

# **TOSHIBA**

## **TX03 ペリフェラルドライバ ユーザーガイド (TMPM375FSDMG)**

第一版  
2017 年 9 月

**東芝デバイス&ストレージ株式会社**

## **本製品取り扱い上のお願い**

- ソフトウェア使用権許諾契約書の同意無しに使用しないで下さい。

**© 2017 Toshiba Electronics Devices & Storage Corporation**

## 目次

<b>1. はじめに</b>	<b>1</b>
<b>2. TX03 ペリフェラルドライバの構成</b>	<b>1</b>
<b>3. ADC</b>	<b>2</b>
3.1 概要	2
3.2 API 関数	2
3.2.1 関数一覧	2
3.2.2 関数の種類	3
3.2.3 関数仕様	3
3.2.4 データ構造	12
<b>4. CG</b>	<b>15</b>
4.1 概要	15
4.2 API 関数	15
4.2.1 関数一覧	15
4.2.2 関数の種類	16
4.2.3 関数仕様	16
4.2.4 データ構造	29
<b>5. ENC</b>	<b>30</b>
5.1 概要	30
5.2 API 関数	30
5.2.1 関数一覧	30
5.2.2 関数の種類	31
5.2.3 関数仕様	31
5.2.4 データ構造	35
<b>6. FC</b>	<b>37</b>
6.1 概要	37
6.2 API 関数	37
6.2.1 関数一覧	37
6.2.2 関数の種類	37
6.2.3 関数仕様	38
6.2.4 データ構造	42
<b>7. GPIO</b>	<b>43</b>
7.1 概要	43
7.2 API 関数	43
7.2.1 関数一覧	43
7.2.2 関数の種類	43
7.2.3 関数仕様	44
7.2.4 データ構造	54
<b>8. SBI</b>	<b>55</b>
8.1 概要	55
8.2 API 関数	55
8.2.1 関数一覧	55
8.2.2 関数の種類	55
8.2.3 関数仕様	56
8.2.4 データ構造	61
<b>9. OFD</b>	<b>63</b>
9.1 概要	63
9.2 API 関数	63
9.2.1 関数一覧	63

9.2.2 関数の種類 .....	63
9.2.3 関数仕様 .....	63
9.2.4 データ構造 .....	65
<b>10. TMRB .....</b>	<b>66</b>
10.1 概要 .....	66
10.2 API 関数 .....	66
10.2.1 関数一覧 .....	66
10.2.2 関数の種類 .....	67
10.2.3 関数仕様 .....	67
10.2.4 データ構造 .....	76
<b>11. SIO/UART .....</b>	<b>78</b>
11.1 概要 .....	78
11.2 API 関数 .....	78
11.2.1 関数一覧 .....	78
11.2.2 関数の種類 .....	79
11.2.3 関数仕様 .....	79
11.2.4 データ構造 .....	94
<b>12. VLTD .....</b>	<b>97</b>
12.1 概要 .....	97
12.2 API 関数 .....	97
12.2.1 関数一覧 .....	97
12.2.2 関数の種類 .....	97
12.2.3 関数仕様 .....	97
12.2.4 データ構造 .....	98
<b>13. WDT .....</b>	<b>99</b>
13.1 概要 .....	99
13.2 API 関数 .....	99
13.2.1 関数一覧 .....	99
13.2.2 関数の種類 .....	99
13.2.3 関数仕様 .....	99
13.2.4 データ構造 .....	102
<b>14. PMD .....</b>	<b>103</b>
14.1 概要 .....	103
14.2 API 関数 .....	103
14.2.1 関数一覧 .....	103
14.2.2 関数の種類 .....	105
14.2.3 関数仕様 .....	105
14.2.4 データ構造 .....	127

## 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。TMPM375FSDMG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TMPM375FSDMG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。

TMPM375FSDMG 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網羅しています。

## 2. TX03 ペリフェラルドライバの構成

### **/Libraries**

TX03 CMSIS ファイルと TMPM375FSDMG 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 TMPM375FSDMG 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**

TMPM375FSDMG 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**

TMPM375FSDMG 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**

TMPM375FSDMG 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**

TMPM375FSDMG 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**

TMPM375FSDMG 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**

TMPM375FSDMG ペリフェラルドライバの使用例が格納されています。

### **/Utilities/TMPM375-EVAL**

TMPM375FSDMG ボードのハードウェアリソース用の設定ファイル、およびドライバファイル (例: led, key) が格納されています。

## 3. ADC

### 3.1 概要

本デバイスは、12ビット逐次変換方式アナログ/デジタルコンバータ(ADコンバータ)を内蔵しています。

ADコンバータユニットBは5本のアナログ入力を持っています。4本はモータ0の測定用に使用可能です。この内1本はIC内蔵でオペアンプの出力に接続されていますので、外部から入力可能なアナログ入力端子は4本です。

4本の外部アナログ入力端子(AINB9～AINB12)は、入出力専用ポートと兼用です。

機能と特徴:

- (1) PMD 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) ソフトウェア起動、常時起動において任意のアナログ入力を変換することができます。
- (3) AD 変換値レジスタが12 個あります。
- (4) TMRB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、チャンネル選択、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\tmpm375_adc.c  
/Libraries/TX03_Periph_Driver\inc\tmpm375_adc.h
```

### 3.2 API 関数

#### 3.2.1 関数一覧

- ◆ void ADC\_SetClk(TSB\_AD\_TypeDef \* **ADx**, uint32\_t **Sample\_HoldTime**,  
ADC\_PRESCALER **Prescaler\_Output**);
- ◆ void ADC\_Enable(TSB\_AD\_TypeDef \* **ADx**);
- ◆ void ADC\_Disable(TSB\_AD\_TypeDef \* **ADx**);
- ◆ void ADC\_Start(TSB\_AD\_TypeDef \* **ADx**, ADC\_TrgType **Trg**);
- ◆ void ADC\_StopConstantTrg(TSB\_AD\_TypeDef \* **ADx**);
- ◆ WorkState ADC\_GetConvertState(TSB\_AD\_TypeDef \* **ADx**, ADC\_TrgType **Trg**);
- ◆ void ADC\_SetMonitor(TSB\_AD\_TypeDef \* **ADx**, ADC\_MonitorTypeDef \* **Monitor**);
- ◆ void ADC\_DisableMonitor(TSB\_AD\_TypeDef \* **ADx**, ADC\_CMPCRx **CMPCRx**);
- ◆ ADC\_Result ADC\_GetConvertResult(TSB\_AD\_TypeDef \* **ADx**,  
ADC\_REGx **ResultREGx**);
- ◆ void ADC\_SelectPMDTrgProgNum(TSB\_AD\_TypeDef \* **ADx**,  
PMD\_TRG\_PROG\_SELx **SELx**, uint8\_t **MacroProgNum**);
- ◆ void ADC\_SetPMDTrgProgINT(TSB\_AD\_TypeDef \* **ADx**,  
PMD\_TrgProgINTTypeDef \* **TrgProgINT**);

- ◆ void ADC\_SetPMDTrg(TSB\_AD\_TypeDef \* **ADx**, PMD\_TrgTypeDef \* **PMDTrg**);
- ◆ void ADC\_SetTimerTrg(TSB\_AD\_TypeDef \* **ADx**,  
ADC\_REGx **ResultREGx**, uint8\_t **MacroAINx**);
- ◆ void ADC\_SetSWTrg(TSB\_AD\_TypeDef \* **ADx**,  
ADC\_REGx **ResultREGx**, uint8\_t **MacroAINx**);
- ◆ void ADC\_SetConstantTrg(TSB\_AD\_TypeDef \* **ADx**,  
ADC\_REGx **ResultREGx**, uint8\_t **MacroAINx**);

## 3.2.2 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) AD 変換設定:  
ADC\_SetClk(), ADC\_SetMonitor(), ADC\_DisableMonitor(),  
ADC\_SelectPMDTrgProgNum(), ADC\_SetPMDTrgProgINT(), ADC\_SetPMDTrg(),  
ADC\_SetTimerTrg(), ADC\_SetSWTrg(), ADC\_SetConstantTrg()
- 2) AD 変換の許可/禁止と開始/終了:  
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\_GetConvertState(), ADC\_GetConvertResult()

## 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アラ出力(SCLK)の設定

関数のプロトタイプ宣言:

```
void
ADC_SetClk(TSB_AD_TypeDef * ADx,
           uint32_t Sample_HoldTime,
           ADC_PRESCALER Prescaler_Output)
```

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_ADB**: AD コンバータユニット B

**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**: TSH<0:3> に“1001b”をライトします。

**Prescaler\_Output**: 以下から ADC プリスケアラ出力(ADCLK)を選択します。

- **ADC\_FC\_DIVIDE\_LEVEL\_NONE**: fc
- **ADC\_FC\_DIVIDE\_LEVEL\_2**: fc / 2
- **ADC\_FC\_DIVIDE\_LEVEL\_4**: fc / 4
- **ADC\_FC\_DIVIDE\_LEVEL\_8**: fc / 8
- **ADC\_FC\_DIVIDE\_LEVEL\_16**: fc / 16

機能:

**ADC\_HOLD\_FIX** としての **Sample\_HoldTime** で ADC サンプルホールド時間を設定し、**Prescaler\_Output** でプリスケアラ出力を設定します。

戻り値:

なし

## 3.2.3.2 ADC\_Enable

AD 変換の許可

**関数のプロトタイプ宣言:**

```
void  
ADC_Enable(TSB_AD_TypeDef * ADx)
```

**引数:**

**ADx:** AD 変換のユニットを指定します。

➤ **TSB\_ADB:** AD コンバータユニット B

**機能:**

AD 変換を許可します。

**戻り値:**

なし

## 3.2.3.3 ADC\_Disable

AD 変換の禁止

**関数のプロトタイプ宣言:**

```
void  
ADC_Disable(TSB_AD_TypeDef * ADx)
```

**引数:**

**ADx:** AD 変換のユニットを指定します。

➤ **TSB\_ADB:** AD コンバータユニット B

**機能:**

AD 変換を禁止します。

**戻り値:**

なし

## 3.2.3.4 ADC\_Start

AD 変換の開始

**関数のプロトタイプ宣言:**

```
void  
ADC_Start(TSB_AD_TypeDef * ADx,  
          ADC_TriggerType Trg)
```

**引数:**

**ADx:** AD 変換のユニットを指定します。

➤ **TSB\_ADB:** AD コンバータユニット B

**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

### 3.2.3.5 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(TSB_AD_TypeDef * ADx)
```

**引数:**

**ADx:** AD 変換のユニットを指定します。

➤ **TSB\_ADB:** AD コンバータユニット B

**機能:**

通常起動時の AD 変換を停止します。

**戻り値:**

なし

### 3.2.3.6 ADC\_GetConvertState

AD 変換状態の確認

**関数のプロトタイプ宣言:**

```
WorkState  
ADC_GetConvertState(TSB_AD_TypeDef * ADx,  
                    ADC_TrgType Trg)
```

**引数:**

**ADx:** AD 変換のユニットを指定します。

➤ **TSB\_ADB:** AD コンバータユニット B

**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

## 3.2.3.7 ADC\_SetMonitor

AD 監視機能の許可

関数のプロトタイプ宣言:

```
void  
ADC_SetMonitor(TSB_AD_TypeDef * ADx,  
               ADC_MonitorTypeDef * Monitor)
```

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_ADB**: AD コンバータユニット B

**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

## 3.2.3.8 ADC\_DisableMonitor

AD 監視機能の禁止

関数のプロトタイプ宣言:

```
void  
ADC_DisableMonitor(TSB_AD_TypeDef * ADx,  
                   ADC_CMPCRx CMPCRx)
```

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_ADB**: AD コンバータユニット B

**CMPCR<sub>x</sub>**: 比較制御レジスタを選択します。

➤ **ADC\_CMPCR\_0**: ADxCMPPCR0

➤ **ADC\_CMPCR\_1**: ADxCMPPCR1

機能:

**CMPCR<sub>x</sub>** で無効にする AD 監視機能を選択します。

戻り値:

なし

## 3.2.3.9 ADC\_GetConvertResult

AD 変換結果の読み出し

**関数のプロトタイプ宣言:**

```
ADC_Result  
ADC_GetConvertResult(TSB_AD_TypeDef * ADx,  
                     ADC_REGx ResultREGx)
```

**引数:**

**ADx:** AD 変換のユニットを指定します。

➤ **TSB\_ADB:** AD コンバータユニット B

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

- **ADC\_REG0:** ADxREG0
- **ADC\_REG1:** ADxREG1
- **ADC\_REG2:** ADxREG2
- **ADC\_REG3:** ADxREG3
- **ADC\_REG4:** ADxREG4
- **ADC\_REG5:** ADxREG5
- **ADC\_REG6:** ADxREG6
- **ADC\_REG7:** ADxREG7
- **ADC\_REG8:** ADxREG8
- **ADC\_REG9:** ADxREG9
- **ADC\_REG10:** ADxREG10
- **ADC\_REG11:** ADxREG11

**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 変換結果のそれぞれのビットの意味は以下です。

- **Stored(Bit0):** AD 変換結果格納状態
- **OverRun(Bit1):** オーバーランフラグ
- **ADResult(Bit4 ~ Bit15):** AD 変換結果

## 3.2.3.10 ADC\_SelectPMDTrgProgNum

AD 変換ユニットの PMD が発生するトリガ信号 PMD6 から PMD11 に対して、起動するプログラム番号 0 から 5 を選択します

**関数のプロトタイプ宣言:**

```
void  
ADC_SelectPMDTrgProgNum(TSB_AD_TypeDef * ADx,  
                        PMD_TRG_PROG_SELx SELx,  
                        uint8_t MacroProgNum)
```

**引数:**

**ADx:** AD 変換のユニットを指定します。

➤ **TSB\_ADB:** AD コンバータユニット B

**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\_PROG\_SEL6:** ADxPSEL6
- **PMD\_TRG\_PROG\_SEL7:** ADxPSEL7
- **PMD\_TRG\_PROG\_SEL8:** ADxPSEL8
- **PMD\_TRG\_PROG\_SEL9:** ADxPSEL9
- **PMD\_TRG\_PROG\_SEL10:** ADxPSEL10
- **PMD\_TRG\_PROG\_SEL11:** ADxPSEL11

**MacroProgNum:** どの PMD<sub>x</sub> (x = 6 ~ 11)トリガによってプログラムを起動するか選択します。

- **TRG\_ENABLE(PMD\_PROG<sub>y</sub>):** 有効にする PMDトリガ用プログラム y ( y = 0 ~ 5)
- **TRG\_DISABLE(PMD\_PROG<sub>y</sub>):** 無効にする PMDトリガ用プログラム y ( y = 0 ~ 5)

**機能:**

**SEL<sub>x</sub>** 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

### 3.2.3.11 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(TSB_AD_TypeDef * ADx,  
                      PMD_TrgProgINTTypeDef * TrgProgINT)
```

**引数:**

**ADx:** AD 変換のユニットを指定します。

- **TSB\_ADB:** AD コンバータユニット B

**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

## 3.2.3.12 ADC\_SetPMDTrg

PMD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_SetPMDTrg(TSB_AD_TypeDef * ADx,  
               PMD_TrgTypeDef * PMDTrg)
```

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_ADB**: AD コンバータユニット B

**PMDTrg**: PMDトリガ用プログラムレジスタの割り込み設定の構造体。

```
typedef struct {  
    PMD_PROGx ProgNum;  
    VE_PHASE Reg0_Phase;  
    VE_PHASE Reg1_Phase;  
    VE_PHASE Reg2_Phase;  
    VE_PHASE Reg3_Phase;  
    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:

**PMDTrg** により PMDトリガ用プログラムレジスタを設定します。

戻り値:

なし

## 3.2.3.13 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(TSB_AD_TypeDef * ADx,  
                 ADC_REGx ResultREGx,  
                 uint8_t MacroAINx)
```

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_ADB**: AD コンバータユニット B

**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**: ADxREG0
- **ADC\_REG1**: ADxREG1
- **ADC\_REG2**: ADxREG2
- **ADC\_REG3**: ADxREG3
- **ADC\_REG4**: ADxREG4
- **ADC\_REG5**: ADxREG5

- **ADC\_REG6:** ADxREG6
- **ADC\_REG7:** ADxREG7
- **ADC\_REG8:** ADxREG8
- **ADC\_REG9:** ADxREG9
- **ADC\_REG10:** ADxREG10
- **ADC\_REG11:** ADxREG11

**MacroAINx:** 以下から、許可/禁止付 AD チャンネルを選択します。

- **TRG\_ENABLE(y):** **ResultREGx** に対する AD チャンネル'y'を許可
- **TRG\_DISABLE(y):** **ResultREGx** に対する AD チャンネル'y'を禁止  
以下から、'y'を選択します。  
ADC\_AIN9 ~ ADC\_AIN12, ADC\_AIN16

**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

### 3.2.3.14 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(TSB_AD_TypeDef * ADx,  
              ADC_REGx ResultREGx,  
              uint8_t MacroAINx)
```

**引数:**

**ADx:** AD 変換のユニットを指定します。

- **TSB\_ADB:** AD コンバータユニット B

**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:** ADxREG0
- **ADC\_REG1:** ADxREG1
- **ADC\_REG2:** ADxREG2
- **ADC\_REG3:** ADxREG3
- **ADC\_REG4:** ADxREG4
- **ADC\_REG5:** ADxREG5
- **ADC\_REG6:** ADxREG6
- **ADC\_REG7:** ADxREG7
- **ADC\_REG8:** ADxREG8
- **ADC\_REG9:** ADxREG9
- **ADC\_REG10:** ADxREG10
- **ADC\_REG11:** ADxREG11

**MacroAINx:** 以下から、許可/禁止付 AD チャンネルを選択します。

- **TRG\_ENABLE(y):** **ResultREGx** に対する AD チャンネル'y'を許可
- **TRG\_DISABLE(y):** **ResultREGx** に対する AD チャンネル'y'を禁止  
以下から、'y'を選択します。

ADC\_AIN9 to ADC\_AIN12, ADC\_AIN16

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

### 3.2.3.15 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(TSB_AD_TypeDef * ADx,  
                   ADC_REGx ResultREGx,  
                   uint8_t MacroAINx)
```

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_ADB**: AD コンバータユニット B

**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**: ADxREG0
- **ADC\_REG1**: ADxREG1
- **ADC\_REG2**: ADxREG2
- **ADC\_REG3**: ADxREG3
- **ADC\_REG4**: ADxREG4
- **ADC\_REG5**: ADxREG5
- **ADC\_REG6**: ADxREG6
- **ADC\_REG7**: ADxREG7
- **ADC\_REG8**: ADxREG8
- **ADC\_REG9**: ADxREG9
- **ADC\_REG10**: ADxREG10
- **ADC\_REG11**: ADxREG11

**MacroAINx**: 以下から、許可/禁止付 AD チャンネルを選択します。

- **TRG\_ENABLE(y)**: **ResultREGx** に対する AD チャンネル'y'を許可
  - **TRG\_DISABLE(y)**: **ResultREGx** に対する AD チャンネル'y'を禁止
- 以下から、'y'を選択します。

ADC\_AIN9 to ADC\_AIN12, ADC\_AIN16

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\_CMPCR<sub>x</sub>

**CMPCR<sub>x</sub>** 以下からコンペア制御レジスタを選択します。

➤ **ADC\_CMPCR\_0**: ADxCMPCR0

➤ **ADC\_CMPCR\_1**: ADxCMPCR1

ADC\_REG<sub>x</sub>

**ResultREG<sub>x</sub>** 以下から AD 変換結果レジスタを選択します。

➤ **ADC\_REG0**: ADxREG0

➤ **ADC\_REG1**: ADxREG1

➤ **ADC\_REG2**: ADxREG2

➤ **ADC\_REG3**: ADxREG3

➤ **ADC\_REG4**: ADxREG4

➤ **ADC\_REG5**: ADxREG5

➤ **ADC\_REG6**: ADxREG6

➤ **ADC\_REG7**: ADxREG7

➤ **ADC\_REG8**: ADxREG8

➤ **ADC\_REG9**: ADxREG9

➤ **ADC\_REG10**: ADxREG10

➤ **ADC\_REG11**: ADxREG11

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 ADxREG<sub>m</sub> と ADxCMP<sub>n</sub> の比較設定を選択します。(m = 0 ~ 11, x = B, n = 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。(x = B)

➤ 0 ~ 4095

### 3.2.4.2 PMD\_TrgProgINTTypeDef

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\_INTADPDB**: INTADPDB 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。

### 3.2.4.3 PMD\_TrgTypeDef

メンバ:

PMD\_PROGx

**ProgNum** 以下から、ADxPSETn (x = B、n = 0 ~ 5) に対するプログラム番号を選択します。

- **PMD\_PROG0**: プログラム番号 0
- **PMD\_PROG1**: プログラム番号 1
- **PMD\_PROG2**: プログラム番号 2
- **PMD\_PROG3**: プログラム番号 3
- **PMD\_PROG4**: プログラム番号 4
- **PMD\_PROG5**: プログラム番号 5

VE\_PHASE

**Reg0\_Phase** 以下から、ADxPSETn (x = B、n = 0 ~ 5) に対する REG0 の相を選択します。

- **VE\_PHASE\_NONE**: 指定なし
- **VE\_PHASE\_U**: U 相
- **VE\_PHASE\_V**: V 相
- **VE\_PHASE\_W**: W 相

VE\_PHASE

**Reg1\_Phase** ADxPSETn (x = B、n = 0 ~ 5) に対する REG1 の相を選択します。  
選択する相は **Reg0\_Phase** と同じです。

VE\_PHASE

**Reg2\_Phase** ADxPSETn (x = B、n = 0 ~ 5) に対する REG2 の相を選択します。  
選択する相は **Reg0\_Phase** と同じです。

VE\_PHASE

**Reg3\_Phase** ADxPSETn (x = B、n = 0 ~ 5) に対する REG3 の相を選択します。  
選択する相は **Reg0\_Phase** と同じです。

uint8\_t

**Reg0\_AINx** 以下から、ADxPSETn の REG0 に対する許可/禁止付 AD チャンネルを選択します。

- **TRG\_ENABLE(y)**: AD チャンネル'y'を許可

- **TRG\_DISABLE(y)**: AD チャンネル'y'を禁止  
以下から、'y'を選択します。  
ADC\_AIN9 ~ ADC\_AIN12, ADC\_AIN16

uint8\_t

**Reg1\_AINx** ADxPSETn の REG1 に対する許可/禁止付 AD チャンネルを選択します。  
選択する AD チャンネルは **Reg0\_AINx** と同じです。

uint8\_t

**Reg2\_AINx** ADxPSETn の REG2 に対する許可/禁止付 AD チャンネルを選択します。  
選択する AD チャンネルは **Reg0\_AINx** と同じです。

uint8\_t

**Reg3\_AINx** ADxPSETn の REG3 に対する許可/禁止付 AD チャンネルを選択します。  
選択する AD チャンネルは **Reg0\_AINx** と同じです。

## 3.2.4.4 ADC\_Result

メンバ:

uint32\_t

**All**: AD 変換結果

**Bit**

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

## 4. CG

### 4.1 概要

本 CG API は TMPM375FSDMG 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、クロック逡倍回路 (PLL) の制御
- プリスケールクロックの制御
- 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

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/tmpm375\_cg.c

/Libraries/TX03\_Periph\_Driver/inc/tmpm375\_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。

**fosc1** : X1, X2 端子より入力されるクロック

**fosc2** : 内蔵発振器より入力されるクロック

**fosc** : fosc1 または fosc2 どちらか選択されたシステムクロック

**fPLL** : PLL により逡倍(4 逡倍) 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 ( プリスケールクロック)

### 4.2 API 関数

#### 4.2.1 関数一覧

- ◆ void CG\_SetFgearLevel(CG\_DivideLevel **DivideFgearFromFc**);
- ◆ CG\_DivideLevel CG\_GetFgearLevel(void);
- ◆ void CG\_SetPhiT0Src(CG\_PhiT0Src **PhiT0Src**);
- ◆ CG\_PhiT0Src CG\_GetPhiT0Src(void);
- ◆ Result CG\_SetPhiT0Level(CG\_DivideLevel **DividePhiT0FromFc**);
- ◆ CG\_DivideLevel CG\_GetPhiT0Level(void);
- ◆ void CG\_SetWarmUpTime(CG\_WarmUpSrc **Source**, uint16\_t **Time**);
- ◆ void CG\_StartWarmUp(void);
- ◆ WorkState CG\_GetWarmUpState(void);
- ◆ Result CG\_SetFPLLValue(uint32\_t NewValue);
- ◆ uint32\_t CG\_GetFPLLValue(void);
- ◆ Result CG\_SetPLL(FunctionalState **NewState**);
- ◆ FunctionalState CG\_GetPLLState(void);
- ◆ Result CG\_SetFosc(CG\_FoscSrc **Source**, FunctionalState **NewState**);
- ◆ void CG\_SetFoscSrc(CG\_FoscSrc **Source**);

- ◆ CG\_FoscSrc CG\_GetFoscSrc(void);
- ◆ FunctionalState CG\_GetFoscState(CG\_FoscSrc **Source**);
- ◆ void CG\_SetPortM(CG\_PortMMode **Mode**);
- ◆ void CG\_SetSTBYMode(CG\_STBYMode **Mode**);
- ◆ CG\_STBYMode CG\_GetSTBYMode(void);
- ◆ void CG\_SetPinStateInStopMode(FunctionalState **NewState**);
- ◆ FunctionalState CG\_GetPinStateInStopMode(void);
- ◆ Result CG\_SetFcSrc(CG\_FcSrc **Source**);
- ◆ CG\_FcSrc CG\_GetFcSrc(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\_SetWarmUpTime(),  
CG\_StartWarmUp(), CG\_GetWarmUpState(), CG\_SetFPLLValue(),  
CG\_GetFPLLValue(), CG\_SetPLL(), CG\_GetPLLState(), CG\_SetFosc(),  
CG\_GetFoscState(), CG\_SetFcSrc(), CG\_GetFcSrc(), CG\_SetFoscSrc(),  
CG\_GetFoscSrc(), CG\_SetPortM()

2) スタンバイモードの設定:

CG\_SetSTBYMode(), CG\_GetSTBYMode(), 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:** 無効データ

## 4.2.3.3 CG\_SetPhiT0Src

PhiT0(ΦT0) ,fc 間の PhiT0(ΦT0) ソースの設定

関数のプロトタイプ宣言:

void

CG\_SetPhiT0Src(CG\_PhiT0Src **PhiT0Src**)

引数:

**PhiT0Src:** 以下から PhiT0 ソースを選択します。

➤ **CG\_PHIT0\_SRC\_FGEAR** : fgear が PhiT0 ソース

➤ **CG\_PHIT0\_SRC\_FC**: fc が PhiT0 ソース

機能:

PhiT0 (ΦT0) ソースを選択します。

戻り値:

なし

## 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 ΦT0 ソース**CG\_PHIT0\_SRC\_FC**: fc が ΦT0 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
- **CG\_DIVIDE\_2**: ΦT0 = fc/2
- **CG\_DIVIDE\_4**: ΦT0 = fc/4
- **CG\_DIVIDE\_8**: ΦT0 = fc/8
- **CG\_DIVIDE\_16**: ΦT0 = fc/16
- **CG\_DIVIDE\_32**: ΦT0 = fc/32
- **CG\_DIVIDE\_64**: ΦT0 = fc/64
- **CG\_DIVIDE\_128**: ΦT0 = fc/128
- **CG\_DIVIDE\_256**: ΦT0 = fc/256
- **CG\_DIVIDE\_512**: ΦT0 = fc/512

**機能:**

プリスケ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( $\Phi T0$ ),  $f_c$  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( $\Phi T0$ ),  $f_c$  間の分周レベルを以下から設定します。

**CG\_DIVIDE\_1:**  $\Phi T0 = f_c$   
**CG\_DIVIDE\_2:**  $\Phi T0 = f_c/2$   
**CG\_DIVIDE\_4:**  $\Phi T0 = f_c/4$   
**CG\_DIVIDE\_8:**  $\Phi T0 = f_c/8$   
**CG\_DIVIDE\_16:**  $\Phi T0 = f_c/16$   
**CG\_DIVIDE\_32:**  $\Phi T0 = f_c/32$   
**CG\_DIVIDE\_64:**  $\Phi T0 = f_c/64$   
**CG\_DIVIDE\_128:**  $\Phi T0 = f_c/128$   
**CG\_DIVIDE\_256:**  $\Phi T0 = f_c/256$   
**CG\_DIVIDE\_512:**  $\Phi T0 = f_c/512$   
**CG\_DIVIDE\_UNKNOWN:** 無効データ

#### 4.2.3.7 CG\_SetWarmUpTime

ウォームアップ時間の設定

**関数のプロトタイプ宣言:**

void  
CG\_SetWarmUpTime (CG\_WarmUpSrc **Source**,  
uint16\_t **Time**)

**引数:**

**Source:** 以下から、ウォームアップカウンタを選択します。

- **CG\_WARM\_UP\_SRC\_OSC1:** fosc1
- **CG\_WARM\_UP\_SRC\_OSC2:** fosc2

**Time:** 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 =  $100 \times 10E(-6) / (1 / (8 \times 10E(6))) / 16 = 0x0320 >> 4 = 0x32$

**戻り値:**

なし

#### 4.2.3.8 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

#### 4.2.3.9 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

#### 4.2.3.10 CG\_SetFPLLValue

PLL の逡倍数を設定

関数のプロトタイプ宣言:  
Result  
CG\_SetFPLLValue(uint32\_t **NewValue**)

引数:  
**NewValue:** 以下から逡倍数を選択します。  
➤ **CG\_FPLL\_8M\_MULTIPLY\_5:** 入力クロック 8MHz、出力クロック 40MHz(5 逡倍)  
➤ **CG\_FPLL\_10M\_MULTIPLY\_4:** 入力クロック 10MHz、出力クロック 40MHz(4 逡倍)



**機能:**

PLL の通倍数を設定します。

**戻り値:**

**SUCCESS:** 成功

**ERROR:** 失敗

#### 4.2.3.11 CG\_GetFPLLValue

PLL の通倍数を取得

**関数のプロトタイプ宣言:**

uint32\_t

CG\_GetFPLLValue(void)

**引数:**

なし。

**機能:**

PLL の通倍数を取得します。

**戻り値:**

PLL の通倍数

- **CG\_FPLL\_8M\_MULTIPLY\_5:** 入力クロック 8MHz、出力クロック 40MHz(5 通倍)
- **CG\_FPLL\_10M\_MULTIPLY\_4:** 入力クロック 10MHz、出力クロック 40MHz(4 通倍)

#### 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\_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

## 4.2.3.19 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\_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

## 4.2.3.20 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\_IDLE: IDLE モード

CG\_STBY\_MODE\_UNKNOWN: 無効データ

#### 4.2.3.21 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

#### 4.2.3.22 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)

#### 4.2.3.23 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

#### 4.2.3.24 CG\_GetFcSrc

fc ソースクロックの取得

**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

#### 4.2.3.25 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\_6** : INT6
- **CG\_INT\_SRC\_7** : INT7
- **CG\_INT\_SRC\_C** : INTC

**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\_RISING**: 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

#### 4.2.3.26 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\_6, CG\_INT\_SRC\_7, CG\_INT\_SRC\_C**

**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\_RISING**: 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

#### 4.2.3.27 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\_6, CG\_INT\_SRC\_7, CG\_INT\_SRC\_C**

**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

#### 4.2.3.28 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

#### 4.2.3.29 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

**VLTDReset (Bit 3) :** VLTD 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* CGNMI ソース起動状態です。

ビットフィールド:

uint32\_t

**WDT**(Bit 0) WDT 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

**VLTDReset** (Bit 3) 低電圧検出フラグ

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. ENC

### 5.1 概要

本デバイスは、エンコーダ入力回路を 1 本内蔵しています(ENC0)。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 通倍 (6 通倍) 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、チャンネル選択、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\\tmpm375\_enc.c  
\\Libraries\\TX03\_Periph\_Driver\\inc\\tmpm375\_enc.h

### 5.2 API 関数

#### 5.2.1 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**);

## 5.2.2 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()

## 5.2.3 関数仕様

### 5.2.3.1 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:** 以下から、ENC チャンネルを選択します。

➤ **EN0**

**機能:**

エンコーダ動作を許可します。

**戻り値:**

なし

### 5.2.3.2 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:** 以下から、ENC チャンネルを選択します。

➤ **EN0**

**機能:**

エンコーダ動作を禁止します。

**戻り値:**

なし

### 5.2.3.3 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**: 以下から、ENC チャンネルを選択します。

➤ **EN0**

**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

## 5.2.3.4 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**: 以下から、ENC チャンネルを選択します。

➤ **EN0**

**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

## 5.2.3.5 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**: 以下から、ENC チャンネルを選択します。

➤ **EN0**

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

## 5.2.3.6 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**: 以下から、ENC チャンネルを選択します。

➤ **EN0**

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

## 5.2.3.7 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**: 以下から、ENC チャンネルを選択します。

➤ **EN0**

**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

## 5.2.3.8 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**: 以下から、ENC チャンネルを選択します。

➤ **EN0**

**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

## 5.2.3.9 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**: 以下から、ENC チャンネルを選択します。

➤ **EN0**

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

## 5.2.3.10 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**: 以下から、ENC チャンネルを選択します。

➤ **EN0**

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

## 5.2.4 データ構造

### 5.2.4.1 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層入力選択

- **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

**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

## 5.2.4.2 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



## 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 64Kbyte 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\tmpm375_fc.c  
\Libraries\TX03_Periph_Driver\inc\tmpm375_fc.h
```

### 6.2 API 関数

#### 6.2.1 関数一覧

- ◆ void FC\_SetBufferState (FunctionalState **NewState**)
- ◆ 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 5 種類に分かれています。

- 1) セキュリティ設定(Flash ROM データの読み出し、デバッグ):  
FC\_SetSecurityBit(), FC\_GetSecurityBit()
- 2) 自動動作状態およびプロテクト状態の取得:  
FC\_GetBusyState(), FC\_GetBlockProtectState().
- 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().
- 5) フラッシュバッファの制御:  
FC\_SetBufferState()

## 6.2.3 関数仕様

### 6.2.3.1 FC\_SetBufferState

フラッシュバッファの無効/有効

関数のプロトタイプ宣言:

void

FC\_SetBufferState (FunctionalState **NewState**)

引数:

**NewState**: 以下からフラッシュバッファの無効/有効を選択します。

- **DISABLE**: 無効(同時にバッファをクリアします)
- **ENABLE**: 有効

機能:

書き込みまたは消去を行った後に無効→許可を行い、バッファクリアを行ってください。

### 6.2.3.2 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および低消費電力モードのSTOP2解除で初期化されます。

戻り値:

なし

### 6.2.3.3 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

## 6.2.3.4 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

## 6.2.3.5 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

➤ **FC\_BLOCK\_1** block 1

**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

## 6.2.3.6 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
- **FC\_BLOCK\_1** block 1

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)

## 6.2.3.7 FC\_EraseBlockProtectState

プロテクトの解除

関数のプロトタイプ宣言:

FC\_Result

FC\_EraseBlockProtectState(uint8\_t **BlockGroup**)

引数:

**BlockGroup**: ブロックグループを指定してください。

- **FC\_BLOCK\_GROUP\_0**: ブロック 0, 1

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)

## 6.2.3.8 FC\_WritePage

ページ単位の書き込み

関数のプロトタイプ宣言:

FC\_Result

FC\_WritePage(uint32\_t **PageAddr**, uint32\_t \* **Data**)

引数:

**PageAddr**: ページの開始アドレスを指定します。

**Data**: 書き込むデータバッファへのポインタを指定します。サイズは 128Byte です。

**機能:**

ページ書き込みを行います。

自動ページ書き込みは、既に消去された 1 ページにつき一回のみ実施されます。データ値が“1” または “0” のいずれかであっても、2 回以上書き込みを実施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)

## 6.2.3.9 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)

## 6.2.3.10 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/ tmpm375 \_gpio.c

/Libraries/TX03\_Periph\_Driver/inc/tmpm375 \_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\_PB**: GPIO port B
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PJ**: GPIO port J
- **GPIO\_PK**: GPIO port K
- **GPIO\_PM**: GPIO port M

機能:

DATAレジスタを読み込みます。

戻り値:

DATAレジスタの値

### 7.2.3.2 GPIO\_ReadDataBit

ビット単位での DATAレジスタの読み込み

関数のプロトタイプ宣言:

uint8\_t

GPIO\_ReadDataBit(GPIO\_Port **GPIO\_x**,  
uint8\_t **Bit\_x**)

引数:

**GPIO\_x**: GPIO ポートを選択します。

- **GPIO\_PB**: GPIO port B
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PJ**: GPIO port J
- **GPIO\_PK**: GPIO port K
- **GPIO\_PM**: GPIO port M

**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:**

DATA データレジスタのビット値:

- **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\_PB:** GPIO port B
- **GPIO\_PE:** GPIO port E
- **GPIO\_PF:** GPIO port F
- **GPIO\_PG:** GPIO port G
- **GPIO\_PJ:** GPIO port J
- **GPIO\_PK:** GPIO port K
- **GPIO\_PM:** GPIO port M

**Data:** DATA レジスタに書き込む値を設定します。

**機能:**

DATA レジスタにデータを書き込みます。

**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\_PB:** GPIO port B

- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PJ**: GPIO port J
- **GPIO\_PK**: GPIO port K
- **GPIO\_PM**: GPIO port M

**Bit\_x**: GPIO 端子を選択します。

- **GPIO\_BIT\_0**: GPIO pin 0
- **GPIO\_BIT\_1**: GPIO pin 1
- **GPIO\_BIT\_2**: GPIO pin 2
- **GPIO\_BIT\_3**: GPIO pin 3
- **GPIO\_BIT\_4**: GPIO pin 4
- **GPIO\_BIT\_5**: GPIO pin 5
- **GPIO\_BIT\_6**: GPIO pin 6
- **GPIO\_BIT\_7**: GPIO pin 7
- **GPIO\_BIT\_ALL**: すべての GPIO 端子

**BitValue**: GPIO ビットに書き込む値

- **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\_PB**: GPIO port B
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PJ**: GPIO port J
- **GPIO\_PK**: GPIO port K
- **GPIO\_PM**: GPIO port M

**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

ポートの出力設定

**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\_PB:** GPIO port B
- **GPIO\_PE:** GPIO port E
- **GPIO\_PF:** GPIO port F
- **GPIO\_PG:** GPIO port G
- **GPIO\_PJ:** GPIO port J
- **GPIO\_PK:** GPIO port K
- **GPIO\_PM:** GPIO port M

**Bit\_x:** GPIO 端子を選択します。

- **GPIO\_BIT\_0:** GPIO pin 0
- **GPIO\_BIT\_1:** GPIO pin 1
- **GPIO\_BIT\_2:** GPIO pin 2
- **GPIO\_BIT\_3:** GPIO pin 3
- **GPIO\_BIT\_4:** GPIO pin 4
- **GPIO\_BIT\_5:** GPIO pin 5
- **GPIO\_BIT\_6:** GPIO pin 6
- **GPIO\_BIT\_7:** GPIO pin 7
- **GPIO\_BIT\_ALL:** 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

ポートの入力設定

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\_PB**: GPIO port B
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PJ**: GPIO port J
- **GPIO\_PK**: GPIO port K
- **GPIO\_PM**: GPIO port M

**Bit\_x**: GPIO 端子を選択します。

- **GPIO\_BIT\_0**: GPIO pin 0
- **GPIO\_BIT\_1**: GPIO pin 1
- **GPIO\_BIT\_2**: GPIO pin 2
- **GPIO\_BIT\_3**: GPIO pin 3
- **GPIO\_BIT\_4**: GPIO pin 4
- **GPIO\_BIT\_5**: GPIO pin 5
- **GPIO\_BIT\_6**: GPIO pin 6
- **GPIO\_BIT\_7**: GPIO pin 7
- **GPIO\_BIT\_ALL**: 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\_PB**: GPIO port B
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PJ**: GPIO port J
- **GPIO\_PK**: GPIO port K

- **GPIO\_PM**: GPIO port M

**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:**

出力ポートの許可/禁止設定を行います。

**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\_PB**: GPIO port B
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PJ**: GPIO port J
- **GPIO\_PK**: GPIO port K
- **GPIO\_PM**: GPIO port M

**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:**

入力ポートの許可/禁止設定を行います。

**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

ポートのプルアップ設定

**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\_PB**: GPIO port B
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PJ**: GPIO port J
- **GPIO\_PK**: GPIO port K
- **GPIO\_PM**: GPIO port M

**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:**

ポートのプルアップ設定を行います。

**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

ポートのプルダウン設定

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\_PB**: GPIO port B
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PJ**: GPIO port J
- **GPIO\_PK**: GPIO port K
- **GPIO\_PM**: GPIO port M

**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:

ポートのプルダウン設定を行います。

**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

ポートの CMOS/オープンドレイン設定

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\_PB**: GPIO port B
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PJ**: GPIO port J
- **GPIO\_PK**: GPIO port K
- **GPIO\_PM**: GPIO port M

**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:

ポートの CMOS/オープンドレイン設定を行います。

**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

ポートの機能設定

関数のプロトタイプ宣言:

```
void  
GPIO_EnableFuncReg(GPIO_Port GPIO_x,  
                   uint8_t FuncReg_x,  
                   uint8_t Bit_x);
```

引数:

**GPIO\_x**: GPIO ポートを選択します。

- **GPIO\_PB**: GPIO port B
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G



**FuncReg\_x:** GPIO 機能レジスタの番号を選択します。

- **GPIO\_FUNC\_REG\_1:** GPIO 機能レジスタ 1
- **GPIO\_FUNC\_REG\_2:** GPIO 機能レジスタ 2
- **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:**

ポートの機能設定を行います。

**戻り値:**

なし

## 7.2.3.14 GPIO\_DisableFuncReg

ポートの機能設定解除

**関数のプロトタイプ宣言:**

```
void  
GPIO_DisableFuncReg(GPIO_Port GPIO_x,  
                     uint8_t FuncReg_x,  
                     uint8_t Bit_x)
```

**引数:**

**GPIO\_x:** GPIO ポートを選択します。

- **GPIO\_PB:** GPIO port B
- **GPIO\_PE:** GPIO port E
- **GPIO\_PF:** GPIO port F
- **GPIO\_PG:** GPIO port G

**FuncReg\_x:** GPIO 機能レジスタの番号を選択します。

- **GPIO\_FUNC\_REG\_1:** GPIO 機能レジスタ 1
- **GPIO\_FUNC\_REG\_2:** GPIO 機能レジスタ 2
- **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:**

ポートの機能設定を解除します。

**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. SBI

### 8.1 概要

本デバイスはシリアルバスインターフェース(SBI)が 1 本あり、マルチマスタ動作、または 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 チャン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/tmpm375\_sbi.c  
/Libraries/TX03\_Periph\_Driver/inc/tmpm375\_sbi.h

### 8.2 API 関数

#### 8.2.1 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**);

#### 8.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。

- 1) 共通機能の設定:  
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()

## 8.2.3 関数仕様

### 8.2.3.1 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 チャンネルを指定します。

機能:

SBI 動作を有効にします。

戻り値:

なし

### 8.2.3.2 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 チャンネルを指定します。

機能:

SBI 動作を無効にします。

戻り値:

なし

### 8.2.3.3 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 チャンネルを指定します。

**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

## 8.2.3.4 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 チャンネルを指定します。

**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

## 8.2.3.5 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 チャンネルを指定します。

**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。

### 8.2.3.6 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 チャンネルを指定します。

**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。

### 8.2.3.7 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 チャンネルを指定します。

**機能:**

SBI 割り込み要求を解除します。

**戻り値:**

なし。

## 8.2.3.8 SBI\_Generatel2CStart

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_Generatel2CStart(TSB_SBI_TypeDef* SBIx)
```

**引数:**

**SBIx:** SBI チャンネルを指定します。

**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。

## 8.2.3.9 SBI\_Generatel2CStop

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_Generatel2CStop(TSB_SBI_TypeDef* SBIx)
```

**引数:**

**SBIx:** SBI チャンネルを指定します。

**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。

## 8.2.3.10 SBI\_GetI2CState

I2C バスモードにおける SBI チャン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 チャンネルを指定します。

**機能:**

I2C バスモード中の SBI チャン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 チャン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 チャンネルの状態。

## 8.2.3.11 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 チャンネルを指定します。

**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 チャンネルは動作します。  
**DISABLE** を選択すると IDLE モード時に禁止されます。

戻り値:

なし。

## 8.2.3.12 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 チャンネルを指定します。

**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。

## 8.2.3.13 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 チャンネルを指定します。

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。

## 8.2.3.14 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 チャンネルを指定します。

**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。

## 8.2.4 データ構造:

### 8.2.4.1 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 チャン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。

## 8.2.4.2 SBI\_I2CState

メンバ:

uint32\_t

**All:** I2C モードの全ての状態

Bit Fields:

uint32\_t

**LastRxBit:** 最終受信ビットモニタ

uint32\_t

**GeneralCall:** ゼネラルコール検出モニタ

uint32\_t

**SlaveAddrMatch:** スレーブアドレス一致モニタ

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

## 9. OFD

### 9.1 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/tmpm375\_ofd.c

/Libraries/TX03\_Periph\_Driver/inc/tmpm375\_ofd.h

### 9.2 API 関数

#### 9.2.1 関数一覧

- ◆ void OFD\_SetRegWriteMode(FunctionalState NewState);
- ◆ void OFD\_Enable(void);
- ◆ void OFD\_Disable(void);
- ◆ void OFD\_SetDetectionFrequency(OFD\_PLL\_State State,  
uint32\_t HigherDetectionCount,  
uint32\_t LowerDetectionCount);

#### 9.2.2 関数の種類

OFD 回路の初期化と設定:

OFD\_SetRegWriteMode(), OFD\_SetDetectionFrequency (), OFD\_Enable (),  
OFD\_Disable ()

#### 9.2.3 関数仕様

##### 9.2.3.1 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** :

OFDCR2/OFDMNPLLON/OFDMNPLLOFF/OFDMXPLLON/OFDMXPLLOFF レジスタの書き込みステータス

下記のいずれかの値を選択します。

- **ENABLE:**  
OFDCR2/OFDMNPLLON/OFDMNPLLOFF/OFDMXPLLON/OFDMXPLLOFF レジスタに書き込み許可
- **DISABLE:**  
OFDCR2/OFDMNPLLON/OFDMNPLLOFF/OFDMXPLLON/OFDMXPLLOFF レジスタに書き込み禁止

**機能:**

本関数は、**NewState** が **ENABLE** の時に、OFDCR2/OFDMNPLLON/OFDMNPLLOFF/OFDMXPLLON/OFDMXPLLOFF 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

### 9.2.3.2 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

### 9.2.3.3 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

## 9.2.3.4 OFD\_SetDetectionFrequency

検出周波数のカウント値を設定

関数のプロトタイプ宣言:

```
void  
OFD_SetDetectionFrequency(OFD_PLL_State State,  
                           Uint32_t HigherDetectionCount,  
                           Uint32_t LowerDetectionCount)
```

引数:

**State**: PLL の状態を下記から選択します。

- **OFD\_PLL\_ON**: PLL ON 時
- **OFD\_PLL\_OFF**: PLL OFF 時

**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、検出周波数カウント値、PLL OFF, PLL ON を設定します。

戻り値:

なし

## 9.2.4 データ構造:

なし

## 10. TMRB

### 10.1 概要

本デバイスは、4 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, TMRB4, TMRB5, TMRB7)を内蔵しています。各チャンネルは下記モードで動作します。

- 16-bit interval timer mode
- 16-bit event counter mode
- 16-bit programmable pulse generation mode (PPG)
- External trigger Programmable pulse generation mode (PPG)
- 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
- 外部トリガ用プログラマブル矩形波出力(PPG)モード

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

本ドライバは、クロック分割、サイクル、デューティ期間、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ン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/tmpm375\_tmr.c  
/Libraries/TX03\_Periph\_Driver/inc/tmpm375\_tmr.h

### 10.2 API 関数

#### 10.2.1 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\_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**);

## 10.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。

- 1) 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\_SetDoubleBuf(), TMRB\_SetExtStartTrg(), TMRB\_SetClkInCoreHalt ()

## 10.2.3 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\_TB4, TSB\_TB5, TSB\_TB7.**

### 10.2.3.1 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 チャンネルを指定します。

機能:

TMRB 動作を有効にします。

戻り値:

なし

### 10.2.3.2 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 チャンネルを指定します。

**機能:**

TMRB 動作を無効にします。

**戻り値:**

なし

### 10.2.3.3 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 チャンネルを指定します。

**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

### 10.2.3.4 TMRB\_Init

TMRB チャンネルの初期化

**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 チャンネルを指定します。

**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、デューティ期間の初期設定を行います。

**戻り値:**



なし

補足:

指定されたチャンネルが TBxIN を持たない場合、*InitStruct->mode* は **TMRB\_INTERVAL\_TIMER** しか選択できません。

## 10.2.3.5 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**: 以下から、TMRB チャンネルを指定します。

TSB\_TB0, TSB\_TB4, TSB\_TB7

**CaptureTiming**: キャプチャタイミングを選択します。

- **TMRB\_DISABLE\_CAPTURE**: キャプチャ機能を無効にします。
- **TMRB\_CAPTURE\_IN\_RISING**: TBxIN↑  
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む
- **TMRB\_CAPTURE\_IN\_RISING\_FALLING**: TBxIN↑  
TBxIN↓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む
- **TMRB\_CAPTURE\_TIMPLS\_RISING\_FALLING**: TIMPLS↑TIMPLS↓  
TIMPLS 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、  
TIMPLS 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む

補足:

TSB\_TB0 の場合、キャプチャタイミングに **TMRB\_CAPTURE\_IN\_RISING** と **TMRB\_CAPTURE\_IN\_RISING\_FALLING** を選択できません。

TSB\_TB4、または TSB\_TB7 の場合、キャプチャタイミングに **TMRB\_CAPTURE\_TIMPLS\_RISING\_FALLING** を選択できません。

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\_TIMPLS\_RISING\_FALLING** の場合、TIMPLS 入力の立ち上がりエッジでキャプチャレジスタ (TBxCP0) にカウント値を取り

込み、TIMPLS 入力の立ち下がりエッジでキャプチャレジスタ(TBxCP1)にカウント値を取り込みます。

戻り値:

なし

### 10.2.3.6 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 チャンネルを指定します。

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

機能:

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

戻り値:

なし

### 10.2.3.7 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 チャンネルを指定します。

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:

**MatchLeadintTiming**(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  
}
```

## 10.2.3.8 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 チャンネルを指定します。

**INTMask**: マスクする割り込みを選択します。

- **TMRB\_MASK\_MATCH\_LEADINGTIMING\_INT**: 一致フラグ(TBxRG0)
- **TMRB\_MASK\_MATCH\_TRAILINGTIMING\_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、オーバーフロー発生時の割り込みは発生しません。

**TMRB\_NO\_INT\_MASK** 選択時、割り込みマスクはすべてクリアされます。

戻り値:

なし

## 10.2.3.9 TMRB\_ChangeLeadintTiming

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 チャンネルを指定します。

**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 の設定と *ClkDiv*(詳細は"データ構造"を参照) の値によります。

**戻り値:**

なし。

**補足:**

*LeadingTiming* は *TrailingTiming* を超えることはできません。

## 10.2.3.10 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 チャンネルを指定します。

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

**機能:**

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

**戻り値:**

なし。

**Note:**

*TrailingTiming* は *LeadingTiming* より小さくすることはできません。また *TBxRG0/1* の値は PPG モードの *TBxRG0*<*TBxRG1* を満たすように設定してください。

## 10.2.3.11 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 チャンネルを指定します。

**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

## 10.2.3.12 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 チャンネルを指定します。

**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。**TBx** は以下のチャンネルのみ指定可能です。  
TSB\_TB0, TSB\_TB4, TSB\_TB7.

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。

## 10.2.3.13 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 チャンネルを指定します。

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

## 10.2.3.14 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 チャンネルを指定します。

**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 チャンネルは動作します。**DISABLE** の場合、IDLE 時は動作を停止します。

戻り値:

なし

## 10.2.3.15 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 チャンネルを指定します。

**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

## 10.2.3.16 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 チャンネルを指定します。

**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

## 10.2.3.17 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 チャンネルを指定します。

**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

## 10.2.4 データ構造:

### 10.2.4.1 TMRB\_InitTypeDef

メンバ:

uint32\_t

**Mode:** タイマモードを選択します。

- **TMRB\_INTERVAL\_TIMER:** インタバルタイマモード
- **TMRB\_EVENT\_CNT:** イベントカウンタモード

**補 足 :** TBxIN を持たないチャネルは **InitStruct->mode** に **TMRB\_INTERVAL\_TIMER** のみ指定可能です。

uint32\_t

**ClkDiv:** インタバルタイマのソースクロックの分周を選択します。

- **TMRB\_CLK\_DIV\_2:** fperiph / 2
- **TMRB\_CLK\_DIV\_8:** fperiph / 8
- **TMRB\_CLK\_DIV\_32:** fperiph / 32
- **TMRB\_CLK\_DIV\_64:** fperiph / 64
- **TMRB\_CLK\_DIV\_128:** fperiph / 128
- **TMRB\_CLK\_DIV\_256:** fperiph / 256
- **TMRB\_CLK\_DIV\_512:** fperiph / 512

uint32\_t

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

uint32\_t

**UpCntCtrl:** アップカウンタの動作を選択します。

- **TMRB\_FREE\_RUN:** 周期が一致した後も、0xFFFF になるまでアップカウンタは停止しません。その後、カウンタがクリアされ、0 からカウントを開始します。
- **TMRB\_AUTO\_CLEAR:** **TrailingTiming** と一致したときに、0 クリアされ、再スタートします。

uint32\_t

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

### 10.2.4.2 TMRB\_FFOutputTypeDef

メンバ:

**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。

## 10.2.4.3 TMRB\_INTFactor

メンバ:

uint32\_t

**All**: TMRB 割り込み要因

**Bit**

uint32\_t

**MatchLeadingTiming**: 1 デューティとの一致検出

uint32\_t

**MatchTrailingTiming**: 1 周期との一致検出

uint32\_t

**OverFlow**: 1 オーバーフロー

uint32\_t

**Reserverd**: 29 -

## 11. SIO/UART

### 11.1 概要

本デバイスは 2 つの動作モードを持っています。チャンネル 0 は UART モード(非同期通信)と SIO モード(同期通信)を選択することができ、チャンネル 1 は UART モードのみ設定可能です。  
UART モードでは、7, 8, 9 ビット長のデータを選択可能です。

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/tmpm375\_uart.c  
/Libraries/TX03\_Periph\_Driver/inc/tmpm375\_uart.h

### 11.2 API 関数

#### 11.2.1 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\_SetInputClock(TSB\_SC\_TypeDef \* **UARTx**, uint8\_t **ClkDivider**)
- ◆ void UART\_FIFOConfig(TSB\_SC\_TypeDef \* UARTx, FunctionalState NewState);
- ◆ void UART\_SetFIFOTransferMode(TSB\_SC\_TypeDef \* UARTx,  
uint32\_t TransferMode);
- ◆ void UART\_TRxAutoDisable(TSB\_SC\_TypeDef \* UARTx,  
UART\_TRxAutoDisable TRxAutoDisable);
- ◆ void UART\_RxFIFOINTCtrl(TSB\_SC\_TypeDef \* UARTx, FunctionalState NewState);
- ◆ void UART\_TxFIFOINTCtrl(TSB\_SC\_TypeDef \* UARTx, FunctionalState NewState);
- ◆ void UART\_RxFIFOByteSel(TSB\_SC\_TypeDef \* UARTx, uint32\_t BytesUsed);
- ◆ void UART\_RxFIFOFillLevel(TSB\_SC\_TypeDef \* UARTx, uint32\_t RxFIFOLevel);
- ◆ void UART\_RxFIFOINTSel(TSB\_SC\_TypeDef \* UARTx, uint32\_t RxINTCondition);
- ◆ void UART\_RxFIFOClear(TSB\_SC\_TypeDef \* UARTx);
- ◆ void UART\_TxFIFOFillLevel(TSB\_SC\_TypeDef \* UARTx, uint32\_t TxFIFOLevel);

- ◆ void UART\_TxFIFOINTSel(TSB\_SC\_TypeDef \* UARTx, uint32\_t TxINTCondition);
- ◆ void UART\_TxFIFOClear(TSB\_SC\_TypeDef \* UARTx);
- ◆ void UART\_TxBufferClear(TSB\_SC\_TypeDef \* UARTx);
- ◆ uint32\_t UART\_GetRxFIFOFillLevelStatus(TSB\_SC\_TypeDef \* UARTx);
- ◆ uint32\_t UART\_GetRxFIFOOverRunStatus(TSB\_SC\_TypeDef \* UARTx);
- ◆ uint32\_t UART\_GetTxFIFOFillLevelStatus(TSB\_SC\_TypeDef \* UARTx);
- ◆ uint32\_t UART\_GetTxFIFOUnderRunStatus(TSB\_SC\_TypeDef \* UARTx);
- ◆ void SIO\_SetInputClock(TSB\_SC\_TypeDef \* SIOx, uint32\_t Clock)
- ◆ void SIO\_Enable(TSB\_SC\_TypeDef\* SIOx)
- ◆ void SIO\_Disable(TSB\_SC\_TypeDef\* SIOx)
- ◆ void SIO\_Init(TSB\_SC\_TypeDef\* SIOx, uint32\_t IOClkSel,  
UART\_InitTypeDef\* InitStruct)
- ◆ uint8\_t SIO\_GetRxData(TSB\_SC\_TypeDef\* SIOx)
- ◆ void SIO\_SetTxData(TSB\_SC\_TypeDef\* SIOx, uint8\_t Data)

## 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) 初期化と設定:  
UART\_Enable(), UART\_Disable(), UART\_Init(), UART\_DefaultConfig(),  
UART\_SetInputClock, SIO\_Enable(), SIO\_Disable(), SIO\_SetInputClock(), SIO\_Init()
- 2) 送受信設定とエラー確認:  
UART\_GetBufState(), UART\_GetRxData(), UART\_SetTxData(),  
UART\_GetErrState(), SIO\_GetRxData() and SIO\_SetTxData
- 3) その他:  
UART\_SWReset(), UART\_SetWakeUpFunc(), UART\_SetIdleMode()
- 4) FIFO モードの設定:  
UART\_FIFOConfig(), UART\_SetFIFOTransferMode(), UART\_RxFIFOINTCtrl(),  
UART\_TxFIFOINTCtrl(), UART\_RxFIFOByteSel(), UART\_RxFIFOFillLevel(),  
UART\_RxFIFOINTSel(), UART\_RxFIFOClear(), UART\_TxFIFOFillLevel(),  
UART\_TxFIFOINTSel(), UART\_TxFIFOClear(), UART\_TxBufferClear(),  
UART\_GetRxFIFOFillLevelStatus(), UART\_GetRxFIFOOverRunStatus(),  
UART\_GetTxFIFOFillLevelStatus(), UART\_GetTxFIFOUnderRunStatus()

## 11.2.3 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.**

また、“TSB\_SC\_TypeDef\* **SIOx**” は、以下から選択してください。

**SIO0**

### 11.2.3.1 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 チャンネルを指定します。

機能:

UART 動作を許可します。

戻り値:  
なし

## 11.2.3.2 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 チャンネルを指定します。

機能:  
UART 動作を禁止します。

戻り値:  
なし

## 11.2.3.3 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 チャンネルを指定します。

**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。

## 11.2.3.4 UART\_SWReset

ソフトウェアリセットの実行

**関数のプロトタイプ宣言:**

void  
UART\_SWReset(TSB\_SC\_TypeDef\* **UARTx**)

**引数:**

**UARTx**: UART チャンネルを指定します。

**機能:**

ソフトウェアリセットを実行します。

**戻り値:**

なし

## 11.2.3.5 UART\_Init

UART チャンネルの初期化

**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 チャンネルを指定します。

**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

## 11.2.3.6 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 チャンネルを指定します。

**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 (シリアルチャンネル) 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。

## 11.2.3.7 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 チャンネルを指定します。

**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 (シリアルチャンネル) 割り込み関数の中で実行してください。

戻り値:

なし

## 11.2.3.8 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 チャンネルを指定します。

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

## 11.2.3.9 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 チャンネルを指定します。

**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

## 11.2.3.10 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 チャンネルを指定します。

**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

## 11.2.3.11 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 チャンネルを指定します。

**NewState:** IDLE 時の動作を選択します。

- **ENABLE:** 動作
- **DISABLE:** 停止

**機能:**

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

**戻り値:**

なし

## 11.2.3.12 UART\_SetInputClock

入力クロックの設定

**関数のプロトタイプ宣言:**

```
void  
UART_SetInputClock(TSB_SC_TypeDef * UARTx, uint8_t ClkDivider)
```

**引数:**

**UARTx:** UART チャンネルを指定します。

**ClkDivider:** 以下から、プリスケアラの入力クロックを選択します。

- **UART\_DIVIDE\_1\_1:**  $\Phi T0$
- **UART\_DIVIDE\_1\_2:**  $\Phi T0/2$

**機能:**

プリスケアラの入力クロックを選択します。

**戻り値:**

なし

## 11.2.3.13 UART\_FIFOConfig

FIFO の許可

**関数のプロトタイプ宣言:**

```
void  
UART_FIFOConfig(TSB_SC_TypeDef * UARTx,  
                 FunctionalState NewState)
```

**引数:**

**UARTx:** UART チャンネルを指定します。

**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

## 11.2.3.14 UART\_SetFIFOTransferMode

転送モードの選択

**関数のプロトタイプ宣言:**

```
void  
UART_SetFIFOTransferMode(TSB_SC_TypeDef * UARTx,  
                          uint32_t TransferMode)
```

**引数:**

**UARTx**: UART チャンネルを指定します。

**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

## 11.2.3.15 UART\_TRxAutoDisable

送信/受信の自動禁止

**関数のプロトタイプ宣言:**

```
void  
UART_TRxAutoDisable (TSB_SC_TypeDef * UARTx,  
                     UART_TRxDisable TRxAutoDisable)
```

**引数:**

**UARTx**: UART チャンネルを指定します。

**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

## 11.2.3.16 UART\_RxFIFOINTCtrl

受信 FIFO 使用時の受信割り込み許可

**関数のプロトタイプ宣言:**

```
void  
UART_RxFIFOINTCtrl (TSB_SC_TypeDef * UARTx,  
                    FunctionalState NewState)
```

**引数:**

**UARTx:** UART チャンネルを指定します。

**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

## 11.2.3.17 UART\_TxFIFOINTCtrl

送信 FIFO 使用時の送信割り込み許可

**関数のプロトタイプ宣言:**

```
void  
UART_TxFIFOINTCtrl (TSB_SC_TypeDef * UARTx,  
                    FunctionalState NewState)
```

**引数:**

**UARTx:** UART チャンネルを指定します。

**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

## 11.2.3.18 UART\_RxFIFOByteSel

受信 FIFO 使用バイト数

**関数のプロトタイプ宣言:**

```
void  
UART_RxFIFOByteSel (TSB_SC_TypeDef * UARTx,  
                    uint32_t BytesUsed)
```

**引数:**

**UARTx:** UART チャンネルを指定します。

**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

## 11.2.3.19 UART\_RxFIFOFillLevel

受信割り込みが発生する受信 FIFO の fill レベルの設定

**関数のプロトタイプ宣言:**

void

UART\_RxFIFOFillLevel (TSB\_SC\_TypeDef \* **UARTx**,  
uint32\_t **RxFIFOLevel**)

**引数:**

**UARTx:** UART チャンネルを指定します。

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

<b>RxFIFOLevel</b>	半二重	全二重
<b>UART_RXFIFO4B_FLEVLE_4_2B</b>	4 バイト	2 バイト
<b>UART_RXFIFO4B_FLEVLE_1_1B</b>	1 バイト	1 バイト
<b>UART_RXFIFO4B_FLEVLE_2_2B</b>	2 バイト	2 バイト
<b>UART_RXFIFO4B_FLEVLE_3_1B</b>	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

## 11.2.3.20 UART\_RxFIFOINTSel

受信割り込み発生条件の選択

**関数のプロトタイプ宣言:**

void

UART\_RxFIFOINTSel (TSB\_SC\_TypeDef \* **UARTx**,  
uint32\_t **RxINTCondition**)

**引数:**

**UARTx:** UART チャンネルを指定します。

**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

## 11.2.3.21 UART\_RxFIFOClear

受信 FIFO クリア

**関数のプロトタイプ宣言:**

void

UART\_RxFIFOClear (TSB\_SC\_TypeDef \* **UARTx**)

**引数:**

**UARTx**: UART チャンネルを指定します。

**機能:**

受信 FIFO をクリアします。

**戻り値:**

なし

## 11.2.3.22 UART\_TxFIFOFillLevel

送信割り込みが発生する送信 FIFO の fill レベルの設定

**関数のプロトタイプ宣言:**

void

UART\_TxFIFOFillLevel (TSB\_SC\_TypeDef \* **UARTx**,  
uint32\_t **TxFIFOLevel**)

**引数:**

**UARTx**: UART チャンネルを指定します。

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

<b>TxFIFOLevel</b>	半二重	全二重
<b>UART_TXFIFO4B_FLEVLE_0_0B</b>	Empty	Empty
<b>UART_TXFIFO4B_FLEVLE_1_1B</b>	1 バイト	1 バイト
<b>UART_TXFIFO4B_FLEVLE_2_0B</b>	2 バイト	Empty
<b>UART_TXFIFO4B_FLEVLE_3_1B</b>	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

## 11.2.3.23 UART\_TxFIFOINTSel

送信割り込み発生条件の選択

関数のプロトタイプ宣言:

```
void  
UART_TxFIFOINTSel (TSB_SC_TypeDef * UARTx,  
uint32_t TxINTCondition)
```

引数:

**UARTx**: UART チャンネルを指定します。

**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

## 11.2.3.24 UART\_TxFIFOClear

送信 FIFO クリア

関数のプロトタイプ宣言:

```
void  
UART_TxFIFOClear (TSB_SC_TypeDef * UARTx)
```

引数:

**UARTx**: UART チャンネルを指定します。

機能:

送信 FIFO をクリアします。

戻り値:

なし

## 11.2.3.25 UART\_TxBufferClear

送信バッファクリア

関数のプロトタイプ宣言:

```
void  
UART_TxBufferClear (TSB_SC_TypeDef * UARTx);
```

引数:

**UARTx**: UART チャンネルを指定します。

**機能:**

送信バッファをクリアします。

**戻り値:**

なし

## 11.2.3.26 UART\_GetRxFIFOFillLevelStatus

受信 FIFO の fill レベルの取得

**関数のプロトタイプ宣言:**

uint32\_t

UART\_GetRxFIFOFillLevelStatus (TSB\_SC\_TypeDef\* **UARTx**);

**引数:**

**UARTx**: UART チャンネルを指定します。

**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

## 11.2.3.27 UART\_GetRxFIFOOverRunStatus

受信 FIFO オーバーラン状態の取得

**関数のプロトタイプ宣言:**

uint32\_t

UART\_GetRxFIFOOverRunStatus (TSB\_SC\_TypeDef\* **UARTx**);

**引数:**

**UARTx**: UART チャンネルを指定します。

**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

## 11.2.3.28 UART\_GetTxFIFOFillLevelStatus

送信 FIFO の fill レベルの取得

**関数のプロトタイプ宣言:**

uint32\_t

UART\_GetTxFIFOFillLevelStatus (TSB\_SC\_TypeDef\* **UARTx**);

引数:

**UARTx**: UART チャンネルを指定します。

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

### 11.2.3.29 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 チャンネルを指定します。

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

### 11.2.3.30 UART\_SetInputClock

入力クロックの設定

関数のプロトタイプ宣言:

void

UART\_SetInputClock (TSB\_SC\_TypeDef \* **UARTx**,  
uint32\_t clock)

引数:

**UARTx**: UART チャンネルを指定します。

**Clock**: 以下から、プリスケアラの入力クロックを選択します。

0 :  $\Phi T0/2$

1 :  $\Phi T0$

機能:

プリスケアラの入力クロックを選択します。

戻り値:

なし

## 11.2.3.31 SIO\_SetInputClock

入力クロックの設定

関数のプロトタイプ宣言:

```
void  
SIO_SetInputClock (TSB_SC_TypeDef * SIOx,  
                  uint32_t Clock)
```

引数:

**SIOx**: SIO チャンネルを指定します。

**Clock**: 以下から、プリスケアラの入力クロックを選択します。

**SIO\_CLOCK\_T0\_HALF** :  $\Phi T0/2$

**SIO\_CLOCK\_T0** :  $\Phi T0$

機能:

プリスケアラの入力クロックを選択します。

戻り値:

なし

## 11.2.3.32 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 チャンネルを指定します。

機能:

SIO 動作を許可します。

戻り値:

なし

## 11.2.3.33 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 チャンネルを指定します。



**機能:**

SIO 動作を禁止します。

**戻り値:**

なし

## 11.2.3.34 SIO\_Init

SIO チャンネルの初期化

**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 チャンネルを指定します。

**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

## 11.2.3.35 SIO\_GetRxData

受信用バッファの取得

**関数のプロトタイプ宣言:**

```
uint32_t  
SIO_GetRxData(TSB_SC_TypeDef* SIOx)
```

**引数:**

**SIOx**: SIO チャンネルを指定します。

**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)

## 11.2.3.36 SIO\_SetTxData

送信用バッファの設定

**関数のプロトタイプ宣言:**

```
void  
SIO_SetTxData(TSB_SC_TypeDef* SIOx,  
              uint8_t Data)
```

**引数:**

**SIOx**: SIO チャンネルを指定します。

**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

## 11.2.4 データ構造:

### 11.2.4.1 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**: 奇数(Odd) パリティ

uint32\_t

**Mode**: 転送モードを選択します。送受信の場合は、送信と受信を OR 演算子によって接続して指定してください。

- **UART\_ENABLE\_TX**: 送信許可
- **UART\_ENABLE\_RX**: 受信許可

uint32\_t

**FlowCtrl** : フローコントロールモードを選択します(\*\*)。

- **UART\_NONE\_FLOW\_CTRL** : CTS 無効

\*: fperiph の周波数が高すぎる、または、低すぎると、ボーレートが正しく設定できない

場合があります。

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

## 11.2.4.2 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 の立ち下がリ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 の立ち下がリエッジで 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。

**TIDLE:** 最終ビット出力後の TXDx 端子の状態を選択します。

- **SIO\_TIDLE\_LOW:** "Low"出力保持
- **SIO\_TIDLE\_HIGH:** "High"出力保持
- **SIO\_TIDLE\_LAST:** 最終ビット保持

uint32\_t

**TXDEMP:** 以下から、クロック入力モード時、アンダーランエラーが発生したときの TXDx 端子の状態を選択します。

- **SIO\_TXDEMP\_LOW:** "Low"出力
- **SIO\_TXDEMP\_HIGH:** "High"出力

uint32\_t

**EHOLDTime:** 以下から、クロック入力モードの TXDx 端子の最終ビットホールド時間を選択します。

- **SIO\_EHOLD\_FC\_2:** 2/fc
- **SIO\_EHOLD\_FC\_4:** 4/fc
- **SIO\_EHOLD\_FC\_8:** 8/fc
- **SIO\_EHOLD\_FC\_16:** 16/fc
- **SIO\_EHOLD\_FC\_32:** 32/fc
- **SIO\_EHOLD\_FC\_64:** 64/fc
- **SIO\_EHOLD\_FC\_128:** 128/fc

uint32\_t

**IntervalTime:** 連続転送時のインターバル時間を選択します。

- **SIO\_SINT\_TIME\_NONE:** なし
- **SIO\_SINT\_TIME\_SCLK\_1:** 1\*SCLK
- **SIO\_SINT\_TIME\_SCLK\_2:** 2\*SCLK
- **SIO\_SINT\_TIME\_SCLK\_4:** 4\*SCLK
- **SIO\_SINT\_TIME\_SCLK\_8:** 8\*SCLK
- **SIO\_SINT\_TIME\_SCLK\_16:** 16\*SCLK
- **SIO\_SINT\_TIME\_SCLK\_32:** 32\*SCLK
- **SIO\_SINT\_TIME\_SCLK\_64:** 64\*SCLK

uint32\_t

**TransferMode:** 転送モードを選択します。

- **SIO\_TRANSFER\_PROHIBIT:** 転送禁止
- **SIO\_TRANSFER\_HALFDPX\_RX:** 半二重(受信)

- **SIO\_TRANSFER\_HALFDPX\_TX**: 半二重(送信)
- **SIO\_TRANSFER\_FULLDPX**: 全二重

uint32\_t

**TransferDir**: 転送方向を選択します。

- **SIO\_LSB\_FRIST**: LSB FRIST
- **SIO\_MSB\_FRIST**: MSB FRIST

uint32\_t

**Mode**: 送受信を制御します。有効ビットの組み合わせが可能です。

- **SIO\_ENABLE\_TX**: 送信許可
- **SIO\_ENABLE\_RX**: 受信許可

uint32\_t

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

- **SIO\_WBUF\_ENABLE**: 許可
- **SIO\_WBUF\_DISABLE**: 禁止

uint32\_t

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

- **SIO\_BR\_CLOCK\_TS0**:  $\phi$ TS0
- **SIO\_BR\_CLOCK\_TS2**:  $\phi$ TS2
- **SIO\_BR\_CLOCK\_TS8**:  $\phi$ TS8
- **SIO\_BR\_CLOCK\_TS32**:  $\phi$ TS32

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

## 12. VLTD

### 12.1 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/tmpm375\_vltd.c  
/Libraries/TX03\_Periph\_Driver/inc/tmpm375\_vltd.h

### 12.2 API 関数

#### 12.2.1 関数一覧

- ◆ void VLTD\_Enable(void);
- ◆ void VLTD\_Disable(void);
- ◆ void VLTD\_SetVoltage(uint32\_t **Voltage**);

#### 12.2.2 関数の種類

関数は、主に以下の 2 種類に分かれています。

- 1) VLTD の許可/禁止:  
VLTD\_Enable()、VLTD\_Disable()
- 2) 検出電圧の設定:  
VLTD\_SetVoltage()

#### 12.2.3 関数仕様

##### 12.2.3.1 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

## 12.2.3.2 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

## 12.2.3.3 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\_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

## 12.2.4 データ構造:

なし

## 13. WDT

### 13.1 概要

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

WDTドライバの API は、検出時間、カウンタのオーバーフロー時の出力、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\\tmpm375\_wdt.c  
\\Libraries\\TX03\_Periph\_Driver\\inc\\tmpm375\_wdt.h

### 13.2 API 関数

#### 13.2.1 関数一覧

- void WDT\_SetDetectTime(uint32\_t **DetectTime**)
- void WDT\_SetIdleMode(FunctionalState **NewState**)
- void WDT\_SetOverflowOutput(uint32\_t **OverflowOutput**)
- void WDT\_Init(WDT\_InitTypeDef \* **InitStruct**)
- void WDT\_Enable(void)
- void WDT\_Disable(void)
- void WDT\_WriteClearCode(void)

#### 13.2.2 関数の種類

関数は、主に以下の 2 種類に分かれています。

- 1) ウォッチドッグタイマ設定:  
WDT\_SetDetectTime(), WDT\_SetOverflowOutput(), WDT\_Init(), WDT\_Enable(),  
WDT\_Disable(), WDT\_WriteClearCode()
- 2) IDLE モード時の開始・停止:  
WDT\_SetIdleMode()

#### 13.2.3 関数仕様

##### 13.2.3.1 WDT\_SetDetectTime

WDT 検出時間の設定

関数のプロトタイプ宣言:

void  
WDT\_SetDetectTime(uint32\_t **DetectTime**)

引数:

**DetectTime**: 以下から検出時間を選択します。

- **WDT\_DETECT\_TIME\_EXP\_15**:  $2^{15}/f_{sys}$

- WDT\_DETECT\_TIME\_EXP\_17: 2<sup>17</sup>/fsys
- WDT\_DETECT\_TIME\_EXP\_19: 2<sup>19</sup>/fsys
- WDT\_DETECT\_TIME\_EXP\_21: 2<sup>21</sup>/fsys
- WDT\_DETECT\_TIME\_EXP\_23: 2<sup>23</sup>/fsys
- WDT\_DETECT\_TIME\_EXP\_25: 2<sup>25</sup>/fsys

**機能:**

WDT の検出時間を設定します。

**戻り値:**

なし

### 13.2.3.2 WDT\_SetIdleMode

IDLE 時の動作選択

**関数のプロトタイプ宣言:**

void  
WDT\_SetIdleMode(FunctionalState **NewState**)

**引数:**

**NewState:** 以下から IDLE 時の WDT 動作を選択します。

- **ENABLE:** 動作
- **DISABLE:** 停止

**機能:**

本関数は、IDLE モード時の WDT カウンタの動作を設定します。

**NewState** が **ENABLE** の時は WDT カウンタ停止

**NewState** が **DISABLE** の時は WDT カウンタ作動

**補足:**

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

**戻り値:**

なし

### 13.2.3.3 WDT\_SetOverflowOutput

暴走検出後の動作選択

**関数のプロトタイプ宣言:**

void  
WDT\_SetOverflowOutput(uint32\_t **OverflowOutput**)

**引数:**

**OverflowOutput:** 以下から暴走検出後の動作を選択します。

- **WDT\_NMIINT:** INTWDT 割り込み要求を発生します。
- **WDT\_WDOUT:** マイコンをリセットします。

**機能:**



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

戻り値:  
なし

### 13.2.3.4 WDT\_Init

WDT の初期化

関数のプロトタイプ宣言:

```
void  
WDT_Init (WDT_InitTypeDef* InitStruct)
```

引数:

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

機能:

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

戻り値:  
なし

### 13.2.3.5 WDT\_Enable

WDT 動作の許可

関数のプロトタイプ宣言:

```
void  
WDT_Enable(void)
```

引数:

なし

機能:

WDT 動作を許可します。

戻り値:  
なし

### 13.2.3.6 WDT\_Disable

WDT 動作の禁止

関数のプロトタイプ宣言:

```
void  
WDT_Disable(void)
```

引数:

なし

機能:

WDT 動作を禁止します。

戻り値:

なし

### 13.2.3.7 WDT\_WriteClearCode

クリアコードの書き込み

関数のプロトタイプ宣言:

void

WDT\_WriteClearCode (void)

引数:

なし

機能:

クリアコードをライトします。

戻り値:

なし

## 13.2.4 データ構造:

### 13.2.4.1 WDT\_InitTypeDef

メンバ:

uint32\_t

**DetectTime** 以下から検出時間を選択します。

- WDT\_DETECT\_TIME\_EXP\_15: 2<sup>15</sup>/fsys
- WDT\_DETECT\_TIME\_EXP\_17: 2<sup>17</sup>/fsys
- WDT\_DETECT\_TIME\_EXP\_19: 2<sup>19</sup>/fsys
- WDT\_DETECT\_TIME\_EXP\_21: 2<sup>21</sup>/fsys
- WDT\_DETECT\_TIME\_EXP\_23: 2<sup>23</sup>/fsys
- WDT\_DETECT\_TIME\_EXP\_25: 2<sup>25</sup>/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。

## 14. PMD

### 14.1 概要

本デバイスはモータ制御回路(PMD)を1チャンネル内蔵しています。本デバイスのPMDはベクトルエンジン(VE+)やアナログ/デジタルコンバータ(ADC)と連携動作してベクトル制御などの3相モータ制御を実現します。パルス幅変調回路、通電制御および同期トリガ生成回路はVEからの指令で動作可能で、同期トリガ生成回路はADCに変換開始指令ができます。

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キャリアが共通で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。
- 保護回路ではEMG入力、OVV入力による緊急出力停止を行いません。
- 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/tmpm375\_pmd.c  
/Libraries/TX03\_Periph\_Driver/inc/tmpm375\_pmd.h

### 14.2 API 関数

#### 14.2.1 関数一覧

- ◆ void PMD\_Enable(TSB\_PMD\_TypeDef \* **PMDx**);
- ◆ void PMD\_Disable(TSB\_PMD\_TypeDef \* **PMDx**);
- ◆ void PMD\_SetPortControl(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **PortMode**);
- ◆ void PMD\_Init(TSB\_PMD\_TypeDef \* **PMDx**,  
PMD\_InitTypeDef \* **InitStruct**);
- ◆ void PMD\_ChangePWMCycle(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **CycleTiming**);
- ◆ uint32\_t PMD\_GetCntFlag(TSB\_PMD\_TypeDef \* **PMDx**);
- ◆ uint16\_t PMD\_GetCntValue(TSB\_PMD\_TypeDef \* **PMDx**);
- ◆ void PMD\_SetCompareValue(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **PMDPhase**,  
uint32\_t **Timing**);
- ◆ void PMD\_SetPortOutputMode(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **Mode**);
- ◆ void PMD\_SetOutputPhasePolarity(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **OutputPhase**,  
uint32\_t **Polarity**);
- ◆ void PMD\_SetReflectTime(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **ReflectedTime**);
- ◆ void PMD\_EnableEMG(TSB\_PMD\_TypeDef \* **PMDx**);
- ◆ void PMD\_DisableEMG(TSB\_PMD\_TypeDef \* **PMDx**);
- ◆ void PMD\_SetEMGNoiseElimination(TSB\_PMD\_TypeDef \* **PMDx**,

- uint32\_t **NoiseElimination**);
- ◆ void PMD\_SetToolBreakOutput(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **Status**);
- ◆ void PMD\_SetEMGMode(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **Mode**);
- ◆ void PMD\_EMGRelease(TSB\_PMD\_TypeDef \* **PMDx**);
- ◆ uint32\_t PMD\_GetEMGAbnormalLevel(TSB\_PMD\_TypeDef \* **PMDx**);
- ◆ uint32\_t PMD\_GetEMGCondition(TSB\_PMD\_TypeDef \* **PMDx**);
- ◆ void PMD\_SetDeadTime(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **Time**);
- ◆ void PMD\_SetAllPhaseCompareValue(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **UPhaseTiming**,  
uint32\_t **VPhaseTiming**,  
uint32\_t **WPhaseTiming**);
- ◆ void PMD\_ChangeDutyMode(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **DutyMode**);
- ◆ Result PMD\_SetPortOutput(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **PMDPhase**,  
uint8\_t **Output**);
- ◆ void PMD\_SetTrgCmpValue(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **TRGCMP0Timing**,  
uint32\_t **TRGCMP1Timing**,  
uint32\_t **TRGCMP2Timing**,  
uint32\_t **TRGCMP3Timing**);
- ◆ void PMD\_SetTrgMode(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **PMDTrg**,  
uint32\_t **Mode**);
- ◆ void PMD\_SetTrgUpdate(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **PMDTrg**,  
uint32\_t **UpdateTiming**);
- ◆ void PMD\_SetEMGTrg(TSB\_PMD\_TypeDef \* **PMDx**,  
FunctionalState **NewState**);
- ◆ void PMD\_SetTrgOutput(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **TrgMode**,  
uint32\_t **TrgChannel**);
- ◆ void PMD\_SetSelectMode(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **Mode**);
- ◆ void PMD\_EnableOVV(TSB\_PMD\_TypeDef \* **PMDx**);
- ◆ void PMD\_DisableOVV(TSB\_PMD\_TypeDef \* **PMDx**);
- ◆ void PMD\_SetOVVNoiseElimination(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **NoiseElimination**);
- ◆ void PMD\_SetADCMonitorInput(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **Monitor**,  
FunctionalState **NewState**);
- ◆ void PMD\_SetOVVMode(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **Mode**);
- ◆ void PMD\_SetOVVInputSrc(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **Source**);
- ◆ void PMD\_SetOVVAutoRelease(TSB\_PMD\_TypeDef \* **PMDx**,  
FunctionalState **NewState**);
- ◆ uint32\_t PMD\_GetOVVAbnormalLevel(TSB\_PMD\_TypeDef \* **PMDx**);
- ◆ void PMD\_SetAutoSwitchCtrl(TSB\_PMD\_TypeDef \* **PMDx**,  
FunctionalState **NewState**);
- ◆ void PMD\_SetPWMEdge(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **PMDPhase**,  
uint32\_t **Edge**);

- ◆ void PMD\_SetBufferUpdateTime(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **UpdateTime**);
- ◆ void PMD\_SetTrgSyncTime(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **SyncTime**);
- ◆ void PMD\_SetTrgUpdateTime(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **UpdateTime**);

## 14.2.2 関数の種類

関数は、主に以下の 7 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(),  
PMD\_SetSelectMode(), PMD\_SetAutoSwitchCtrl(), PMD\_SetPWMEdge(),  
PMD\_SetBufferUpdateTime()
- 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\_GetCntFlag(), PMD\_GetCntValue(), PMD\_GetEMGAbnormalLevel(),  
PMD\_GetEMGCondition(), PMD\_GetOVVAbnormalLevel(), PMD\_GetOVVCondition()
- 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(), PMD\_SetTrgOutput(), PMD\_SetTrgSyncTime(),  
PMD\_SetTrgUpdateTime()
- 7) OVV 保護制御回路の設定:  
PMD\_EnableOVV(), PMD\_DisableOVV(), PMD\_SetOVVNoiseElimination(),  
PMD\_SetADCMonitorInput(), PMD\_SetOVVMode(), PMD\_SetOVVAutoRelease(),  
PMD\_SetOVVInputSrc()

## 14.2.3 関数仕様

**補足:** 下記の全 API において、パラメータ “TSB\_PMD\_TypeDef \* **PMDx**” は 以下のいずれか  
を選択してください。

**PMD1**

### 14.2.3.1 PMD\_Enable

PMD 機能の許可

**関数のプロトタイプ宣言:**

void  
PMD\_Enable(TSB\_PMD\_TypeDef \* **PMDx**)

**引数:**

**PMDx:** PMD チャンネルを指定します。

**機能:**

PMD 機能を許可します。

戻り値:  
なし

## 14.2.3.2 PMD\_Disable

PMD 機能の禁止

関数のプロトタイプ宣言:

```
void  
PMD_Disable(TSB_PMD_TypeDef * PMDx)
```

引数:

*PMDx*: PMD チャンネルを指定します。

機能:

PMD 機能を禁止します。

戻り値:  
なし

## 14.2.3.3 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_PMD_TypeDef * PMDx  
                   uint32_t PortMode)
```

引数:

*PMDx*: PMD チャンネルを指定します。

*PortMode*: ポート制御の種類を選択します。

- **PMD\_PORT\_MODE\_0**: 上相 High-z / 下相 High-z
- **PMD\_PORT\_MODE\_1**: 上相 High-z / 下相 PMD 出力
- **PMD\_PORT\_MODE\_2**: 上相 PMD 出力 / 下相 High-z
- **PMD\_PORT\_MODE\_3**: 上相 PMD 出力 / 下相 PMD 出力

機能:

ポート制御を設定します。

戻り値:  
なし

## 14.2.3.4 PMD\_Init

PMD の初期化

関数のプロトタイプ宣言:

```
void  
PMD_Init(TSB_PMD_TypeDef * PMDx,  
         PMD_InitTypeDef * InitStruct)
```

**引数:**

**PMDx**: PMD チャンネルを指定します。

**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

## 14.2.3.5 PMD\_ChangePWMCycle

PWM 周期の設定

**関数のプロトタイプ宣言:**

```
void  
PMD_ChangePWMCycle(TSB_PMD_TypeDef * PMDx,  
                   uint32_t CycleTiming)
```

**引数:**

**PMDx**: PMD チャンネルを指定します。

**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せん)

## 14.2.3.6 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_PMD_TypeDef * PMDx)
```

**引数:**

**PMDx**: PMD チャンネルを指定します。

**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

## 14.2.3.7 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\_PMD\_TypeDef \* **PMDx**)

**引数:**

**PMDx**: PMD チャンネルを指定します。

**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

## 14.2.3.8 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\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **PMDPhase**,  
uint32\_t **Timing**)

**引数:**

**PMDx**: PMD チャンネルを指定します。

**PMDPhase**: 3 相のいずれか、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



## 14.2.3.9 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_PMD_TypeDef * PMDx,  
                      uint32_t Mode)
```

**引数:**

**PMDx:** PMD チャンネルを指定します。

**Mode:** U,V,W 相のポート出力を設定します。

- **PMD\_PORT\_OUTPUT\_MODE\_0:** PMDxMDCR<SYNTMD>=0
- **PMD\_PORT\_OUTPUT\_MODE\_1:** PMDxMDCR<SYNTMD>=1

**機能:**

U,V,W 相のポート出力設定を行います。

**補足:**

PMDxMDCR<SYNTMD>, PMDxMDPOT<POLH><POLL>, PMDxMDOUT  
<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 PMDxMDCR<SYNTMD>を設定します。  
PMD\_SetOutputPhasePolarity()により PMDxMDPOT<POLH><POLL>を設定しま  
す  
PMD\_SetPortOutput()により PMDxMDOUT<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	PWM	PWM
0	1	L	H	L	PWM
1	0	H	L	PWM	L
1	1	H	H	PWM	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	PWM
0	1	H	L	H	PWM
1	0	L	H	PWM	H
1	1	L	L	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	PWM	PWM
0	1	L	H	L	PWM
1	0	H	L	PWM	L
1	1	H	H	PWM	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	PWM
0	1	H	L	H	PWM
1	0	L	H	PWM	H
1	1	L	L	PWM	PWM

戻り値:

なし

## 14.2.3.10PMD\_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\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **OutputPhase**,  
uint32\_t **Polarity**)

引数:

**PMDx**: PMD チャンネルを指定します。

**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 を無効の状態を選択を行ってください。

**戻り値:**

なし

## 14.2.3.11 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_PMD_TypeDef * PMDx,  
uint32_t ReflectedTime)
```

**引数:**

**PMDx**: PMD チャンネルを指定します。

**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=  
PMDxMDPRD<MDPRD>(最大)の時、反映
- **PMD\_REFLECTED\_TIME\_MIN\_MAX**: PWM カウンタ MDCNT="1"(最小)お  
よび PMDxMDPRD<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 を無効の状態を選択を行ってください。

**戻り値:**

なし

## 14.2.3.12 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_PMD_TypeDef * PMDx)
```

**引数:**

**PMDx**: PMD チャンネルを指定します。

**機能:**

EMG 保護回路を許可します。

戻り値:  
なし

## 14.2.3.13 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_PMD_TypeDef * PMDx)
```

引数:

**PMDx**: PMD チャンネルを指定します。

機能:

EMG 保護回路を禁止します。

戻り値:  
なし

## 14.2.3.14 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_PMD_TypeDef * PMDx,  
                             uint32_t NoiseElimination)
```

引数:

**PMDx**: PMD チャンネルを指定します。

**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

## 14.2.3.15PMD\_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_PMD_TypeDef * PMDx,  
                        uint32_t Status)
```

**引数:**

**PMDx**: PMD チャンネルを指定します。

**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

## 14.2.3.16PMD\_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_PMD_TypeDef * PMDx,  
                uint32_t Mode)
```

**引数:**

**PMDx**: PMD チャンネルを指定します。

**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

## 14.2.3.17PMD\_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\_PMD\_TypeDef \* **PMDx**)

**引数:**

**PMDx:** PMD チャンネルを指定します。

**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、PMDxMDOUT<UPWN><VPWN><WPWN>、  
PMDxMDOUT<UOC> <VOC> <WOC>に 0 を設定します。(x=0, 1)

**戻り値:**

なし

## 14.2.3.18PMD\_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\_PMD\_TypeDef \* **PMDx**)

**引数:**

**PMDx:** PMD チャンネルを指定します。

**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"

## 14.2.3.19PMD\_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\_PMD\_TypeDef \* **PMDx**)

**引数:**

**PMDx:** PMD チャンネルを指定します。

**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

## 14.2.3.20PMD\_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_PMD_TypeDef * PMDx,  
                 uint32_t Time)
```

引数:

**PMDx**: PMD チャンネルを指定します。

**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 を無効の状態を選択を行ってください。

戻り値:

なし

## 14.2.3.21PMD\_SetAllPhaseCompareValue

PWM パルス幅の設定

関数のプロトタイプ宣言:

```
void  
PMD_SetAllPhaseCompareValue(TSB_PMD_TypeDef * PMDx,  
                             uint32_t UPhaseTiming,  
                             uint32_t VPhaseTiming,  
                             uint32_t WPhaseTiming)
```

引数:

**PMDx**: PMD チャンネルを指定します。

**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

## 14.2.3.22PMD\_ChangeDutyMode

DUTY モードの設定

関数のプロトタイプ宣言:

```
void  
PMD_ChangeDutyMode(TSB_PMD_TypeDef * PMDx,  
                    uint32_t DutyMode)
```

引数:

**PMDx**: PMD チャンネルを指定します。

**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

## 14.2.3.23PMD\_SetPortOutput

UVW 相出力の設定

関数のプロトタイプ宣言:

```
Result  
PMD_SetPortOutput(TSB_PMD_TypeDef * PMDx,  
                  uint32_t PMDPhase,  
                  uint8_t Output)
```

引数:

**PMDx**: PMD チャンネルを指定します。

**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。

## 14.2.3.24PMD\_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_PMD_TypeDef * PMDx,  
                    uint32_t TRGCMP0Timing,  
                    uint32_t TRGCMP1Timing,  
                    uint32_t TRGCMP2Timing,  
                    uint32_t TRGCMP3Timing)
```

**引数:**

**PMDx:** PMD チャンネルを指定します。

**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。

**TRGCMP2Timing:** トリガコンペアレジスタ 2 の値を 0x0001~[<MDPRO>-1]の間で設定してください。

**TRGCMP3Timing:** トリガコンペアレジスタ 3 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。

## 14.2.3.25 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_PMD_TypeDef * PMDx,  
                uint32_t PMDTrg,  
                uint32_t Mode)
```

引数:

**PMDx**: PMD チャンネルを指定します。

**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

## 14.2.3.26 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_PMD_TypeDef * PMDx,  
                  uint32_t PMDTrg,  
                  uint32_t UpdateTiming)
```

引数:

**PMDx**: PMD チャンネルを指定します。

**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
- **PMD\_ADC\_TRG\_2:** トリガ 2
- **PMD\_ADC\_TRG\_3:** トリガ 3

**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

## 14.2.3.27 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_PMD_TypeDef * PMDx,  
               FunctionalState NewState)
```

**引数:**

**PMDx:** PMD チャンネルを指定します。

**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

## 14.2.3.28 PMD\_SetTrgOutput

トリガ出力の設定

**関数のプロトタイプ宣言:**

```
void  
PMD_SetTrgOutput(TSB_PMD_TypeDef * PMDx,  
                  uint32_t TrgMode,  
                  uint32_t TrgChannel);
```

**引数:**

**PMDx:** PMD チャンネルを指定します。

**TrgMode:** トリガ出力モードを選択します。

- **PMD\_TRG\_FIXED\_OUTPUT:** トリガ固定出力
- **PMD\_TRG\_VARIABLE\_OUTPUT:** トリガ選択出力

**TrgChannel:** トリガ出力ポートを選択します。

**TrgMode == PMD\_TRG\_FIXED\_OUTPUT の場合:**

- **PMD\_TRG\_OUTPUT\_0:** PMDTRG0 より出力
- **PMD\_TRG\_OUTPUT\_1:** PMDTRG1 より出力
- **PMD\_TRG\_OUTPUT\_2:** PMDTRG2 より出力
- **PMD\_TRG\_OUTPUT\_3:** PMDTRG3 より出力

**TrgMode == PMD\_TRG\_VARIABLE\_OUTPUT の場合:**

- **PMD\_TRG\_OUTPUT\_0:** PMDTRG0 より出力
- **PMD\_TRG\_OUTPUT\_1:** PMDTRG1 より出力
- **PMD\_TRG\_OUTPUT\_2:** PMDTRG2 より出力
- **PMD\_TRG\_OUTPUT\_3:** PMDTRG3 より出力
- **PMD\_TRG\_OUTPUT\_4:** PMDTRG4 より出力
- **PMD\_TRG\_OUTPUT\_5:** PMDTRG5 より出力

**機能:**

トリガ出力を設定します。

**戻り値:**

なし

## 14.2.3.29PMD\_SetSelectMode

モードの選択

**関数のプロトタイプ宣言:**

```
void  
PMD_SetSelectMode(TSB_PMD_TypeDef * PMDx,  
                  uint32_t Mode);
```

**引数:**

**PMDx:** PMD チャンネルを指定します。

**Mode:** モードを選択します。

- **PMD\_BUS\_MODE:** ダブルバッファ後段へ入力するデータをバスから選択したレジスタ値を使用します。(バスモード)
- **PMD\_VE\_MODE:** ダブルバッファ後段へ入力するデータをベクトルエンジン(VE)からの値を使用します。(VE モード)

**機能:**

モードを選択します。

**戻り値:**

なし

## 14.2.3.30PMD\_EnableOVV

OVV 保護回路の許可

**関数のプロトタイプ宣言:**

void  
PMD\_EnableOVV(TSB\_PMD\_TypeDef \* **PMDx**);

**引数:**

**PMDx**: PMD チャンネルを指定します。

**機能:**

OVV 保護回路を許可します。

**戻り値:**

なし

## 14.2.3.31 PMD\_DisableOVV

OVV 保護回路の禁止

**関数のプロトタイプ宣言:**

void  
PMD\_DisableOVV(TSB\_PMD\_TypeDef \* **PMDx**);

**引数:**

**PMDx**: PMD チャンネルを指定します。

**機能:**

OVV 保護回路を禁止します。

**戻り値:**

なし

## 14.2.3.32 PMD\_SetOVVNoiseElimination

OVV 入力検出時間の設定

**関数のプロトタイプ宣言:**

void  
PMD\_SetOVVNoiseElimination(TSB\_PMD\_TypeDef \* **PMDx**,  
uint32\_t **NoiseElimination**);

**引数:**

**PMDx**: PMD チャンネルを指定します。

**NoiseElimination**: OVV 入力検出時間を選択します。

- **PMD\_NOISE\_ELIMINATION\_16**: OVV 入力検出時間 X16/fsys[s]
- **PMD\_NOISE\_ELIMINATION\_32**: OVV 入力検出時間 X32/fsys[s]
- **PMD\_NOISE\_ELIMINATION\_48**: OVV 入力検出時間 X48/fsys[s]
- **PMD\_NOISE\_ELIMINATION\_64**: OVV 入力検出時間 X64/fsys[s]
- **PMD\_NOISE\_ELIMINATION\_80**: OVV 入力検出時間 X80/fsys[s]
- **PMD\_NOISE\_ELIMINATION\_96**: OVV 入力検出時間 X96/fsys[s]
- **PMD\_NOISE\_ELIMINATION\_112**: OVV 入力検出時間 X112/fsys[s]

- **PMD\_NOISE\_ELIMINATION\_128:** OV V 入力検出時間 X128/fsys[s]
- **PMD\_NOISE\_ELIMINATION\_144:** OV V 入力検出時間 X144/fsys[s]
- **PMD\_NOISE\_ELIMINATION\_160:** OV V 入力検出時間 X160/fsys[s]
- **PMD\_NOISE\_ELIMINATION\_176:** OV V 入力検出時間 X176/fsys[s]
- **PMD\_NOISE\_ELIMINATION\_192:** OV V 入力検出時間 X192/fsys[s]
- **PMD\_NOISE\_ELIMINATION\_208:** OV V 入力検出時間 X208/fsys[s]
- **PMD\_NOISE\_ELIMINATION\_224:** OV V 入力検出時間 X224/fsys[s]
- **PMD\_NOISE\_ELIMINATION\_240:** OV V 入力検出時間 X240/fsys[s]

**機能:**

OV V 入力検出時間を設定します。

**戻り値:**

なし

### 14.2.3.33 PMD\_SetADCMonitorInput

ADC 監視割り込み入力の許可/禁止

**関数のプロトタイプ宣言:**

```
void  
PMD_SetADCMonitorInput(TSB_PMD_TypeDef * PMDx,  
                        uint32_t Monitor,  
                        FunctionalState NewState);
```

**引数:**

**PMDx:** PMD チャンネルを指定します。

**Monitor:** OV V 保護用 ADC 監視割り込み入力を選択します。

- **PMD\_ADC\_MONITOR\_A:** ADC A 監視割り込み
- **PMD\_ADC\_MONITOR\_B:** ADC B 監視割り込み

**NewState:** ADC 監視割り込み入力の許可/禁止を選択します。

- **ENABLE:** 入力許可
- **DIABLE:** 入力禁止

**機能:**

OV V 保護用 ADC 監視割り込み入力の許可/禁止を選択します。

**戻り値:**

なし

### 14.2.3.34 PMD\_SetOVVMode

OV V 保護モードの選択

**関数のプロトタイプ宣言:**

```
void  
PMD_SetOVVMode(TSB_PMD_TypeDef * PMDx,  
                uint32_t Mode);
```

引数:

**PMDx**: PMD チャンネルを指定します。

**Mode**: OVV 保護モードを選択します。

- **PMD\_OVV\_MODE\_0**: 出力制御なし。
- **PMD\_OVV\_MODE\_1**: 全上相オン、全下相オフ。
- **PMD\_OVV\_MODE\_2**: 全上相オフ、全下相オン。
- **PMD\_OVV\_MODE\_3**: 全相オフ (オン = High, OFF = Low [ハイアクティブ (PLL/H=1)時])

機能:

OVV 保護モードを選択します。

戻り値:

なし

## 14.2.3.35 PMD\_SetOVVInputSrc

OVV 入力選択

関数のプロトタイプ宣言:

```
void  
PMD_SetOVVInputSrc(TSB_PMD_TypeDef * PMDx,  
                    uint32_t Source);
```

引数:

**PMDx**: PMD チャンネルを指定します。

**Source**: OVV 入力を選択します。

- **PMD\_OVV\_PORT\_INPUT**: ポート入力
- **PMD\_OVV\_ADC\_MONITOR**: ADC 監視信号

機能:

OVV 入力を選択します。

戻り値:

なし

## 14.2.3.36 PMD\_SetOVVAutoRelease

OVV 保護状態からの自動復帰設定

関数のプロトタイプ宣言:

```
void  
PMD_SetOVVAutoRelease(TSB_PMD_TypeDef * PMDx,  
                       FunctionalState NewState);
```

引数:

**PMDx**: PMD チャンネルを指定します。

**NewState**: OVV 保護状態からの自動復帰の許可/禁止を選択します。

- **ENABLE:** 保護状態からの自動復帰許可
- **DIABLE:** 保護状態からの自動復帰禁止

**機能:**

OVV 保護状態からの自動復帰設定を行います。

**戻り値:**

なし

## 14.2.3.37PMD\_GetOVVAbnormalLevel

OVV 入力状態の取得

**関数のプロトタイプ宣言:**

uint32\_t

PMD\_GetOVVAbnormalLevel(TSB\_PMD\_TypeDef \* **PMDx**)

**引数:**

**PMDx:** PMD チャンネルを指定します。

**機能:**

OVV 入力状態を取得します。

**戻り値:**

OVV 入力状態

**PMD\_ABNORMAL\_LEVEL\_L:** OVV 保護入力が"L"

**PMD\_ABNORMAL\_LEVEL\_H:** OVV 保護入力が"H"

## 14.2.3.38PMD\_GetOVVCondition

OVV 保護状態の取得

**関数のプロトタイプ宣言:**

uint32\_t

PMD\_GetOVVCondition(TSB\_PMD\_TypeDef \* **PMDx**)

**引数:**

**PMDx:** PMD チャンネルを指定します。

**機能:**

OVV 保護状態を取得します。

**戻り値:**

OVV 保護状態:

**PMD\_OVV\_NORMAL:** 通常動作中

**PMD\_OVV\_PROTECTED:** 保護中

## 14.2.3.39PMD\_SetAutoSwitchCtrl

VE レジスタと PMD レジスタの自動切り替え許可/禁止



**関数のプロトタイプ宣言:**

```
void  
PMD_SetAutoSwitchCtrl(TSB_PMD_TypeDef * PMDx,  
                      FunctionalState NewState);
```

**引数:**

**PMDx**: PMD チャンネルを指定します。

**NewState**: 自動切り替え許可/禁止を選択します。

- **ENABLE**: 2レジスタ切り替え許可
- **DIABLE**: 2レジスタ切り替え禁止

**機能:**

VE レジスタと PMD レジスタの自動切り替えを許可/禁止します。

**戻り値:**

なし

## 14.2.3.40 PMD\_SetPWMEdge

PWM エッジ設定

**関数のプロトタイプ宣言:**

```
void  
PMD_SetPWMEdge(TSB_PMD_TypeDef * PMDx,  
               uint32_t PMDPhase,  
               uint32_t Edge);
```

**引数:**

**PMDx**: PMD チャンネルを指定します。

**PMDPhase**: 3 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**: すべての相

**Edge**: エッジを選択します。

- **PMD\_PWM\_EDGE\_UNFIXED**: エッジ固定解除
- **PMD\_PWM\_RISING\_EDGE\_FIXED**: PWM 立ち上がりエッジ固定
- **PMD\_PWM\_FALLING\_EDGE\_FIXED**: PWM 立ち下りエッジ固定

**機能:**

PWM エッジを設定します。

**戻り値:**

なし

## 14.2.3.41 PMD\_SetBufferUpdateTime

Duty コンペアレジスタと PWM 周期レジスタのダブルバッファ更新タイミングの設定

**関数のプロトタイプ宣言:**

```
void  
PMD_SetBufferUpdateTime(TSB_PMD_TypeDef * PMDx,  
                        uint32_t UpdateTime);
```

**引数:**

**PMDx**: PMD チャンネルを指定します。

**UpdateTime**: 更新タイミングを選択します。

- **PMD\_UPDATE\_TIME\_WRITE**: 割り込み周期設定(INTPRD)によります。
- **PMD\_UPDATE\_TIME\_MIN**: PWM キャリアボトムで更新
- **PMD\_UPDATE\_TIME\_MAX**: PWM キャリアピークで更新
- **PMD\_UPDATE\_TIME\_MIN\_MAX**: PWM キャリアのピークとボトムで更新

**機能:**

Duty コンペアレジスタと PWM 周期レジスタのダブルバッファ更新タイミングを設定します。

**戻り値:**

なし

## 14.2.3.42 PMD\_SetTrgSyncTime

トリガ同期設定

**関数のプロトタイプ宣言:**

```
void  
PMD_SetTrgSyncTime (TSB_PMD_TypeDef * PMDx,  
                    uint32_t SyncTime);
```

**引数:**

**PMDx**: PMD チャンネルを指定します。

**SyncTime**: 同期タイミングを選択します。

- **PMD\_TRGGER\_TIME\_ASYNC**: 非同期
- **PMD\_TRGGER\_TIME\_ENC**: INTENC (ENC 割り込み要求)発生時
- **PMD\_TRGGER\_TIME\_TMRB**: INTTB00 (TMRB 割り込み要求)発生時

**機能:**

トリガ同期設定を行います。

**戻り値:**

なし

## 14.2.3.43 PMD\_SetTrgUpdateTime

通電制御レジスタの更新タイミング選択

**関数のプロトタイプ宣言:**

```
void
```

```
PMD_SetTrgUpdateTime(TSB_PMD_TypeDef * PMDx,  
                    uint32_t UpdateTime);
```

引数:

**PMDx**: PMD チャンネルを指定します。

**UpdateTime**: 通電制御レジスタの更新タイミングを選択します。

- **PMD\_UPDATE\_TIME\_WRITE**: PWM 非同期
- **PMD\_UPDATE\_TIME\_MIN**: PWM キャリアボトム
- **PMD\_UPDATE\_TIME\_MAX**: PWM キャリアピーク
- **PMD\_UPDATE\_TIME\_MIN\_MAX**: キャリアピークおよびキャリアボトムを選択します。

機能:

通電制御レジスタの更新タイミングを選択します。

戻り値:

なし

## 14.2.4 データ構造

### 14.2.4.1 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=MTPD×MDPRD<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。