarm(), RTC_ResetAlarm(), RTC_SetAlarmValue(), RTC_GetAlarmValue()
5) その他:
   RTC_SetAlarmOutput(), RTC_SetRTCINT()

10.2.3 関数仕様

10.2.3.1 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

10.2.3.2 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

10.2.3.3 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 までの分桁を設定します。

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

戻り値:
なし

10.2.3.4 RTC_GetMin

時計/アラームの分桁読み込み

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

10.2.3.5 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

10.2.3.6 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

10.2.3.7 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

10.2.3.8 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

10.2.3.9 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

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

10.2.3.10 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

10.2.3.11 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

10.2.3.12 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

10.2.3.13 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

10.2.3.14 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

10.2.3.15 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

10.2.3.16 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

10.2.3.17 RTC_SetHourMode

24 時間時計/12 時間時計の選択

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

引数:
HourMode: 時間モードを選択します。

- **RTC_12_HOUR_MODE**: 12 時間時計
- **RTC_24_HOUR_MODE**.: 24 時間時計

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

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

戻り値:
なし

10.2.3.18 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

10.2.3.19 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

10.2.3.20 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。

10.2.3.21 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

10.2.3.22 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

10.2.3.23 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

10.2.3.24 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

10.2.3.25 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

10.2.3.26 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

10.2.3.27 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

10.2.3.28 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

10.2.3.29 RTC_ResetClockSec

時計秒カウンタのリセット

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

引数:
なし

機能:
時計秒カウンタをリセットします。

戻り値:
なし

10.2.3.30 RTC_GetResetClockSecReq

時計秒カウンタのリセット要求状態の読み込み

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

引数:
なし

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

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

10.2.3.31 RTC_ResetAlarm

RTC アラームのリセット

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

引数:
なし

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

戻り値:
なし

10.2.3.32 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

10.2.3.33 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

10.2.3.34 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

10.2.3.35 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

10.2.3.36 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

10.2.3.37 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

10.2.3.38 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()** が実行されます。

戻り値:
なし

10.2.3.39 RTC_GetAlarmValue

アラームの日時の読み込み

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()** を実行します。

戻り値:
なし

10.2.4 データ構造

10.2.4.1 RTC_DateTypeDef

メンバ:
uint8_t
LeapYear **:** うるう年を設定します:
- **RTC_LEAP_YEAR_0:** 現在の年(今年)がうるう年
- **RTC_LEAP_YEAR_1:** 現在がうるう年から 1 年目
- **RTC_LEAP_YEAR_2:** 現在がうるう年から 2 年目
- **RTC_LEAP_YEAR_3:** 現在がうるう年から 3 年目

uint8_t
Year 年桁の値(0～99)。

uint8_t
Month 月桁の値(1～12)。

uint8_t
Date の日桁の値(1～31)。

uint8_t
Day 週桁の値を以下。
- **RTC_SUN:** 日曜日
- **RTC_MON:** 月曜日
- **RTC_TUE:** 火曜日
- **RTC_WED:**水曜日
- **RTC_THU:**木曜日
- **RTC_FRI:** 金曜日
- **RTC_SAT:** 土曜日

10.2.4.2 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。

10.2.4.3 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)。

11. SBI

11.1 概要

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

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

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm380_sbi.c
/Libraries/TX03_Periph_Driver/inc/tmpm380_sbi.h

11.2 API 関数

11.2.1 関数一覧

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

11.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。:

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

4) その他:
SBI_SWReset(), SBI_SetIdleMode(), SBI_EnableI2CfreeDataMode()

11.2.3 関数仕様

補足: 引数"TSB_SBI_TypeDef* ***SBIx***"は **TSB_SBI0** または **TSB_SBI1** のいずれかを指定してください。

11.2.3.1 SBI_Enable

シリアルバスインタフェース動作の許可

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

引数:
SBIx: SBI チャネルを指定します。

機能:
SBI 動作を有効にします。

戻り値:
なし

11.2.3.2 SBI_Disable

シリアルバスインタフェース動作の禁止

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

引数:
SBIx: SBI チャネルを指定します。

機能:
SBI 動作を無効にします。

戻り値:
なし

11.2.3.3 SBI_SetI2CACK

I2C バスモードにおける ACK 選択

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

引数:

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

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

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

戻り値:
なし

11.2.3.4 SBI_InitI2C

I2C バスモードにおける通信の初期化

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

引数:
SBIx: SBI チャネルを指定します。

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

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

戻り値:
なし

11.2.3.5 SBI_SetI2CBitNum

I2C バスモードにおける転送ビット数の選択

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

引数:
SBIx: SBI チャネルを指定します。

I2CBitNum: 転送ビット数(1～8)を選択します。

- **SBI_I2C_DATA_LEN_8:** データ長 8
- **SBI_I2C_DATA_LEN_1:** データ長 1
- **SBI_I2C_DATA_LEN_2:** データ長 2
- **SBI_I2C_DATA_LEN_3:** データ長 3
- **SBI_I2C_DATA_LEN_4:** データ長 4

- **SBI_I2C_DATA_LEN_5:** データ長 5
- **SBI_I2C_DATA_LEN_6:** データ長 6
- **SBI_I2C_DATA_LEN_7:** データ長 7

機能:
転送ビット数を選択します。

戻り値:
なし

11.2.3.6 SBI_SWReset

ソフトウェアリセットの発生

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

引数:
SBIx: SBI チャネルを指定します。

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

戻り値:
なし

11.2.3.7 SBI_ClearI2CINTReq

I2C バスモードにおける INTSBIx 割り込み要求解除

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

引数:
SBIx: SBI チャネルを指定します。

機能:
SBI 割り込み要求を解除します。

戻り値:
なし

11.2.3.8 SBI_GenerateI2CStart

I2C バスモードにおけるスタート状態の発生

関数のプロトタイプ宣言:

void
SBI_GenerateI2CStart(TSB_SBI_TypeDef* ***SBIx***)

引数:
SBIx: SBI チャネルを指定します。

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

戻り値:
なし

11.2.3.9 SBI_GenerateI2CStop

I2C バスモードにおけるストップ状態の発生

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

引数:
SBIx: SBI チャネルを指定します。

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

戻り値:
なし

11.2.3.10 SBI_GetI2CState

I2C バスモードにおける SBI チャネルの状態の読み込み

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

引数:
SBIx: SBI チャネルを指定します。

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

戻り値:
I2C モードでの SBI チャネルの状態

11.2.3.11 SBI_SetIdleMode

IDLE モード時の動作の許可/禁止

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

引数:
SBIx: SBI チャネルを指定します。

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

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

戻り値:
なし

11.2.3.12 SBI_SetSendData

データ送信

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

引数:
SBIx: SBI チャネルを指定します。

Data: 送信データ。(最大値は 0xFF です)

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

戻り値:
なし

11.2.3.13 SBI_GetReceiveData

データ受信

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

引数:
SBIx: SBI チャネルを指定します。

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

戻り値:
受信データ

11.2.3.14 SBI_SetI2CFreeDataMode

アドレス認識モードの指定

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

引数:
SBIx: SBI チャネルを指定します。

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

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

戻り値:
なし

11.2.4 データ構造

11.2.4.1 SBI_InitI2CTypeDef

メンバ:

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

uint32_t
***I2CDataLen*:** I2C モードにおける SBI チャネルの転送ビット数を指定します。
- **SBI_I2C_DATA_LEN_8:** データ長 8
- **SBI_I2C_DATA_LEN_1:** データ長 1
- **SBI_I2C_DATA_LEN_2:** データ長 2
- **SBI_I2C_DATA_LEN_3:** データ長 3
- **SBI_I2C_DATA_LEN_4:** データ長 4
- **SBI_I2C_DATA_LEN_5:** データ長 5
- **SBI_I2C_DATA_LEN_6:** データ長 6
- **SBI_I2C_DATA_LEN_7:** データ長 7

uint32_t
***I2CClkDiv*:** I2C 転送のソースクロックを選択します。

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

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

- **ENABLE:** 有効。
- **DISABLE:** 無効。

11.2.4.2 SBI_I2CState

メンバ:

uint32_t
***All*:** I2C モードの全ての状態

ビットフィールド:

uint32_t
***LastRxBit*:** 最終受信ビットモニタ

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

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

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

uint32_t
***INTReq*:** 割り込み要求状態モニタ

uint32_t
***BusState*:** バス状態モニタ

uint32_t
***TRx*:** 送信/受信選択状態モニタ

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

12. SSP

12.1 概要

本デバイスは、同期式シリアルインターフェースを (SSP：Synchronous Serial Port) を 2 チャネル内蔵しています。(SSP0, SSP1)

同期式シリアルインターフェースは、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ーフェースで行います。

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

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm380_ssp.c
/Libraries/TX03_Periph_Driver/inc/tmpm380_ssp.h

12.2 API 関数

12.2.1 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***)

12.2.2 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) SSP_Init()を用いた共通関数
   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()

12.2.3 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**

12.2.3.1 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

12.2.3.2 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

12.2.3.3 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)
```
typedef struct {
    SSP_FrameFormat FrameFormat;
    uint8_t PreScale;
    uint8_t ClkRate;
    SSP_ClkPolarity ClkPolarity;
    SSP_ClkPhase ClkPhase;
    uint8_t DataSize;
    SSP_MS_Mode Mode;
} SSP_InitTypeDef;
```

機能:
SSP 通信の初期化を行います。

戻り値:
なし

12.2.3.4 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()** により呼び出されます。

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

12.2.3.5 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()** により呼び出されます。

戻り値:
なし

12.2.3.6 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()** により呼び出されます。

戻り値:
なし

12.2.3.7 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()** により呼び出されます。

戻り値:
なし

12.2.3.8 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()** により呼び出されます。

戻り値:
なし

12.2.3.9 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

12.2.3.10 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

12.2.3.11 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

12.2.3.12 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

12.2.3.13 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

12.2.3.14 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

12.2.3.15 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。

12.2.3.16 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

12.2.3.17 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。

12.2.3.18 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。

12.2.3.19 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。

12.2.3.20 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

12.2.3.21 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

12.2.4 データ構造

12.2.4.1 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 4～16 ビットの間で設定します。

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

- **SSP_SLAVE**: デバイスがスレーブ

12.2.4.2 SSP_INTState

メンバ for this union:

uint32_t
All: 割り込み要因

ビットフィールド:

uint32_t ***OverRun***:	1	オーバー欄割り込み
uint32_t ***TimeOut***:	1	受信タイムアウト
uint32_t ***Rx***:	1	受信
uint32_t ***Tx***:	1	送信
uint32_t ***Reserved***:	28	未使用

13. TMRB

13.1 概要

本デバイスは、8 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 ～ TMRB7)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。

- パルス幅測定
- 外部トリガパルスからのワンショットパルス出力
- 時間差測定

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

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

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm380_tmrb.c
/Libraries/TX03_Periph_Driver/inc/tmpm380_tmrb.h

13.2 API 関数

13.2.1 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***);

13.2.2 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()

13.2.3 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_TB_MPT0, TSB_TB_MPT1, TSB_TB_MPT2

13.2.3.1 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

13.2.3.2 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

13.2.3.3 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

13.2.3.4 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

13.2.3.5 TMRB_SetCaptureTiming

キャプチャタイミングの設定

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

引数:
***TBx*:** TMRB チャネルを指定します。

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

機能:
CaptureTiming が **TMRB_CAPTURE_IN_RISING** の場合、TBxIN 端子入力の立ち上がりでキャプチャレジスタ 0 (TBxCP0)にカウント値を取り込みます。
CaptureTiming が **TMRB_CAPTURE_IN_RISING_FALLING** の場合、TBxIN 端子入力の立ち上がりでキャプチャレジスタ 0 (TBxCP0)にカウント値を取り込み、TBxIN 端子入力の立ち下がりでキャプチャレジスタ 1 (TBxCP1)にカウント値を取り込みます。
CaptureTiming が **TMRB_CAPTURE_OUTPUT_EDGE** の場合、16 ビットタイマー致出力(TBxOUT)の立ち上がりでキャプチャレジスタ 0 (TBnCP0)にカウント値を取り込み、TBxOUT の立ち下がりでキャプチャレジスタ 1 (TBnCP1)にカウント値を取り込みます。(TMRB2, TMRB5, TMRB7）
補足: TMRB_CAPTURE_OUTPUT_EDGE は TMRB0~TMRBB まで有効です。

TMRB3~5: TB2OUT
TMRB6~7: TB5OUT
TMRB0~2: TB7OUT

戻り値:
なし

13.2.3.6 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

13.2.3.7 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
}
```

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

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

戻り値:
なし

13.2.3.9 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。

13.2.3.10 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 の校正と ***ClkDiv***(詳細は"データ構造"を参照)の値によります。

戻り値:
なし

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

13.2.3.11 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

13.2.3.12 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:
キャプチャされた値

13.2.3.13 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

13.2.3.14 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

13.2.3.15 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

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

機能:
TMRB1~TMRB3 を同期モードに設定すると、TMRB0 のスタートに同期して動作がスタートし、TMRB5~TMRB7 を同期モードに設定すると、TMRB4 のスタートに同期して動作がスタートします。

戻り値:
なし

補足:
同期モードを使用するために、TMRB0, TMRB4 のカウントを開始する前に、**TMRB_SetRunState()** によって TMRB1~TMRB3、TMRB5~TMRB7 をスタートしてください。

13.2.3.16 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:** 許可。
- **DISABLE:** 禁止。

***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

13.2.3.17 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

13.2.3.18 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

13.2.4 データ構造

13.2.4.1 TMRB_InitTypeDef

メンバ:

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

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

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

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

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

13.2.4.2 TMRB_FFOutputTypeDef

メンバ:

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

uint32_t
FlipflopReverseTrg: 以下から、フリップフロップの反転トリガを選択します。
- **TMRB_DISALBE_FLIPFLOP**: 反転トリガを無効にします。

- **TMRB_FLIPFLOP_TAKE_CATPURE_0**: アップカウンタの値がキャプチャレジスタ 0 に取り込まれた時にタイマフリップフロップを反転します。
- **TMRB_FLIPFLOP_TAKE_CATPURE_1**: アップカウンタの値がキャプチャレジスタ 1 に取り込まれた時にタイマフリップフロップを反転します。
- **TMRB_FLIPFLOP_MATCH_TRAILINGTIMING**: アップカウンタと周期との一致時にタイマフリップフロップを反転します。
- **TMRB_FLIPFLOP_MATCH_LEADINGTIMING**: アップカウンタとデューティとの一致時にタイマフリップフロップを反転します。

13.2.4.3 TMRB_INTFactor

メンバ:

uint32_t
All: TMRB 割り込み要因

ビットフィールド:

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

uint32_t
MatchTrailingTiming: 1周期との一致検出

uint32_t
OverFlow : 1 オーバーフロー

uint32_t
Reserverd : 29 -

14. SIO/UART

14.1 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)と 7, 8, 9 ビット長の UART モード(非同期通信)を実装しています。9 ビット UART モードでは、シリアルリンク(マルチコントローラ・システム) でマスタコントローラがスレーブコントローラを起動するときにウェイクアップ機能が使用されます。

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

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm380_uart.c
/Libraries/TX03_Periph_Driver/inc/tmpm380_uart.h

14.2 API 関数

14.2.1 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_TRxDisable TRxAutoDisable)
- void UART_RxFIFOINTCtrl(TSB_SC_TypeDef * UARTx, FunctionalState NewState)
- void UART_TxFIFOINTCtrl(TSB_SC_TypeDef * UARTx, FunctionalState NewState)
- void UART_RxFIFOByteSel(TSB_SC_TypeDef * UARTx, uint32_t BytesUsed)
- void UART_RxFIFOFillLevel(TSB_SC_TypeDef * UARTx,uint32_t RxFIFOLevel)
- void UART_RxFIFOINTSel(TSB_SC_TypeDef * UARTx, uint32_t RxINTCondition)
- void UART_RxFIFOClear(TSB_SC_TypeDef * UARTx)
- void UART_TxFIFOFillLevel(TSB_SC_TypeDef * UARTx, uint32_t TxFIFOLevel)
- void UART_TxFIFOINTSel(TSB_SC_TypeDef * UARTx, uint32_t TxINTCondition)
- void UART_TxFIFOClear(TSB_SC_TypeDef * UARTx)
- uint32_t UART_GetRxFIFOFillLevelStatus(TSB_SC_TypeDef * UARTx)
- uint32_t UART_GetRxFIFOOverRunStatus(TSB_SC_TypeDef * UARTx)
- uint32_t UART_GetTxFIFOFillLevelStatus(TSB_SC_TypeDef * UARTx)

- uint32_t UART_GetTxFIFOUnderRunStatus(TSB_SC_TypeDef * UARTx)
- void SIO_Enable(TSB_SC_TypeDef * SIOx)
- void SIO_Disable(TSB_SC_TypeDef * SIOx)
- uint8_t SIO_GetRxData(TSB_SC_TypeDef * SIOx)
- void SIO_SetTxData(TSB_SC_TypeDef * SIOx, uint8_t Data)
- void SIO_Init(TSB_SC_TypeDef * SIOx, uint32_t IOClkSel, SIO_InitTypeDef * InitStruct)

14.2.2 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_Init(), UART_DefaultConfig(),
   SIO_Enable(), SIO_Disable(), SIO_Init()
2) 送受信設定とエラー確認:
   UART_GetBufState(), UART_GetRxData(), UART_SetTxData(),
   UART_GetErrState(), SIO_GetRxData(), SIO_SetTxData()
3) その他:
   UART_SWReset(), UART_SetWakeUpFunc(), UART_SetIdleMode()
4) FIFO 制御:
   UART_FIFOConfig(), UART_SetFIFOTransferMode(), UART_RxFIFOINTCtrl(),
   UART_TxFIFOINTCtrl(), UART_RxFIFOByteSel(), UART_RxFIFOFillLevel(),
   UART_RxFIFOINTSel(), UART_RxFIFOClear(), UART_TxFIFOFillLevel(),
   UART_TxFIFOINTSel(), UART_TxFIFOClear(), UART_GetRxFIFOFillLevelStatus(),
   UART_GetRxFIFOOverRunStatus(), UART_GetTxFIFOFillLevelStatus(),
   UART_GetTxFIFOUnderRunStatus()

14.2.3 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**
また、“TSB_SC_TypeDef* ***SIOx***” は、以下から選択してください。
SIO0, **SIO1**, **SIO2**, **SIO3**, **SIO4**

14.2.3.1 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

14.2.3.2 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

14.2.3.3 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

14.2.3.4 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

14.2.3.5 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

14.2.3.6 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

14.2.3.7 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

14.2.3.8 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

14.2.3.9 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

14.2.3.10 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

14.2.3.11 UART_SetIdleMode

IDLE 時の動作

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

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

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

戻り値:
なし

14.2.3.12 UART_FIFOConfig

FIFO の許可

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

引数:
UARTx: UART チャネルを指定します。
NewState: FIFO の許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

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

戻り値:
なし

14.2.3.13 UART_SetFIFOTransferMode

転送モードの選択

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

引数:
UARTx: UART チャネルを指定します。
TransferMode: 転送モードを選択します。
- **UART_TRANSFER_PROHIBIT** : 転送禁止
- **UART_TRANSFER_HALFDPX_RX** : 半二重(受信)
- **UART_TRANSFER_HALFDPX_TX** : 半二重(送信)
- **UART_TRANSFER_FULLDPX** : 全二重

機能:
転送モードを選択します。

戻り値:
なし

14.2.3.14 UART_TRxAutoDisable

送信/受信の自動禁止

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

引数:
UARTx: UART チャネルを指定します。
TRxAutoDisable: 送信/受信の自動禁止機能を制御します。
- **UART_RXTXCNT_NONE**: なし
- **UART_RXTXCNT_AUTODISABLE**: 自動禁止

機能:
送信/受信の自動禁止機能を制御します。

戻り値:
なし

14.2.3.15 UART_RxFIFOINTCtrl

受信 FIFO 使用時の受信割り込み許可

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

引数:
UARTx: UART チャネルを指定します。
NewState: 受信 FIFO 使用時の受信割り込みの許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

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

戻り値:
なし

14.2.3.16 UART_TxFIFOINTCtrl

送信 FIFO 使用時の送信割り込み許可

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

引数:

UARTx: UART チャネルを指定します。
NewState: 送信 FIFO 使用時の送信割り込みの許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

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

戻り値:
なし

14.2.3.17 UART_RxFIFOByteSel

受信 FIFO 使用バイト数

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

引数:
UARTx: UART チャネルを指定します。
BytesUsed: 受信 FIFO 使用バイト数を設定します。
- **UART_RXFIFO_MAX**: 最大
- **UART_RXFIFO_RXFLEVEL**: 受信 FIFO の FILL レベルに同じ

機能:
受信 FIFO 使用バイト数を設定します。

戻り値:
なし

14.2.3.18 UART_RxFIFOFillLevel

受信割り込みが発生する受信 FIFO の fill レベルの設定

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

引数:
UARTx: UART チャネルを指定します。
RxFIFOLevel: 受信 FIFO の fill レベルを選択します。

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

機能:

受信割り込みが発生する受信 FIFO の fill レベルを選択します。

戻り値:
なし

14.2.3.19 UART_RxFIFOINTSel

受信割り込み発生条件の選択

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

引数:
UARTx: UART チャネルを指定します。
RxINTCondition: 受信 割り込み発生条件を選択します。

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

機能:
受信割り込み発生条件を選択します。

戻り値:
なし

14.2.3.20 UART_RxFIFOClear

受信 FIFO クリア

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

引数:
UARTx: UART チャネルを指定します。

機能:
受信 FIFO をクリアします。

戻り値:
なし

14.2.3.21 UART_TxFIFOFillLevel

送信割り込みが発生する送信 FIFO の fill レベルの設定

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

uint32_t ***TxFIFOLevel***)

引数:
UARTx: UART チャネルを指定します。
TxFIFOLevel: 受信 FIFO の fill レベルを選択します。

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

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

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

14.2.3.22 UART_TxFIFOINTSel

送信割り込み発生条件の選択

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

引数:
UARTx: UART チャネルを指定します。
TxINTCondition: 受信 割り込み発生条件を選択します。

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

機能:
送信割り込み発生条件を選択します。

機能:
送信割り込み発生条件を選択します。

戻り値:
なし

14.2.3.23 UART_TxFIFOClear

送信 FIFO クリア

関数のプロトタイプ宣言:
void

UART_TxFIFOClear (TSB_SC_TypeDef * ***UARTx***)

引数:
UARTx: UART チャネルを指定します。

機能:
送信 FIFO をクリアします。

戻り値:
なし

14.2.3.24 UART_GetRxFIFOFillLevelStatus

受信 FIFO の fill レベルの取得

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

引数:
UARTx: UART チャネルを指定します。

機能:
受信 FIFO の fill レベルを取得します。

戻り値:
- **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

14.2.3.25 UART_GetRxFIFOOverRunStatus

受信 FIFO オーバーラン状態の取得

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

引数:
UARTx: UART チャネルを指定します。

機能:
受信 FIFO オーバーラン状態を取得します。

戻り値:
UART_RXFIFO_OVERRUN: オーバーラン発生

14.2.3.26 UART_GetTxFIFOFillLevelStatus

送信 FIFO の fill レベルの取得

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

引数:
UARTx: UART チャネルを指定します。

機能:
送信 FIFO の fill レベルの取得

戻り値:
- **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

14.2.3.27 UART_GetTxFIFOUnderRunStatus

送信 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_ GetTxFIFOUnderRunStatus (TSB_SC_TypeDef* ***UARTx***);

引数:
UARTx: UART チャネルを指定します。

機能:
送信 FIFO オーバーラン状態を取得します。

戻り値:
UART_TXFIFO_UNDERRUN: オーバーラン発生

14.2.3.28 SIO_Enable

SIO 動作の許可

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

引数:
SIOx: SIO チャネルを指定します。

機能:
SIO 動作を許可します。

戻り値:
なし

14.2.3.29 SIO_Disable

SIO 動作の禁止

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

引数:
SIOx: SIO チャネルを指定します。

機能:
SIO 動作を禁止します。

戻り値:
なし

14.2.3.30 SIO_GetRxData

受信用バッファの取得

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

引数:
SIOx: SIO チャネルを指定します。

機能:
受信用バッファを取得します。

戻り値:
受信用バッファ(値の範囲は 0x00 ~ 0xFF です)

14.2.3.31 SIO_SetTxData

送信用バッファの設定

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

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

機能:

送信用バッファを指定します。

戻り値:
なし

14.2.3.32 SIO_Init

SIO チャネルの初期化

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

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

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

機能:
ボーレート、転送方向、転送モードなどの初期設定を行います。

戻り値:
なし

14.2.4 データ構造

14.2.4.1 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

*: 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。
**: UART_NONE_FLOW_CTRL のみサポートしています。

14.2.4.2 SIO_InitTypeDef

メンバ:
uint32_t
InputClkEdge: 入力クロックエッジを選択します。
- **SIO_SCLKS_TXDF_RXDR**: SCLKx の立ち下がりエッジで送信バッファのデータを 1bit ずつ TXDx 端子へ出力します。SCLKx の立ち上がりエッジで RXDx 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、SCLKx は High レベルからスタートします。
- **SIO_SCLKS_TXDR_RXDF**: SCLKx の立ち上がりエッジで送信バッファのデータを 1bit ずつ TXDx 端子へ出力します。SCLKx の立ち下がりエッジで RXDx 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、SCLKx は Low レベルからスタートします。

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

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

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

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

- **SIO_ENABLE_RX**: 受信許可

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

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

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

15. VLTD

15.1 概要

電圧検出回路は、電源電圧の低下を検出し、リセット信号を発生します。

VLTD ドライバ API は、VLTD 機能の許可/禁止、検出電圧の設定、電源電圧の状態の取得を設定する関数セットです。

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm380_vltd.c
/Libraries/TX03_Periph_Driver/inc/tmpm380_vltd.h

15.2 API 関数

15.2.1 関数一覧

- void VLTD_Enable(void);
- void VLTD_Disable(void);
- void VLTD_SetVoltage(uint32_t ***Voltage***);
- uint32_t VLTD_GetStatus(void);

15.2.2 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) VLTD の許可/禁止:
   VLTD_SetVoltage(), VLTD_Enable(), VLTD_Disable()
2) 検出電圧の設定:
   VLTD_GetStatus()

15.2.3 関数仕様

15.2.3.1 VLTD_Enable

電圧検出の許可

関数のプロトタイプ宣言:
void
VLTD_Enable(void)

引数:
なし

機能:
電圧検出を許可します。

戻り値:
なし

15.2.3.2 VLTD_Disable

電圧検出の禁止

関数のプロトタイプ宣言:
void
VLTD_Disable(void)

引数:
なし

機能:
電圧検出を禁止します。

戻り値:
なし

15.2.3.3 VLTD_SetVoltage

検出電圧レベルの選択

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

引数:
***Voltage*:** 以下から検出電圧レベルを選択します。
- **VLTD_DETECT_VOLTAGE_38:** 3.8V ± 0.2V
- **VLTD_DETECT_VOLTAGE_41:** 4.1V ± 0.2V
- **VLTD_DETECT_VOLTAGE_44:** 4.4V ± 0.2V
- **VLTD_DETECT_VOLTAGE_46:** 4.6V ± 0.2V

機能:
検出電圧レベルを選択します。

戻り値:
なし

15.2.3.4 VLTD_GetStatus

電圧検出ステータスの取得

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

引数:
なし

機能:
電圧検出ステータスを取得します。

戻り値:
電圧検出ステータス:
0: 電源電圧は検出電圧以上
1: 電源電圧は検出電圧以下

15.2.4 データ構造:

なし

16. WDT

16.1 概要

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

WDT ドライバの API は、検出時間、カウンタのオーバーフロ時の出力、IDLE モードでの動作設定などの引数等、ウォッチドッグタイマの設定を行う関数を提供します。

本ドライバは、以下のファイルで構成されています。
\Libraries\TX03_Periph_Driver\src\tmpm380_wdt.c
\Libraries/TX03_Periph_Driver\inc\tmpm380_wdt.h

16.2 API 関数

16.2.1 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)

16.2.2 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() functions
2) IDLE モード時の開始・停止:
   WDT_SetIdleMode()

16.2.3 関数仕様

16.2.3.1 WDT_SetDetectTime

WDT 検出時間の設定

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

引数:
***DetectTime*: 以下から検出時間を選択します。**

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

- **WDT_DETECT_TIME_EXP_25:** 2^25/fsys

機能:
WDT の検出時間を設定します。

戻り値:
なし

16.2.3.2 WDT_SetIdleMode

IDLE 時の動作選択

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

引数:
NewState: 以下から IDLE 時の WDT 動作を選択します。
- **ENABLE**: 動作
- **DISABLE**. 停止

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

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

戻り値:
なし

16.2.3.3 WDT_SetOverflowOutput

暴走検出後の動作選択

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

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

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

戻り値:

なし

16.2.3.4 WDT_Init

WDT の初期化

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

引数:
InitStruct: カウンタオーバーフロー発生時の WDT 検出時間、WDT 出力の設定を含む WDT 設定に関する構造体。(詳細は“データ構造:”を参照)

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

戻り値:
なし

16.2.3.5 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

16.2.3.6 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

16.2.3.7 WDT_WriteClearCode

クリアコードの書き込み

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

引数:
なし

機能:
クリアコードをライトします。

戻り値:
なし

16.2.4 データ構造

16.2.4.1 WDT_InitTypeDef

メンバ:

uint32_t
DetectTime 以下から検出時間を選択します。

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

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

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

17. ENC

17.1 概要

本デバイスは、エンコーダ入力回路を 2 チャネル内蔵しています(ENC0/1)。インクリメンタルエンコーダの信号を直接入力し、モータの絶対位置を用意に得ることができます。

エンコーダ入力回路は、エンコーダモード、センサモード(2 種類)、タイマモードの 4 つの動作モードに対応しています。

- インクリメンタルエンコーダおよびホール IC センサ対応（センサ信号を直接入力可能）
- 汎用 24 ビットタイマ機能
- 4 逓倍 (6 逓倍) 回路内蔵
- 回転方向検出回路内蔵
- カウンタ (24 ビット) 内蔵
- コンペア許可／禁止設定可能
- 割り込み出力1本
- 入力信号についてデジタルノイズフィルタ内蔵

ENC 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、比較機能設定、ソフトウェアキャプチャ設定、ステータスリード、ENC カウント値の取得などの機能を提供します。

本ドライバは、以下のファイルで構成されています。
\Libraries\TX03_Periph_Driver\src\tmpm380_enc.c
\Libraries\TX03_Periph_Driver\inc\tmpm380_enc.h

17.2 API 関数

17.2.1 関数一覧

- void ENC_Enable(TSB_EN_TypeDef * ***ENx***);
- void ENC_Disable(TSB_EN_TypeDef * ***ENx***);
- void ENC_Init(TSB_EN_TypeDef * ***ENx***, ENC_InitTypeDef * ***InitStruct***);
- void ENC_SetSWCapture(TSB_EN_TypeDef * ***ENx***, uint32_t ***ENC_Mode***);
- void ENC_ClearCounter (TSB_EN_TypeDef * ***ENx***);
- ENC_FlagStatus ENC_GetENCFlag (TSB_EN_TypeDef * ***ENx***);
- void ENC_SetCounterReload (TSB_EN_TypeDef **ENx*, uint32_t ***ENC_Mode***,
  uint32_t ***PeriodValue***);
- void ENC_SetCompareValue(TSB_EN_TypeDef * ***ENx***,uint32_t ***ENC_Mode***,
  uint32_t ***CompareValue***);
- uint32_t ENC_GetCompareValue(TSB_EN_TypeDef * ***ENx***);
- uint32_t ENC_GetCounterValue(TSB_EN_TypeDef * ***ENx***);

17.2.2 関数の種類

上記関数は 3 つのグループに分けられます。

1) ENC の設定:

ENC_Init(), ENC_ClearCounter(), ENC_SetCounterReload(), ENC_SetSWCapture(), ENC_SetCompareValue()

2) ENC の許可/禁止:
ENC_Enable(), ENC_Disable()

3) ENC 状態、またはデータリード:
ENC_GetENCFlag(), ENC_GetCounterValue(), ENC_GetCompareValue()

17.2.3 関数仕様

補足: 下記の全 API において、パラメータ “TSB_EN_TypeDef * ***ENx***” は 以下のいずれかを選択してください。
EN0, EN1

17.2.3.1 ENC_Enable

エンコーダ動作の許可

関数のプロトタイプ宣言:
void
ENC_Enable(TSB_EN_TypeDef * ***ENx***)

引数:
ENx: ENC チャネルを選択します。

機能:
エンコーダ動作を許可します。

戻り値:
なし

17.2.3.2 ENC_Disable

エンコーダ動作の禁止

関数のプロトタイプ宣言:
void
ENC_Disable(TSB_EN_TypeDef * ***ENx***)

引数:
ENx: ENC チャネルを選択します。

機能:
エンコーダ動作を禁止します。

戻り値:
なし

17.2.3.3 ENC_Init

エンコーダ動作の初期化

関数のプロトタイプ宣言:
void
ENC_Init(TSB_EN_TypeDef * ***ENx***, ENC_InitTypeDef * ***InitStruct***)

引数:
ENx: ENC チャネルを選択します。

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

機能:
エンコーダ動作の初期設定を行います。

戻り値:
なし

17.2.3.4 ENC_SetSWCapture

ソフトキャプチャの実行(タイマモード/センサモード (タイマカウント)時)

関数のプロトタイプ宣言:
void
ENC_SetSWCapture(TSB_EN_TypeDef * ***ENx***, uint32_t ***ENC_Mode***)

引数:
ENx: ENC チャネルを選択します。

ENC_Mode: 以下から、エンコーダ動作モードを選択します。
- **ENC_TIMER_MODE**: タイマモード
- **ENC_SENSOR_TIME_MODE**: センサモード

機能:
ソフトキャプチャの実行を行います。

戻り値:
なし

17.2.3.5 ENC_ClearCounter

エンコーダパルスカウンタクリア

関数のプロトタイプ宣言:
void
ENC_ClearCounter(TSB_EN_TypeDef * ***ENx***)

引数:
ENx: ENC チャネルを選択します。

機能:
エンコーダパルスカウンタをクリアします。

戻り値:

なし

17.2.3.6 ENC_GetENCFlag

エンコーダコンペアフラグ/反転エラーフラグ/ Z 相通過検出/エンコーダ回転方向の取得

関数のプロトタイプ宣言:
ENC_FlagStatus
ENC_GetENCFlag (TSB_EN_TypeDef * ***ENx***)

引数:
ENx: ENC チャネルを選択します。

機能:
エンコーダコンペアフラグ/反転エラーフラグ/ Z 相通過検出/エンコーダ回転方向を取得します。各フラグの意味については、MCU データシートを参照してください。

戻り値:
エンコーダフラグです。
ZPhaseDetectFlag(bit12): Z 相通過検出
RotationDirection (bit13): エンコーダ回転方向
ReverseErrorFlag (bit14): 反転エラーフラグ
CompareFlag (bit15): エンコーダコンペアフラグ

17.2.3.7 ENC_SetCounterReload

エンコーダカウンタの周期設定

関数のプロトタイプ宣言:
void
ENC_SetCounterReload (TSB_EN_TypeDef * ***ENx***, uint32_t ***PeriodValue***)

引数:
ENx: ENC チャネルを選択します。

PeriodValue: エンコーダカウンタの周期を選択します。値は **0x0000 ～ 0xFFFF** まで選択可能です。

機能:
エンコーダカウンタの周期を設定します。

戻り値:
なし

17.2.3.8 ENC_SetCompareValue

カウンタ比較値の設定

関数のプロトタイプ宣言:
void

ENC_SetCompareValue(TSB_EN_TypeDef * ***ENx***,uint32_t ***ENC_Mode***,
uint32_t ***CompareValue***)

引数:
ENx: ENC チャネルを選択します。

ENC_Mode: 以下から、エンコーダ動作モードを選択します。
- **ENC_ENCODER _MODE**: エンコーダモード
- **ENC_SENSOR_EVENT_MODE**: センサモード(イベントカウント)
- **ENC_SENSOR_TIME_MODE**: センサモード(タイマカウント)
- **ENC_TIMER_MODE**: タイマモード

CompareValue: 以下から、カウンタ比較値を設定します。
エンコーダモードとセンサモード(イベントカウント)の場合: **0x0000 - 0xFFFF**
センサモード(タイマカウント)とタイマモードの場合: **0x000000 - 0xFFFFFF**

機能:
カウンタ比較値を設定します。

戻り値:
なし

17.2.3.9 ENC_GetCompareValue

カウンタ比較値の取得

関数のプロトタイプ宣言:
uint32_t
ENC_GetCompareValue(TSB_EN_TypeDef * **ENx**)

引数:
ENx: ENC チャネルを選択します。

機能:
カウンタ比較値を取得します。

戻り値:
カウンタ比較値

17.2.3.10 ENC_GetCounterValue

エンコードカウンタ/キャプチャ値の取得

関数のプロトタイプ宣言:
uint32_t
ENC_GetCounterValue(TSB_EN_TypeDef * ***ENx***)

引数:
ENx: ENC チャネルを選択します。

機能:
エンコードカウンタ/キャプチャ値を取得します。

戻り値:
エンコードカウンタ/キャプチャ値の取得

17.2.4 データ構造

17.2.4.1 ENC_InitTypeDef

メンバ:

uint32_t
ModeType **:** エンコーダ入力モード:
- **ENC_ENCODER_MODE**: エンコーダモード
- **ENC_SENSOR_EVENT_MODE**: センサモード(イベントカウンタ)
- **ENC_SENSOR_TIME_MODE**: センサモード(タイマカウント)
- **ENC_TIMER_MODE**: タイマモード

uint32_t
PhaseType: 2 相/3 相入力選択:
- **ENC_TWO_PHASE**: 2 相入力
- **ENC_THREE_PHASE**: 3 相入力

uint32_t
EdgeType: ENCZ の使用エッジ選択:
- **ENC_RISING_EDGE**: 立ち上がりエッジ
- **ENC_FALLING_EDGE**: 立下りエッジ

uint32_t
CompareStatus: コンペアイネーブル:
- **ENC_COMPARE_DISABLE**: コンペア実行しない
- **ENC_COMPARE_ENABLE**: コンペア実行する

uint32_t
ZphaseStatus: Z 相イネーブル:
- **ENC_ZPHASE_DISABLE**: 禁止
- **ENC_ZPHASE_ENABLE**: 許可

uint32_t
FilterValue: ノイズフィルタ:
- **ENC_NO_FILTER**: ノイズフィルタなし
- **ENC_FILTER_VALUE31**: 31/fsys 未満のパルスはノイズとして除去
- **ENC_FILTER_VALUE63**: 63/fsys 未満のパルスはノイズとして除去
- **ENC_FILTER_VALUE127**: 127/fsys 未満のパルスはノイズとして除去

uint32_t
IntEn ENC 割り込みの許可/禁止
- **ENC_INTERRUPT_DISABLE**: 禁止
- **ENC_INTERRUPT_ENABLE**: 許可
-

uint32_t
PulseDivFactor: エンコーダパルス分周比:
- **ENC_PULSE_DIV1**: 1 分周
- **ENC_PULSE_DIV2**: 2 分周
- **ENC_PULSE_DIV4**: 4 分周
- **ENC_PULSE_DIV8**: 8 分周
- **ENC_PULSE_DIV16**: 16 分周
- **ENC_PULSE_DIV32**: 32 分周
- **ENC_PULSE_DIV64**: 64 分周
- **ENC_PULSE_DIV128**: 128 分周

17.2.4.2 ENC_FlagStatus

メンバ:

uint32_t
All **:** 全ての ENC フラグの状態

uint32_t
ZPhaseDetectFlag(bit12): Z 相通過検出

uint32_t
RotationDirection (bit13): 回転方向

uint32_t
ReverseErrorFlag (bit14): 反転エラーフラグ(センサモード(タイマカウント)時)

uint32_t
CompareFlag (bit15): コンペア発生フラグ

18. PMD

18.1 概要

本デバイスはモーター制御回路(PMD)を2チャンネル内蔵しています。本製品のPMDは1シャントセンサレスモータ制御を実現する為に通電出力制御や、DC 過電圧検出入力を追加し、ADCを連携させたモータ制御を可能としています。

PMD（プログラマブルモータドライバ）回路は波形生成回路と同期トリガ生成回路の2ブロックから成り、波形生成回路はパルス幅変調回路、通電制御回路、保護制御回路、デッドタイム制御回路で構成されています。
- パルス幅変調回路はPWM 周波数が等しい3 相の独立したPWM 波形を生成します。
- 通電制御回路はU、V、W 相の各上下相の出力パターンを決定します。
- 保護制御回路では異常検出入力による緊急出力停止を行ないます。
- デッドタイム制御回路では上下相の切り替え時の短絡を防止します。
- 同期トリガ生成回路ではADC への同期トリガ信号を生成します。

全ドライバ API は、アプリで使用する API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver\src\tmpm380_pmd.c
/Libraries/TX03_Periph_Driver\inc\tmpm380_pmd.h

18.2 API 関数

18.2.1 関数一覧

- void PMD_Enable(TSB_MTPD_TypeDef * ***PMDx***);
- void PMD_Disable(TSB_MTPD_TypeDef * ***PMDx***);
- void PMD_SetPortControl(TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***PortMode***);
- void PMD_Init(TSB_MTPD_TypeDef * ***PMDx***,
  PMD_InitTypeDef * ***InitStruct***);
- void PMD_ChangePWMCycle(TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***CycleTiming***);
- uint32_t PMD_GetCntFlag(TSB_MTPD_TypeDef * ***PMDx***);
- uint16_t PMD_GetCntValue(TSB_MTPD_TypeDef * ***PMDx***);
- void PMD_SetCompareValue(TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***PMDPhase***,
  uint32_t ***Timing***);
- void PMD_SetPortOutputMode(TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***Mode***);
- void PMD_SetOutputPhasePolarity(TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***OutputPhase***,
  uint32_t ***Polarity***);
- void PMD_SetReflectTime(TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***ReflectedTime***);
- void PMD_EnableEMG(TSB_MTPD_TypeDef * ***PMDx***);
- void PMD_DisableEMG(TSB_MTPD_TypeDef * ***PMDx***);
- void PMD_SetEMGNoiseElimination(TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***NoiseElimination***);

- void PMD_SetToolBreakOutput(TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***Status***);
- void PMD_SetEMGMode(TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***Mode***);
- void PMD_EMGRelease(TSB_MTPD_TypeDef * ***PMDx***);
- uint32_t PMD_GetEMGAbnormalLevel(TSB_MTPD_TypeDef * ***PMDx***);
- uint32_t PMD_GetEMGCondition(TSB_MTPD_TypeDef * ***PMDx***);
- void PMD_SetDeadTime(TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***Time***);
- void PMD_SetAllPhaseCompareValue(TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***UPhaseTiming***,
  uint32_t ***VPhaseTiming***,
  uint32_t ***WPhaseTiming***)
- void PMD_ChangeDutyMode(TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***DutyMode***);
- Result PMD_SetPortOutput (TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***PMDPhase***,
  uint8_t ***Output***);
- void PMD_SetTrgCmpValue(TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***TRGCMP0Timing***,
  uint32_t ***TRGCMP1Timing***);
- void PMD_SetTrgMode(TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***PMDTrg***,
  uint32_t ***Mode***);
- void PMD_SetTrgUpdate(TSB_MTPD_TypeDef * ***PMDx***,
  uint32_t ***PMDTrg***,
  uint32_t ***UpdateTiming***);
- void PMD_SetEMGTrg(TSB_MTPD_TypeDef * ***PMDx***,
  FunctionalState ***NewState***);

18.2.2 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) PMD の共通設定:
   PMD_Enable(), PMD_Disable(), PMD_SetPortControl(), PMD_Init(),
   PMD_ChangePWMCycle(), PMD_SetCompareValue(),
   PMD_SetAllPhaseCompareValue(), PMD_ChangeDutyMode()
2) PMD ポート出力の設定:
   PMD_SetPortOutputMode(), PMD_SetOutputPhasePolarity(),
   PMD_SetReflectTime(), PMD_SetPortOutput()
3) EMG 保護制御回路の設定:
   PMD_EnableEMG(), PMD_DisableEMG(), PMD_SetEMGNoiseElimination(),
   PMD_SetToolBreakOutput(), PMD_SetEMGMode(), PMD_EMGRelease()
4) 動作状態の取得:
   PMD_GetCntrFlag(), PMD_GetCntValue(), PMD_GetEMGAbnormalLevel(),
   PMD_GetEMGCondition()
5) デッドタイム制御:
   PMD_SetDeadTime()
6) ADC トリガ要求:
   PMD_SetTrgCmpValue(), PMD_SetTrgMode(), PMD_SetTrgUpdate(),
   PMD_SetEMGTrg()

18.2.3 関数仕様

補足: 下記の全 API において、パラメータ “TSB_MTPD_TypeDef * ***PMDx***” は 以下のいずれかを選択してください。
PMD0, PMD1

18.2.3.1 PMD_Enable

PMD 機能の許可

関数のプロトタイプ宣言:
void
PMD_Enable(TSB_MTPD_TypeDef * ***PMDx***)

引数:
PMDx: PMD チャネルを指定します。

機能:
PMD 機能を許可します。

戻り値:
なし

18.2.3.2 PMD_Disable

PMD 機能の禁止

関数のプロトタイプ宣言:
void
PMD_Disable(TSB_MTPD_TypeDef * ***PMDx***)

引数:
PMDx: PMD チャネルを指定します。

機能:
PMD 機能を禁止します。

戻り値:
なし

18.2.3.3 PMD_SetPortControl

ポート制御の設定

関数のプロトタイプ宣言:
void
PMD_SetPortControl(TSB_MTPD_TypeDef * ***PMDx***
uint32_t ***PortMode***)

引数:
PMDx: PMD チャネルを指定します。

PortMode: ポート制御の種類を選択します。
- **PMD_PORT_HIGH_IMPEDANCE:** ハイ・インピーダンス

- **PMD_PORT_PMD_OUTPUT:** PMD 出力

機能:
ポート制御を設定します。

戻り値:
なし

18.2.3.4 PMD_Init

PMD の初期化

関数のプロトタイプ宣言:
void
PMD_Init(TSB_MTPD_TypeDef * ***PMDx***,
PMD_InitTypeDef * ***InitStruct***)

引数:
PMDx: PMD チャネルを指定します。

InitStruct: PMD の基本設定内容を格納した構造体を指定します。
(詳細は “データ構造” 参照)

機能:
PMD を初期化します。

戻り値:
なし

18.2.3.5 PMD_ChangePWMCycle

PWM 周期の設定

関数のプロトタイプ宣言:
void
PMD_ChangePWMCycle(TSB_MTPD_TypeDef * ***PMDx***,
uint32_t ***CycleTiming***)

引数:
PMDx: PMD チャネルを指定します。

CycleTiming: PWM 周期を 0x0000 ~ 0xFFFF の間で設定します。

機能:
PWM 周期を設定します。

戻り値:
なし

補足:

設定値は 0x10 以上の値を設定してください。0x10 未満の値を設定した場合、0x10 が設定されたものとして動作します。(設定値を取得すると設定した値が読み出せます)

18.2.3.6 PMD_GetCntFlag

PWM カウンタフラグの取得

関数のプロトタイプ宣言:
uint32_t
PMD_GetCntFlag(TSB_MTPD_TypeDef * ***PMDx***)

引数:
PMDx: PMD チャネルを指定します。

機能:
PWM カウンタフラグを取得します。

戻り値:
PWM カウンタフラグ:
PMD_COUNTER_UP: アップカウント中
PMD_COUNTER_DOWN : ダウンカウント中

18.2.3.7 PMD_GetCntValue

PWM 周期カウント値の取得

関数のプロトタイプ宣言:
uint16_t
PMD_GetCntValue(TSB_MTPD_TypeDef * ***PMDx***)

引数:
PMDx: PMD チャネルを指定します。

機能:
PWM 周期カウント値を取得します。

戻り値:
PWM 周期カウント値

18.2.3.8 PMD_SetCompareValue

PWM パルス幅の設定

関数のプロトタイプ宣言:
void
PMD_SetCompareValue(TSB_MTPD_TypeDef * ***PMDx***,
uint32_t ***PMDPhase***,
uint32_t ***Timing***)

引数:

PMDx: PMD チャネルを指定します。

PMDPhase: 3 相のいぞれか、または 3 相すべてを選択します。
- **PMD_PHASE_U:** U 相
- **PMD_PHASE_V:** V 相
- **PMD_PHASE_W:** W 相
- **PMD_PHASE_ALL:** 3 相すべて

Timing: コンペア値を 0x0000 ~ 0xFFFF の間で設定します。

機能:
PWM パルス幅を設定します。

戻り値:
なし

18.2.3.9 PMD_SetPortOutputMode

U,V,W 相のポート出力設定

関数のプロトタイプ宣言:
void
PMD_SetPortOutputMode(TSB_MTPD_TypeDef * ***PMDx***,
uint32_t ***Mode***)

引数:
PMDx: PMD チャネルを指定します。

Mode: U,V,W 相のポート出力を設定します。
- **PMD_PORT_OUTPUT_MODE_0:** MTPDxMDCR<SYNTMD>=0
- **PMD_PORT_OUTPUT_MODE_1:** MTPDxMDCR<SYNTMD>=1

機能:
U,V,W 相のポート出力設定を行います。

補足:
MTPDxMDCR<SYNTMD>, MTPDxMDPOT<POLH><POLL>, MTPDxMDOUT <UPWN><VPWN><WPWN> <UOC> <VOC> <WOC>の内容により出力ポートの制御を行います。(x=0, 1)

PMD_SetPortOutputMode()により MTPDxMDCR<SYNTMD>を設定します。
PMD_SetOutputPhasePolarity()により MTPDxMDPOT<POLH><POLL>を設定します
PMD_SetPortOutput()により MTPDxMDOUT<UPWN><VPWN> <WPWN> <UOC> <VOC> <WOC>を設定します。

上記による設定によって得られる端子出力の関係については下表を参照してください。

MTPDxMDCR<SYNTMD>=0

Polarity: high-active(MTPDxMDPOT<POLH><POLL>="11")

MDOUT output control		MTPDxMDOUT <WPWM><VPWM><UPWM> H/L/PWM output selection			
<WOC[1]> <VOC[1]> <UOC[1]> (Upper)	<WOC[0]> <VOC[0]> <UOC[0]> (Lower)	0 : H/L output		1 : PWM output	
		Upper output	Lower output	Upper output	Lower output
0	0	L	L	$\overline{PWM}$	PWM
0	1	L	H	L	PWM
1	0	H	L	PWM	L
1	1	H	H	PWM	$\overline{PWM}$

MTPDxMDCR<SYNTMD>=0

Polarity: low-active(MTPDxMDPOT<POLH><POLL>="00")

MDOUT output control		MTPDxMDOUT <WPWM><VPWM><UPWM> H/L/PWM output selection			
<WOC[1]> <VOC[1]> <UOC[1]> (Upper)	<WOC[0]> <VOC[0]> <UOC[0]> (Lower)	0 : H/L output		1 : PWM output	
		Upper output	Lower output	Upper output	Lower output
0	0	H	H	PWM	$\overline{PWM}$
0	1	H	L	H	$\overline{PWM}$
1	0	L	H	$\overline{PWM}$	H
1	1	L	L	$\overline{PWM}$	PWM

MTPDxMDCR<SYNTMD>=1

Polarity: high-active(MTPDxMDPOT<POLH><POLL>="11")

MDOUT output control		MTPDxMDOUT <WPWM><VPWM><UPWM> H/L/PWM output selection			
<WOC[1]> <VOC[1]> <UOC[1]> (Upper)	<WOC[0]> <VOC[0]> <UOC[0]> (Lower)	0 : H/L output		1 : PWM output	
		Upper output	Lower output	Upper output	Lower output
0	0	L	L	$\overline{PWM}$	PWM
0	1	L	H	L	$\overline{PWM}$
1	0	H	L	PWM	L
1	1	H	H	PWM	$\overline{PWM}$

MTPDxMDCR<SYNTMD>=1

Polarity: low-active(MTPDxMDPOT<POLH><POLL>="00")

MDOUT output control		MTPDxMDOUT <WPWM><VPWM><UPWM> H/L/PWM output selection			
<WOC[1]> <VOC[1]> <UOC[1]> (Upper)	<WOC[0]> <VOC[0]> <UOC[0]> (Lower)	0 : H/L output		1 : PWM output	
		Upper output	Lower output	Upper output	Lower output
0	0	H	H	PWM	$\overline{PWM}$
0	1	H	L	H	PWM
1	0	L	H	$\overline{PWM}$	H
1	1	L	L	$\overline{PWM}$	PWM

戻り値:
なし

18.2.3.10 PMD_SetOutputPhasePolarity

上相/下相の出力ポート極性の選択

関数のプロトタイプ宣言:
void
PMD_SetOutputPhasePolarity(TSB_MTPD_TypeDef * ***PMDx***,
uint32_t ***OutputPhase***,
uint32_t ***Polarity***)

引数:
PMDx: PMD チャネルを指定します。

OutputPhase: 出力ポートの上相/下相を選択します。
- **PMD_OUTPUT_PHASE_UPPER:** 上相の出力ポート
- **PMD_OUTPUT_PHASE_LOWER:** 下相の出力ポート

Polarity: 極性を選択します。
- **PMD_POLARITY_LOW:** ロー・アクティブ
- **PMD_POLARITY_HIGH:** ハイ・アクティブ

機能:
上相/下相の出力ポートの極性を選択します。

補足:
1 詳細は PMD_SetPortOutputMode() 関数を参照してください。
2 PWM を無効の状態で選択を行ってください。

戻り値:
なし

18.2.3.11 PMD_SetReflectTime

U, V, W 相出力設定のポート出力反映時のタイミング選択

関数のプロトタイプ宣言:
void
PMD_SetReflectTime(TSB_MTPD_TypeDef * ***PMDx***,
uint32_t ***ReflectedTime***)

引数:
PMDx: PMD チャネルを指定します。

ReflectedTime: U, V, W 相出力設定のポート出力反映時のタイミングを選択します。
- **PMD_REFLECTED_TIME_WRITE:** 書き込み時に反映
- **PMD_REFLECTED_TIME_MIN:** PWM カウンタ MDCNT= "1"(最小)の時、反映
- **PMD_REFLECTED_TIME_MAX:** PWM カウンタ MDCNT= MTPDxMDPRD<MDPRD>(最大)の時、反映
- **PMD_REFLECTED_TIME_MIN_MAX:** PWM カウンタ MDCNT="1"(最小)および MTPDxMDPRD<MDPRD>(最大)の時、反映

機能:
U, V, W 相出力設定のポート出力反映時のタイミングを選択します。

補足:
PWM を無効の状態で選択を行ってください。

戻り値:
なし

18.2.3.12 PMD_EnableEMG

EMG 保護回路の許可

関数のプロトタイプ宣言:
void
PMD_EnableEMG(TSB_MTPD_TypeDef * ***PMDx***)

引数:
PMDx: PMD チャネルを指定します。

機能:
EMG 保護回路を許可します。

戻り値:
なし

18.2.3.13 PMD_DisableEMG

EMG 保護回路の禁止

関数のプロトタイプ宣言:
void
PMD_DisableEMG(TSB_MTPD_TypeDef * ***PMDx***)

引数:
PMDx: PMD チャネルを指定します。

機能:
EMG 保護回路を禁止します。

戻り値:
なし

18.2.3.14 PMD_SetEMGNoiseElimination

異常検出入力のノイズ除去時間の設定

関数のプロトタイプ宣言:
void
PMD_SetEMGNoiseElimination(TSB_MTPD_TypeDef * ***PMDx***,
uint32_t ***NoiseElimination***)

引数:
PMDx: PMD チャネルを指定します。

NoiseElimination:異常検出入力のノイズ除去時間を選択します。

- **PMD_NOISE_ELIMINATION_NONE:** ノイズフィルタを経由しません。
- **PMD_NOISE_ELIMINATION_16:** 入力ノイズ除去時間 16/fsys[s]
- **PMD_NOISE_ELIMINATION_32:** 入力ノイズ除去時間 32/fsys[s]
- **PMD_NOISE_ELIMINATION_48:** 入力ノイズ除去時間 48/fsys[s]
- **PMD_NOISE_ELIMINATION_64:** 入力ノイズ除去時間 64/fsys[s]
- **PMD_NOISE_ELIMINATION_80:** 入力ノイズ除去時間 80/fsys[s]
- **PMD_NOISE_ELIMINATION_96:** 入力ノイズ除去時間 96/fsys[s]
- **PMD_NOISE_ELIMINATION_112:** 入力ノイズ除去時間 112/fsys[s]
- **PMD_NOISE_ELIMINATION_128:** 入力ノイズ除去時間 128/fsys[s]
- **PMD_NOISE_ELIMINATION_144:** 入力ノイズ除去時間 144/fsys[s]
- **PMD_NOISE_ELIMINATION_160:** 入力ノイズ除去時間 160/fsys[s]
- **PMD_NOISE_ELIMINATION_176:** 入力ノイズ除去時間 176/fsys[s]
- **PMD_NOISE_ELIMINATION_192:** 入力ノイズ除去時間 192/fsys[s]
- **PMD_NOISE_ELIMINATION_208:** 入力ノイズ除去時間 208/fsys[s]
- **PMD_NOISE_ELIMINATION_224:** 入力ノイズ除去時間 224/fsys[s]
- **PMD_NOISE_ELIMINATION_240:** 入力ノイズ除去時間 240/fsys[s]

機能:
異常検出入力のノイズ除去時間を設定します。

戻り値:
なし

18.2.3.15 PMD_SetToolBreakOutput

ツールブレーク時の PWM 出力状態の選択

関数のプロトタイプ宣言:
void
PMD_SetToolBreakOutput(TSB_MTPD_TypeDef * ***PMDx***,
uint32_t ***Status***)

引数:
PMDx: PMD チャネルを指定します。

Status: ツールブレーク時の PWM 出力状態を選択します。
- **PMD_BREAK_STATUS_PMD:** PMD 出力継続
- **PMD_BREAK_STATUS_HIGH_IMPEDANCE:** ハイ・インピーダンス

機能:
ツールブレーク時の PWM 出力状態を選択します。

戻り値:
なし

18.2.3.16 PMD_SetEMGMode

EMG 保護モードの選択

関数のプロトタイプ宣言:
void
PMD_SetEMGMode(TSB_MTPD_TypeDef * ***PMDx***,
uint32_t ***Mode***)

引数:
PMDx: PMD チャネルを指定します。

Mode: EMG 保護モードを選択します。
- **PMD_EMG_MODE_0:** 全相オン/PORT ハイ・インピーダンス
- **PMD_EMG_MODE_1:** 全相オフ/PORT ハイ・インピーダンス
- **PMD_EMG_MODE_2:** 全相オン/PORT 出力許可
- **PMD_EMG_MODE_3:** 全相オフ/PORT ハイ・インピーダンス

機能:
EMG 保護モードを選択します。

戻り値:
なし

18.2.3.17 PMD_EMGRelease

EMG 保護状態からの復帰

関数のプロトタイプ宣言:
void
PMD_EMGRelease(TSB_MTPD_TypeDef * ***PMDx***)

引数:
PMDx: PMD チャネルを指定します。

機能:
EMG 保護状態から復帰します。

補足:
本関数をコールすると、MTPDxMDOUT<UPWN><VPWN><WPWN>、
MTPDxMDOUT<UOC> <VOC> <WOC>に 0 を設定します。(x=0)

戻り値:
なし

18.2.3.18 PMD_GetEMGAbnormalLevel

異常状態入力のレベルモニタ

関数のプロトタイプ宣言:
uint32_t
PMD_GetEMGAbnormalLevel(TSB_MTPD_TypeDef * ***PMDx***)

引数:
PMDx: PMD チャネルを指定します。

機能:
異常状態入力のレベルをモニタします。

戻り値:
異常状態入力の状態
PMD_ABNORMAL_LEVEL_L: 異常状態入力のレベルが"L"
PMD_ABNORMAL_LEVEL_H : 異常状態入力のレベルが"H"

18.2.3.19 PMD_GetEMGCondition

EMG 保護の状態モニタ

関数のプロトタイプ宣言:
uint32_t
PMD_GetEMGCondition(TSB_MTPD_TypeDef * ***PMDx***)

引数:
PMDx: PMD チャネルを指定します。

機能:

EMG 保護の状態をモニタします。

戻り値:
EMG 保護の状態
0：通常動作中
1：EMG 保護中

18.2.3.20 PMD_SetDeadTime

デッドタイムの設定

関数のプロトタイプ宣言:
void
PMD_SetDeadTime(TSB_MTPD_TypeDef * ***PMDx***,
uint32_t ***Time***)

引数:
PMDx: PMD チャネルを指定します。

Time: デッドタイムを 0x00 ～ 0xFF の間で設定します。

機能:
デッドタイムを設定します。

補足:
PWM を無効の状態で選択を行ってください。

戻り値:
なし

18.2.3.21 PMD_SetAllPhaseCompareValue

PWM パルス幅の設定

関数のプロトタイプ宣言:
void
PMD_SetAllPhaseCompareValue(TSB_MTPD_TypeDef * ***PMDx***,
uint32_t ***UPhaseTiming***,
uint32_t ***VPhaseTiming***,
uint32_t ***WPhaseTiming***)

引数:
PMDx: PMD チャネルを指定します。

UPhaseTiming: U 相に出力するパルス幅を 0x0000～0xFFFF の間で設定します。

VPhaseTiming: V 相に出力するパルス幅を 0x0000～0xFFFF の間で設定します。

WPhaseTiming: W 相に出力するパルス幅を 0x0000～0xFFFF の間で設定します。

機能:
PWM パルス幅の設定をします。

戻り値:
なし

18.2.3.22 PMD_ChangeDutyMode

DUTY モードの設定

関数のプロトタイプ宣言:
void
PMD_ChangeDutyMode(TSB_MTPD_TypeDef * ***PMDx***,
uint32_t ***DutyMode***)

引数:
PMDx: PMD チャネルを指定します。

DutyMode: DUTY モードを選択します。
- **PMD_DUTY_MODE_U_PHASE:** U 相共通
- **PMD_DUTY_MODE_3_PHASE:** 3 相独立

機能:
DUTY モードを設定します。

戻り値:
なし

18.2.3.23 PMD_SetPortOutput

UVW 相出力の設定

関数のプロトタイプ宣言:
Result
PMD_SetPortOutput(TSB_MTPD_TypeDef * ***PMDx***,
uint32_t ***PMDPhase***,
uint8_t ***Output***)

引数:
PMDx: PMD チャネルを指定します。

PMDPhase: UVW 相を選択します。
- **PMD_PHASE_U:** U 相
- **PMD_PHASE_V:** V 相
- **PMD_PHASE_W:** W 相
- **PMD_PHASE_ALL:** 全相

Output: 出力を選択します。
- **PMD_OUTPUT_L_L:** 上相出力"L"、下相出力"L"
- **PMD_OUTPUT_L_H:** 上相出力"L",下相出力"H"
- **PMD_OUTPUT_H_L:** 上相出力"H", 下相出力"L"
- **PMD_OUTPUT_H_H:** 上相出力"H", 下相出力"H"
- **PMD_OUTPUT_PWM_IPWM:** 上相出力"PWM", 下相出力 IPWM

- **PMD_OUTPUT_IPWM_PWM:** 上相出力"IPWM", 下相出力 PWM
- **PMD_OUTPUT_H_PWM:** 上相出力"H", 下相出力"PWM"
- **PMD_OUTPUT_L_PWM:** 上相出力"L", 下相出力"PWM"
- **PMD_OUTPUT_PWM_L:** 上相出力"PWM", 下相出力"L"
- **PMD_OUTPUT_H_IPWM:** 上相出力"H", 下相出力"IPWM"
- **PMD_OUTPUT_L_IPWM:** 上相出力"L", 下相出力"IPWM"
- **PMD_OUTPUT_IPWM_H:** 上相出力"IPWM", 下相出力 H"

機能:
UVW 相出力を設定します。

戻り値:
実行結果:
SUCCESS: PMD 出力設定成功
ERROR: PMD 出力設定失敗

補足:
1. IPWM は PWM の反転です。
2. 詳細は PMD_SetPortOutputMode()関数を参照してください。

18.2.3.24 PMD_SetTrgCmpValue

トリガコンペアレジスタ値の設定

関数のプロトタイプ宣言:
```
void
PMD_SetTrgCmpValue(TSB_MTPD_TypeDef * PMDx,
                   uint32_t TRGCMP0Timing,
                   uint32_t TRGCMP1Timing)
```

引数:
PMDx: PMD チャネルを指定します。

TRGCMP0Timing: トリガコンペアレジスタ 0 の値を 0x0001～[<MDPRO>-1]の間で設定してください。

TRGCMP1Timing: トリガコンペアレジスタ 1 の値を 0x0001～[<MDPRO>-1]の間で設定してください。

機能:
トリガコンペアレジスタ値を設定します。

戻り値:
なし

補足: PMDnTRGCMPx (x=0, 1)は 1～ [<MDPRD> – 1]の間で設定してください。

18.2.3.25 PMD_SetTrgMode

トリガモードの設定

関数のプロトタイプ宣言:
void
PMD_SetTrgMode (TSB_MTPD_TypeDef * ***PMDx***,
uint32_t ***PMDTrg***,
uint32_t ***Mode***)

引数:
PMDx: PMD チャネルを指定します。

PMDTrg: PMD トリガを選択します。
- **PMD_ADC_TRG_0:** トリガ 0
- **PMD_ADC_TRG_1:** トリガ 1

Mode: PMD トリガを選択します。
- **PMD_TRG_MODE_0:** トリガ出力禁止
- **PMD_TRG_MODE_1:** ダウンカウント時の一致でトリガ出力
- **PMD_TRG_MODE_2:** アップカウント時の一致でトリガ出力
- **PMD_TRG_MODE_3:** アップ/ダウンカウント時の一致でトリガ出力
- **PMD_TRG_MODE_4:** PWM キャリアピークでトリガ出力
- **PMD_TRG_MODE_5:** PWM キャリアボトムでトリガ出力
- **PMD_TRG_MODE_6:** PWM キャリアピーク/ボトムでトリガ出力
- **PMD_TRG_MODE_7:** トリガ出力禁止

機能:
トリガモードを設定します。

戻り値:
なし

18.2.3.26 PMD_SetTrgUpdate

トリガコンペアレジスタの更新タイミング

関数のプロトタイプ宣言:
void
PMD_SetTrgUpdate (TSB_MTPD_TypeDef * ***PMDx***,
uint32_t ***PMDTrg***,
uint32_t ***UpdateTiming***)

引数:
PMDx: PMD チャネルを指定します。

PMDTrg: PMD トリガを選択します。
- **PMD_ADC_TRG_0:** トリガ 0
- **PMD_ADC_TRG_1:** トリガ 1

Mode: PMDTRG0 ～ PMDTRG1 を更新タイミングを選択します。
- **PMD_TRG_UPDATE_SYNC:** PWM 同期更新
- **PMD_TRG_UPDATE_ASYNC:** 非同期更新(バッファの非同期更新を許可します。書き込み後、直ちに反映)

機能:

トリガコンペアレジスタの更新タイミングを設定します。

戻り値:
なし

18.2.3.27 PMD_SetEMGTrg

EMG 保護動作中の出力許可設定

関数のプロトタイプ宣言:
void
PMD_SetEMGTrg (TSB_MTPD_TypeDef * ***PMDx***,
FunctionalState ***NewState***)

引数:
PMDx: PMD チャネルを指定します。

NewState: EMG 保護動作中の出力許可/禁止を選択します。
- **ENABLE:** 保護動作時トリガ出力許可
- **DIABLE:** 保護動作時トリガ出力禁止

機能:
EMG 保護動作中の出力許可を設定します。

戻り値:
なし

18.2.4 データ構造

18.2.4.1 PMD_InitTypeDef

メンバ:
uint32_t
CycleMode: PWM 周期延長モードを指定します。
- **PMD_PWM_NORMAL_CYCLE:** 通常周期
- **PMD_PWM_4_FOLD_CYCLE:** 4 倍周期

uint32_t
DutyMode: DUTY モードを指定します。
- **PMD_DUTY_MODE_U_PHASE:** U 相共通
- **PMD_DUTY_MODE_3_PHASE:** 3 相独立

uint32_t
IntTiming: PWM モード 1(三角波)の時の PWM 割り込みタイミングを選択します。
- **PMD_PWM_INT_TIMING_MINIMUM:** PWM カウンタ MDCNT="1"の時(最小)割り込み要求
- **PMD_PWM_INT_TIMING_MAXIMUM:** PWM カウンタ MDCNT= MTPDxMDPRD<MDPRD>の時 (最大)割り込み要求

uint32_t
IntCycle: PWM 割り込み周期を選択します。

- **PMD_PWM_INT_CYCLE_HALF:** PWM 0.5 周期毎に割り込み (PWM モード 1(三角波)のみ設定可能です)
- **PMD_PWM_INT_CYCLE_1:** PWM 1 周期毎に割り込み
- **PMD_PWM_INT_CYCLE_2:** PWM 2 周期毎に割り込み
- **PMD_PWM_INT_CYCLE_4:** PWM 4 周期毎に割り込み

uint32_t
CarrierMode: PWM キャリア波形を指定します。
- **PMD_CARRIER_WAVE_MODE_0:** PWM モード 0 (エッジ PWM, ノコギリ波)
- **PMD_CARRIER_WAVE_MODE_1:** PWM モード 1 (センターPWM, 三角波)

uint32_t
CycleTiming: PWM 周期を 0x0000～0xFFFF の間で指定します。

補足:
設定値が 0x10 以下の 場合は 0x10 が設定されたものとして動作します。