

# **TOSHIBA**

## **TX04 ペリフェラルドライバ ユーザーガイド (TMPM462)**

**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。

## 目次

<b>1. はじめに</b>	<b>1</b>
<b>2. TX04 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	4
3.2.4 データ構造	19
<b>4. CEC</b>	<b>21</b>
4.1 概要	21
4.2 API 関数	21
4.2.1 関数一覧	21
4.2.2 関数の種類	22
4.2.3 関数仕様	23
4.2.4 データ構造	39
<b>5. EXB</b>	<b>41</b>
5.1 概要	41
5.2 API 関数	41
5.2.1 関数一覧	41
5.2.2 関数の種類	41
5.2.3 関数仕様	42
5.2.4 データ構造	45
<b>6. CG</b>	<b>48</b>
6.1 概要	48
6.2 API 関数	48
6.2.1 関数一覧	48
6.2.2 関数の種類	49
6.2.3 関数仕様	49
6.2.4 データ構造	67
<b>7. FC</b>	<b>69</b>
7.1 概要	69
7.2 API 関数	69
7.2.1 関数一覧	69
7.2.2 関数の種類	69
7.2.3 関数仕様	70
7.2.4 データ構造	80
<b>8. FUART</b>	<b>81</b>
8.1 概要	81
8.2 API 関数	81
8.2.1 関数一覧	81
8.2.2 関数の種類	82
8.2.3 関数仕様	82
8.2.4 データ構造	93
<b>9. GPIO</b>	<b>97</b>
9.1 概要	97
9.2 API 関数	97

9.2.1 関数一覧.....	97
9.2.2 関数の種類.....	97
9.2.3 関数仕様.....	98
9.2.4 データ構造.....	110
<b>10. I2C .....</b>	<b>113</b>
10.1 概要.....	113
10.2 API 関数.....	113
10.2.1 関数一覧.....	113
10.2.2 関数の種類.....	113
10.2.3 関数仕様.....	114
10.2.4 データ構造.....	120
<b>11. IGBT .....</b>	<b>123</b>
11.1 概要.....	123
11.2 API 関数.....	123
11.2.1 関数一覧.....	123
11.2.2 関数の種類.....	123
11.2.3 関数仕様.....	124
11.2.4 データ構造.....	131
<b>12. LVD.....</b>	<b>134</b>
12.1 概要.....	134
12.2 API 関数.....	134
12.2.1 関数一覧.....	134
12.2.2 関数の種類.....	134
12.2.3 関数仕様.....	134
12.2.4 データ構造.....	139
<b>13. OFD .....</b>	<b>140</b>
13.1 概要.....	140
13.2 API 関数.....	140
13.2.1 関数一覧.....	140
13.2.2 関数の種類.....	140
13.2.3 関数仕様.....	140
13.2.4 データ構造.....	143
<b>14. RMC.....</b>	<b>144</b>
14.1 概要.....	144
14.2 API 関数.....	144
14.2.1 関数一覧.....	144
14.2.2 関数の種類.....	145
14.2.3 関数仕様.....	145
14.2.4 データ構造.....	152
<b>15. RTC .....</b>	<b>154</b>
15.1 概要.....	154
15.2 API 関数.....	154
15.2.1 関数一覧.....	154
15.2.2 関数の種類.....	155
15.2.3 関数仕様.....	155
15.2.4 データ構造.....	174
<b>16. SSP.....</b>	<b>176</b>
16.1 概要.....	176
16.2 API 関数.....	176
16.2.1 関数一覧.....	176
16.2.2 関数の種類.....	177

16.2.3 関数仕様.....	177
16.2.4 データ構造.....	186
<b>17. TMRB.....</b>	<b>188</b>
17.1 概要.....	188
17.2 API 関数.....	188
17.2.1 関数一覧.....	188
17.2.2 関数の種類.....	189
17.2.3 関数仕様.....	189
17.2.4 データ構造.....	198
<b>18. SIO/UART.....</b>	<b>201</b>
18.1 概要.....	201
18.2 API 関数.....	201
18.2.1 関数一覧.....	201
18.2.2 関数の種類.....	202
18.2.3 関数仕様.....	202
18.2.4 データ構造.....	216
<b>19. uDMAC.....</b>	<b>219</b>
19.1 概要.....	219
19.2 API 関数.....	219
19.2.1 関数一覧.....	219
19.2.2 関数の種類.....	220
19.2.3 関数仕様.....	221
19.2.4 データ構造.....	232
<b>20. WDT.....</b>	<b>239</b>
20.1 概要.....	239
20.2 API 関数.....	239
20.2.1 関数一覧.....	239
20.2.2 関数の種類.....	239
20.2.3 関数仕様.....	239
1.1.2 データ構造.....	242

## 1. はじめに

本製品は、東芝TX04シリーズマイコン用ペリフェラルドライバセットです。TMPM462x (※2参照) ペリフェラルドライバは、東芝TX04ペリフェラルドライバのTMPM462シリーズMCU用です。

TX04ペリフェラルドライバでは、ユーザーアプリケーション内で各ペリフェラルを簡単に使用するためのマクロ、データ構造、関数および使用例を用意しています。

TMPM462 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。

※2 本ドキュメントではTMPM462F10FG/TMPM462F15FGを表す部分を"TMPM462x"で表現します。

## 2. TX04 ペリフェラルドライバの構成

### **/Libraries**

TX04 CMSIS ファイルと TMPM462 ペリフェラルドライバが格納されています。

### **/Libraries/ TX04\_CMSIS**

このフォルダには TMPM462 CMSIS ファイルのデバイス・ペリフェラル・アクセス・レイヤーが格納されています。

### **/Libraries/TX04\_Periph\_Driver**

TMPM462 ペリフェラルドライバの全てのソースコードが格納されています。

### **/Libraries/TX04\_Periph\_Driver/inc**

TMPM462 ペリフェラルドライバのヘッダファイルが格納されています。

### **/Libraries/TX04\_Periph\_Driver/src**

TMPM462 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**

TMPM462 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**

TMPM462 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**

TMPM462 ペリフェラルドライバの使用例が格納されています。

### **/Utilities/TMPM462-EVAL**

TMPM462 ボードのハードウェアリソース用の設定ファイル、およびドライバファイル (例: led, key) が格納されています。

## 3. ADC

### 3.1 概要

本デバイスは、12 ビット逐次変換方式アナログ/デジタルコンバータ(AD コンバータ)を 1 ユニット内蔵し、合計 15 チャンネルの通常アナログ入力と 5 チャンネルの拡張アナログ入力を有します。

15 チャンネルのアナログ入力端子(AIN0 ~ AIN14)と、5 チャンネルの拡張アナログ入力端子 (AIN15 ~ AIN19) は、入出力ポートと兼用です。

12 ビット A/D コンバータは、以下のような特徴があります。

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

ADCドライバ API は、各モジュールの設定機能を持ち、チャンネル選択、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/TX04\_Periph\_Driver/src/tmpm462\_adc.c  
/Libraries/TX04\_Periph\_Driver/inc/tmpm462\_adc.h

補足:

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

### 3.2 API 関数

#### 3.2.1 関数一覧

- ◆ void ADC\_SWReset(TSB\_AD\_TypeDef \* **ADx**)
- ◆ void ADC\_SetClk(TSB\_AD\_TypeDef \* **ADx**,  
uint32\_t **Sample\_HoldTime**,  
uint32\_t **Prescaler\_Output**)
- ◆ void ADC\_Start(TSB\_AD\_TypeDef \* **ADx**)
- ◆ void ADC\_SetScanMode(TSB\_AD\_TypeDef \* **ADx**,

- ◆ void ADC\_SetRepeatMode(TSB\_AD\_TypeDef \* **ADx**,  
FunctionalState **NewState**)
- ◆ void ADC\_SetINTMode(TSB\_AD\_TypeDef \* **ADx**, uint32\_t **INTMode**)
- ◆ void ADC\_SetInputChannel(TSB\_AD\_TypeDef \* **ADx**, ADC\_AINx **InputChannel**)
- ◆ void ADC\_SetScanChannel(TSB\_AD\_TypeDef \* **ADx**,  
ADC\_AINx **StartChannel**,  
uint32\_t **Range**)
- ◆ void ADC\_SetVrefCut(TSB\_AD\_TypeDef \* **ADx**, uint32\_t **VrefCtrl**)
- ◆ void ADC\_SetIdleMode(TSB\_AD\_TypeDef \* **ADx**, FunctionalState **NewState**)
- ◆ void ADC\_SetVref(TSB\_AD\_TypeDef \* **ADx**, FunctionalState **NewState**)
- ◆ void ADC\_SetInputChannelTop(TSB\_AD\_TypeDef \* **ADx**,  
ADC\_AINx **TopInputChannel**)
- ◆ void ADC\_StartTopConvert(TSB\_AD\_TypeDef \* **ADx**)
- ◆ void ADC\_SetMonitor(TSB\_AD\_TypeDef \* **ADx**,  
ADC\_CMPCRx **ADCMPx**,  
FunctionalState **NewState**)
- ◆ void ADC\_ConfigMonitor(TSB\_AD\_TypeDef \* **ADx**,  
ADC\_CMPCRx **ADCMPx**,  
ADC\_MonitorTypeDef \* **Monitor**)
- ◆ void ADC\_SetHWTrg(TSB\_AD\_TypeDef \* **ADx**,  
uint32\_t **HWSrc**,  
FunctionalState **NewState**)
- ◆ void ADC\_SetHWTrgTop(TSB\_AD\_TypeDef \* **ADx**,  
uint32\_t **HWSrc**,  
FunctionalState **NewState**)
- ◆ ADC\_State ADC\_GetConvertState(TSB\_AD\_TypeDef \* **ADx**)
- ◆ ADC\_Result ADC\_GetConvertResult(TSB\_AD\_TypeDef \* **ADx**,  
ADC\_REGx **ADREGx**)
- ◆ void ADC\_EnableTrigger(void)
- ◆ void ADC\_DisableTrigger(void)
- ◆ ADC\_SetTriggerStartup(ADC\_TRGx **TriggerStartup**)
- ◆ ADC\_SetTriggerStartupTop(ADC\_TRGx **TopTriggerStartup**)
- ◆ ADC\_DisableExtension(void)
- ◆ ADC\_EnableExtension(void)
- ◆ ADC\_SetExtChannel(ADC\_EXT\_AINx **ExtChannel**)
- ◆ ADC\_SetExtClkSource(uint32\_t **ClkSrc**)
- ◆ ADC\_SetExtClk (uint32\_t **Sample\_HoldTime**)

## 3.2.2 関数の種類

関数は、主に以下の 6 種類に分かれています:

- 1) AD 変換設定:  
ADC\_SetClk(), ADC\_SetScanMode(), ADC\_SetRepeatMode(), ADC\_SetINTMode(),  
ADC\_SetInputChannel(), ADC\_SetScanChannel(), ADC\_SetVref(),  
ADC\_SetInputChannelTop(), ADC\_SetMonitor(), ADC\_ConfigMonitor(),  
ADC\_SetHWTrg(), ADC\_SetHWTrgTop()
- 2) AD 変換開始:  
ADC\_Start(), ADC\_StartTopConvert()
- 3) AD 変換ステータス/結果の読み出し:  
ADC\_GetConvertState(), ADC\_GetConvertResult()
- 4) その他:  
ADC\_SWReset(), ADC\_SetVrefCut(), ADC\_SetIdleMode()



- 5) AD 変換起動:  
ADC\_EnableTrigger(), ADC\_DisableTrigger(), ADC\_SetTriggerStartup(),  
ADC\_SetTriggerStartupTop()
- 6) 拡張 AD 変換の設定:  
ADC\_DisableExtension(), ADC\_EnableExtension(), ADC\_SetExtChannel(),  
ADC\_SetExtClkSource(), ADC\_SetExtClk()

## 3.2.3 関数仕様

### 3.2.3.1 ADC\_SWReset

AD 変換機能のソフトウェアリセット

関数のプロトタイプ宣言:

```
void  
ADC_SWReset(TSB_AD_TypeDef * ADx)
```

引数:

**ADx**: AD 変換のユニットを指定します。  
➤ **TSB\_AD**: AD コンバータユニット

機能:

AD 変換機能をソフトウェアリセットします。

補足:

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

戻り値:

なし

### 3.2.3.2 ADC\_SetClk

AD 変換サンプルホールド時間とプリスケアラ出力の設定

関数のプロトタイプ宣言:

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

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_AD**: AD コンバータユニット

**Sample\_HoldTime**: 以下から ADC サンプルホールド時間を選択します。

- **ADC\_CONVERSION\_CLK\_10**: 10 x <ADCLK>
- **ADC\_CONVERSION\_CLK\_20**: 20 x <ADCLK>
- **ADC\_CONVERSION\_CLK\_30**: 30 x <ADCLK>
- **ADC\_CONVERSION\_CLK\_40**: 40 x <ADCLK>
- **ADC\_CONVERSION\_CLK\_80**: 80 x <ADCLK>
- **ADC\_CONVERSION\_CLK\_160**: 160 x <ADCLK>
- **ADC\_CONVERSION\_CLK\_320**: 320 x <ADCLK>

**Prescaler\_Output:** 以下から ADC プリスケール出力(ADCLK)を選択します。

- **ADC\_FC\_DIVIDE\_LEVEL\_1:** fc
- **ADC\_FC\_DIVIDE\_LEVEL\_2:** fc / 2
- **ADC\_FC\_DIVIDE\_LEVEL\_4:** fc / 4
- **ADC\_FC\_DIVIDE\_LEVEL\_8:** fc / 8
- **ADC\_FC\_DIVIDE\_LEVEL\_16:** fc / 16

**機能:**

**Sample\_HoldTime** で ADC サンプルホールド時間を設定し、**Prescaler\_Output** でプリスケール出力を設定します。

**補足:**

AD変換中にこの関数を使用して、AD変換用クロックの設定を変更しないでください。

**ADC\_GetConvertState()** を使用して、AD変換状態が **BUSY** 以外のときに本関数を実行してください。

**戻り値:**

なし

### 3.2.3.3 ADC\_Start

AD 変換の開始

**関数のプロトタイプ宣言:**

void

ADC\_Start(TSB\_AD\_TypeDef \* **ADx**)

**引数:**

**ADx:** AD 変換のユニットを指定します。

- **TSB\_AD:** AD コンバータユニット

**機能:**

通常(ソフト)AD 変換を開始します。

**補足:**

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

チャンネル固定シングル変換モード

チャンネルスキャンシングル変換モード

チャンネル固定リピート変換モード

チャンネルスキャンリピート変換モード

詳細は下記関数の機能を参照してください。

**ADC\_SetScanMode(), ADC\_SetRepeatMode(), ADC\_SetInputChannel(),  
ADC\_SetScanChannel()**

AD 変換をスタートさせる場合は、**ADC\_SetVref (ENABLE)** を実行して Vref を有効にし、内部回路状態が安定するまで 3  $\mu$ s 待ってから **ADC\_Start()** を実行してください。

**戻り値:**

なし

## 3.2.3.4 ADC\_SetScanMode

AD 変換スキャンモードの有効/無効切り替え

関数のプロトタイプ宣言:

```
void  
ADC_SetScanMode(TSB_AD_TypeDef * ADx,  
                 FunctionalState NewState)
```

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_AD**: AD コンバータユニット

**NewState**: AD 変換スキャンモードの状態を指定します。

➤ **ENABLE**: スキャンモードを有効

➤ **DISABLE**: スキャンモードを無効

機能:

AD 変換スキャンモードの有効/無効を切り替えます。

戻り値:

なし

## 3.2.3.5 ADC\_SetRepeatMode

AD 変換リピートモードの有効/無効切り替え

関数のプロトタイプ宣言:

```
void  
ADC_SetRepeatMode(TSB_AD_TypeDef * ADx,  
                  FunctionalState NewState)
```

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_AD**: AD コンバータユニット

**NewState**: AD 変換リピートモードの状態を指定します。

➤ **ENABLE**: リピートモードを有効

➤ **DISABLE**: リピートモードを無効

機能:

AD 変換リピートモードの有効/無効を切り替えます。

戻り値:

なし

## 3.2.3.6 ADC\_SetINTMode

チャンネル固定リピート変換モードにおける AD 変換 割り込みモードの設定

関数のプロトタイプ宣言:

```
void  
ADC_SetINTMode(TSB_AD_TypeDef * ADx,  
               uint32_t INTMode)
```

**引数:**

**ADx:** AD 変換のユニットを指定します。

- **TSB\_AD:** AD コンバータユニット

**INTMode:** AD 変換割り込みモードを選択します。

- **ADC\_INT\_SINGLE:** 1 回変換ごと割り込み発生。
- **ADC\_INT\_CONVERSION\_2:** 2 回変換ごと割り込み発生。
- **ADC\_INT\_CONVERSION\_3:** 3 回変換ごと割り込み発生。
- **ADC\_INT\_CONVERSION\_4:** 4 回変換ごと割り込み発生。
- **ADC\_INT\_CONVERSION\_5:** 5 回変換ごと割り込み発生。
- **ADC\_INT\_CONVERSION\_6:** 6 回変換ごと割り込み発生。
- **ADC\_INT\_CONVERSION\_7:** 7 回変換ごと割り込み発生。
- **ADC\_INT\_CONVERSION\_8:** 8 回変換ごと割り込み発生。

**機能:**

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

**補足:**

本関数はチャンネル固定リピート変換モード設定後に使用してください。

以下はチャンネル固定リピートモードの例です。

1. **ADC\_SetScanMode(DISABLE)**
2. **ADC\_SetRepeatMode(ENABLE)**

**戻り値:**

なし

### 3.2.3.7 ADC\_SetInputChannel

AD 変換入力チャンネルの設定

**関数のプロトタイプ宣言:**

```
void  
ADC_SetInputChannel(TSB_AD_TypeDef * ADx,  
                    ADC_AINx InputChannel)
```

**引数:**

**ADx:** AD 変換のユニットを指定します。

- **TSB\_AD:** AD コンバータユニット

**InputChannel:** AD 変換入力チャンネルを選択します。

- **ADC\_AN\_00, ADC\_AN\_01, ADC\_AN\_02, ADC\_AN\_03, ADC\_AN\_04, ADC\_AN\_05, ADC\_AN\_06, ADC\_AN\_07, ADC\_AN\_08, ADC\_AN\_09, ADC\_AN\_10, ADC\_AN\_11, ADC\_AN\_12, ADC\_AN\_13, ADC\_AN\_14, ADC\_AN\_EXT**

**機能:**

**InputChannel** により、AD 変換入力チャンネルを設定します。

**\*補足:**

- 1 **ADC\_AN\_00~ADC\_AN\_EXT** の内 1 チャンネルだけ通常変換入力を選択可能です。
- 2 **InputChannel** が **ADC\_AN\_EXT** の場合、**AIN15** から **AIN19** のいずれか 1 つを選択してください。(ADC\_SetExtChannel())の説明を参照してください)

戻り値:  
なし

### 3.2.3.8 ADC\_SetScanChannel

AD 変換スキャンチャンネルの選択

関数のプロトタイプ宣言:

```
void
ADC_SetScanChannel(TSB_AD_TypeDef * ADx,
                   ADC_AINx StartChannel,
                   uint32_t Range)
```

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_AD**: AD コンバータユニット

**StartChannel**: スキャン開始チャンネルを指定します。

➤ **ADC\_AN\_00**, **ADC\_AN\_01**, **ADC\_AN\_02**, **ADC\_AN\_03**, **ADC\_AN\_04**,  
**ADC\_AN\_05**, **ADC\_AN\_06**, **ADC\_AN\_07**, **ADC\_AN\_08**, **ADC\_AN\_09**,  
**ADC\_AN\_10**, **ADC\_AN\_11**, **ADC\_AN\_12**, **ADC\_AN\_13**, **ADC\_AN\_14**

**Range**: チャンネルスキャンの範囲を設定します。

➤ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 (補足: **StartChannel** + **Range** <= 15)

機能:

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

補足:

有効なチャンネルスキャンの設定値を以下に示します:

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

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

戻り値:  
なし

### 3.2.3.9 ADC\_SetVrefCut

AVREFH-AVREFL 間のリファレンス電流制御

関数のプロトタイプ宣言:

```
void  
ADC_SetVrefCut(TSB_AD_TypeDef * ADx,  
               uint32_t VrefCtrl)
```

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_AD**: AD コンバータユニット

**VrefCtrl**: AVREFH-AVREFL 間のリファレンス電流の制御方法を指定します。

➤ **ADC\_APPLY\_VREF\_IN\_CONVERSION**: 変換中のみ通電。

➤ **ADC\_APPLY\_VREF\_AT\_ANY\_TIME**: リセット時以外常時通電。

機能:

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

戻り値:  
なし

### 3.2.3.10 ADC\_SetIdleMode

IDLE モード時の ADC 動作制御の指定

関数のプロトタイプ宣言:

```
void  
ADC_SetIdleMode(TSB_AD_TypeDef * ADx,  
                FunctionalState NewState)
```

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_AD**: AD コンバータユニット

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

➤ **ENABLE**: 動作

➤ **DISABLE**: 停止

機能:

IDLE モード時の ADC 動作制御の動作/停止を指定します。

システムが IDLE モードに遷移する前に実行する必要があります。

戻り値:  
なし

## 3.2.3.11 ADC\_SetVref

ADC Vref アプリケーションの回路 ON/OFF 制御

関数のプロトタイプ宣言:

```
void  
ADC_SetVref(TSB_AD_TypeDef * ADx,  
            FunctionalState NewState)
```

引数:

**ADx**: AD 変換のユニットを指定します。

- **TSB\_AD**: AD コンバータユニット

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

- **ENABLE**: Vref ON
- **DISABLE**: Vref OFF

機能:

ADC Vref アプリケーションの回路 ON/OFF を制御します。

補足:

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

戻り値:

なし

## 3.2.3.12 ADC\_SetInputChannelTop

最優先 AD 変換入力チャネルの設定

関数のプロトタイプ宣言:

```
void  
ADC_SetInputChannelTop(TSB_AD_TypeDef * ADx,  
                      ADC_AINx TopInputChannel)
```

引数:

**ADx**: AD 変換のユニットを指定します。

- **TSB\_AD**: AD コンバータユニット

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

- **ADC\_AN\_00**, **ADC\_AN\_01**, **ADC\_AN\_02**, **ADC\_AN\_03**, **ADC\_AN\_04**,  
**ADC\_AN\_05**, **ADC\_AN\_06**, **ADC\_AN\_07**, **ADC\_AN\_08**, **ADC\_AN\_09**,  
**ADC\_AN\_10**, **ADC\_AN\_11**, **ADC\_AN\_12**, **ADC\_AN\_13**, **ADC\_AN\_14**,  
**ADC\_AN\_EXT**

機能:

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

補足:

1 最優先 AD 変換入力チャネルには、**ADC\_AN\_0~ADC\_AN\_EXT** のうちのの一つを選ぶことができます。

2 **TopInputChannel** が **ADC\_AN\_EXT** の場合、**AIN15** から **AIN19** のいずれか 1 つを選択してください。(ADC\_SetExtChannel())の説明を参照してください)

戻り値:

なし

### 3.2.3.13 ADC\_StartTopConvert

最優先 AD 変換の開始

関数のプロトタイプ宣言:

```
void  
ADC_StartTopConvert(TSB_AD_TypeDef * ADx)
```

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_AD**: AD コンバータユニット

機能:

最優先 AD 変換を開始します。

補足:

本関数を実行する前に、**ADC\_SetInputChannelTop()** を実行してください。

戻り値:

なし

### 3.2.3.14 ADC\_SetMonitor

AD 監視機能の設定

関数のプロトタイプ宣言:

```
void  
ADC_SetMonitor(TSB_AD_TypeDef * ADx,  
               ADC_CMPCRx ADCMPx,  
               FunctionalState NewState)
```

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_AD**: AD コンバータユニット

**ADCMPx**: AD 監視機能の比較レジスタを選択します。

➤ **ADC\_CMPCR\_0**: ADCMPCR0

➤ **ADC\_CMPCR\_1**: ADCMPCR1

**NewState**: AD 監視機能の有効/無効を選択します。

➤ **ENABLE**: ADC 監視の有効

➤ **DISABLE**: ADC 監視の無効

機能:

AD 監視チャンネルは、チャンネル 0 とチャンネル 1 の 2 種類です。

**ADCMPx** で AD 監視チャンネルを設定し、**NewState** で有効/無効の設定をします。

戻り値:

なし



## 3.2.3.15 ADC\_ConfigMonitor

ADC 監視モジュールの設定

関数のプロトタイプ宣言:

```
void  
ADC_ConfigMonitor(TSB_AD_TypeDef * ADx,  
                  ADC_CMPCRx ADCMPx,  
                  ADC_MonitorTypeDef * Monitor)
```

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_AD**: AD コンバータユニット

**ADCMPx**: AD 監視機能の比較レジスタを選択します。

➤ **ADC\_CMPCR\_0**: ADCMPCR0

➤ **ADC\_CMPCR\_1**: ADCMPCR1

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

機能:

AD 監視チャンネルは、チャンネル 0 とチャンネル 1 の 2 種類です。

**ADCMPx** で AD 監視チャンネルを設定し、**Monitor** で AD 監視設定を行います。

補足: 本関数の実行前に AD 監視モジュールを無効にしてください。

戻り値:

なし

## 3.2.3.16 ADC\_SetHWTrg

通常 AD 変換のハードウェア起動と起動ソースの設定

関数のプロトタイプ宣言:

```
void  
ADC_SetHWTrg(TSB_AD_TypeDef * ADx,  
              uint32_t HWSrc,  
              FunctionalState NewState)
```

引数:

**ADx**: AD 変換のユニットを指定します。

➤ **TSB\_AD**: AD コンバータユニット

**HWSrc**: 通常 AD 変換の起動ソースを選択します。

➤ **ADC\_EXTERADTRG**: ADTRG 端子

➤ **ADC\_INTERTRIGGER**: 内部トリガ (ADILVTRGSEL<TRGSEL>にて選択)

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

➤ **ENABLE**: ハードウェアトリガを有効

➤ **DISABLE**: ハードウェアトリガを無効

機能:

**HWSrc** により、通常 AD 変換のハードウェア起動と起動ソースを設定します。また

**NewState** により、通常 AD 変換のハードウェアトリガの有効/無効を指定します。

**\*補足:**

最優先 AD 変換のハードウェア起動を使用する場合、通常 AD 変換のハードウェア起動用の外部トリガを使用することはできません。

**戻り値:**

なし

### 3.2.3.17 ADC\_SetHWTrgTop

最優先 AD 変換のハードウェア起動と起動ソースの設定

**関数のプロトタイプ宣言:**

```
void  
ADC_SetHWTrgTop(TSB_AD_TypeDef * ADx,  
                uint32_t HWSrc,  
                FunctionalState NewState)
```

**引数:**

**ADx**: AD 変換のユニットを指定します。

- **TSB\_AD**: AD コンバータユニット

**HWSrc**: 最優先 AD 変換の起動ソースを選択します。

- **ADC\_EXTERADTRG**:  $\overline{\text{ADTRG}}$  端子
- **ADC\_INTERTRIGGER**: 内部トリガ (ADILVTRGSEL<HPTRGSEL>にて選択)

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

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

**機能:**

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

**\*補足:**

最優先 AD 変換のハードウェア起動を使用する場合、通常 AD 変換のハードウェア起動用の外部トリガを使用することはできません。

**戻り値:**

なし

### 3.2.3.18 ADC\_GetConvertState

通常 AD 変換完了フラグおよび最優先 AD 変換完了フラグの確認

**関数のプロトタイプ宣言:**

```
ADC_State  
ADC_GetConvertState(TSB_AD_TypeDef * ADx)
```

**引数:**

**ADx**: AD 変換のユニットを指定します。

- **TSB\_AD**: AD コンバータユニット

**機能:**

通常 AD 変換状態および最優先 AD 変換状態を確認します。本関数を実行することで、AD 変換が終了したかどうか確認できます。

戻り値:

通常 AD 変換状態:

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

### 3.2.3.19 ADC\_GetConvertResult

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

関数のプロトタイプ宣言:

ADC\_Result

ADC\_GetConvertResult(TSB\_AD\_TypeDef \* **ADx**,  
ADC\_REGx **ADREGx**)

引数:

**ADx**: AD 変換のユニットを指定します。

- **TSB\_AD**: AD コンバータユニット

**ADREGx**: AD 変換結果レジスタを選択します。

- **ADC\_REG\_00, ADC\_REG\_01, ADC\_REG\_02, ADC\_REG\_03, ADC\_REG\_04, ADC\_REG\_05, ADC\_REG\_06, ADC\_REG\_07, ADC\_REG\_08, ADC\_REG\_09, ADC\_REG\_10, ADC\_REG\_11, ADC\_REG\_12, ADC\_REG\_13, ADC\_REG\_14, ADC\_REG\_15, ADC\_REG\_SP**

機能:

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

アナログ入力チャネルと AD 変換結果レジスタの関係を下表に示します。

チャネル固定シングルモード		
ユニット	チャネル	変換結果レジスタ
<b>TSB_AD</b>	ADC_AN_00	ADC_REG_00
	ADC_AN_01	ADC_REG_01
	ADC_AN_02	ADC_REG_02
	ADC_AN_03	ADC_REG_03
	ADC_AN_04	ADC_REG_04
	ADC_AN_05	ADC_REG_05
	ADC_AN_06	ADC_REG_06
	ADC_AN_07	ADC_REG_07
	ADC_AN_08	ADC_REG_08
	ADC_AN_09	ADC_REG_09
	ADC_AN_10	ADC_REG_10
	ADC_AN_11	ADC_REG_11
	ADC_AN_12	ADC_REG_12
	ADC_AN_13	ADC_REG_13
	ADC_AN_14	ADC_REG_14
	ADC_AN_EXT	ADC_REG_15

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

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

AD 変換モードの詳細は、関連 API を参照ください。

**\*補足:**

- 1 最優先 AD 変換の結果は "ADC\_REG\_SP" に格納されます。
- 2 12 ビット拡張 A/D 変換の結果は "ADC\_REG\_15" に格納されます。

**戻り値:**

AD 変換結果:

**ADResult** (Bit 0 - Bit 11) : AD 変換値が格納されます。

**Stored** (Bit 12) : AD 変換値が格納されると '1' がセットされます。  
このフラグはリードすると '0' にクリアされます。

**OverRun** (Bit 13) 新しい AD 変換値が上書きされると '1' がセットされます。  
このフラグはリードすると '0' にクリアされます。

## 3.2.3.20 ADC\_EnableTrigger

トリガ動作の有効

**関数のプロトタイプ宣言:**

void  
ADC\_EnableTrigger(void)

**引数:**

なし

**機能:**

トリガを有効にします。

**戻り値:**

なし

## 3.2.3.21 ADC\_DisableTrigger

トリガ動作の無効

**関数のプロトタイプ宣言:**

void  
ADC\_DisableTrigger(void)

**引数:**

なし

**機能:**

トリガを無効にします。

**戻り値:**

なし

## 3.2.3.22 ADC\_SetTriggerStartup

通常 AD 変換起動トリガの選択

**関数のプロトタイプ宣言:**

void  
ADC\_SetTriggerStartup(ADC\_TRGx *TriggerStartup*)

**引数:**

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

- ADC\_TRG\_00, ADC\_TRG\_01, ADC\_TRG\_02, ADC\_TRG\_03,  
ADC\_TRG\_04, ADC\_TRG\_05, ADC\_TRG\_06, ADC\_TRG\_07,  
ADC\_TRG\_08, ADC\_TRG\_09

**機能:**

通常 AD 変換起動トリガを選択します。

**戻り値:**

なし

### 3.2.3.23 ADC\_SetTriggerStartupTop

最優先 AD 変換起動トリガの選択

関数のプロトタイプ宣言:

void

ADC\_SetTriggerStartupTop(ADC\_TRGx *TopTriggerStartup*)

引数:

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

- ADC\_TRG\_00, ADC\_TRG\_01, ADC\_TRG\_02, ADC\_TRG\_03,  
ADC\_TRG\_04, ADC\_TRG\_05, ADC\_TRG\_06, ADC\_TRG\_07,  
ADC\_TRG\_08, ADC\_TRG\_09

機能:

最優先 AD 変換起動トリガを選択します。

戻り値:

なし

### 3.2.3.24 ADC\_DisableExtension

拡張チャネルの使用禁止

関数のプロトタイプ宣言:

void

ADC\_DisableExtension(void)

引数:

なし

機能:

拡張チャネルを使用しません。

戻り値:

なし

### 3.2.3.25 ADC\_EnableExtension

拡張チャネルの使用許可

関数のプロトタイプ宣言:

void

ADC\_EnableExtension(void)

引数:

なし

機能:

拡張チャネルを使用します。

戻り値:

なし

### 3.2.3.26 ADC\_SetExtChannel

拡張チャネルの選択

関数のプロトタイプ宣言:

void

ADC\_SetExtChannel(ADC\_EXT\_AINx **ExtChannel**)

引数:

**ExtChannel**: 拡張チャネルを選択します。

- ADC\_EXT\_AN\_15, ADC\_EXT\_AN\_16, ADC\_EXT\_AN\_17, ADC\_EXT\_AN\_18, ADC\_EXT\_AN\_19

機能:

拡張チャネルを選択します。

戻り値:

なし

### 3.2.3.27 ADC\_SetExtClkSource

拡張 AD 変換のサンプリング期間のソース選択

関数のプロトタイプ宣言:

void

ADC\_SetExtClkSource(uint32\_t **ClkSrc**)

引数:

**ClkSrc**: 拡張 AD 変換のサンプリング期間のソースを選択します。

- ADC\_EXT\_CLK\_SRC\_EXCR: EXCR0 の設定のサンプリング期間を選択します。
- ADC\_EXT\_CLK\_SRC\_ADCLK: ADCLK<ADSH[3:0]>の設定を選択します。

機能:

拡張 AD 変換のサンプリング期間のソースを選択します。

戻り値:

なし

### 3.2.3.28 ADC\_SetExtClk

拡張チャネルのサンプリング期間を選択します。

関数のプロトタイプ宣言:

void

ADC\_SetExtClk (uint32\_t **Sample\_HoldTime**)

引数:

**Sample\_HoldTime**: 拡張 AD 変換のサンプリング期間を選択します。

- **ADC\_CONVERSION\_CLK\_10**: 10 x <ADCLK>
- **ADC\_CONVERSION\_CLK\_20**: 20 x <ADCLK>

機能:

拡張 AD 変換のサンプリング期間を選択します。

戻り値:

なし

## 3.2.4 データ構造

### 3.2.4.1 ADC\_MonitorTypeDef

メンバ:

ADC\_AINx

**CmpChannel**: ADC チャンネルを指定します。

**ADC\_AN\_00 - ADC\_AN\_14, ADC\_AN\_EXT** (16 チャンネル)

uint32\_t

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

ADC\_CmpCondition

**Condition** 大小判定を設定します。

- **ADC\_LARGER\_THAN\_CMP\_REG**: 比較レジスタ 0 よりも変換結果レジスタの値が大きいと割り込みを発生します。
- **ADC\_SMALLER\_THAN\_CMP\_REG**: 比較レジスタ 0 よりも変換結果レジスタの値が小さいと割り込みを発生します。

ADC\_CmpCntMode

**CntMode** 判定カウント条件を指定します。

- **ADC\_SEQUENCE\_CMP\_MODE**: 連続方式
- **ADC\_CUMULATION\_CMP\_MODE**: 蓄積方式

uint32\_t

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

(補足: 詳細はデータシートの“AD 監視機能”を参照してください。)

### 3.2.4.2 ADC\_State

メンバ:

uint32\_t

**All** AD 変換の状態を指定します。

ビットフィールド

uint32\_t

**NormalBusy**(Bit 0)

通常 AD 変換 BUSY フラグ(ADBF)



'1': 変換中  
'0': 変換停止

uint32\_t  
**NormalComplete** (Bit 1) 通常 AD 変換終了フラグ (EOCF)  
'1': 変換終了  
'0': 変換前または変換中

uint32\_t  
**TopBusy** (Bit 2) 最優先 AD 変換 BUSY フラグ (HPADBF)  
'1': 変換中  
'0': 変換停止

uint32\_t  
**TopComplete** (Bit 3) 最優先 AD 変換終了フラグ (HPEOCF)  
'1': 変換終了  
'0': 変換前または変換中

uint32\_t  
**Reserved** (Bit 4 - Bit 31) 未使用

### 3.2.4.3 ADC\_Result

メンバ:  
uint32\_t  
**All** AD 変換結果

ビットフィールド:

uint32\_t  
**ADResult** (Bit 0 - Bit 11) AD 変換結果値  
uint32\_t  
**Stored** (Bit 12) AD 変換結果格納フラグ  
'1': 変換結果あり  
'0': 変換結果なし

uint32\_t  
**OverRun** (Bit 13) オーバーランフラグ  
'1': 発生あり  
'0': 発生なし

uint32\_t  
**Reserved** (Bit 14 - Bit 31) 未使用

## 4. CEC

### 4.1 概要

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

#### 受信

- 32kHz クロックまたは 16 ビットタイマフリップフロップ出力(TBxOUT)でサンプリング
  1. ノイズキャンセル時間を調整可能
- 1byte 毎にデータを受信
  1. データサンプリングポイントを調整可能
  2. ディスティネーションアドレス不一致でも受信可能
- エラー検出
  1. 周期違反(最少/最大)
  2. ACK 衝突
  3. 波形エラー

#### 送信

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

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

/Libraries/TX04\_Periph\_Driver/src/tmpm462\_cec.c

/Libraries/TX04\_Periph\_Driver/inc/tmpm462\_cec.h

### 4.2 API 関数

#### 4.2.1 関数一覧

- ◆ void CEC\_Enable(void)
- ◆ void CEC\_Disable(void)
- ◆ void CEC\_SWReset(void)
- ◆ Result CEC\_DefaultConfig(void)
- ◆ Result CEC\_SetLogicalAddr(CEC\_LogicalAddr **LogicalAddr**)
- ◆ Result CEC\_AddLogicalAddr(CEC\_LogicalAddr **LogicalAddr**)
- ◆ Result CEC\_RemoveLogicalAddr(CEC\_LogicalAddr **LogicalAddr**)
- ◆ CEC\_AddrListTypeDef CEC\_GetLogicalAddr(void)
- ◆ void CEC\_SetRxCtrl( FunctionalState **NewState** )
- ◆ Result CEC\_StartTx(void)
- ◆ void CEC\_StopTx(void)
- ◆ CEC\_DataTypeDef CEC\_GetRxData(void)
- ◆ void CEC\_SetTxData(uint8\_t **Data**,CEC\_EOMBit **EOM\_Flag**)

- ◆ FunctionalState CEC\_GetRxState(void)
- ◆ WorkState CEC\_GetTxState(void)
- ◆ CEC\_RxINTState CEC\_GetRxINTState(void)
- ◆ CEC\_TxINTState CEC\_GetTxINTState(void)
- ◆ Result CEC\_SetACKResponseMode(FunctionalState **NewState**)
- ◆ Result CEC\_SetNoiseCancellation(CEC\_LowCancellation **LowCancellation**,  
CEC\_HighCancellation **HighCancellation**)
  
- ◆ Result CEC\_SetCycleConfig(CEC\_CycleMin **CycleMin**, CEC\_CycleMax **CycleMax**)
- ◆ Result CEC\_SetDataValidTime(CEC\_ValidTime **ValidTime**)
- ◆ Result CEC\_SetTimeOutMode(CEC\_TimeOut **TimeOut**);
- ◆ Result CEC\_SetRxErrINTSuspend(FunctionalState **NewState**)
- ◆ Result CEC\_SetSnoopMode(FunctionalState **NewState**)
- ◆ Result CEC\_SetRxDetectWaveConfig (  
CEC\_Logical1RisingTimeMin **Logical1RisingTimeMin**,  
CEC\_Logical1RisingTimeMax **Logical1RisingTimeMax**,  
CEC\_Logical0RisingTimeMin **Logical0RisingTimeMin**,  
CEC\_Logical0RisingTimeMax **Logical0RisingTimeMax**)
  
- ◆ Result CEC\_SetRxStartBitWaveConfig(  
CEC\_StartBitRisingTimeMin **RisingTimeMin**,  
CEC\_StartBitRisingTimeMax **RisingTimeMax**,  
CEC\_StartBitCycleMin **CycleMin**,  
CEC\_StartBitCycleMax **CycleMax**)
  
- ◆ Result CEC\_SetRxWaveErrDetect(FunctionalState **NewState**)
- ◆ Result CEC\_SetTxWaveConfig(  
CEC\_TxDataBitCycle **DataBitCycle**,  
CEC\_TxDataBitRisingTime **DataBitRisingTime**,  
CEC\_TxStartBitCycle **StartBitCycle**,  
CEC\_TxStartBitRisingTime **StartBitRisingTime**)
  
- ◆ Result CEC\_SetTxBroadcast(FunctionalState **NewState**)
- ◆ Result CEC\_SetBusFreeTime(CEC\_BusFree **BusFree**)
- ◆ Result CEC\_SetRxStartBitDetect(FunctionalState **NewState**)
- ◆ void CEC\_SetSamplingClk(CEC\_SamplingClockSrc **ClkSrc**)

## 4.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています:

- 1) CEC の初期化と各種設定:  
CEC\_Enable(), CEC\_Disable(), CEC\_DefaultConfig(), CEC\_SetLogicalAddr(),  
CEC\_AddLogicalAddr(), CEC\_RemoveLogicalAddr(), CEC\_GetLogicalAddr(),  
CEC\_SetACKResponseMode(), CEC\_SetNoiseCancellation(),  
CEC\_SetCycleConfig(), CEC\_SetDataValidTime(), CEC\_SetTimeOutMode(),  
CEC\_SetRxErrINTSuspend(), CEC\_SetSnoopMode(),  
CEC\_SetRxDetectWaveConfig(), CEC\_SetRxStartBitWaveConfig(),  
CEC\_SetRxWaveErrDetect(), CEC\_SetTxWaveConfig(), CEC\_SetTxBroadcast(),  
CEC\_SetBusFreeTime(), CEC\_SetRxStartBitDetect(), CEC\_SetSamplingClk()
- 2) 送受信とエラーチェック:  
CEC\_SetRxCtrl(), CEC\_StartTx(), CEC\_StopTx(), CEC\_GetRxData(),  
CEC\_SetTxData(), CEC\_GetRxState(), CEC\_GetTxState(), CEC\_GetRxINTState(),  
CEC\_GetTxINTState()
- 3) その他:  
CEC\_SWReset()

## 4.2.3 関数仕様

### 4.2.3.1 CEC\_Enable

CEC 動作の許可

**関数のプロトタイプ宣言:**

void  
CEC\_Enable(void)

**引数:**

なし

**機能:**

CEC 動作を許可します。使用前に CEC を許可してください。

**戻り値:**

なし

### 4.2.3.2 CEC\_Disable

CEC 動作の禁止

**関数のプロトタイプ宣言:**

void  
CEC\_Disable(void)

**引数:**

なし

**機能:**

CEC 動作を禁止します。

**戻り値:**

なし

### 4.2.3.3 CEC\_SWReset

ソフトウェアリセットの発生

**関数のプロトタイプ宣言:**

void  
CEC\_SWReset(void)

**引数:**

なし

**機能:**

CEC 回路を初期化するリセット信号を発生します。リセット後、すべての制御レジスタやステータスフラグはリセット後の値に初期化されます。

本関数をコールした後、リセットが終了したか確認可能な以下の関数を呼び出すことが可能です。

**CEC\_GetRxState()** (戻り値は **DISABLE** になります)

**CEC\_GetTxState()** (戻り値は **CEC\_TX\_STOPED** になります)

戻り値:

なし

#### 4.2.3.4 CEC\_DefaultConfig

デフォルト値の設定

関数のプロトタイプ宣言:

Result

CEC\_DefaultConfig(void)

引数:

なし

機能:

CEC を以下の値で設定します。

Idle Mode: on

Noise Cancellation Time: H: 1 cycle L: 1 cycle

Cycle Range: 2.05ms~2.75ms

Data Valid Time: 1.05ms

Time Out: 1 Bit

Rx Start Wave configure: Min of start: 3.5ms; Max of start: 3.9ms

Min of cycle: 4.3ms; Max of cycle: 4.7ms

Receive Bit Wave configure: Min of "1": 0.4ms; Max of "1": 0.8ms

Min of "0": 1.3ms; Max of "0": 1.7

Send Bit Wave configure: RV

Bus free configure: 5 bit cycle

Snoop mode: On

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

戻り値:

➤ **SUCCESS** 成功

➤ **ERROR** エラー

#### 4.2.3.5 CEC\_SetLogicalAddr

ロジカルアドレスの設定

関数のプロトタイプ宣言:

Result

CEC\_SetLogicalAddr(CEC\_LogicalAddr **LogicalAddr**)

引数:

**LogicalAddr**: 以下から設定するロジカルアドレスを選択します。

- **CEC\_TV, CEC\_RECORDING\_DEVICE\_1, CEC\_RECORDING\_DEVICE\_2, CEC\_STB\_1, CEC\_DVD\_1, CEC\_AUDIO\_SYSTEM, CEC\_STB\_2, CEC\_STB\_3,**

CEC\_DVD\_2, CEC\_RECORDING\_DEVICE\_3, CEC\_FREE\_USE,  
CEC\_BROADCAST

**機能:**

ロジカルアドレスを設定します。

本関数が実行されると、以前設定されたロジカルアドレスはクリアされ、新しいロジカルアドレスが設定されます。

CEC が受信許可、または送信中の場合、本関数の戻り値は **ERROR** となり、設定変更は行いません。

**戻り値:**

- **SUCCESS** 成功
- **ERROR** エラー

## 4.2.3.6 CEC\_AddLogicalAddr

ロジカルアドレスの追加

**関数のプロトタイプ宣言:**

Result

CEC\_AddLogicalAddr(CEC\_LogicalAddr **LogicalAddr**)

**引数:**

**LogicalAddr:** 以下から追加するロジカルアドレスを指定します。

- CEC\_TV, CEC\_RECORDING\_DEVICE\_1,  
CEC\_RECORDING\_DEVICE\_2, CEC\_STB\_1, CEC\_DVD\_1,  
CEC\_AUDIO\_SYSTEM, CEC\_STB\_2, CEC\_STB\_3, CEC\_DVD\_2,  
CEC\_RECORDING\_DEVICE\_3, CEC\_FREE\_USE, CEC\_BROADCAST

**機能:**

ロジカルアドレスを追加します。本関数が実行されると、以前に設定されたロジカルアドレスを残したまま、新しいロジカルアドレスを追加します。

CEC が受信許可、または、送信中の場合、本関数の戻り値は **ERROR** となり、設定を変更しません。

**戻り値:**

- **SUCCESS** 成功
- **ERROR** エラー

## 4.2.3.7 CEC\_RemoveLogicalAddr

ロジカルアドレスの削除

**関数のプロトタイプ宣言:**

Result

CEC\_RemoveLogicalAddr(CEC\_LogicalAddr **LogicalAddr**)

**引数:**

**LogicalAddr:** 以下から削除するロジカルアドレスを指定します。

- CEC\_TV, CEC\_RECORDING\_DEVICE\_1,  
CEC\_RECORDING\_DEVICE\_2, CEC\_STB\_1, CEC\_DVD\_1,

CEC\_AUDIO\_SYSTEM, CEC\_STB\_2, CEC\_STB\_3, CEC\_DVD\_2,  
CEC\_RECORDING\_DEVICE\_3, CEC\_FREE\_USE, CEC\_BROADCAST

**機能:**

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

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

**戻り値:**

- **SUCCESS** 成功
- **ERROR** エラー

#### 4.2.3.8 CEC\_GetLogicalAddr

ロジカルアドレスの取得

**関数のプロトタイプ宣言:**

CEC\_AddrListTypeDef  
CEC\_GetLogicalAddr(void)

**引数:**

なし

**機能:**

ロジカルアドレス情報を取得します。

**戻り値:**

ロジカルアドレス情報:

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

#### 4.2.3.9 CEC\_SetRxCtrl

データ受信制御

**関数のプロトタイプ宣言:**

void  
CEC\_SetRxCtrl(FunctionalState **NewState**)

**引数:**

**NewState**: データ受信機能有効 / 無効を選択します。

- **ENABLE** : 有効
- **DISABLE** : 無効

**機能:**

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

**戻り値:**

なし

## 4.2.3.10 CEC\_StartTx

送信の開始

**関数のプロトタイプ宣言:**

Result

CEC\_StartTx(void)

**引数:**

なし

**機能:**

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

**戻り値:**

- **SUCCESS** 成功
- **ERROR** エラー

## 4.2.3.11 CEC\_StopTx

送信の停止

**関数のプロトタイプ宣言:**

void

CEC\_StopTx(void)

**引数:**

なし

**機能:**

送信を停止します。

**戻り値:**

なし

## 4.2.3.12 CEC\_GetRxData

受信データの読み出し

**関数のプロトタイプ宣言:**

CEC\_DataTypeDef

CEC\_GetRxData (void)

**引数:**

なし

**機能:**



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

**戻り値:**

受信バッファからの受信データ

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

#### 4.2.3.13 CEC\_SetTxData

送信データの設定

**関数のプロトタイプ宣言:**

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

**引数:**

**Data**: 送信データを指定します。データ長は 1 バイトです。

**EOM\_Flag**: 送信する EOM ビットを設定します。

- **CEC\_EOM**: フレームの最終データの場合に設定します。
- **CEC\_NO\_EOM**: フレームの最終データ以外の場合に設定します。

**機能:**

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

**戻り値:**

なし

#### 4.2.3.14 CEC\_GetRxState

受信状態の取得

**関数のプロトタイプ宣言:**

```
FunctionalState  
CEC_GetRxState (void)
```

**引数:**

なし

**機能:**

受信状態を取得します。

**戻り値:**

- **ENABLE**: 動作中
- **DISABLE**: 停止中

## 4.2.3.15 CEC\_GetTxState

送信状態の取得

**関数のプロトタイプ宣言:**

WorkState

CEC\_GetTxState(void)

**引数:**

なし

**機能:**

送信状態を取得します。

**戻り値:**

- **BUSY:** 送信中
- **DONE:** 送信していない

## 4.2.3.16 CEC\_GetRxINTState

受信割り込みステータスの取得

**関数のプロトタイプ宣言:**

CEC\_RxINTState

CEC\_GetRxINTState(void)

**引数:**

なし

**機能:**

受信割り込みステータスを取得します。

**戻り値:**

受信割り込みステータス:

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

## 4.2.3.17 CEC\_GetTxINTState

送信割り込みステータスの取得

**関数のプロトタイプ宣言:**

CEC\_TxINTState

CEC\_GetTxINTState(void)

**引数:**

なし

**機能:**

送信割り込みステータスを取得します。

**戻り値:**

送信割り込みステータス:

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

#### 4.2.3.18 CEC\_SetACKResponseMode

ACK 応答モードの設定

**関数のプロトタイプ宣言:**

Result

CEC\_SetACKResponseMode(FunctionalState **NewState**)

**引数:**

**NewState:** 以下から ACK モードを設定します。

- **ENABLE** : 許可
- **DISABLE** : 禁止

**機能:**

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

**戻り値:**

- **SUCCESS** 成功
- **ERROR** エラー

#### 4.2.3.19 CEC\_SetNoiseCancellation

ノイズキャンセルモードの設定

**関数のプロトタイプ宣言:**

Result

CEC\_SetNoiseCancellation(CEC\_LowCancellation **LowCancellation**,  
CEC\_HighCancellation **HighCancellation**)

**引数:**

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

- **CEC\_LOW\_CANCELLATION\_1**: 1 cycle
- **CEC\_LOW\_CANCELLATION\_2**: 2 cycles
- **CEC\_LOW\_CANCELLATION\_3**: 3 cycles
- **CEC\_LOW\_CANCELLATION\_4**: 4 cycles

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

- **CEC\_HIGH\_CANCELLATION\_1**: 1 cycle
- **CEC\_HIGH\_CANCELLATION\_2**: 2 cycles
- **CEC\_HIGH\_CANCELLATION\_3**: 3 cycles
- **CEC\_HIGH\_CANCELLATION\_4**: 4 cycles

**機能:**

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

**戻り値:**

- **SUCCESS** 成功
- **ERROR** エラー

## 4.2.3.20 CEC\_SetCycleConfig

周期違反検出時間の設定

**関数のプロトタイプ宣言:**

Result

CEC\_SetCycleConfig(CEC\_CycleMin **CycleMin**,  
CEC\_CycleMax **CycleMax**)

**引数:**

**CycleMin**: 以下から最少周期違反検出時間を選択します。

- **CEC\_CYCLE\_MIN\_0**: 2.05ms
- **CEC\_CYCLE\_MIN\_1**: 2.05ms+1cycle
- **CEC\_CYCLE\_MIN\_2**: 2.05ms+2cycles
- **CEC\_CYCLE\_MIN\_3**: 2.05ms+3cycles
- **CEC\_CYCLE\_MIN\_4**: 2.05ms-1cycles
- **CEC\_CYCLE\_MIN\_5**: 2.05ms-2cycles
- **CEC\_CYCLE\_MIN\_6**: 2.05ms-3cycles
- **CEC\_CYCLE\_MIN\_7**: 2.05ms-4cycles

**CycleMax**: 以下から最大周波数違反検出時間を選択します。

- **CEC\_CYCLE\_MAX\_0**: 2.75ms
- **CEC\_CYCLE\_MAX\_1**: 2.75ms+1cycles
- **CEC\_CYCLE\_MAX\_2**: 2.75ms+2cycles
- **CEC\_CYCLE\_MAX\_3**: 2.75ms+3cycles
- **CEC\_CYCLE\_MAX\_4**: 2.75ms-1cycles
- **CEC\_CYCLE\_MAX\_5**: 2.75ms-2cycles
- **CEC\_CYCLE\_MAX\_6**: 2.75ms-3cycles
- **CEC\_CYCLE\_MAX\_7**: 2.75ms-4cycles

**機能:**

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

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

**戻り値:**

- **SUCCESS** 成功
- **ERROR** エラー

## 4.2.3.21 CEC\_SetDataValidTime

データ 0/1 判別タイミングの設定

関数のプロトタイプ宣言:

Result

CEC\_SetDataValidTime(CEC\_ValidTime **ValidTime**)

引数:

**ValidTime**: 以下から、データ 0/1 判別タイミングを選択します。

- **CEC\_VALID\_TIME\_0**: 1.05ms
- **CEC\_VALID\_TIME\_1**: 1.05ms+2cycles
- **CEC\_VALID\_TIME\_2**: 1.05ms+4cycles
- **CEC\_VALID\_TIME\_3**: 1.05ms+6cycles
- **CEC\_VALID\_TIME\_4**: 1.05ms-2cycles
- **CEC\_VALID\_TIME\_5**: 1.05ms-4cycles
- **CEC\_VALID\_TIME\_6**: 1.05ms-6cycles

機能:

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

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

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

## 4.2.3.22 CEC\_SetTimeOutMode

タイムアウト判定時間の設定

関数のプロトタイプ宣言:

Result

CEC\_SetTimeOutMode(CEC\_TimeOut **TimeOut**)

引数:

**TimeOut**: 以下から、タイムアウト判定時間を選択します。

- **CEC\_TIME\_OUT\_1\_BIT**: 1 bit cycle
- **CEC\_TIME\_OUT\_2\_BIT**: 2 bit cycle
- **CEC\_TIME\_OUT\_3\_BIT**: 3 bit cycle

機能:

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

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

## 4.2.3.23 CEC\_SetRxErrINTSuspend

受信エラー割り込み保留設定

関数のプロトタイプ宣言:

Result

CEC\_SetRxErrINTSuspend(FunctionalState **NewState**)

引数:

**NewState**: 以下から、受信エラー割り込み保留の有効/無効を選択します。

- **ENABLE**: 受信エラー割り込みを有効にする
- **DISABLE**: 受信エラー割り込みを無効にする

機能:

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

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

## 4.2.3.24 CEC\_SetSnoopMode

ロジカルアドレス不一致時に、データ受信動作を行うかどうかを設定

関数のプロトタイプ宣言:

Result

CEC\_SetSnoopMode(FunctionalState **NewState**)

引数:

**NewState**: スヌープモード有効 / 無効を設定します。

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

機能:

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

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

## 4.2.3.25 CEC\_SetRxDetectWaveConfig

波形エラー検出範囲の設定

関数のプロトタイプ宣言:

## Result

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

## 引数:

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

- CEC\_LOGICAL\_1\_RISING\_TIME\_MIN\_0: 0.4ms
- CEC\_LOGICAL\_1\_RISING\_TIME\_MIN\_1: 0.4ms-1cycle
- CEC\_LOGICAL\_1\_RISING\_TIME\_MIN\_2: 0.4ms-2cycles
- CEC\_LOGICAL\_1\_RISING\_TIME\_MIN\_3: 0.4ms-3cycles
- CEC\_LOGICAL\_1\_RISING\_TIME\_MIN\_4: 0.4ms-4cycles
- CEC\_LOGICAL\_1\_RISING\_TIME\_MIN\_5: 0.4ms-5cycles
- CEC\_LOGICAL\_1\_RISING\_TIME\_MIN\_6: 0.4ms-6cycles
- CEC\_LOGICAL\_1\_RISING\_TIME\_MIN\_7: 0.4ms-7cycles

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

- CEC\_LOGICAL\_1\_RISING\_TIME\_MAX\_0: 0.8ms
- CEC\_LOGICAL\_1\_RISING\_TIME\_MAX\_1: 0.8ms+1cycle
- CEC\_LOGICAL\_1\_RISING\_TIME\_MAX\_2: 0.8ms+2cycles
- CEC\_LOGICAL\_1\_RISING\_TIME\_MAX\_3: 0.8ms+3cycles
- CEC\_LOGICAL\_1\_RISING\_TIME\_MAX\_4: 0.8ms+4cycles
- CEC\_LOGICAL\_1\_RISING\_TIME\_MAX\_5: 0.8ms+5cycles
- CEC\_LOGICAL\_1\_RISING\_TIME\_MAX\_6: 0.8ms+6cycles
- CEC\_LOGICAL\_1\_RISING\_TIME\_MAX\_7: 0.8ms+7cycles

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

- CEC\_LOGICAL\_0\_RISING\_TIME\_MIN\_0: 1.3ms
- CEC\_LOGICAL\_0\_RISING\_TIME\_MIN\_1: 1.3ms -1cycle
- CEC\_LOGICAL\_0\_RISING\_TIME\_MIN\_2: 1.3ms -2cycles
- CEC\_LOGICAL\_0\_RISING\_TIME\_MIN\_3: 1.3ms -3cycles
- CEC\_LOGICAL\_0\_RISING\_TIME\_MIN\_4: 1.3ms -4cycles
- CEC\_LOGICAL\_0\_RISING\_TIME\_MIN\_5: 1.3ms -5cycles
- CEC\_LOGICAL\_0\_RISING\_TIME\_MIN\_6: 1.3ms -6cycles
- CEC\_LOGICAL\_0\_RISING\_TIME\_MIN\_7: 1.3ms -7cycles

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

- CEC\_LOGICAL\_0\_RISING\_TIME\_MAX\_0: 1.7ms
- CEC\_LOGICAL\_0\_RISING\_TIME\_MAX\_1: 1.7ms +1cycle
- CEC\_LOGICAL\_0\_RISING\_TIME\_MAX\_2: 1.7ms +2cycles
- CEC\_LOGICAL\_0\_RISING\_TIME\_MAX\_3: 1.7ms +3cycles
- CEC\_LOGICAL\_0\_RISING\_TIME\_MAX\_4: 1.7ms +4cycles
- CEC\_LOGICAL\_0\_RISING\_TIME\_MAX\_5: 1.7ms +5cycles
- CEC\_LOGICAL\_0\_RISING\_TIME\_MAX\_6: 1.7ms +6cycles
- CEC\_LOGICAL\_0\_RISING\_TIME\_MAX\_7: 1.7ms +7cycles

## 機能:

受信波形が設定された波形エラー検出範囲外の場合に、エラー検出をするかどうかを設定します。検出時間は、**Logical1RisingTimeMin**、**Logical1RisingTimeMax**、

**Logical0RisingTimeMin**、**Logical0RisingTimeMax** で設定します。

詳細はデータシートの CEC 章「(10) 波形エラー検出」を参照ください。

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

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

## 4.2.3.26 CEC\_SetRxStartBitWaveConfig

スタートビット検出時の立ち上がりタイミングを設定

関数のプロトタイプ宣言:

Result

```
CEC_SetRxStartBitWaveConfig(  
    CEC_StartBitRisingTimeMin RisingTimeMin,  
    CEC_StartBitRisingTimeMax RisingTimeMax,  
    CEC_StartBitCycleMin CycleMin,  
    CEC_StartBitCycleMax CycleMax)
```

引数:

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

- **CEC\_START\_BIT\_RISING\_TIME\_MIN\_0**: 3.5ms
- **CEC\_START\_BIT\_RISING\_TIME\_MIN\_1**: 3.5ms-1cycle
- **CEC\_START\_BIT\_RISING\_TIME\_MIN\_2**: 3.5ms-2cycles
- **CEC\_START\_BIT\_RISING\_TIME\_MIN\_3**: 3.5ms-3cycles
- **CEC\_START\_BIT\_RISING\_TIME\_MIN\_4**: 3.5ms-4cycles
- **CEC\_START\_BIT\_RISING\_TIME\_MIN\_5**: 3.5ms-5cycles
- **CEC\_START\_BIT\_RISING\_TIME\_MIN\_6**: 3.5ms-6cycles
- **CEC\_START\_BIT\_RISING\_TIME\_MIN\_7**: 3.5ms-7cycles

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

- **CEC\_START\_BIT\_RISING\_TIME\_MAX\_0**: 3.9ms
- **CEC\_START\_BIT\_RISING\_TIME\_MAX\_1**: 3.9ms+1cycle
- **CEC\_START\_BIT\_RISING\_TIME\_MAX\_2**: 3.9ms+2cycles
- **CEC\_START\_BIT\_RISING\_TIME\_MAX\_3**: 3.9ms+3cycles
- **CEC\_START\_BIT\_RISING\_TIME\_MAX\_4**: 3.9ms+4cycles
- **CEC\_START\_BIT\_RISING\_TIME\_MAX\_5**: 3.9ms+5cycles
- **CEC\_START\_BIT\_RISING\_TIME\_MAX\_6**: 3.9ms+6cycles
- **CEC\_START\_BIT\_RISING\_TIME\_MAX\_7**: 3.9ms+7cycles

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

- **CEC\_START\_BIT\_CYCLE\_MIN\_0**: 4.3ms
- **CEC\_START\_BIT\_CYCLE\_MIN\_1**: 4.3ms-1cycle
- **CEC\_START\_BIT\_CYCLE\_MIN\_2**: 4.3ms -2cycles
- **CEC\_START\_BIT\_CYCLE\_MIN\_3**: 4.3ms -3cycles
- **CEC\_START\_BIT\_CYCLE\_MIN\_4**: 4.3ms -4cycles
- **CEC\_START\_BIT\_CYCLE\_MIN\_5**: 4.3ms -5cycles
- **CEC\_START\_BIT\_CYCLE\_MIN\_6**: 4.3ms -6cycles
- **CEC\_START\_BIT\_CYCLE\_MIN\_7**: 4.3ms -7cycles

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

- **CEC\_START\_BIT\_CYCLE\_MAX\_0**: 4.7ms
- **CEC\_START\_BIT\_CYCLE\_MAX\_1**: 4.7ms+1cycle
- **CEC\_START\_BIT\_CYCLE\_MAX\_2**: 4.7ms +2cycles
- **CEC\_START\_BIT\_CYCLE\_MAX\_3**: 4.7ms +3cycles
- **CEC\_START\_BIT\_CYCLE\_MAX\_4**: 4.7ms +4cycles
- **CEC\_START\_BIT\_CYCLE\_MAX\_5**: 4.7ms +5cycles



- **CEC\_START\_BIT\_CYCLE\_MAX\_6**: 4.7ms +6cycles
- **CEC\_START\_BIT\_CYCLE\_MAX\_7**: 4.7ms +7cycles

**機能:**

立ち上がりのタイミングとスタートビット検出条件を設定します。

**RisingTimeMin** : 立ち上がりタイミングの最小値

**RisingTimeMax** : 立ち上がりタイミングの最大値

**CycleMin** : スタートビット検出時の周期の最小値

**CycleMax** : スタートビット検出時の周期の最大値

詳細はデータシートの CEC 章「(9) スタートビット検出」を参照ください。

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

**戻り値:**

- **SUCCESS** 成功
- **ERROR** エラー

#### 4.2.3.27 CEC\_SetRxWaveErrDetect

波形エラー検出の有効/無効設定、および波形エラー割り込みの発生の設定

**関数のプロトタイプ宣言:**

Result

CEC\_SetRxWaveErrDetect(FunctionalState **NewState**)

**引数:**

**NewState**: 以下から、波形エラー検出の有効/無効を選択します。

- **ENABLE**: 波形エラー検出許可
- **DISABLE**: 波形エラー検出禁止

**機能:**

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

**戻り値:**

- **SUCCESS** 成功
- **ERROR** エラー

#### 4.2.3.28 CEC\_SetTxWaveConfig

送信波形の設定

**関数のプロトタイプ宣言:**

Result

CEC\_SetTxWaveConfig(CEC\_TxDataBitCycle **DataBitCycle** ,  
CEC\_TxDataBitRisingTime **DataBitRisingTime** ,  
CEC\_TxStartBitCycle **StartBitCycle** ,  
CEC\_TxStartBitRisingTime **StartBitRisingTime**)

**引数:**

**DataBitCycle**: 以下から、データビットの周期を選択します。

- **CEC\_TX\_DATA\_BIT\_CYCLE\_0**: RV (参考値: 約 2.4ms)
- **CEC\_TX\_DATA\_BIT\_CYCLE\_1**: RV-1cycle

- CEC\_TX\_DATA\_BIT\_CYCLE\_2: RV-2cycles
- CEC\_TX\_DATA\_BIT\_CYCLE\_3: RV-3cycles
- CEC\_TX\_DATA\_BIT\_CYCLE\_4: RV-4cycles
- CEC\_TX\_DATA\_BIT\_CYCLE\_5: RV-5cycles
- CEC\_TX\_DATA\_BIT\_CYCLE\_6: RV-6cycles
- CEC\_TX\_DATA\_BIT\_CYCLE\_7: RV-7cycles
- CEC\_TX\_DATA\_BIT\_CYCLE\_8: RV-8cycles
- CEC\_TX\_DATA\_BIT\_CYCLE\_9: RV-9cycles
- CEC\_TX\_DATA\_BIT\_CYCLE\_10: RV-10cycles
- CEC\_TX\_DATA\_BIT\_CYCLE\_11: RV-11cycles
- CEC\_TX\_DATA\_BIT\_CYCLE\_12: RV-12cycles
- CEC\_TX\_DATA\_BIT\_CYCLE\_13: RV-13cycles
- CEC\_TX\_DATA\_BIT\_CYCLE\_14: RV-14cycles
- CEC\_TX\_DATA\_BIT\_CYCLE\_15: RV-15cycles

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

- CEC\_TX\_DATA\_BIT\_RISING\_TIME\_0: RV (logical “1”: 約 0.6 ms, logical “0”: 約 1.5 ms)
- CEC\_TX\_DATA\_BIT\_RISING\_TIME\_1: RV-1cycle
- CEC\_TX\_DATA\_BIT\_RISING\_TIME\_2: RV-2cycles
- CEC\_TX\_DATA\_BIT\_RISING\_TIME\_3: RV-3cycles

**StartBitCycle:** 以下から、スタートビットの周期を選択します。

- CEC\_TX\_START\_BIT\_CYCLE\_0: RV (約 4.5ms)
- CEC\_TX\_START\_BIT\_CYCLE\_1: RV-1cycle
- CEC\_TX\_START\_BIT\_CYCLE\_2: RV-2cycles
- CEC\_TX\_START\_BIT\_CYCLE\_3: RV-3cycles
- CEC\_TX\_START\_BIT\_CYCLE\_4: RV-4cycles
- CEC\_TX\_START\_BIT\_CYCLE\_5: RV-5cycles
- CEC\_TX\_START\_BIT\_CYCLE\_6: RV-6cycles
- CEC\_TX\_START\_BIT\_CYCLE\_7: RV-7cycles

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

- CEC\_TX\_START\_BIT\_RISING\_TIME\_0: RV (約 3.7ms)
- CEC\_TX\_START\_BIT\_RISING\_TIME\_1: RV-1cycle
- CEC\_TX\_START\_BIT\_RISING\_TIME\_2: RV-2cycles
- CEC\_TX\_START\_BIT\_RISING\_TIME\_3: RV-3cycles
- CEC\_TX\_START\_BIT\_RISING\_TIME\_4: RV-4cycles
- CEC\_TX\_START\_BIT\_RISING\_TIME\_5: RV-5cycles
- CEC\_TX\_START\_BIT\_RISING\_TIME\_6: RV-6cycles
- CEC\_TX\_START\_BIT\_RISING\_TIME\_7: RV-7cycles

## 機能:

送信波形のデータビット/スタートビットの周期と立ち上がりタイミングを設定します。

**DataBitCycle**, **DataBitRisingTime**, **StartBitCycle**, **StartBitRisingTime** で、立ち上がりと周期の最も早いタイミングから標準値の間で設定をします。詳細はデータシートの CEC 章「(3) 送信波形調整」を参照ください。

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

## 戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

### 4.2.3.29 CEC\_SetTxBroadcast

ブロードキャスト送信の設定

**関数のプロトタイプ宣言:**

Result

CEC\_SetTxBroadcast(FunctionalState **NewState**)**引数:****NewState:** 以下から、ブロードキャスト送信モードの有効/無効を選択します。

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

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

- **SUCCESS** 成功
- **ERROR** エラー

#### 4.2.3.30 CEC\_SetBusFreeTime

送信開始前に確認するバスフリー時間の設定

**関数のプロトタイプ宣言:**

Result

CEC\_SetBusFreeTime (CEC\_BusFree **BusFree**)**引数:****BusFree:** 以下から、バスフリー待ち時間を選択します。

- **CEC\_BUS\_FREE\_1\_BIT:** 1 bit cycle
- **CEC\_BUS\_FREE\_2\_BIT:** 2 bit cycles
- **CEC\_BUS\_FREE\_3\_BIT:** 3 bit cycles
- **CEC\_BUS\_FREE\_4\_BIT:** 4 bit cycles
- **CEC\_BUS\_FREE\_5\_BIT:** 5 bit cycles
- **CEC\_BUS\_FREE\_6\_BIT:** 6 bit cycles
- **CEC\_BUS\_FREE\_7\_BIT:** 7 bit cycles
- **CEC\_BUS\_FREE\_8\_BIT:** 8 bit cycles
- **CEC\_BUS\_FREE\_9\_BIT:** 9 bit cycles
- **CEC\_BUS\_FREE\_10\_BIT:** 10 bit cycles
- **CEC\_BUS\_FREE\_11\_BIT:** 11 bit cycles
- **CEC\_BUS\_FREE\_12\_BIT:** 12 bit cycles
- **CEC\_BUS\_FREE\_13\_BIT:** 13 bit cycles
- **CEC\_BUS\_FREE\_14\_BIT:** 14 bit cycles
- **CEC\_BUS\_FREE\_15\_BIT:** 15 bit cycles
- **CEC\_BUS\_FREE\_16\_BIT:** 16 bit cycles

**機能:**

送信開始前に確認するバスフリー時間の設定を行います。1 サイクルから 16 サイクルの間で設定します。バスフリー状態の確認は、最終ビットから開始します。

**CEC\_BUS\_FREE\_1\_BIT** がバスフリーの場合、送信開始します。詳細はデータシートの CEC 章「(3) バスフリー待ち時間」を参照ください。送信中の場合、本関数の戻り値は **ERROR** となります。

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

#### 4.2.3.31 CEC\_SetRxStartBitDetect

受信時のスタートビット割り込み設定

関数のプロトタイプ宣言:

Result

CEC\_SetRxStartBitDetect(FunctionalState **NewState**)

引数:

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

- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:

受信時のスタートビット割り込み許可/禁止を選択します。

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

戻り値:

- **SUCCESS** 成功
- **ERROR** エラー

#### 4.2.3.32 CEC\_SetSamplingClk

サンプリングクロックの選択

関数のプロトタイプ宣言:

void

void CEC\_SetSamplingClk(CEC\_SamplingClockSrc **ClkSrc**)

引数:

**ClkSrc**: 以下からサンプリングクロックを選択します。

- **CEC\_SAMPLING\_CLK\_SRC\_LOW\_SPEED**: 低速クロック (fs)
- **CEC\_SAMPLING\_CLK\_SRC\_TBXOUT**: タイマ出力 (TBxOUT)

機能:

サンプリングクロックを選択します。

戻り値:

なし

### 4.2.4 データ構造

#### 4.2.4.1 CEC\_DataTypeDef

メンバ:

uint8\_t

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

CEC\_ACKState

**ACKBit:** 受信 ACKビットです。

- **CEC\_ACK:** ACKビットが "1"
- **CEC\_NO\_ACK:** ACKビットが "0"

CEC\_EOMBit

**EOMBit:** 受信 EOMビットです。

- **CEC\_EOM:** EOMビットが "1"
- **CEC\_NO\_EOM:** EOMビットが "0"

#### 4.2.4.2 CEC\_AddrListTypeDef

メンバ:

uint8\_t

**AddrNumber:** ロジカルアドレス番号

CEC\_LogicalAddr

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

CEC\_TV, CEC\_RECORDING\_DEVICE\_1, CEC\_RECORDING\_DEVICE\_2,  
CEC\_STB\_1, CEC\_DVD\_1, CEC\_AUDIO\_SYSTEM, CEC\_STB\_2,  
CEC\_STB\_3, CEC\_DVD\_2, CEC\_RECORDING\_DEVICE\_3, CEC\_FREE\_USE,  
CEC\_BROADCAST

## 5. EXB

### 5.1 概要

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

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

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

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

/Libraries/TX04\_Periph\_Driver/src/tmpm462\_exb.c

/Libraries/TX04\_Periph\_Driver/inc/tmpm462\_exb.h

### 5.2 API 関数

#### 5.2.1 関数一覧

- ◆ void EXB\_SetBusMode(uint8\_t **BusMode**);
- ◆ void EXB\_SetBusCycleExtension(uint8\_t **Cycle**);
- ◆ void EXB\_Enable(uint8\_t **ChipSelect**);
- ◆ void EXB\_Disable(uint8\_t **ChipSelect**);
- ◆ void EXB\_Init(uint8\_t **ChipSelect**, EXB\_InitTypeDef\* **InitStruct**);
- ◆ Result EXB\_SetClkOutputDivision(uint8\_t **ClkDiv**);
- ◆ void EXB\_EnableClkOutput(void);
- ◆ void EXB\_DisableClkOutput(void);
- ◆ FunctionalState EXB\_GetClkOutputState(void);

#### 5.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています:

- 1) EXB バスモード、バスサイクルウェイト拡張、データバス幅、チップセクタを元にした外部バスサイクルの設定  
EXB\_SetBusMode(), EXB\_SetBusCycleExtension(), EXB\_Init()
- 2) 許可/禁止制御  
EXB\_Enable(), EXB\_Disable()
- 2) クロック出力の設定  
EXB\_SetClkOutputDivision(), EXB\_EnableClkOutput(),  
EXB\_DisableClkOutput(), EXB\_GetClkOutputState()

## 5.2.3 関数仕様

### 5.2.3.1 EXB\_SetBusMode

EXB 外部バスモードの設定

関数のプロトタイプ宣言:

```
void  
EXB_SetBusMode(uint8_t BusMode)
```

引数:

**BusMode**: 以下から EXB 外部バスモードを選択します。

- **EXB\_BUS\_MULTIPLEX**: マルチプレクスバスモード
- **EXB\_BUS\_SEPARATE**: セパレートバスモード

機能:

外部バスモードを設定します。

戻り値:

なし

### 5.2.3.2 EXB\_SetBusCycleExtension

バスサイクルウェイト拡張の設定

関数のプロトタイプ宣言:

```
void  
EXB_SetBusCycleExtension(uint8_t Cycle)
```

引数:

**Cycle**: バスサイクルウェイト拡張を指定します。

- **EXB\_CYCLE\_NONE**: 拡張なし
- **EXB\_CYCLE\_DOUBLE**: 2 倍
- **EXB\_CYCLE\_QUADRUPLE**: 4 倍

機能:

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

戻り値:

なし

### 5.2.3.3 EXB\_Enable

チップセレクトの許可

関数のプロトタイプ宣言:

```
void  
EXB_Enable(uint8_t ChipSelect)
```

引数:

**ChipSelect**: チップセレクトを選択します。

- **EXB\_CS0**: CS0
- **EXB\_CS1**: CS1
- **EXB\_CS2**: CS2
- **EXB\_CS3**: CS3

**機能:**

チップセレクトを許可します。

**戻り値:**

なし

## 5.2.3.4 EXB\_Disable

チップセレクトの禁止

**関数のプロトタイプ宣言:**

void

EXB\_Disable(uint8\_t **ChipSelect**)

**引数:**

**ChipSelect**: チップセレクトを選択します。

- **EXB\_CS0**: CS0
- **EXB\_CS1**: CS1
- **EXB\_CS2**: CS2
- **EXB\_CS3**: CS3

**機能:**

チップセレクトを禁止します。

**戻り値:**

なし

## 5.2.3.5 EXB\_Init

チップセレクト設定の初期化

**関数のプロトタイプ宣言:**

void

EXB\_Init (uint8\_t **ChipSelect**,  
EXB\_InitTypeDef\* **InitStruct**)

**引数:**

**ChipSelect**: チップセレクトを選択します。

- **EXB\_CS0**: CS0
- **EXB\_CS1**: CS1
- **EXB\_CS2**: CS2
- **EXB\_CS3**: CS3

**InitStruct**: 下記設定を行います。

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

**機能:**



チップセレクト設定を初期化します。

戻り値:

なし

## 5.2.3.6 EXB\_SetClkOutputDivision

クロック分周の設定

関数のプロトタイプ宣言:

Result

EXB\_SetClkOutputDivision (uint8\_t **ClkDiv**)

引数:

**ClkDiv**: 以下からクロック分周値を選択します。

- EXB\_CLK\_DIVISION\_FSYS\_2: fsys/2
- EXB\_CLK\_DIVISION\_FSYS\_4: fsys/4
- EXB\_CLK\_DIVISION\_FSYS\_8: fsys/8

機能:

クロック分周を設定します。

\*補足:

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

戻り値:

- **SUCCESS**: 成功
- **ERROR**: 失敗

## 5.2.3.7 EXB\_EnableClkOutput

外部クロックの許可

関数のプロトタイプ宣言:

void

EXB\_EnableClkOutput(void)

引数:

なし

機能:

外部クロックを許可します。

戻り値:

なし

## 5.2.3.8 EXB\_DisableClkOutput

外部クロックの禁止

**関数のプロトタイプ宣言:**

void  
EXB\_DisableClkOutput(void)

**引数:**

なし

**機能:**

外部クロックを禁止します。

**戻り値:**

なし

## 5.2.3.9 EXB\_GetClkOutputState

外部クロック制御状態の取得

**関数のプロトタイプ宣言:**

void  
EXB\_GetClkOutputState(void)

**引数:**

なし

**機能:**

外部クロックの制御状態を取得します。

**戻り値:**

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

## 5.2.4 データ構造

### 5.2.4.1 EXB\_InitTypeDef

**メンバ:**

uint8\_t

**AddrSpaceSize:** アドレス空間を設定します。

- **EXB\_16M\_BYTE:** アドレス空間 16Mbyte
- **EXB\_8M\_BYTE:** アドレス空間 8Mbyte
- **EXB\_4M\_BYTE:** アドレス空間 4Mbyte
- **EXB\_2M\_BYTE:** アドレス空間 2Mbyte
- **EXB\_1M\_BYTE:** アドレス空間 1Mbyte
- **EXB\_512K\_BYTE:** アドレス空間 512Kbyte
- **EXB\_256K\_BYTE:** アドレス空間 256Kbyte
- **EXB\_128K\_BYTE:** アドレス空間 128Kbyte
- **EXB\_64K\_BYTE:** アドレス空間 64Kbyte

uint8\_t

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

uint8\_t

**BusWidth**: データバス幅を設定します。

- **EXB\_BUS\_WIDTH\_BIT\_8**: データバス幅 8bit,
- **EXB\_BUS\_WIDTH\_BIT\_16**: データバス幅 16bit.

EXB\_CyclesTypeDef

**Cycles**: 外部バス周期を設定します。

**InternalWait**, **ReadSetupCycle**, **WriteSetupCycle**, **ALEWaitCycle** (マルチプレクスバスモードのみ), **ReadRecoveryCycle**, **WriteRecoveryCycle**, **ChipSelectRecoveryCycle**. (詳細は “EXB\_CyclesTypeDef” を参照)

uint8\_t

**WaitSignal**: ウェイト信号を選択します。

- **EXB\_WAIT\_SIGNAL\_LOW**: "Low"アクティブ
- **EXB\_WAIT\_SIGNAL\_HIGH**: "High"アクティブ

uint8\_t

**WaitFunction**: ウェイト機能を選択します。

- **EXB\_WAIT\_FUNCTION\_INT**: 内部ウェイト
- **EXB\_WAIT\_FUNCTION\_EXT**: 外部ウェイト

## 5.2.4.2 EXB\_CyclesType Def

メンバ:

uint8\_t

**InternalWait**: 内部ウェイト(自動挿入)を設定します。

- **EXB\_INTERNAL\_WAIT\_0**: 0 wait
- **EXB\_INTERNAL\_WAIT\_1**: 1 wait
- **EXB\_INTERNAL\_WAIT\_2**: 2 wait
- **EXB\_INTERNAL\_WAIT\_3**: 3 wait
- **EXB\_INTERNAL\_WAIT\_4**: 4 wait
- **EXB\_INTERNAL\_WAIT\_5**: 5 wait
- **EXB\_INTERNAL\_WAIT\_6**: 6 wait
- **EXB\_INTERNAL\_WAIT\_7**: 7 wait
- **EXB\_INTERNAL\_WAIT\_8**: 8 wait
- **EXB\_INTERNAL\_WAIT\_9**: 9 wait
- **EXB\_INTERNAL\_WAIT\_10**: 10 wait
- **EXB\_INTERNAL\_WAIT\_11**: 11 wait
- **EXB\_INTERNAL\_WAIT\_12**: 12 wait
- **EXB\_INTERNAL\_WAIT\_13**: 13 wait
- **EXB\_INTERNAL\_WAIT\_14**: 14 wait
- **EXB\_INTERNAL\_WAIT\_15**: 15 wait

uint8\_t

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

- **EXB\_CYCLE\_0**: 0 cycle
- **EXB\_CYCLE\_1**: 1 cycle
- **EXB\_CYCLE\_2**: 2 cycle
- **EXB\_CYCLE\_4**: 4 cycle

uint8\_t

**WriteSetupCycle**: ライト(WRn)セットアップサイクルを設定します。

- **EXB\_CYCLE\_0**: 0 cycle
- **EXB\_CYCLE\_1**: 1 cycle

- **EXB\_CYCLE\_2**: 2 cycle
- **EXB\_CYCLE\_4**: 4 cycle

uint8\_t

**ALEWaitCycle**: ALE ウェイトサイクル(マルチプレクスバスモード時)を選択します。

- **EXB\_CYCLE\_0**: 0 cycle
- **EXB\_CYCLE\_1**: 1 cycle
- **EXB\_CYCLE\_2**: 2 cycle
- **EXB\_CYCLE\_4**: 4 cycle

uint8\_t

**ReadRecoveryCycle**: リード(RDn)リカバリサイクルを選択します。

- **EXB\_CYCLE\_0**: 0 cycle
- **EXB\_CYCLE\_1**: 1 cycle
- **EXB\_CYCLE\_2**: 2 cycle
- **EXB\_CYCLE\_3**: 3 cycle
- **EXB\_CYCLE\_4**: 4 cycle
- **EXB\_CYCLE\_5**: 5 cycle
- **EXB\_CYCLE\_6**: 6 cycle
- **EXB\_CYCLE\_8**: 8 cycle

uint8\_t

**WriteRecoveryCycle**: ライト(WRn)リカバリサイクルを選択します。

- **EXB\_CYCLE\_0**: 0 cycle
- **EXB\_CYCLE\_1**: 1 cycle
- **EXB\_CYCLE\_2**: 2 cycle
- **EXB\_CYCLE\_3**: 3 cycle
- **EXB\_CYCLE\_4**: 4 cycle
- **EXB\_CYCLE\_5**: 5 cycle
- **EXB\_CYCLE\_6**: 6 cycle
- **EXB\_CYCLE\_8**: 8 cycle

uint8\_t

**ChipSelectRecoveryCycle** : チップセレクト(CSxn)リカバリサイクルを選択します。

- **EXB\_CYCLE\_0**: 0 cycle
- **EXB\_CYCLE\_1**: 1 cycle
- **EXB\_CYCLE\_2**: 2 cycle
- **EXB\_CYCLE\_4**: 4 cycle

## 6. CG

### 6.1 概要

本 CG API は TMPM462x CG における以下の機能を提供します。

- 高速/低速発振器、PLL(逡倍回路)の設定
- クロックギア、プリスケールクロック、PLL、発振器の設定
- ウォームアップタイマの設定と結果の読み出し
- 低消費電力モードの設定
- 動作モードの変更 (ノーマルモード、低速モード、低消費電力モード)
- スタンバイモードに関する割り込みの設定

本ドライバは、以下のファイルで構成されています。

/Libraries/TX04\_Periph\_Driver/src/tmpm462\_cg.c

/Libraries/TX04\_Periph\_Driver/inc/tmpm462\_cg.h

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

**EHCLKIN** : X1 端子より入力されるクロック

**EHOSC** : 外部高速発振器から出力されるクロック

**ELOSC** : 外部低速発振器から出力されるクロック

**IHOSC** : 内部高速発振器から出力されるクロック (SYS 用)

**IHOSC2** : 内部高速発振器から出力されるクロック (OFD 基準クロック用)

**FOSCHI** : CGOSCCR<HOSCON>で選択したクロック

**fosc** : CGOSCCR<OSCSEL>により選択されたクロック

**fpll** : PLLにより逡倍されたクロック

**fc** : CGPLLSEL<PLLSEL>により選択されたクロック (高速クロック)

**fgear** : CGSYSCR<GEAR[2:0]>により選択されたクロック

**fsys** : CGSYSCR<GEAR[2:0]>により選択されたクロック (システムクロック)

**fperiph** : CGSYSCR<FPSEL[2:0]>により選択されたクロック

**ΦT0** : CGSYSCR<PRCK[2:0]>により選択されたクロック (プリスケールクロック)

### 6.2 API 関数

#### 6.2.1 関数一覧

- ◆ void CG\_SetFgearLevel(CG\_DivideLevel *DivideFgearFromFc*)
- ◆ CG\_DivideLevel CG\_GetFgearLevel(void)
- ◆ void CG\_SetPhiT0Src(CG\_PhiT0Src *PhiT0Src*)
- ◆ CG\_PhiT0Src CG\_GetPhiT0Src(void)
- ◆ Result CG\_SetPhiT0Level(CG\_DivideLevel *DividePhiT0FromFc*)
- ◆ CG\_DivideLevel CG\_GetPhiT0Level(void)
- ◆ void CG\_SetSCOUTSrc(CG\_SCOUTSrc *Source*)
- ◆ CG\_SCOUTSrc CG\_GetSCOUTSrc(void)
- ◆ void CG\_SetWarmUpTime(CG\_WarmUpSrc *Source*, uint16\_t *Time*)
- ◆ void CG\_StartWarmUp(void)
- ◆ WorkState CG\_GetWarmUpState(void)
- ◆ Result CG\_SetFPLLValue(CG\_FpllValue *NewValue*)

- ◆ CG\_FppllValue CG\_GetFPLLValue(void)
- ◆ Result CG\_SetPLL(FunctionalState **NewState**)
- ◆ FunctionalState CG\_GetPLLState(void)
- ◆ Result CG\_SetFosc(CG\_FoscSrc Source, FunctionalState **NewState**)
- ◆ void CG\_SetFoscSrc(CG\_FoscSrc **Source**)
- ◆ CG\_FoscSrc CG\_GetFoscSrc(void)
- ◆ FunctionalState CG\_GetFoscState(CG\_FoscSrc **Source**)
- ◆ void CG\_SetSTBYMode(CG\_STBYMode **Mode**)
- ◆ CG\_STBYMode CG\_GetSTBYMode(void)
- ◆ void CG\_SetPortKeepInStop2Mode(FunctionalState **NewState**)
- ◆ FunctionalState CG\_GetPortKeepInStop2Mode(void)
- ◆ Result CG\_SetFcSrc(CG\_FcSrc **Source**)
- ◆ CG\_FcSrc CG\_GetFcSrc(void)
- ◆ void CG\_SetProtectCtrl(FunctionalState **NewState**)
- ◆ void CG\_SetSTBYReleaseINTSrc(CG\_INTSrc **INTSource**,  
CG\_INTActiveState **ActiveState**,  
FunctionalState **NewState**)
- ◆ CG\_INTActiveState CG\_GetSTBYReleaseINTState(CG\_INTSrc **INTSource**)
- ◆ void CG\_ClearINTReq(CG\_INTSrc **INTSource**)
- ◆ CG\_NMIFactor CG\_GetNMIFlag(void)
- ◆ FunctionalState CG\_GetIOSCFlashFlag(void)
- ◆ CG\_ResetFlag CG\_GetResetFlag(void)
- ◆ void CG\_SetADCClkSupply(FunctionalState **NewState**)
- ◆ void CG\_SetInternalOscForOFD(FunctionalState **NewState**)
- ◆ void CG\_SetFcPeriphA(uint32\_t **Periph**, FunctionalState **NewState**)
- ◆ void CG\_SetFcPeriphB(uint32\_t **Periph**, FunctionalState **NewState**)
- ◆ void CG\_SetFs(FunctionalState **NewState**)

## 6.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。

- 1) クロックの選択:  
CG\_SetFgearLevel(), CG\_GetFgearLevel(), CG\_SetPhiT0Src(), CG\_GetPhiT0Src(),  
CG\_SetPhiT0Level(), CG\_GetPhiT0Level(), CG\_SetSCOUTSrc(),  
CG\_GetSCOUTSrc(), CG\_SetWarmUpTime(), CG\_StartWarmUp(),  
CG\_GetWarmUpState(), CG\_SetFPLLValue(), CG\_GetFPLLValue(), CG\_SetPLL(),  
CG\_GetPLLState(), CG\_SetFosc(), CG\_SetFoscSrc(), CG\_GetFoscSrc(),  
CG\_GetFoscState(), CG\_SetFcSrc(), CG\_GetFcSrc(), CG\_SetProtectCtrl()
- 2) スタンバイモードの設定:  
CG\_SetSTBYMode(), CG\_GetSTBYMode(), CG\_SetPortKeepInStop2Mode(),  
CG\_GetPortKeepInStop2Mode()
- 3) 割り込みの設定:  
CG\_SetSTBYReleaseINTSrc(), CG\_GetSTBYReleaseINTState(), CG\_ClearINTReq(),  
CG\_GetNMIFlag(), CG\_GetResetFlag()
- 4) 周辺機能へのクロック供給:  
CG\_SetADCClkSupply(), CG\_SetInternalOscForOFD(), CG\_SetFcPeriphA(),  
CG\_SetFcPeriphB(), CG\_GetIOSCFlashFlag(), CG\_SetFs()

## 6.2.3 関数仕様

### 6.2.3.1 CG\_SetFgearLevel

fgear, fc 間の分周レベル設定

関数のプロトタイプ宣言:

void  
CG\_SetFgearLevel(CG\_DivideLevel **DivideFgearFromFc**)

**引数:**

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

- **CG\_DIVIDE\_1:** fgear = fc
- **CG\_DIVIDE\_2:** fgear = fc/2
- **CG\_DIVIDE\_4:** fgear = fc/4
- **CG\_DIVIDE\_8:** fgear = fc/8
- **CG\_DIVIDE\_16:** fgear = fc/16

**機能:**

fgear,fc 間の分周レベルを設定します。

**戻り値:**

なし

## 6.2.3.2 CG\_GetFgearLevel

fgear,fc 間の分周レベルの取得

**関数のプロトタイプ宣言:**

CG\_DivideLevel  
CG\_GetFgearLevel(void)

**引数:**

なし

**機能:**

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

**戻り値:**

fgear, fc 間の分周レベルで、下記のいずれかの値になります。

- **CG\_DIVIDE\_1:** fgear = fc
- **CG\_DIVIDE\_2:** fgear = fc/2
- **CG\_DIVIDE\_4:** fgear = fc/4
- **CG\_DIVIDE\_8:** fgear = fc/8
- **CG\_DIVIDE\_16:** fgear = fc/16
- **CG\_DIVIDE\_UNKNOWN:** 無効なデータ

## 6.2.3.3 CG\_SetPhiT0Src

PhiT0(ΦT0) ,fc 間の PhiT0(ΦT0) ソースの設定

**関数のプロトタイプ宣言:**

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

**引数:**

**PhiT0Src:** 以下から PhiT0 ソースを選択します。

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

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

**機能:**

PhiT0 (ΦT0) ソースを選択します。

**戻り値:**

なし

## 6.2.3.4 CG\_GetPhiT0Src

PhiT0 (ΦT0) ソースの取得

**関数のプロトタイプ宣言:**

CG\_PhiT0Src

CG\_GetPhiT0Src(void)

**引数:**

なし

**機能:**

PhiT0 (ΦT0) ソースを取得します。

**戻り値:**

- **CG\_PHIT0\_SRC\_FGEAR**: fgear が PhiT0 ソース
- **CG\_PHIT0\_SRC\_FC**: fc が PhiT0 ソース

## 6.2.3.5 CG\_SetPhiT0Level

PhiT0 (ΦT0) と fc 間の分周レベルの設定

**関数のプロトタイプ宣言:**

Result

CG\_SetPhiT0Level(CG\_DivideLevel ***DividePhiT0FromFc***)

**引数:**

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

- **CG\_DIVIDE\_1**: ΦT0 = fc
- **CG\_DIVIDE\_2**: ΦT0 = fc/2
- **CG\_DIVIDE\_4**: ΦT0 = fc/4
- **CG\_DIVIDE\_8**: ΦT0 = fc/8
- **CG\_DIVIDE\_16**: ΦT0 = fc/16
- **CG\_DIVIDE\_32**: ΦT0 = fc/32
- **CG\_DIVIDE\_64**: ΦT0 = fc/64
- **CG\_DIVIDE\_128**: ΦT0 = fc/128
- **CG\_DIVIDE\_256**: ΦT0 = fc/256
- **CG\_DIVIDE\_512**: ΦT0 = fc/512

**機能:**

プリスケラクロックの分周レベルを設定します。

**戻り値:**

- **SUCCESS**: 成功



➤ **ERROR:** 失敗

## 6.2.3.6 CG\_GetPhiT0Level

PhiT0( $\Phi T0$ ), 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( $\Phi T0$ ), fc 間の分周レベルを取得します。レジスタから読み出した値が“Reserved”の場合、**CG\_DIVIDE\_UNKNOWN** を返します。

**戻り値:**

PhiT0( $\Phi T0$ ), fc 間の分周レベル:

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

## 6.2.3.7 CG\_SetSCOUTSrc

SCOUT ソースクロック設定

**関数のプロトタイプ宣言:**

void

CG\_SetSCOUTSrc(CG\_SCOUTSrc **Source**)

**引数:**

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

- **CG\_SCOUT\_SRC\_FSYS:** fsys
- **CG\_SCOUT\_SRC\_FS:** fs

**機能:**

SCOUT のソースクロックを設定します。

**戻り値:**

なし

## 6.2.3.8 CG\_GetSCOUTSrc

SCOUT ソースクロック設定の取得

関数のプロトタイプ宣言:

SCOUTSrc

CG\_GetSCOUTSrc(void)

引数:

なし

機能:

SCOUT のソースクロック設定を取得します。

戻り値:

SCOUT のソースクロック:

➤ CG\_SCOUT\_SRC\_FSYS: fsys

➤ CG\_SCOUT\_SRC\_FS: fs

## 6.2.3.9 CG\_SetWarmUpTime

ウォームアップ時間の設定

関数のプロトタイプ宣言:

void

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

引数:

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

➤ CG\_WARM\_UP\_SRC\_OSC\_INT\_HIGH: 内部高速発振

➤ CG\_WARM\_UP\_SRC\_OSC\_EXT\_HIGH: 外部高速発振

➤ CG\_WARM\_UP\_SRC\_OSC\_EXT\_LOW: 外部低速発振

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

機能:

ウォームアップサイクル数の計算式は下記になります。

ウォーミングアップサイクル数 = (ウォーミングアップ時間) / (ウォーミングアップクロック周期)

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

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

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

戻り値:

なし

## 6.2.3.10 CG\_StartWarmUp

ウォームアップ開始

関数のプロトタイプ宣言:

void  
CG\_StartWarmUp(void)

引数:  
なし

機能:  
ウォームアップを開始します。

戻り値:  
なし

## 6.2.3.11 CG\_GetWarmUpState

ウォーミングアップ動作状態 (動作中、完了)の確認

関数のプロトタイプ宣言:  
WorkState  
CG\_GetWarmUpState(void)

引数:  
なし

機能:  
ウォーミングアップ動作状態を確認します。

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

戻り値:  
ウォーミングアップ状態:  
➤ **DONE**: ウォーミングアップ動作終了  
➤ **BUSY**: ウォーミングアップ動作中

## 6.2.3.12 CG\_SetFPLLValue

PLL 通倍値の設定

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

引数:  
**NewValue**: 以下から PLL 通倍値を選択します。  
➤ **CG\_8M\_MUL\_4\_FPLL**: 入力クロック 8MHz, 出力クロック 32MHz (4 通倍)  
➤ **CG\_8M\_MUL\_6\_FPLL**: 入力クロック 8MHz, 出力クロック 48MHz (6 通倍)  
➤ **CG\_8M\_MUL\_8\_FPLL**: 入力クロック 8MHz, 出力クロック 64MHz (8 通倍)  
➤ **CG\_8M\_MUL\_10\_FPLL**: 入力クロック 8MHz, 出力クロック 80MHz (10 通倍)  
➤ **CG\_8M\_MUL\_12\_FPLL**: 入力クロック 8MHz, 出力クロック 96MHz (12 通倍)

- **CG\_10M\_MUL\_4\_FPLL**: 入力クロック 10MHz, 出力クロック 40MHz (4 通倍)
- **CG\_10M\_MUL\_5\_FPLL**: 入力クロック 10MHz, 出力クロック 50MHz (5 通倍)
- **CG\_10M\_MUL\_6\_FPLL**: 入力クロック 10MHz, 出力クロック 60MHz (6 通倍)
- **CG\_10M\_MUL\_10\_FPLL**: 入力クロック 10MHz, 出力クロック 100MHz (10 通倍)
- **CG\_10M\_MUL\_12\_FPLL**: 入力クロック 10MHz, 出力クロック 120MHz (12 通倍)
- **CG\_12M\_MUL\_4\_FPLL**: 入力クロック 12MHz, 出力クロック 48MHz (4 通倍)
- **CG\_12M\_MUL\_10\_FPLL**: 入力クロック 12MHz, 出力クロック 120MHz (10 通倍)
- **CG\_16M\_MUL\_4\_FPLL**: 入力クロック 16MHz, 出力クロック 64MHz (4 通倍)
- **CG\_16M\_MUL\_6\_FPLL**: 入力クロック 16MHz, 出力クロック 96MHz (6 通倍)

**機能:**

PLL 通倍値を設定します。

**戻り値:**

- **SUCCESS**: 成功
- **ERROR**: 失敗

### 6.2.3.13 CG\_GetFPLLValue

PLL 通倍値の取得

**関数のプロトタイプ宣言:**

uint32\_t

CG\_GetFPLLValue(void)

**引数:**

なし

**機能:**

PLL 通倍値を取得します。

取得した値が “Reserved” の場合、**CG\_FPLL\_MULTIPLY\_UNKNOWN** を返却します。

**戻り値:**

PLL 通倍値:

- **CG\_8M\_MUL\_4\_FPLL**: 入力クロック 8MHz, 出力クロック 32MHz (4 通倍)
- **CG\_8M\_MUL\_6\_FPLL**: 入力クロック 8MHz, 出力クロック 48MHz (6 通倍)
- **CG\_8M\_MUL\_8\_FPLL**: 入力クロック 8MHz, 出力クロック 64MHz (8 通倍)
- **CG\_8M\_MUL\_10\_FPLL**: 入力クロック 8MHz, 出力クロック 80MHz (10 通倍)
- **CG\_8M\_MUL\_12\_FPLL**: 入力クロック 8MHz, 出力クロック 96MHz (12 通倍)
- **CG\_10M\_MUL\_4\_FPLL**: 入力クロック 10MHz, 出力クロック 40MHz (4 通倍)
- **CG\_10M\_MUL\_5\_FPLL**: 入力クロック 10MHz, 出力クロック 50MHz (5 通倍)
- **CG\_10M\_MUL\_6\_FPLL**: 入力クロック 10MHz, 出力クロック 60MHz (6 通倍)
- **CG\_10M\_MUL\_10\_FPLL**: 入力クロック 10MHz, 出力クロック 100MHz (10 通倍)
- **CG\_10M\_MUL\_12\_FPLL**: 入力クロック 10MHz, 出力クロック 120MHz (12 通倍)
- **CG\_12M\_MUL\_4\_FPLL**: 入力クロック 12MHz, 出力クロック 48MHz (4 通倍)

- **CG\_12M\_MUL\_10\_FPLL:** 入力クロック 12MHz, 出力クロック 120MHz (10 逡倍)
- **CG\_16M\_MUL\_4\_FPLL:** 入力クロック 16MHz, 出力クロック 64MHz (4 逡倍)
- **CG\_16M\_MUL\_6\_FPLL:** 入力クロック 16MHz, 出力クロック 96MHz (6 逡倍)

## 6.2.3.14 CG\_SetPLL

PLL 回路の設定

**関数のプロトタイプ宣言:**

Result

CG\_SetPLL(FunctionalState **NewState**)

**引数:**

**NewState:**

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

**機能:**

PLL 回路の有効/無効を設定します。

**戻り値:**

- **SUCCESS:** 成功
- **ERROR:** 失敗

## 6.2.3.15 CG\_GetPLLState

PLL 回路の状態の取得

**関数のプロトタイプ宣言:**

FunctionalState

CG\_GetPLLState(void)

**引数:**

なし

**機能:**

PLL 回路の状態を取得します。

**戻り値:**

PLL 回路の設定状態:

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

## 6.2.3.16 CG\_SetFosc

高速発振器(fosc)の有効/無効設定

**関数のプロトタイプ宣言:**

Result

CG\_SetFosc(CG\_FoscSrc **Source**,

FunctionalState **NewState**)

引数:

**Source:** fosc のソースクロックを選択します。

- **CG\_FOSC\_OSC\_EXT:** 外部高速発信
- **CG\_FOSC\_OSC\_INT:** 内部高速発信

**NewState**

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

機能:

高速発信器の有効/無効を設定します。

戻り値:

- **SUCCESS:** 成功
- **ERROR:** 失敗

## 6.2.3.17 CG\_SetFoscSrc

高速発振器(fosc)のソース設定

関数のプロトタイプ宣言:

void  
CG\_SetFoscSrc(CG\_FoscSrc Source)

引数:

**Source:** fosc のソースを選択します。

- **CG\_FOSC\_OSC\_EXT:** 外部高速発信子
- **CG\_FOSC\_CLKIN\_EXT:** 外部クロック入力
- **CG\_FOSC\_OSC\_INT:** 内部高速発信器

機能:

高速発振器(fosc)のソースを設定します。

戻り値:

なし

## 6.2.3.18 CG\_GetFoscSrc

高速発振器のソース取得

関数のプロトタイプ宣言:

CG\_FoscSrc  
CG\_GetFoscSrc(void)

引数:

なし

機能:

高速発振器のソースを取得します。

戻り値:

高速発振器のソース

- **CG\_FOSC\_OSC\_EXT**: 外部高速発信子
- **CG\_FOSC\_CLKIN\_EXT**: 外部クロック入力
- **CG\_FOSC\_OSC\_INT**: 内部高速発信器

## 6.2.3.19 CG\_GetFoscState

高速発信器の状態

関数のプロトタイプ宣言:

FunctionalState

CG\_GetFoscState(CG\_FoscSrc Source)

引数:

**Source**: fosc のソースを指定します。

- **CG\_FOSC\_OSC\_EXT**: 外部高速発信
- **CG\_FOSC\_OSC\_INT**: 内部高速発信

機能:

高速発信器の状態を取得します。

戻り値:

fosc の状態:

- **ENABLE**: 有効
- **DISABLE**: 無効

## 6.2.3.20 CG\_SetSTBYMode

スタンバイモードの選択

関数のプロトタイプ宣言:

void

CG\_SetSTBYMode(CG\_STBYMode **Mode**)

引数:

**Mode**: スタンバイモードを選択します。

- **CG\_STBY\_MODE\_STOP1**: STOP1 モード (内部発振器も含めてすべての内部回路が停止)
- **CG\_STBY\_MODE\_STOP2**: STOP2 モード (一部の機能を保持して内部電源を遮断)
- **CG\_STBY\_MODE\_IDLE**: IDLE モード (CPU が停止)

機能:

スタンバイモードを選択します。

戻り値:

なし

## 6.2.3.21 CG\_GetSTBYMode

スタンバイモード設定状態の取得

**関数のプロトタイプ宣言:**

CG\_STBYMode

CG\_GetSTBYMode(void)

**引数:**

なし

**機能:**

スタンバイモード設定状態を取得します。

“Reserved”の場合、“CG\_STBY\_MODE\_UNKNOWN”を返却します。

**戻り値:**

スタンバイモード:

- CG\_STBY\_MODE\_STOP1: STOP1 モード
- CG\_STBY\_MODE\_STOP2: STOP2 モード
- CG\_STBY\_MODE\_IDLE: IDLE モード
- CG\_STBY\_MODE\_UNKNOWN: 無効なモード

## 6.2.3.22 CG\_SetPortKeepInStop2Mode

STOP2 モード中の I/O 制御信号保持状態の設定

**関数のプロトタイプ宣言:**

void

CG\_SetPortKeepInStop2Mode(FunctionalState **NewState**)

**引数:**

**NewState:**

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

STOP2 モード中の I/O 制御信号保持の詳細については、MCU データシートの“低消費電力モード”を参照してください。

**機能:**

STOP2 モード中の I/O 制御信号保持の有効/無効を切り替えます。

**戻り値:**

なし

## 6.2.3.23 CG\_GetPortKeepInStop2Mode

STOP2 モード中の I/O 制御信号保持状態の取得

**関数のプロトタイプ宣言:**

FunctionalState

CG\_GetPinStateInStopMode(void)

**引数:**

なし

**機能:**



STOP2 モード中の I/O 制御信号保持状態を取得します。

戻り値:

STOP2 モード中の I/O 制御信号保持状態:

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

## 6.2.3.24 CG\_SetFcSrc

fc のソース選択

関数のプロトタイプ宣言:

Result

CG\_SetFcSrc(CG\_FcSrc **Source**)

引数:

**Source:** fc のソースを選択します。

- **CG\_FC\_SRC\_FOSC:** fosc 使用
- **CG\_FC\_SRC\_FPLL:** fpll 使用

機能:

fc のソースクロックを選択します。

戻り値:

**SUCCESS:** 成功

**ERROR:** 失敗

## 6.2.3.25 CG\_GetFcSrc

fc ソースの設定状態取得

関数のプロトタイプ宣言:

CG\_FcSrc

CG\_GetFosc(void)

引数:

なし

機能:

fc ソースの設定状態を取得します。

戻り値:

fc ソースの設定状態

- **CG\_FC\_SRC\_FOSC:** fosc 使用
- **CG\_FC\_SRC\_FPLL:** fpll 使用

## 6.2.3.26 CG\_SetProtectCtrl

CG レジスタの書き込み制御

関数のプロトタイプ宣言:

```
void
CG_SetProtectCtrl(FunctionalState NewState)
```

引数:

**NewState**

- **DISABLE**: 書き込み禁止
- **ENABLE**: 書き込み許可

機能:

CG レジスタの書き込み許可/禁止を設定します。

戻り値:

なし

## 6.2.3.27 CG\_SetSTBYReleaseINTSrc

スタンバイモードの解除割り込みソースの設定

関数のプロトタイプ宣言:

```
void
CG_SetSTBYReleaseINTSrc(CG_INTSrc INTSource,
                        CG_INTActiveState ActiveState,
                        FunctionalState NewState)
```

引数:

**INTSource**: スタンバイモードの解除割り込みソースを選択します。

- **CG\_INT\_SRC\_0**: INT0
- **CG\_INT\_SRC\_1**: INT1
- **CG\_INT\_SRC\_2**: INT2
- **CG\_INT\_SRC\_3**: INT3
- **CG\_INT\_SRC\_4**: INT4
- **CG\_INT\_SRC\_5**: INT5
- **CG\_INT\_SRC\_6**: INT6
- **CG\_INT\_SRC\_7**: INT7
- **CG\_INT\_SRC\_8**: INT8
- **CG\_INT\_SRC\_9**: INT9
- **CG\_INT\_SRC\_A**: INTA
- **CG\_INT\_SRC\_B**: INTB
- **CG\_INT\_SRC\_C**: INTC
- **CG\_INT\_SRC\_D**: INTD
- **CG\_INT\_SRC\_E**: INTE
- **CG\_INT\_SRC\_F**: INTF
- **CG\_INT\_SRC\_CECRX**: INTCECRX
- **CG\_INT\_SRC\_CECTX**: INTCECTX
- **CG\_INT\_SRC\_RMCRX0**: INTRMCRX0
- **CG\_INT\_SRC\_RTC**: INTRTC
- **CG\_INT\_SRC\_RMCRX1**: INTRMCRX1

**ActiveState**: 解除トリガのアクティブ状態を選択します。

割り込み要因	選択できるアクティブレベル	説明
<b>CG_INT_SRC_RTC</b>	<b>CG_INT_ACTIVE_STATE_FALLING</b>	↓エッジ
<b>CG_INT_SRC_CECRX , CG_INT_SRC_CECTX , CG_INT_SRC_RMCRX0, CG_INT_SRC_RMCRX1</b>	<b>CG_INT_ACTIVE_STATE_RISING</b>	↑エッジ

上記以外	CG_INT_ACTIVE_STATE_L	"Low"レベル
	CG_INT_ACTIVE_STATE_H	"High"レベル
	CG_INT_ACTIVE_STATE_FALLING	↓エッジ
	CG_INT_ACTIVE_STATE_RISING	↑エッジ
	CG_INT_ACTIVE_STATE_BOTH_EDGES	両エッジ

**NewState:** 解除トリガの有効/無効を選択します。

- **ENABLE:** 許可
- **DISABLE:** 禁止

**機能:**

スタンバイモードの解除割り込みソースを設定します。

**戻り値:**

なし

## 6.2.3.28 CG\_GetSTBYReleaseINTState

スタンバイモードの解除割り込みソースのアクティブ状態の取得

**関数のプロトタイプ宣言:**

CG\_INT\_ActiveState

CG\_GetSTBYReleaseINTSrc(CG\_INTSrc **INTSource**)

**引数:**

**INTSource:** 解除割り込みソースの選択

- CG\_INT\_SRC\_0, CG\_INT\_SRC\_1, CG\_INT\_SRC\_2, CG\_INT\_SRC\_3,  
CG\_INT\_SRC\_4, CG\_INT\_SRC\_5, CG\_INT\_SRC\_6, CG\_INT\_SRC\_7,  
CG\_INT\_SRC\_8, CG\_INT\_SRC\_9, CG\_INT\_SRC\_A, CG\_INT\_SRC\_B,  
CG\_INT\_SRC\_C, CG\_INT\_SRC\_D, CG\_INT\_SRC\_E, CG\_INT\_SRC\_F,  
CG\_INT\_SRC\_CECRX, CG\_INT\_SRC\_CECTX, CG\_INT\_SRC\_RMCRX0,  
CG\_INT\_SRC\_RTC, CG\_INT\_SRC\_RMCRX1

**機能:**

スタンバイモードの解除割り込みソースのアクティブ状態を取得します。

**戻り値:**

解除割り込みソースのアクティブ状態

- **CG\_INT\_ACTIVE\_STATE\_FALLING:** ↓エッジ
- **CG\_INT\_ACTIVE\_STATE\_RISING:** ↑エッジ
- **CG\_INT\_ACTIVE\_STATE\_BOTH\_EDGES:** 両エッジ
- **CG\_INT\_ACTIVE\_STATE\_INVALID:** 無効な値

## 6.2.3.29 CG\_ClearINTReq

スタンバイ解除割り込み要求のクリア

**関数のプロトタイプ宣言:**

void

CG\_ClearINTReq(CG\_INTSrc **INTSource**)

**引数:**

**INTSource:** 解除割り込みソースを選択します。

- CG\_INT\_SRC\_0, CG\_INT\_SRC\_1, CG\_INT\_SRC\_2, CG\_INT\_SRC\_3,  
CG\_INT\_SRC\_4, CG\_INT\_SRC\_5, CG\_INT\_SRC\_6, CG\_INT\_SRC\_7,  
CG\_INT\_SRC\_8, CG\_INT\_SRC\_9, CG\_INT\_SRC\_A, CG\_INT\_SRC\_B,  
CG\_INT\_SRC\_C, CG\_INT\_SRC\_D, CG\_INT\_SRC\_E, CG\_INT\_SRC\_F,  
CG\_INT\_SRC\_CECRX, CG\_INT\_SRC\_CECTX, CG\_INT\_SRC\_RMCRX0,  
CG\_INT\_SRC\_RTC, CG\_INT\_SRC\_RMCRX1

**機能:**

スタンバイ解除割り込み要求をクリアします。

**戻り値:**

なし

## 6.2.3.30 CG\_GetNMIFlag

NMI 発生要因フラグの取得

**関数のプロトタイプ宣言:**

CG\_NMI\_Factor

CG\_GetNMIFlag (void)

**引数:**

なし

**機能:**

NMI 発生要因フラグを取得します。

**戻り値:**

NMI 発生要因

- **WDT** (Bit 0) : WDT による NMI 発生
- **NMIPin** (Bit 1) : NMI 端子による NMI 発生
- **DetectLowVoltage** (Bit 2) : LVD で電源電圧が設定電圧より下がったことによる NMI 発生
- **ReturnLowVoltage** (Bit 3) : LVD で電源電圧が設定電圧より上がったことによる NMI 発生

## 6.2.3.31 CG\_GetIOSCFIashFlag

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

**関数のプロトタイプ宣言:**

FunctionalState

CG\_GetIOSCFIashFlag(void)

**引数:**

なし

**機能:**

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

**補足:**

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

**戻り値:**

内部高速発振器停止/FLASH E/W 可能フラグ:

- **ENABLE:** 許可
- **DISABLE:** 禁止

## 6.2.3.32 CG\_GetResetFlag

リセットフラグの取得とクリア

**関数のプロトタイプ宣言:**

CG\_ResetFlag  
CG\_GetResetFlag(void)

**引数:**

なし

**機能:**

リセットフラグの取得とクリアを行います。

**戻り値:**

リセットフラグ:

- **ResetPin** (Bit0) RESET 端子によるリセット
- **WDTReset** (Bit 2) WDT によるリセット
- **STOP2Reset**(Bit3) STOP2 モード解除によるリセット
- **DebugReset** (Bit 4) <SYSRESETREQ>によるリセット
- **OFDReset** (Bit5) OFD によるリセット
- **LVDReset** (Bit6) LVD によるリセット

## 6.2.3.33 CG\_SetADCClkSupply

ADC クロックの選択

**関数のプロトタイプ宣言:**

void  
CG\_SetADCClkSupply(FunctionalState **NewState**)

**引数:**

**NewState:** ADC クロックを選択します。

- **ENABLE:** 動作
- **DISABLE:** 停止

**機能:**

ADC クロックを選択します。

**戻り値:**

なし

## 6.2.3.34 CG\_SetInternalOscForOFD

内部高速発信器(OFD 用)の発振/停止設定

関数のプロトタイプ宣言:

void

CG\_SetInternalOscForOFD(FunctionalState **NewState**)

引数:

**NewState**

- **ENABLE**: 発振
- **DISABLE**: 停止

機能:

内部高速発信器(OFD 用)の発振/停止を選択します。

戻り値:

なし

## 6.2.3.35 CG\_SetFcPeriphA

周辺機能へのクロック供給停止設定

関数のプロトタイプ宣言:

void

CG\_SetFcPeriphA(uint32\_t **Periph**, FunctionalState **NewState**)

引数:

**Periph**: クロック供給を停止する周辺機能を選択します。

- **CG\_FC\_PERIPH\_PORTA**: PORT A
- **CG\_FC\_PERIPH\_PORTB**: PORT B
- **CG\_FC\_PERIPH\_PORTC**: PORT C
- **CG\_FC\_PERIPH\_PORTD**: PORT D
- **CG\_FC\_PERIPH\_PORTE**: PORT E
- **CG\_FC\_PERIPH\_PORTF**: PORT F
- **CG\_FC\_PERIPH\_PORTG**: PORT G
- **CG\_FC\_PERIPH\_PORTH**: PORT H
- **CG\_FC\_PERIPH\_PORTJ**: PORT J
- **CG\_FC\_PERIPH\_PORTK**: PORT K
- **CG\_FC\_PERIPH\_PORTL**: PORT L
- **CG\_FC\_PERIPH\_PORTM**: PORT M
- **CG\_FC\_PERIPH\_PORTN**: PORT N
- **CG\_FC\_PERIPH\_TMRB0**: TMRB0
- **CG\_FC\_PERIPH\_TMRB1**: TMRB1
- **CG\_FC\_PERIPH\_TMRB2**: TMRB2
- **CG\_FC\_PERIPH\_TMRB3**: TMRB3
- **CG\_FC\_PERIPH\_TMRB4**: TMRB4
- **CG\_FC\_PERIPH\_TMRB5**: TMRB5
- **CG\_FC\_PERIPH\_TMRB6**: TMRB6
- **CG\_FC\_PERIPH\_TMRB7**: TMRB7
- **CG\_FC\_PERIPH\_TMRB8**: TMRB8
- **CG\_FC\_PERIPH\_TMRB9**: TMRB9
- **CG\_FC\_PERIPH\_TMRB10**: TMRB10
- **CG\_FC\_PERIPH\_TMRB11**: TMRB11

- CG\_FC\_PERIPH\_TMRB12: TMRB12
- CG\_FC\_PERIPH\_TMRB13: TMRB13
- CG\_FC\_PERIPH\_TMRB14: TMRB14
- CG\_FC\_PERIPH\_TMRB15: TMRB15
- CG\_FC\_PERIPH\_MPT0: MPT0
- CG\_FC\_PERIPH\_MPT1: MPT1
- CG\_FC\_PERIPH\_TRACE: TRACE
- CG\_FC\_PERIPHA\_ALL: すべて

**NewState**

- ENABLE: 動作
- DISABLE: 停止

**機能:**

周辺機能へのクロック供給を制御します。

**戻り値:**

なし

## 6.2.3.36 CG\_SetFcPeriphB

周辺機能へのクロック供給停止設定

**関数のプロトタイプ宣言:**

void

CG\_SetFcPeriphB(uint32\_t **Periph**, FunctionalState **NewState**)

**引数:**

**Periph:** クロック供給を停止する周辺機能を選択します。

- CG\_FC\_PERIPH\_SIO\_UART0: SIO/UART0
- CG\_FC\_PERIPH\_SIO\_UART1: SIO/UART1
- CG\_FC\_PERIPH\_SIO\_UART2: SIO/UART2
- CG\_FC\_PERIPH\_SIO\_UART3: SIO/UART3
- CG\_FC\_PERIPH\_SIO\_UART4: SIO/UART4
- CG\_FC\_PERIPH\_SIO\_UART5: SIO/UART5
- CG\_FC\_PERIPH\_SIO\_UART6: SIO/UART6
- CG\_FC\_PERIPH\_SIO\_UART7: SIO/UART7
- CG\_FC\_PERIPH\_SIO\_UART8: SIO/UART8
- CG\_FC\_PERIPH\_SIO\_UART9: SIO/UART9
- CG\_FC\_PERIPH\_UART0: UART0
- CG\_FC\_PERIPH\_UART1: UART1
- CG\_FC\_PERIPH\_I2C0: I2C0
- CG\_FC\_PERIPH\_I2C1: I2C1
- CG\_FC\_PERIPH\_I2C2: I2C2
- CG\_FC\_PERIPH\_I2C3: I2C3
- CG\_FC\_PERIPH\_I2C4: I2C4
- CG\_FC\_PERIPH\_SSP0: SSP0
- CG\_FC\_PERIPH\_SSP1: SSP1
- CG\_FC\_PERIPH\_SSP2: SSP2
- CG\_FC\_PERIPH\_EBIF: EBIF
- CG\_FC\_PERIPH\_DMACA: DMAC A
- CG\_FC\_PERIPH\_DMACB: DMAC B
- CG\_FC\_PERIPH\_DMACC: DMAC C
- CG\_FC\_PERIPH\_DMAIF: DMACIF
- CG\_FC\_PERIPH\_ADC: ADC

- **CG\_FC\_PERIPH\_WDT**: WDT
- **CG\_FC\_PERIPH\_OFD**: OFD
- **CG\_FC\_PERIPHB\_ALL**: すべて

**NewState**

- **ENABLE**: 動作
- **DISABLE**: 停止

**機能:**

周辺機能へのクロック供給を制御します。

**戻り値:**

なし

### 6.2.3.37 CG\_SetFs

低速発振器(fs)の動作選択

**関数のプロトタイプ宣言:**

void

CG\_SetFs(FunctionalState **NewState**)

**引数:**

**NewState**

- **ENABLE**: 発振
- **DISABLE**: 停止

**機能:**

低速発振器(fs)の動作を選択します。

**戻り値:**

なし

## 6.2.4 データ構造

### 6.2.4.1 CG\_NMIFactor

**メンバ:**

uint32\_t

**All** CGNMI ソース起動状態を指定します。

**ビットフィールド:**

uint32\_t

**WDT**(Bit 0) WDT による NMI 発生

uint32\_t

**NMIPin**(Bit 1) NMI 端子による NMI 発生

uint32\_t

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

uint32\_t

**Reserved1** (Bit3) 未使用



uint32\_t

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

uint32\_t

**Reserved2** (Bit5~bit31) 未使用

## 6.2.4.2 CG\_ResetFlag

メンバ:

uint32\_t

**All** CG リセット要因を指定します。

ビットフィールド:

uint32\_t

**ResetPin**(Bit0) RESET 端子によるリセット

uint32\_t

**Reserved1** (Bit1) 未使用

uint32\_t

**WDTReset**(Bit2) WDT によるリセット

uint32\_t

**STOP2Reset**(Bit3) STOP2 モード解除によるリセット

uint32\_t

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

uint32\_t

**OFDReset**(Bit5) OFD によるリセット

uint32\_t

**LVDReset**(Bit6) LVD によるリセット

uint32\_t

**Reserved2** (Bit7~bit31) 未使用

## 7. FC

### 7.1 概要

本デバイスは、フラッシュメモリを内蔵しています。  
フラッシュメモリのサイズは TPM462F10 の場合 1024Kbyte で、TPM462F15 の場合 1536KByte です。

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

ブロック構成は、デバイスのデータシートを参照してください。

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

\\Libraries\\TX04\_Periph\_Driver\\src\\tmpm462\_fc.c  
\\Libraries\\TX04\_Periph\_Driver\\inc\\tmpm462\_fc.h

### 7.2 API 関数

#### 7.2.1 関数一覧

- ◆ void FC\_SetSecurityBit(FunctionalState **NewState**)
- ◆ FunctionalState FC\_GetSecurityBit(void)
- ◆ WorkState FC\_GetBusyState(void)
- ◆ FunctionalState FC\_GetBlockProtectState(uint8\_t **BlockNum**)
- ◆ FunctionalState FC\_GetPageProtectState(uint8\_t **PageNum**)
- ◆ FunctionalState FC\_GetAbortState(void)
- ◆ uint32\_t FC\_GetSwapSize(void);
- ◆ uint32\_t FC\_GetSwapState(void);
- ◆ void FC\_SelectArea(uint8\_t **AreaNum**, FunctionalState **NewState**);
- ◆ void FC\_SetAbortion(void);
- ◆ void FC\_ClearAbortion(void);
- ◆ void FC\_SetClkDiv(uint8\_t **ClkDiv**);
- ◆ void FC\_SetProgramCount(uint8\_t **ProgramCount**);
- ◆ void FC\_SetEraseCounter(uint8\_t **EraseCounter**);
- ◆ FC\_Result FC\_ProgramBlockProtectState(uint8\_t **BlockNum**);
- ◆ FC\_Result FC\_ProgramPageProtectState(uint8\_t **PageNum**);
- ◆ FC\_Result FC\_EraseProtectState(void);
- ◆ FC\_Result FC\_WritePage(uint32\_t **PageAddr**, uint32\_t \* **Data**);
- ◆ FC\_Result FC\_EraseBlock(uint32\_t **BlockAddr**);
- ◆ FC\_Result FC\_EraseArea(uint32\_t **AreaAddr**);
- ◆ FC\_Result FC\_ErasePage(uint32\_t **PageAddr**);
- ◆ FC\_Result FC\_EraseChip(void);
- ◆ FC\_Result FC\_SetSwpsrBit(uint8\_t **BitNum**);
- ◆ uint32\_t FC\_GetSwpsrBitValue(uint8\_t **BitNum**);

#### 7.2.2 関数の種類

関数は、主に以下の 5 種類に分かれています:

- 1) セキュリティ設定(Flash ROM データの読み出し、デバッグ):  
FC\_SetSecurityBit(), FC\_GetSecurityBit()
- 2) 自動動作状態およびプロテクト状態の取得:  
FC\_GetBusyState(), FC\_GetBlockProtectState(), FC\_GetPageProtectState()
- 3) プロテクト設定と解除:  
FC\_ProgramBlockProtectState(), FC\_ProgramPageProtectState(),  
FC\_EraseProtectState()
- 4) 自動動作コマンド:  
FC\_WritePage(), FC\_EraseBlock(), FC\_EraseChip(), FC\_EraseArea(),  
FC\_ErasePage(), FC\_SetSwpsrBit()
- 5) その他:  
FC\_GetAbortState(), FC\_GetSwapSize(), FC\_GetSwapState(), FC\_SelectArea(),  
FC\_SetAbortion(), FC\_ClearAbortion(), FC\_SetClkDiv(), FC\_SetProgramCount(),  
FC\_SetEraseCounter(), FC\_GetSwpsrBitValue()

## 7.2.3 関数仕様

### 7.2.3.1 FC\_SetSecurityBit

セキュリティビットの設定

関数のプロトタイプ宣言:

void  
FC\_SetSecurityBit (FunctionalState **NewState**)

引数:

**NewState**: セキュリティビットを設定します。

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

機能:

- 1) 書き込み/消去プロテクト用のすべてのプロテクトビット (PSRA<BLKn>)を"1"にします。
  - 2) FCSECBIT<SECBIT>を"1"にします。
- 上記の 2 つの条件が成立すると、セキュリティ機能が有効になります。セキュリティ機能が有効な状態の制限内容は次の通りです。
- ROM 領域のデータの読み出し。
  - JTAG/SW、トレースの通信

したがって、この API を使用する場合は、注意して実行してください。

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

戻り値:

なし

### 7.2.3.2 FC\_GetSecurityBit

セキュリティビットの設定状態の取得

関数のプロトタイプ宣言:

FunctionalState

FC\_GetSecurityBit(void)

引数:

なし

機能:

セキュリティビットの設定状態を取得します。

戻り値:

セキュリティビットの設定状態:

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

### 7.2.3.3 FC\_GetBusyState

自動動作状態の取得

関数のプロトタイプ宣言:

WorkState

FC\_GetBusyState (void)

引数:

なし

機能:

自動動作状態を取得します。

戻り値:

自動動作状態:

- **BUSY**: 自動動作中
- **DONE**: 自動動作終了

### 7.2.3.4 FC\_GetBlockProtectState

ブロックのプロテクト状態の取得

関数のプロトタイプ宣言:

FunctionalState

FC\_GetBlockProtectState(uint8\_t **BlockNum**).

引数:

**BlockNum**: ブロック番号を選択します。

**TMPM462F10FG**:

- FC\_BLOCK\_1 ~ FC\_BLOCK\_31

**TMPM462F15FG**:

- FC\_BLOCK\_1 ~ FC\_BLOCK\_47

機能:

各ブロックのプロテクト状態を示します。プロテクト状態の時には、書き込み、消去ができません。

戻り値:

ブロックプロテクトの状態:

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

## 7.2.3.5 FC\_GetPageProtectState

ページのプロテクト状態の取得

関数のプロトタイプ宣言:

FunctionalState

FC\_GetPageProtectState(uint8\_t *PageNum*)

引数:

*PageNum*: ページ番号を選択します。

- **FC\_PAGE\_0 ~ FC\_PAGE\_7**

機能:

各ページのプロテクト状態を示します。プロテクト状態の時には、書き込み、消去ができません。

戻り値:

ページプロテクトの状態:

**DISABLE:** プロテクト状態ではない。

**ENABLE:** プロテクト状態

## 7.2.3.6 FC\_GetAbortState

自動実行コマンドの中止状態の取得

関数のプロトタイプ宣言:

FunctionalState

FC\_GetAbortState(void)

引数:

なし

機能:

自動実行コマンドの中止状態を取得します。

戻り値:

自動実行コマンドの中止を行うと **DONE** を返します。

## 7.2.3.7 FC\_GetSwapSize

メモリスワップサイズの取得

関数のプロトタイプ宣言:

uint32\_t

FC\_GetSwapSize(void)

引数:

なし

**機能:**

メモリスワップサイズを取得します。

**戻り値:**

メモリスワップサイズ:

**FC\_SWAP\_SIZE\_4K**: 4K バイト

**FC\_SWAP\_SIZE\_8K**: 8K バイト

**FC\_SWAP\_SIZE\_16K**: 16K バイト

**FC\_SWAP\_SIZE\_32K**: 32K バイト

**FC\_SWAP\_SIZE\_AREA1**: エリア 0 とエリア 1

**FC\_SWAP\_SIZE\_AREA2**: エリア 0 とエリア 2

## 7.2.3.8 FC\_GetSwapState

メモリスワップ状態の取得

**関数のプロトタイプ宣言:**

uint32\_t

FC\_GetSwapState(void)

**引数:**

なし

**機能:**

メモリスワップ状態を取得します。

**戻り値:**

メモリスワップ状態:

**FC\_SWAP\_RELEASE**: スワップ解除

**FC\_SWAP\_PROHIBIT**: 設定禁止

**FC\_SWAPPING**: スワップ中

**FC\_SWAP\_INITIAL**: スワップ解除(初期化状態)

## 7.2.3.9 FC\_SelectArea

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

**関数のプロトタイプ宣言:**

void

FC\_SelectArea(uint8\_t **AreaNum**)

**引数:**

**AreaNum**: 以下のいずれかのエリア番号を選択します。

**TMPM462F15FG**:

➤ **FC\_AREA\_0**, **FC\_AREA\_1**, **FC\_AREA\_2**, **FC\_AREA\_ALL**

**TMPM462F10FG**:

➤ **FC\_AREA\_0**, **FC\_AREA\_1**, **FC\_AREA\_ALL**

**NewState**: 選択/非選択を設定します。

➤ **ENABLE**: 選択

➤ **DISABLE**: 非選択

**機能:**

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

**戻り値:**

なし

## 7.2.3.10 FC\_SetAbortion

自動実行コマンドの中止

**関数のプロトタイプ宣言:**

```
void  
FC_SetAbortion(void)
```

**引数:**

なし

**機能:**

自動実行コマンドを中止します。

**戻り値:**

なし

## 7.2.3.11 FC\_ClearAbortion

自動実行コマンド中止フラグのクリア

**関数のプロトタイプ宣言:**

```
void  
FC_ClearAbortion(void)
```

**引数:**

なし

**機能:**

自動実行コマンド中止フラグをクリアします。

**戻り値:**

なし

## 7.2.3.12 FC\_SetClkDiv

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

**関数のプロトタイプ宣言:**

```
void  
FC_SetClkDiv(uint8_t ClkDiv)
```

**引数:**

**ClkDiv:** 以下のいずれかの分周比を選択します。

➤ FC\_Clk\_Div\_1 ~ FC\_Clk\_Div\_32

**機能:**

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

**戻り値:**

なし

### 7.2.3.13 FC\_SetProgramCount

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

**関数のプロトタイプ宣言:**

void

FC\_SetProgramCount(uint8\_t **ProgramCount**)

**引数:**

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

➤ FC\_PROG\_CNT\_250, FC\_PROG\_CNT\_300, FC\_PROG\_CNT\_350

**機能:**

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

**戻り値:**

なし

### 7.2.3.14 FC\_SetEraseCounter

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

**関数のプロトタイプ宣言:**

void

FC\_SetEraseCounter (uint8\_t **EraseCounter**)

**引数:**

**EraseCounter:** 以下のいずれかのカウント数を選択します。

- FC\_ERAS\_CNT\_85, FC\_ERAS\_CNT\_90
- FC\_ERAS\_CNT\_95, FC\_ERAS\_CNT\_100
- FC\_ERAS\_CNT\_105, FC\_ERAS\_CNT\_110
- FC\_ERAS\_CNT\_115, FC\_ERAS\_CNT\_120
- FC\_ERAS\_CNT\_125, FC\_ERAS\_CNT\_130
- FC\_ERAS\_CNT\_135, FC\_ERAS\_CNT\_140

**機能:**

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



戻り値:  
なし

## 7.2.3.15 FC\_ProgramBlockProtectState

ブロックのプロテクト設定

関数のプロトタイプ宣言:

FC\_Result

FC\_ProgramProtectState(uint8\_t **BlockNum**)

引数:

**BlockNum**: 以下のいずれかのブロック番号を選択します。

TMPM462F10FG:

➤ FC\_BLOCK\_1 ~ FC\_BLOCK\_31

TMPM462F15FG:

➤ FC\_BLOCK\_1 ~ FC\_BLOCK\_47

機能:

ブロックプロテクトを設定します。

戻り値:

実行結果:

FC\_SUCCESS: 成功

FC\_ERROR\_PROTECTED: 設定済

FC\_ERROR\_OVER\_TIME: タイマオーバーフロー

## 7.2.3.16 FC\_ProgramPageProtectState

ページプロテクトの設定

関数のプロトタイプ宣言:

FC\_Result

FC\_ProgramProtectState(uint8\_t **PageNum**)

引数:

**PageNum**: 以下のいずれかのページ番号を選択します。

➤ FC\_PAGE\_0 ~ FC\_PAGE\_7

機能:

ページプロテクトを設定します。

戻り値:

実行結果:

FC\_SUCCESS: 成功

FC\_ERROR\_PROTECTED: 設定済

FC\_ERROR\_OVER\_TIME: タイマオーバーフロー

## 7.2.3.17 FC\_EraseProtectState

プロテクトの解除

**関数のプロトタイプ宣言:**

FC\_Result

FC\_EraseBlockProtectState(void)

**引数:**

なし

**機能:**

プロテクトビットを"0"にすることでプロテクトを解除します。

**戻り値:**

実行結果:

**FC\_SUCCESS:** プロテクト解除の成功

**FC\_ERROR\_OVER\_TIME:** プロテクト解除の失敗(自動動作のタイムアウト)

## 7.2.3.18 FC\_WritePage

ページ単位の書き込み

**関数のプロトタイプ宣言:**

FC\_Result

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

**引数:**

**PageAddr:** ページの開始アドレスを指定します。

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

**機能:**

ページ書き込みを行います。

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

**\*補足:**

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

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

**戻り値:**

実行結果:

- **FC\_SUCCESS:** 消去成功
- **FC\_ERROR\_PROTECTED:** 消去失敗(ブロックにプロテクトが設定されている)
- **FC\_ERROR\_OVER\_TIME:** 消去の失敗(自動動作のタイムアウト)

## 7.2.3.19 FC\_EraseBlock

ブロック単位の消去

**関数のプロトタイプ宣言:**

FC\_Result

FC\_EraseBlock(uint32\_t **BlockAddr**)**引数:****BlockAddr**: ブロック開始アドレスを指定してください。**機能:**

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

**戻り値:**

実行結果:

- **FC\_SUCCESS**: 消去成功
- **FC\_ERROR\_PROTECTED**: 消去失敗(ブロックにプロテクトが設定されている)
- **FC\_ERROR\_OVER\_TIME**: 消去の失敗(自動動作のタイムアウト)

## 7.2.3.20 FC\_EraseArea

エリア単位の消去

**関数のプロトタイプ宣言:**

FC\_Result

FC\_EraseArea(uint32\_t **AreaAddr**)**引数:****AreaAddr**: エリア開始アドレスを指定してください。**機能:**

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

**戻り値:**

実行結果:

- **FC\_SUCCESS**: 消去成功
- **FC\_ERROR\_PROTECTED**: 消去失敗(ブロックにプロテクトが設定されている)
- **FC\_ERROR\_OVER\_TIME**: 消去の失敗(自動動作のタイムアウト)

## 7.2.3.21 FC\_ErasePage

ページ単位の消去

**関数のプロトタイプ宣言:**

FC\_Result

FC\_ErasePage(uint32\_t **PageAddr**)**引数:****PageAddr**: ページ開始アドレスを指定してください。**機能:**

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

戻り値:

実行結果:

- **FC\_SUCCESS**: 消去成功
- **FC\_ERROR\_PROTECTED**: 消去失敗(ブロックにプロテクトが設定されている)
- **FC\_ERROR\_OVER\_TIME**: 消去の失敗(自動動作のタイムアウト)

## 7.2.3.22 FC\_EraseChip

チップ消去

関数のプロトタイプ宣言:

FC\_Result

FC\_EraseChip(void)

引数:

なし

機能:

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

戻り値:

実行結果:

- **FC\_SUCCESS**: 消去成功
- **FC\_ERROR\_PROTECTED**: 消去失敗(ブロックにプロテクトが設定されている)
- **FC\_ERROR\_OVER\_TIME**: 消去の失敗(自動動作のタイムアウト)

## 7.2.3.23 FC\_SetSwapsrBit

FCSWPSR[10:0]レジスタの設定

関数のプロトタイプ宣言:

FC\_Result

FC\_SetSwapsrBit(uint8\_t **BitNum**)

引数:

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

- **FC\_SWPSR\_BIT\_0** ~ **FC\_SWPSR\_BIT\_10**

機能:

FCSWPSR[10:0]レジスタを設定します。

戻り値:

FCSWPSR ビット変更の実行結果:

- **FC\_SUCCESS**: 設定成功
- **FC\_ERROR\_OVER\_TIME**: 設定失敗(自動動作のタイムアウト)

## 7.2.3.24 FC\_GetSwpsrBitValue

FCSWPSR[10:0]レジスタ値の取得

関数のプロトタイプ宣言:

uint32\_t

FC\_GetSwpsrBitValue(uint8\_t *BitNum*)

引数:

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

➤ FC\_SWPSR\_BIT\_0 ~ FC\_SWPSR\_BIT\_10

機能:

FCSWPSR[10:0]レジスタ値を取得します。

戻り値:

指定ビットの値:

➤ FC\_BIT\_VALUE\_0: 0

➤ FC\_BIT\_VALUE\_1: 1

## 7.2.4 データ構造

なし

## 8. FUART

### 8.1 概要

TMPM462 は非同期のシリアルチャネル (Full UART)とモデム制御を内蔵します。本製品は 1 チャネルの Full UART(FUART0 と FUART1)を内蔵します。

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

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

/Libraries/TX04\_Periph\_Driver/src/tmpm462\_fuart.c

/Libraries/TX04\_Periph\_Driver/inc/tmpm462\_fuart.h

### 8.2 API 関数

#### 8.2.1 関数一覧

- ◆ void FUART\_Enable(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_Disable(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ uint32\_t FUART\_GetRxData(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_SetTxData(TSB\_FUART\_TypeDef \* **FUARTx**, uint32\_t **Data**)
- ◆ FUART\_Err FUART\_GetErrStatus(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_ClearErrStatus(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ WorkState FUART\_GetBusyState(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ FUART\_StorageStatus FUART\_GetStorageStatus(  
TSB\_FUART\_TypeDef \* **FUARTx**, FUART\_Direction **Direction**)
- ◆ void FUART\_SetIrDADivisor(TSB\_FUART\_TypeDef \* **FUARTx**, uint32\_t **Divisor**)
- ◆ void FUART\_Init(  
TSB\_FUART\_TypeDef \* **FUARTx**, FUART\_InitTypeDef \* **InitStruct**)
- ◆ void FUART\_EnableFIFO(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_DisableFIFO(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_SetSendBreak(  
TSB\_FUART\_TypeDef \* **FUARTx**, FunctionalState **NewState**)
- ◆ void FUART\_SetIrDAEncodeMode(  
TSB\_FUART\_TypeDef \* **FUARTx**, uint32\_t **Mode**)
- ◆ Result FUART\_EnableIrDA(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_DisableIrDA(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_SetINTFIFOLevel(  
TSB\_FUART\_TypeDef \* **FUARTx**, uint32\_t **RxLevel**, uint32\_t **TxLevel**)
- ◆ void FUART\_SetINTMask(TSB\_FUART\_TypeDef \* **FUARTx**, uint32\_t **IntMaskSrc**)
- ◆ FUART\_INTStatus FUART\_GetINTMask(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ FUART\_INTStatus FUART\_GetRawINTStatus(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ FUART\_INTStatus FUART\_GetMaskedINTStatus(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_ClearINT(  
TSB\_FUART\_TypeDef \* **FUARTx**, FUART\_INTStatus **INTStatus**)
- ◆ void FUART\_SetDMAOnErr(

- TSB\_FUART\_TypeDef \* **FUARTx**, FunctionalState **NewState**)
- ◆ void FUART\_SetFIFODMA(TSB\_FUART\_TypeDef \* **FUARTx**,  
FUART\_Direction **Direction**, FunctionalState **NewState**)
  - ◆ FUART\_AllModemStatus FUART\_GetModemStatus(  
TSB\_FUART\_TypeDef \* **FUARTx**)
  - ◆ void FUART\_SetRTSSStatus(  
TSB\_FUART\_TypeDef \* **FUARTx**, FUART\_ModemStatus **Status**)
  - ◆ void FUART\_SetDTRStatus(  
TSB\_FUART\_TypeDef \* **FUARTx**, FUART\_ModemStatus **Status**)

## 8.2.2 関数の種類

関数は、主に以下の 5 種類に分かれています:

- 1) Full UART 構成と初期化、共通動作:  
FUART\_Enable(), FUART\_Disable(), FUART\_Init(), FUART\_GetRxData(),  
FUART\_SetTxData(), FUART\_GetErrStatus(), FUART\_ClearErrStatus(),  
FUART\_GetBusyState(), FUART\_GetStorageStatus(), FUART\_SetSendBreak()
- 2) FIFO と DMA の設定  
FUART\_EnableFIFO(), FUART\_DisableFIFO(), FUART\_SetINTFIFOLevel(),  
FUART\_SetFIFODMA(), FUART\_SetDMAOnErr()
- 3) 割り込み制御と割り込み状態の取得:  
FUART\_SetINTMask(), FUART\_GetINTMask(), FUART\_GetRawINTStatus(),  
FUART\_GetMaskedINTStatus(), FUART\_ClearINT()
- 4) モデム制御:  
FUART\_GetModemStatus(), FUART\_SetRTSSStatus(), FUART\_SetDTRStatus()
- 5) IrDA の設定  
FUART\_EnableIrDA(), FUART\_DisableIrDA(), FUART\_SetIrDAEncodeMode(),  
FUART\_SetIrDADivisor()

## 8.2.3 関数仕様

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

### 8.2.3.1 FUART\_Enable

Full UART チャンネルの有効化

関数のプロトタイプ宣言:

void  
FUART\_Enable(TSB\_FUART\_TypeDef \* **FUARTx**)

引数:

**FUARTx**: Full UART チャンネルを指定します。

機能:

Full UART チャンネルを有効にします。

戻り値:

なし

## 8.2.3.2 FUART\_Disable

Full UART チャンネルの無効化

関数のプロトタイプ宣言:

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

引数:

**FUARTx**: Full UART チャンネルを指定します。

機能:

Full UART チャンネルを無効にします。

戻り値:

なし

## 8.2.3.3 FUART\_GetRxData

受信データの取得

関数のプロトタイプ宣言:

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

引数:

**FUARTx**: Full UART チャンネルを指定します。

機能:

受信データを取得します。

本 API は、**FUART\_GetStorageStatus(FUARTx, FUART\_RX)**の戻り値が **FUART\_STORAGE\_NORMAL** あるいは **FUART\_STORAGE\_FULL** の場合に使用してください。

戻り値:

受信データ

## 8.2.3.4 FUART\_SetTxData

送信データの設定

関数のプロトタイプ宣言:

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

引数:

**FUARTx**: Full UART チャンネルを指定します。

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

機能:



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

戻り値:

なし

### 8.2.3.5 FUART\_GetErrStatus

受信エラーステータスの取得

関数のプロトタイプ宣言:

FUART\_Err

FUART\_GetErrStatus(TSB\_FUART\_TypeDef \* **FUARTx**)

引数:

**FUARTx**: Full UART チャンネルを指定します。

機能:

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

戻り値:

**FUART\_NO\_ERR**: エラーはありません

**FUART\_OVERRUN**: オーバーランエラー

**FUART\_PARITY\_ERR**: パリティエラー

**FUART\_FRAMING\_ERR**: フレミングエラー

**FUART\_BREAK\_ERR**: ブレークエラー

**FUART\_ERRS**: 2 つ以上のエラー

### 8.2.3.6 FUART\_ClearErrStatus

受信エラーステータスのクリア

関数のプロトタイプ宣言:

void

FUART\_ClearErrStatus(TSB\_FUART\_TypeDef \* **FUARTx**)

引数:

**FUARTx**: Full UART チャンネルを指定します。

機能:

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

戻り値:

なし

### 8.2.3.7 FUART\_GetBusyState

データ送信状態の取得

**関数のプロトタイプ宣言:**

WorkState

FUART\_GetBusyState(TSB\_FUART\_TypeDef \* **FUARTx**)

**引数:**

**FUARTx**: Full UART チャンネルを指定します。

**機能:**

データ送信中であるか停止中であるか状態を取得します。

**戻り値:**

データ送信状態:

**BUSY**: データ送信中

**DONE**: データ送信が停止中

## 8.2.3.8 FUART\_GetStorageStatus

送受信 FIFO または送受信保持レジスタの取得

**関数のプロトタイプ宣言:**

FUART\_StorageStatus

FUART\_GetStorageStatus(TSB\_FUART\_TypeDef \* **FUARTx**,  
FUART\_Direction **Direction**)

**引数:**

**FUARTx**: Full UART チャンネルを指定します。

**Direction**: 送信または受信のどちらかを選択します。

➤ **FUART\_RX**: 受信 FIFO または受信保持レジスタ

➤ **FUART\_TX**: 送信 FIFO または送信保持レジスタ

**機能:**

FIFO が許可されている場合、送受信 FIFO のステータスを取得します。

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

**戻り値:**

**FUART\_STORAGE\_EMPTY**: FIFO または保持レジスタが empty 状態

**FUART\_STORAGE\_NORMAL**: FIFO または保持レジスタが正常状態

**FUART\_STORAGE\_INVALID**: FIFO または保持レジスタが無効状態

**FUART\_STORAGE\_FULL**: FIFO または保持レジスタが full 状態

## 8.2.3.9 FUART\_SetIrDADivisor

IrDA 低電力除数の設定

**関数のプロトタイプ宣言:**

void

FUART\_SetIrDADivisor(TSB\_FUART\_TypeDef \* **FUARTx**,  
uint32\_t **Divisor**)

**引数:**

**FUARTx**: Full UART チャンネルを指定します。

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

**機能:**

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

本 API をコールする前に、IrDA 回路を許可してください。

**戻り値:**

なし

## 8.2.3.10 FUART\_Init

Full UART チャンネルの設定

**関数のプロトタイプ宣言:**

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

**引数:**

**FUARTx**: Full UART チャンネルを指定します。

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

**機能:**

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

**戻り値:**

なし

## 8.2.3.11 FUART\_EnableFIFO

送受信 FIFO の有効化

**関数のプロトタイプ宣言:**

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

**引数:**

**FUARTx**: Full UART チャンネルを指定します。

**機能:**

送受信 FIFO を許可します。

**戻り値:**

なし

## 8.2.3.12 FUART\_DisableFIFO

送受信 FIFO の無効化

関数のプロトタイプ宣言:

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

引数:

**FUARTx**: Full UART チャンネルを指定します。

機能:

送受信 FIFO を禁止し、モードをキャラクタモードに変更します。

戻り値:

なし

## 8.2.3.13 FUART\_SetSendBreak

ブレーク付き送信の選択

関数のプロトタイプ宣言:

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

引数:

**FUARTx**: Full UART チャンネルを指定します。

**NewState**: ブレーク付き送信の許可/禁止を選択します。

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

機能:

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

戻り値:

なし

## 8.2.3.14 FUART\_SetIrDAEncodeMode

IrDA SIR 低電力モードの設定

関数のプロトタイプ宣言:

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

引数:

**FUARTx**: Full UART チャンネルを指定します。

**Mode**: IrDA SIR 低電力モードを選択します。

- **FUART\_IRDA\_3\_16\_BIT\_PERIOD\_MODE**: ノーマルモード
- **FUART\_IRDA\_3\_TIMES\_IRLPBAUD16\_MODE**: 低電力モード

**機能:**

IrDA SIR 低電力モードを設定します。IrDA SIR 低電力モードとして

**FUART\_IRDA\_3\_TIMES\_IRLPBAUD16\_MODE** を選択すると、消費電力を軽減できますが、送信距離が短くなる可能性があります。

**戻り値:**

なし

### 8.2.3.15 FUART\_EnableIrDA

SIR 許可

**関数のプロトタイプ宣言:**

Result

FUART\_EnableIrDA(TSB\_FUART\_TypeDef \* **FUARTx**)

**引数:**

**FUARTx**: Full UART チャンネルを指定します。

**機能:**

IrDA 回路が許可されます。UART が無効の場合、本 API は何もせず、戻り値がエラーとなります。

**戻り値:**

**SUCCESS**: 成功

**ERROR**: 失敗

### 8.2.3.16 FUART\_DisableIrDA

SIR 禁止

**関数のプロトタイプ宣言:**

void

FUART\_DisableIrDA(TSB\_FUART\_TypeDef \* **FUARTx**)

**引数:**

**FUARTx**: Full UART チャンネルを指定します。

**機能:**

Full UART が許可の場合、本 API は IrDA を禁止にします。UART が無効の場合、本 API は何もせず、戻り値がエラーとなります。

**戻り値:**

なし

### 8.2.3.17 FUART\_SetINTFIFOLevel

送受信割り込み FIFO レベルの選択

## 関数のプロトタイプ宣言:

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

## 引数:

**FUARTx**: Full UART チャンネルを指定します。

**RxLevel**: 受信割り込み FIFO レベルを選択します。(受信 FIFO は 32 段です)

- **FUART\_RX\_FIFO\_LEVEL\_4**: 受信 FIFO が 4 バイト以上
- **FUART\_RX\_FIFO\_LEVEL\_8**: 受信 FIFO が 8 バイト以上
- **FUART\_RX\_FIFO\_LEVEL\_16**: 受信 FIFO が 16 バイト以上
- **FUART\_RX\_FIFO\_LEVEL\_24**: 受信 FIFO が 24 バイト以上
- **FUART\_RX\_FIFO\_LEVEL\_28**: 受信 FIFO が 28 バイト以上

**TxLevel**: 送信割り込み FIFO レベルを選択します。(送信 FIFO は 32 段です)

- **FUART\_TX\_FIFO\_LEVEL\_4**: 送信 FIFO が 4 バイト以上
- **FUART\_TX\_FIFO\_LEVEL\_8**: 送信 FIFO が 8 バイト以上
- **FUART\_TX\_FIFO\_LEVEL\_16**: 送信 FIFO が 16 バイト以上
- **FUART\_TX\_FIFO\_LEVEL\_24**: 送信 FIFO が 24 バイト以上
- **FUART\_TX\_FIFO\_LEVEL\_28**: 送信 FIFO が 28 バイト以上

## 機能:

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

## 戻り値:

なし

### 8.2.3.18 FUART\_SetINTMask

割り込み発生要因の設定

## 関数のプロトタイプ宣言:

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

## 引数:

**FUARTx**: Full UART チャンネルを指定します。

**IntMaskSrc**: 割り込み発生要因を選択します。

- **FUART\_NONE\_INT\_MASK**: すべての割り込みを禁止します。
- **FUART\_RIN\_MODEM\_INT\_MASK**: RIN モデム割り込みを許可します。
- **FUART\_CTS\_MODEM\_INT\_MASK**: CTS モデム割り込みを許可します。
- **FUART\_DCD\_MODEM\_INT\_MASK**: DCD モデム割り込みを許可します。
- **FUART\_DSR\_MODEM\_INT\_MASK**: DSR モデム割り込みを許可します。
- **FUART\_RX\_FIFO\_INT\_MASK**: 受信割り込みを許可します。
- **FUART\_TX\_FIFO\_INT\_MASK**: 送信割り込みを許可します。
- **FUART\_RX\_TIMEOUT\_INT\_MASK**: 受信タイムアウト割り込みを許可します。
- **FUART\_FRAMING\_ERR\_INT\_MASK**: フレーミングエラー割り込みを許可します。

- **FUART\_PARITY\_ERR\_INT\_MASK**: パリティエラー割り込みを許可します。
- **FUART\_BREAK\_ERR\_INT\_MASK**: ブレークエラー割り込みを許可します。
- **FUART\_OVERRUN\_ERR\_INT\_MASK**: オーバーランエラー割り込みを許可します。
- **FUART\_ALL\_INT\_MASK**: すべての割り込みを許可します。

**機能:**

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

**戻り値:**

なし

### 8.2.3.19 FUART\_GetINTMask

割り込み発生要因の取得

**関数のプロトタイプ宣言:**

FUART\_INTStatus

FUART\_GetINTMask(TSB\_FUART\_TypeDef \* **FUARTx**)

**引数:**

**FUARTx**: Full UART チャンネルを指定します。

**機能:**

割り込み発生 of 許可/禁止状態を要因毎に取得します。

**戻り値:**

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

### 8.2.3.20 FUART\_GetRawINTStatus

割り込み許可/禁止設定前の割り込みステータスの取得

**関数のプロトタイプ宣言:**

FUART\_INTStatus

FUART\_GetRawINTStatus(TSB\_FUART\_TypeDef \* **FUARTx**)

**引数:**

**FUARTx**: Full UART チャンネルを指定します。

**機能:**

割り込み許可/禁止設定前の割り込みステータスを取得します。

**戻り値:**

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

## 8.2.3.21 FUART\_GetMaskedINTStatus

割り込み許可/禁止設定後の割り込みステータスの取得

関数のプロトタイプ宣言:

FUART\_INTStatus

FUART\_GetMaskedINTStatus(TSB\_FUART\_TypeDef \* **FUARTx**)

引数:

**FUARTx**: Full UART チャンネルを指定します。

機能:

割り込み許可/禁止設定後の割り込みステータスを取得します。

戻り値:

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

## 8.2.3.22 FUART\_ClearINT

割り込み要因のクリア

関数のプロトタイプ宣言:

void

FUART\_ClearINT(TSB\_FUART\_TypeDef \* **FUARTx**,  
FUART\_INTStatus **INTStatus**)

引数:

**FUARTx**: Full UART チャンネルを指定します。

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

機能:

割り込み要因をクリアします。

戻り値:

なし

## 8.2.3.23 FUART\_SetDMAOnErr

DMA オンエラーの許可/禁止選択

関数のプロトタイプ宣言:

void

FUART\_SetDMAOnErr(TSB\_FUART\_TypeDef \* **FUARTx**,  
FunctionalState **NewState**)

引数:

**FUARTx**: Full UART チャンネルを指定します。

**NewState**: DMA オンエラーの許可/禁止を選択します。

- **ENABLE**: 許可。
- **DISABLE**: 禁止。



**機能:**

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

**戻り値:**

なし

## 8.2.3.24 FUART\_SetFIFODMA

送受信 DMA の許可/禁止選択

**関数のプロトタイプ宣言:**

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

**引数:**

**FUARTx**: Full UART チャンネルを指定します。

**Direction**: 送信または受信のどちらかを選択します。

➤ **FUART\_RX**: 受信 FIFO または受信保持レジスタ

➤ **FUART\_TX**: 送信 FIFO または送信保持レジスタ

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

➤ **ENABLE**: 許可。

➤ **DISABLE**: 禁止。

**機能:**

送信 DMA または受信 DMA の許可/禁止を選択します。

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

**戻り値:**

なし

## 8.2.3.25 FUART\_GetModemStatus

モデム状態の取得

**関数のプロトタイプ宣言:**

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

**引数:**

**FUARTx**: Full UART チャンネルを指定します。

**機能:**

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

**戻り値:**

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

## 8.2.3.26 FUART\_SetRTSStatus

Full UART の RTS(送信要求)モデム状態の設定

**関数のプロトタイプ宣言:**

void

FUART\_SetRTSStatus(TSB\_FUART\_TypeDef \* **FUARTx**,  
FUART\_ModemStatus **Status**)

**引数:**

**FUARTx:** Full UART チャンネルを指定します。

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

- **FUART\_MODEM\_STATUS\_0:** モデムステータス出力を 0 にします。
- **FUART\_MODEM\_STATUS\_1:** モデムステータス出力を 1 にします。

**機能:**

Full UART の RTS(送信要求)モデム状態を設定します。

**戻り値:**

なし

## 8.2.3.27 FUART\_SetDTRStatus

Full UART DTR(データ送信準備完了)状態の設定

**関数のプロトタイプ宣言:**

void

FUART\_SetDTRStatus(TSB\_FUART\_TypeDef \* **FUARTx**,  
FUART\_ModemStatus **Status**)

**引数:**

**FUARTx:** Full UART チャンネルを指定します。

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

- **FUART\_MODEM\_STATUS\_0:** モデムステータス出力を 0 にします。
- **FUART\_MODEM\_STATUS\_1:** モデムステータス出力を 1 にします。

**機能:**

Full UART DTR(データ送信準備完了)状態を設定します。

**戻り値:**

なし

## 8.2.4 データ構造

### 8.2.4.1 FUART\_InitTypeDef

**メンバ:**

uint32\_t

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

uint32\_t

**DataBits**: フレームで送受信されたデータビットの数を設定します。

- **UART\_DATA\_BITS\_5**: 5bit
- **UART\_DATA\_BITS\_6**: 6bit
- **UART\_DATA\_BITS\_7**: 7bit
- **UART\_DATA\_BITS\_8**: 8bit

uint32\_t

**StopBits**: 送信ストップビット長を設定します。

- **UART\_STOP\_BITS\_1**: 1bit
- **UART\_STOP\_BITS\_2**: 2bit

uint32\_t

**Parity**: パリティ状態を設定します。

- **UART\_NO\_PARITY**: パリティの送信およびチェックなし
- **UART\_0\_PARITY**: パリティビットとして"0"を送信または受信
- **UART\_1\_PARITY**: パリティビットとして"1"を送信または受信
- **UART\_EVEN\_PARITY**: パリティビットとして偶数パリティを送信または受信
- **UART\_ODD\_PARITY**: パリティビットとして奇数パリティを送信または受信

uint32\_t

**Mode**: 受信、送信あるいは両方の許可/禁止を設定します。

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

uint32\_t

**FlowCtrl**: ハードウェアフロー制御を設定します。

- **UART\_NONE\_FLOW\_CTRL**: フロー制御なし
- **UART\_CTS\_FLOW\_CTRL**: CTS フロー制御許可
- **UART\_RTS\_FLOW\_CTRL**: RTS フロー制御許可
- **UART\_CTS\_FLOW\_CTRL | UART\_RTS\_FLOW\_CTRL**: CTS/RTS フロー制御許可

## 8.2.4.2 FUART\_INTStatus

メンバ:

uint32\_t

**All**: Full UART 割り込みステータス、または割り込み制御

ビットフィールド:

uint32\_t

**RIN**: 1                      RIN モデム割り込み

uint32\_t

**CTS**: 1                      CTS モデム割り込み

uint32\_t

**DCD**: 1                      DCD モデム割り込み

uint32\_t  
**DSR**: 1            DSR モデム割り込み

uint32\_t  
**RxFIFO**: 1        受信 FIFO 割り込み

uint32\_t  
**TxFIFO**: 1        送信 FIFO 割り込み

uint32\_t  
**RxTimeout**: 1 受信タイムアウト割り込み

uint32\_t  
**FramingErr**: 1    フレーミングエラー割り込み

uint32\_t  
**ParityErr**: 1    パリティエラー割り込み

uint32\_t  
**BreakErr**: 1    ブレークエラー割り込み

uint32\_t  
**OverrunErr**: 1    オーバーランエラー割り込み

uint32\_t  
**Reserved**: 21 未使用

## 8.2.4.3 FUART\_AllModemStatus

メンバ:

uint32\_t  
**All**: Full UART の全モデムステータス

ビットフィールド:

uint32\_t  
**CTS**: 1            CTS モデムステータス

uint32\_t  
**DSR**: 1            DSR モデムステータス

uint32\_t  
**DCD**: 1            DCD モデムステータス

uint32\_t  
**Reserved1**: 5 未使用

uint32\_t  
**RI**: 1            RIN モデムステータス

uint32\_t  
**Reserved2**: 1 未使用

uint32\_t  
**DTR**: 1            DTR モデムステータス

uint32\_t

**RTS:** 1                      RTS モデムステータス

uint32\_t

**Reserved3:** 20      未使用

## 9. GPIO

### 9.1 概要

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

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

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

/Libraries/TX04\_Periph\_Driver/src/tmpm462\_gpio.c

/Libraries/TX04\_Periph\_Driver/inc/tmpm462\_gpio.h

### 9.2 API 関数

#### 9.2.1 関数一覧

- uint16\_t GPIO\_ReadData(GPIO\_Port **GPIO\_x**)
- uint16\_t GPIO\_ReadDataBit(GPIO\_Port **GPIO\_x**, uint16\_t **Bit\_x**)
- void GPIO\_WriteData(GPIO\_Port **GPIO\_x**, uint16\_t **Data**)
- void GPIO\_WriteDataBit(GPIO\_Port **GPIO\_x**, uint16\_t **Bit\_x**, uint16\_t **BitValue**)
- void GPIO\_Init(GPIO\_Port **GPIO\_x**, uint16\_t **Bit\_x**,  
GPIO\_InitTypeDef \* **GPIO\_InitStruct**)
- void GPIO\_SetOutput(GPIO\_Port **GPIO\_x**, uint16\_t **Bit\_x**)
- void GPIO\_SetInput(GPIO\_Port **GPIO\_x**, uint16\_t **Bit\_x**);
- void GPIO\_SetOutputEnableReg(GPIO\_Port **GPIO\_x**, uint16\_t **Bit\_x**,  
FunctionalState **NewState**)
- void GPIO\_SetInputEnableReg(GPIO\_Port **GPIO\_x**, uint16\_t **Bit\_x**,  
FunctionalState **NewState**)
- void GPIO\_SetPullUp(GPIO\_Port **GPIO\_x**, uint16\_t **Bit\_x**,  
FunctionalState **NewState**)
- void GPIO\_SetPullDown(GPIO\_Port **GPIO\_x**, uint16\_t **Bit\_x**,  
FunctionalState **NewState**)
- void GPIO\_SetOpenDrain(GPIO\_Port **GPIO\_x**, uint16\_t **Bit\_x**,  
FunctionalState **NewState**)
- void GPIO\_EnableFuncReg(GPIO\_Port **GPIO\_x**, uint16\_t **FuncReg\_x**, uint16\_t **Bit\_x**)
- void GPIO\_DisableFuncReg(GPIO\_Port **GPIO\_x**, uint16\_t **FuncReg\_x**, uint16\_t **Bit\_x**)

#### 9.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています:

- 1) 入出力ポートへの書き込み/読み出し:  
GPIO\_ReadData(), GPIO\_ReadDataBit(), GPIO\_WriteData(), GPIO\_WriteDataBit()
- 2) 入出力ポートの初期化と設定:  
GPIO\_SetOutput(), GPIO\_SetInput(), GPIO\_SetOutputEnableReg(),  
GPIO\_SetInputEnableReg(), GPIO\_SetPullUp(), GPIO\_SetPullDown(),  
GPIO\_SetOpenDrain(), GPIO\_Init()

- 3) その他:  
GPIO\_EnableFuncReg(), GPIO\_DisableFuncReg()

## 9.2.3 関数仕様

### 9.2.3.1 GPIO\_ReadData

DATA データレジスタの読み込み

関数のプロトタイプ宣言:

```
uint16_t  
GPIO_ReadData(GPIO_Port GPIO_x)
```

引数:

**GPIO\_x**: GPIO ポートを選択します。

- **GPIO\_PA**: GPIO port A
- **GPIO\_PB**: GPIO port B
- **GPIO\_PC**: GPIO port C
- **GPIO\_PD**: GPIO port D
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PH**: GPIO port H
- **GPIO\_PJ**: GPIO port J
- **GPIO\_PK**: GPIO port K
- **GPIO\_PL**: GPIO port L
- **GPIO\_PM**: GPIO port M
- **GPIO\_PN**: GPIO port N

機能:

DATA レジスタを読み込みます。

戻り値:

DATA レジスタの値

### 9.2.3.2 GPIO\_ReadDataBit

ビット単位での DATA レジスタの読み込み

関数のプロトタイプ宣言:

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

引数:

**GPIO\_x**: GPIO ポートを選択します。

- **GPIO\_PA**: GPIO port A
- **GPIO\_PB**: GPIO port B
- **GPIO\_PC**: GPIO port C
- **GPIO\_PD**: GPIO port D
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PH**: GPIO port H
- **GPIO\_PJ**: GPIO port J

- **GPIO\_PK:** GPIO port K
- **GPIO\_PL:** GPIO port L
- **GPIO\_PM:** GPIO port M
- **GPIO\_PN:** GPIO port N

**Bit\_x:** GPIO 端子を選択します。

- **GPIO\_BIT\_0:** GPIO pin 0
- **GPIO\_BIT\_1:** GPIO pin 1
- **GPIO\_BIT\_2:** GPIO pin 2
- **GPIO\_BIT\_3:** GPIO pin 3
- **GPIO\_BIT\_4:** GPIO pin 4
- **GPIO\_BIT\_5:** GPIO pin 5
- **GPIO\_BIT\_6:** GPIO pin 6
- **GPIO\_BIT\_7:** GPIO pin 7
- **GPIO\_BIT\_8:** GPIO pin 8
- **GPIO\_BIT\_9:** GPIO pin 9
- **GPIO\_BIT\_10:** GPIO pin 10
- **GPIO\_BIT\_11:** GPIO pin 11
- **GPIO\_BIT\_12:** GPIO pin 12
- **GPIO\_BIT\_13:** GPIO pin 13
- **GPIO\_BIT\_14:** GPIO pin 14
- **GPIO\_BIT\_15:** GPIO pin 15

**機能:**

ビット単位で DATA データレジスタを読み込みます。

**戻り値:**

GPIO 端子値

- **GPIO\_BIT\_VALUE\_0:** 0
- **GPIO\_BIT\_VALUE\_1:** 1

### 9.2.3.3 GPIO\_WriteData

DATA レジスタへの書き込み

**関数のプロトタイプ宣言:**

void

GPIO\_WriteData(GPIO\_Port **GPIO\_x**,  
uint16\_t **Data**)

**引数:**

**GPIO\_x:** GPIO ポートを選択します。

- **GPIO\_PA:** GPIO port A
- **GPIO\_PB:** GPIO port B
- **GPIO\_PC:** GPIO port C
- **GPIO\_PD:** GPIO port D
- **GPIO\_PE:** GPIO port E
- **GPIO\_PF:** GPIO port F
- **GPIO\_PG:** GPIO port G
- **GPIO\_PH:** GPIO port H
- **GPIO\_PJ:** GPIO port J.
- **GPIO\_PK:** GPIO port K
- **GPIO\_PL:** GPIO port L
- **GPIO\_PM:** GPIO port M
- **GPIO\_PN:** GPIO port N

**Data:** DATA レジスタへのライトデータを指定します。



**機能:**

DATA レジスタへ指定された値を書き込みます。

**戻り値:**

なし

## 9.2.3.4 GPIO\_WriteDataBit

ビット単位での DATA レジスタの書き込み

**関数のプロトタイプ宣言:**

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

**引数:**

**GPIO\_x**: GPIO ポートを選択します。

- **GPIO\_PA**: GPIO port A
- **GPIO\_PB**: GPIO port B
- **GPIO\_PC**: GPIO port C
- **GPIO\_PD**: GPIO port D
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PH**: GPIO port H
- **GPIO\_PJ**: GPIO port J
- **GPIO\_PK**: GPIO port K
- **GPIO\_PL**: GPIO port L
- **GPIO\_PM**: GPIO port M
- **GPIO\_PN**: GPIO port N

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

- **GPIO\_BIT\_0**: GPIO pin 0
- **GPIO\_BIT\_1**: GPIO pin 1
- **GPIO\_BIT\_2**: GPIO pin 2
- **GPIO\_BIT\_3**: GPIO pin 3
- **GPIO\_BIT\_4**: GPIO pin 4
- **GPIO\_BIT\_5**: GPIO pin 5
- **GPIO\_BIT\_6**: GPIO pin 6
- **GPIO\_BIT\_7**: GPIO pin 7
- **GPIO\_BIT\_8**: GPIO pin 8
- **GPIO\_BIT\_9**: GPIO pin 9
- **GPIO\_BIT\_10**: GPIO pin 10
- **GPIO\_BIT\_11**: GPIO pin 11
- **GPIO\_BIT\_12**: GPIO pin 12
- **GPIO\_BIT\_13**: GPIO pin 13
- **GPIO\_BIT\_14**: GPIO pin 14
- **GPIO\_BIT\_15**: GPIO pin 15
- **GPIO\_BIT\_ALL**: GPIO pin[0:15]

**BitValue**: 設定ビットを指定します。

- **GPIO\_BIT\_VALUE\_0**: 0

➤ **GPIO\_BIT\_VALUE\_1:** 1

**機能:**

ビット単位で DATA データレジスタを書き込みます。

**戻り値:**

なし

## 9.2.3.5 GPIO\_Init

GPIO ポートの初期設定

**関数のプロトタイプ宣言:**

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

**引数:**

**GPIO\_x:** GPIO ポートを選択します。

- **GPIO\_PA:** GPIO port A
- **GPIO\_PB:** GPIO port B
- **GPIO\_PC:** GPIO port C
- **GPIO\_PD:** GPIO port D
- **GPIO\_PE:** GPIO port E
- **GPIO\_PF:** GPIO port F
- **GPIO\_PG:** GPIO port G
- **GPIO\_PH:** GPIO port H
- **GPIO\_PJ:** GPIO port J
- **GPIO\_PK :** GPIO port K
- **GPIO\_PL:** GPIO port L
- **GPIO\_PM:** GPIO port M
- **GPIO\_PN:** GPIO port N

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

- **GPIO\_BIT\_0:** GPIO pin 0
- **GPIO\_BIT\_1:** GPIO pin 1
- **GPIO\_BIT\_2:** GPIO pin 2
- **GPIO\_BIT\_3:** GPIO pin 3
- **GPIO\_BIT\_4:** GPIO pin 4
- **GPIO\_BIT\_5:** GPIO pin 5
- **GPIO\_BIT\_6:** GPIO pin 6
- **GPIO\_BIT\_7:** GPIO pin 7
- **GPIO\_BIT\_8:** GPIO pin 8
- **GPIO\_BIT\_9:** GPIO pin 9
- **GPIO\_BIT\_10:** GPIO pin 10
- **GPIO\_BIT\_11:** GPIO pin 11
- **GPIO\_BIT\_12:** GPIO pin 12
- **GPIO\_BIT\_13:** GPIO pin 13
- **GPIO\_BIT\_14:** GPIO pin 14
- **GPIO\_BIT\_15:** GPIO pin 15
- **GPIO\_BIT\_ALL:** GPIO pin[0:15]

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

**機能:**

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

**戻り値:**

なし

## 9.2.3.6 GPIO\_SetOutput

出力ポートの設定

**関数のプロトタイプ宣言:**

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

**引数:**

**GPIO\_x**: GPIO ポートを選択します。

- **GPIO\_PA**: GPIO port A
- **GPIO\_PB**: GPIO port B
- **GPIO\_PC**: GPIO port C
- **GPIO\_PD**: GPIO port D
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PH**: GPIO port H
- **GPIO\_PJ**: GPIO port J
- **GPIO\_PK**: GPIO port K
- **GPIO\_PL**: GPIO port L
- **GPIO\_PM**: GPIO port M
- **GPIO\_PN**: GPIO port N

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

- **GPIO\_BIT\_0**: GPIO pin 0
- **GPIO\_BIT\_1**: GPIO pin 1
- **GPIO\_BIT\_2**: GPIO pin 2
- **GPIO\_BIT\_3**: GPIO pin 3
- **GPIO\_BIT\_4**: GPIO pin 4
- **GPIO\_BIT\_5**: GPIO pin 5
- **GPIO\_BIT\_6**: GPIO pin 6
- **GPIO\_BIT\_7**: GPIO pin 7
- **GPIO\_BIT\_8**: GPIO pin 8
- **GPIO\_BIT\_9**: GPIO pin 9
- **GPIO\_BIT\_10**: GPIO pin 10
- **GPIO\_BIT\_11**: GPIO pin 11
- **GPIO\_BIT\_12**: GPIO pin 12
- **GPIO\_BIT\_13**: GPIO pin 13
- **GPIO\_BIT\_14**: GPIO pin 14
- **GPIO\_BIT\_15**: GPIO pin 15
- **GPIO\_BIT\_ALL**: GPIO pin[0:15]

**機能:**

出力ポートに設定します。

戻り値:  
なし

## 9.2.3.7 GPIO\_SetInput

入力ポートの設定

関数のプロトタイプ宣言:

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

引数:

**GPIO\_x**: GPIO ポートを選択します。

- **GPIO\_PA**: GPIO port A
- **GPIO\_PB**: GPIO port B
- **GPIO\_PC**: GPIO port C
- **GPIO\_PD**: GPIO port D
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PH**: GPIO port H
- **GPIO\_PJ**: GPIO port J
- **GPIO\_PK**: GPIO port K
- **GPIO\_PL**: GPIO port L
- **GPIO\_PM**: GPIO port M
- **GPIO\_PN**: GPIO port N

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

- **GPIO\_BIT\_0**: GPIO pin 0
- **GPIO\_BIT\_1**: GPIO pin 1
- **GPIO\_BIT\_2**: GPIO pin 2
- **GPIO\_BIT\_3**: GPIO pin 3
- **GPIO\_BIT\_4**: GPIO pin 4
- **GPIO\_BIT\_5**: GPIO pin 5
- **GPIO\_BIT\_6**: GPIO pin 6
- **GPIO\_BIT\_7**: GPIO pin 7
- **GPIO\_BIT\_8**: GPIO pin 8
- **GPIO\_BIT\_9**: GPIO pin 9
- **GPIO\_BIT\_10**: GPIO pin 10
- **GPIO\_BIT\_11**: GPIO pin 11
- **GPIO\_BIT\_12**: GPIO pin 12
- **GPIO\_BIT\_13**: GPIO pin 13
- **GPIO\_BIT\_14**: GPIO pin 14
- **GPIO\_BIT\_15**: GPIO pin 15
- **GPIO\_BIT\_ALL**: GPIO pin[0:15]

機能:

入力ポートに設定します。

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

戻り値:  
なし

## 9.2.3.8 GPIO\_SetOutputEnableReg

出力ポートの許可/禁止設定

関数のプロトタイプ宣言:

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

引数:

**GPIO\_x**: GPIO ポートを選択します。

- **GPIO\_PA**: GPIO port A
- **GPIO\_PB**: GPIO port B
- **GPIO\_PC**: GPIO port C
- **GPIO\_PD**: GPIO port D
- **GPIO\_PE**: GPIO port E
- **GPIO\_PF**: GPIO port F
- **GPIO\_PG**: GPIO port G
- **GPIO\_PH**: GPIO port H
- **GPIO\_PJ**: GPIO port J
- **GPIO\_PK**: GPIO port K
- **GPIO\_PL**: GPIO port L
- **GPIO\_PM**: GPIO port M
- **GPIO\_PN**: GPIO port N

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

- **GPIO\_BIT\_0**: GPIO pin 0
- **GPIO\_BIT\_1**: GPIO pin 1
- **GPIO\_BIT\_2**: GPIO pin 2
- **GPIO\_BIT\_3**: GPIO pin 3
- **GPIO\_BIT\_4**: GPIO pin 4
- **GPIO\_BIT\_5**: GPIO pin 5
- **GPIO\_BIT\_6**: GPIO pin 6
- **GPIO\_BIT\_7**: GPIO pin 7
- **GPIO\_BIT\_8**: GPIO pin 8
- **GPIO\_BIT\_9**: GPIO pin 9
- **GPIO\_BIT\_10**: GPIO pin 10
- **GPIO\_BIT\_11**: GPIO pin 11
- **GPIO\_BIT\_12**: GPIO pin 12
- **GPIO\_BIT\_13**: GPIO pin 13
- **GPIO\_BIT\_14**: GPIO pin 14
- **GPIO\_BIT\_15**: GPIO pin 15
- **GPIO\_BIT\_ALL**: GPIO pin[0:15]

**NewState**:

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

機能:

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

戻り値:  
なし

## 9.2.3.9 GPIO\_SetInputEnableReg

入力ポートの許可/禁止設定

関数のプロトタイプ宣言:

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

引数:

**GPIO\_x**: GPIO ポートを選択します。

- **GPIO\_PA**: GPIO port A.
- **GPIO\_PB**: GPIO port B.
- **GPIO\_PC**: GPIO port C.
- **GPIO\_PD**: GPIO port D.
- **GPIO\_PE**: GPIO port E.
- **GPIO\_PF**: GPIO port F.
- **GPIO\_PG**: GPIO port G.
- **GPIO\_PH**: GPIO port H.
- **GPIO\_PJ**: GPIO port J.
- **GPIO\_PK**: GPIO port K.
- **GPIO\_PL**: GPIO port L.
- **GPIO\_PM**: GPIO port M.
- **GPIO\_PN**: GPIO port N.

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

- **GPIO\_BIT\_0**: GPIO pin 0,
- **GPIO\_BIT\_1**: GPIO pin 1,
- **GPIO\_BIT\_2**: GPIO pin 2,
- **GPIO\_BIT\_3**: GPIO pin 3,
- **GPIO\_BIT\_4**: GPIO pin 4,
- **GPIO\_BIT\_5**: GPIO pin 5,
- **GPIO\_BIT\_6**: GPIO pin 6,
- **GPIO\_BIT\_7**: GPIO pin 7,
- **GPIO\_BIT\_8**: GPIO pin 8,
- **GPIO\_BIT\_9**: GPIO pin 9,
- **GPIO\_BIT\_10**: GPIO pin 10,
- **GPIO\_BIT\_11**: GPIO pin 11,
- **GPIO\_BIT\_12**: GPIO pin 12,
- **GPIO\_BIT\_13**: GPIO pin 13,
- **GPIO\_BIT\_14**: GPIO pin 14,
- **GPIO\_BIT\_15**: GPIO pin 15,
- **GPIO\_BIT\_ALL**: GPIO pin[0:15],

**NewState**:

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

機能:

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

戻り値:  
なし

## 9.2.3.10 GPIO\_SetPullUp

内蔵プルアップの設定

関数のプロトタイプ宣言:

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

引数:

**GPIO\_x**: GPIO ポートを選択します。

- **GPIO\_PA**: GPIO port A.
- **GPIO\_PB**: GPIO port B.
- **GPIO\_PC**: GPIO port C.
- **GPIO\_PD**: GPIO port D.
- **GPIO\_PE**: GPIO port E.
- **GPIO\_PF**: GPIO port F.
- **GPIO\_PG**: GPIO port G.
- **GPIO\_PH**: GPIO port H.
- **GPIO\_PJ**: GPIO port J.
- **GPIO\_PK**: GPIO port K.
- **GPIO\_PL**: GPIO port L.
- **GPIO\_PM**: GPIO port M.
- **GPIO\_PN**: GPIO port N.

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

- **GPIO\_BIT\_0**: GPIO pin 0,
- **GPIO\_BIT\_1**: GPIO pin 1,
- **GPIO\_BIT\_2**: GPIO pin 2,
- **GPIO\_BIT\_3**: GPIO pin 3,
- **GPIO\_BIT\_4**: GPIO pin 4,
- **GPIO\_BIT\_5**: GPIO pin 5,
- **GPIO\_BIT\_6**: GPIO pin 6,
- **GPIO\_BIT\_7**: GPIO pin 7,
- **GPIO\_BIT\_8**: GPIO pin 8,
- **GPIO\_BIT\_9**: GPIO pin 9,
- **GPIO\_BIT\_10**: GPIO pin 10,
- **GPIO\_BIT\_11**: GPIO pin 11,
- **GPIO\_BIT\_12**: GPIO pin 12,
- **GPIO\_BIT\_13**: GPIO pin 13,
- **GPIO\_BIT\_14**: GPIO pin 14,
- **GPIO\_BIT\_15**: GPIO pin 15,
- **GPIO\_BIT\_ALL**: GPIO pin[0:15],

**NewState**:

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

機能:

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

戻り値:  
なし

## 9.2.3.11 GPIO\_SetPullDown

内蔵プルダウンの設定

関数のプロトタイプ宣言:

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

引数:

**GPIO\_x**: GPIO ポートを選択します。

➤ **GPIO\_PE**: GPIO port E.

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

➤ **GPIO\_BIT\_4**: GPIO pin 4,

➤ **GPIO\_BIT\_ALL**: GPIO pin[0:15],

**NewState**:

➤ **ENABLE**: 内蔵プルダウン有効

➤ **DISABLE**: 内蔵プルダウン無効

機能:

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

戻り値:  
なし

## 9.2.3.12 GPIO\_SetOpenDrain

CMOS/オープンドレインポートの設定

関数のプロトタイプ宣言:

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

引数:

**GPIO\_x**: GPIO ポートを選択します。

➤ **GPIO\_PA**: GPIO port A.

➤ **GPIO\_PB**: GPIO port B.

➤ **GPIO\_PC**: GPIO port C.

➤ **GPIO\_PD**: GPIO port D.

➤ **GPIO\_PE**: GPIO port E.

➤ **GPIO\_PF**: GPIO port F.

➤ **GPIO\_PG**: GPIO port G.

➤ **GPIO\_PK**: GPIO port K.

➤ **GPIO\_PL**: GPIO port L.

➤ **GPIO\_PM**: GPIO port M.

➤ **GPIO\_PN**: GPIO port N.



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

- **GPIO\_BIT\_0:** GPIO pin 0,
- **GPIO\_BIT\_1:** GPIO pin 1,
- **GPIO\_BIT\_2:** GPIO pin 2,
- **GPIO\_BIT\_3:** GPIO pin 3,
- **GPIO\_BIT\_4:** GPIO pin 4,
- **GPIO\_BIT\_5:** GPIO pin 5,
- **GPIO\_BIT\_6:** GPIO pin 6,
- **GPIO\_BIT\_7:** GPIO pin 7,
- **GPIO\_BIT\_8:** GPIO pin 8,
- **GPIO\_BIT\_9:** GPIO pin 9,
- **GPIO\_BIT\_10:** GPIO pin 10,
- **GPIO\_BIT\_11:** GPIO pin 11,
- **GPIO\_BIT\_12:** GPIO pin 12,
- **GPIO\_BIT\_13:** GPIO pin 13,
- **GPIO\_BIT\_14:** GPIO pin 14,
- **GPIO\_BIT\_15:** GPIO pin 15,
- **GPIO\_BIT\_ALL:** GPIO pin[0:15],

**NewState:**

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

**機能:**

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

**戻り値:**

なし

### 9.2.3.13 GPIO\_EnableFuncReg

機能ポートの有効設定

**関数のプロトタイプ宣言:**

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

**引数:**

**GPIO\_x:** GPIO ポートを選択します。

- **GPIO\_PA:** GPIO port A
- **GPIO\_PB:** GPIO port B
- **GPIO\_PC:** GPIO port C
- **GPIO\_PD:** GPIO port D
- **GPIO\_PE:** GPIO port E
- **GPIO\_PF:** GPIO port F
- **GPIO\_PG:** GPIO port G
- **GPIO\_PJ:** GPIO port J
- **GPIO\_PK :** GPIO port K
- **GPIO\_PL:** GPIO port L
- **GPIO\_PM:** GPIO port M
- **GPIO\_PN:** GPIO port N.

**FuncReg\_x:** GPIO 機能レジスタの番号を選択します。

- **GPIO\_FUNC\_REG\_1** GPIO 機能レジスタ 1
- **GPIO\_FUNC\_REG\_2** GPIO 機能レジスタ 2
- **GPIO\_FUNC\_REG\_3** GPIO 機能レジスタ 3
- **GPIO\_FUNC\_REG\_4** GPIO 機能レジスタ 4
- **GPIO\_FUNC\_REG\_5** GPIO 機能レジスタ 5

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

- **GPIO\_BIT\_0:** GPIO pin 0
- **GPIO\_BIT\_1:** GPIO pin 1
- **GPIO\_BIT\_2:** GPIO pin 2
- **GPIO\_BIT\_3:** GPIO pin 3
- **GPIO\_BIT\_4:** GPIO pin 4
- **GPIO\_BIT\_5:** GPIO pin 5
- **GPIO\_BIT\_6:** GPIO pin 6
- **GPIO\_BIT\_7:** GPIO pin 7
- **GPIO\_BIT\_8:** GPIO pin 8
- **GPIO\_BIT\_9:** GPIO pin 9
- **GPIO\_BIT\_10:** GPIO pin 10
- **GPIO\_BIT\_11:** GPIO pin 11
- **GPIO\_BIT\_12:** GPIO pin 12
- **GPIO\_BIT\_13:** GPIO pin 13
- **GPIO\_BIT\_14:** GPIO pin 14
- **GPIO\_BIT\_15:** GPIO pin 15
- **GPIO\_BIT\_ALL:** GPIO pin[0:15]

**機能:**

GPIO 端子の機能を有効に設定します。

**戻り値:**

なし

## 9.2.3.14 GPIO\_DisableFuncReg

機能ポートの無効設定

**関数のプロトタイプ宣言:**

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

**引数:**

**GPIO\_x:** GPIO ポートを選択します。

- **GPIO\_PA:** GPIO port A.
- **GPIO\_PB:** GPIO port B.
- **GPIO\_PC:** GPIO port C.
- **GPIO\_PD:** GPIO port D.
- **GPIO\_PE:** GPIO port E.
- **GPIO\_PF:** GPIO port F.
- **GPIO\_PG:** GPIO port G.
- **GPIO\_PJ:** GPIO port J.
- **GPIO\_PK :** GPIO port K.
- **GPIO\_PL:** GPIO port L.

➤ **GPIO\_PM:** GPIO port M.

➤ **GPIO\_PN:** GPIO port N.

**FuncReg\_x:** GPIO 機能レジスタの番号を選択します。

➤ **GPIO\_FUNC\_REG\_1** GPIO 機能レジスタ 1

➤ **GPIO\_FUNC\_REG\_2** GPIO 機能レジスタ 2

➤ **GPIO\_FUNC\_REG\_3** GPIO 機能レジスタ 3

➤ **GPIO\_FUNC\_REG\_4** GPIO 機能レジスタ 4

➤ **GPIO\_FUNC\_REG\_5** GPIO 機能レジスタ 5

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

➤ **GPIO\_BIT\_0:** GPIO pin 0,

➤ **GPIO\_BIT\_1:** GPIO pin 1,

➤ **GPIO\_BIT\_2:** GPIO pin 2,

➤ **GPIO\_BIT\_3:** GPIO pin 3,

➤ **GPIO\_BIT\_4:** GPIO pin 4,

➤ **GPIO\_BIT\_5:** GPIO pin 5,

➤ **GPIO\_BIT\_6:** GPIO pin 6,

➤ **GPIO\_BIT\_7:** GPIO pin 7.

➤ **GPIO\_BIT\_8:** GPIO pin 8,

➤ **GPIO\_BIT\_9:** GPIO pin 9,

➤ **GPIO\_BIT\_10:** GPIO pin 10,

➤ **GPIO\_BIT\_11:** GPIO pin 11,

➤ **GPIO\_BIT\_12:** GPIO pin 12,

➤ **GPIO\_BIT\_13:** GPIO pin 13,

➤ **GPIO\_BIT\_14:** GPIO pin 14,

➤ **GPIO\_BIT\_15:** GPIO pin 15,

➤ **GPIO\_BIT\_ALL:** GPIO pin[0:15],

**機能:**

GPIO 端子の機能を無効に設定します。

**戻り値:**

なし

## 9.2.4 データ構造

### 9.2.4.1 GPIO\_InitTypeDef

**メンバ:**

**uint16\_t**

**IOMode** ポートの入出力を選択します。

➤ **GPIO\_INPUT:** 入力ポートに設定します。

➤ **GPIO\_OUTPUT:** 出力ポートに設定します。

➤ **GPIO\_IO\_MODE\_NONE:** 入出力モードを変更しません。

**uint16\_t**

**PullUp** 内蔵プルアップの有効/無効を選択します。

➤ **GPIO\_PULLUP\_ENABLE:** 内蔵プルアップを有効にします。

➤ **GPIO\_PULLUP\_DISABLE:** 内蔵プルアップを無効にします。

➤ **GPIO\_PULLUP\_NONE:** 内蔵プルアップ機能がない、または設定変更しません。

**uint16\_t**

**OpenDrain** オープンドレインポート/CMOSポートを選択します。

➤ **GPIO\_OPEN\_DRAIN\_ENABLE:** オープンドレインポートに設定

- **GPIO\_OPEN\_DRAIN\_DISABLE**: CMOS ポートに設定
- **GPIO\_OPEN\_DRAIN\_NONE**: オープンドレイン機能がない、または設定変更しません。

uint16\_t

**PullDown** 内蔵プルダウンの有効/無効を選択します。

- **GPIO\_PULLDOWN\_ENABLE**: 内蔵プルダウンを有効にします。
- **GPIO\_PULLDOWN\_DISABLE**: 内蔵プルダウンを無効にします。
- **GPIO\_PULLDOWN\_NONE**: 内蔵プルダウンがない、または設定変更しません。

## 9.2.4.2 GPIO\_RegTypeDef

メンバ:

uint16\_t

**PinDATA** DATA レジスタのマスク値

uint16\_t

**PinCR** CR レジスタのマスク値

uint16\_t

**PinFR[FRMAX]** FR レジスタのマスク値

uint16\_t

**PinOD** OD レジスタのマスク値

uint16\_t

**PinPUP** PUP レジスタのマスク値

uint16\_t

**PinPDN** PDN レジスタのマスク値

uint16\_t

**PinPIE** IE レジスタのマスク値

## 9.2.4.3 TSB\_Port\_TypeDef

メンバ:

\_\_IO uint32\_t

**DATA** DATA レジスタのリードデータまたはライトデータです。

\_\_IO uint32\_t

**PinCR** CR レジスタのリードデータまたはライトデータです。

\_\_IO uint32\_t

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

uint32\_t

**RESERVED0[RESER]** 未使用

\_\_IO uint32\_t

**PinOD** OD レジスタのリードデータまたはライトデータです。

\_\_IO uint32\_t

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

\_\_IO uint32\_t

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

uint32\_t

**RESERVED1[RESER]** 未使用

\_\_IO uint32\_t

**PinPIE** IE レジスタのリードデータまたはライトデータです。

## 10. I2C

### 10.1 概要

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

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

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

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

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

/Libraries/TX04\_Periph\_Driver/src/tmpm462\_i2c.c

/Libraries/TX04\_Periph\_Driver/inc/tmpm462\_i2c.h

### 10.2 API 関数

#### 10.2.1 関数一覧

- ◆ void I2C\_SetACK(TSB\_I2C\_TypeDef\* **I2Cx**, FunctionalState **NewState**);
- ◆ void I2C\_Init(TSB\_I2C\_TypeDef\* **I2Cx**, I2C\_InitTypeDef\* **InitI2CStruct**);
- ◆ void I2C\_SetBitNum(TSB\_I2C\_TypeDef\* **I2Cx**, uint32\_t **I2CBitNum**);
- ◆ void I2C\_SWReset(TSB\_I2C\_TypeDef\* **I2Cx**);
- ◆ void I2C\_ClearINTReq(TSB\_I2C\_TypeDef\* **I2Cx**);
- ◆ void I2C\_GenerateStart(TSB\_I2C\_TypeDef\* **I2Cx**);
- ◆ void I2C\_GenerateStop(TSB\_I2C\_TypeDef\* **I2Cx**);
- ◆ I2C\_State I2C\_GetState(TSB\_I2C\_TypeDef\* **I2Cx**);
- ◆ void I2C\_SetSendData(TSB\_I2C\_TypeDef\* **I2Cx**, uint32\_t **Data**);
- ◆ uint32\_t I2C\_GetReceiveData(TSB\_I2C\_TypeDef\* **I2Cx**);
- ◆ void I2C\_SetFreeDataMode(TSB\_I2C\_TypeDef\* **I2Cx**, FunctionalState **NewState**);
- ◆ FunctionalState I2C\_GetSlaveAddrMatchState(TSB\_I2C\_TypeDef \* **I2Cx**);
- ◆ void I2C\_SetPrescalerClock(TSB\_I2C\_TypeDef \* **I2Cx**, uint32\_t **PrescalerClock**);
- ◆ void I2C\_SetINTReq(TSB\_I2C\_TypeDef \* **I2Cx**,FunctionalState **NewState**);
- ◆ FunctionalState I2C\_GetINTStatus(TSB\_I2C\_TypeDef \* **I2Cx**);
- ◆ void I2C\_ClearINTOutput(TSB\_I2C\_TypeDef \* **I2Cx**);

#### 10.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています:

- 1) 共通設定:  
I2C\_SetACK(), I2C\_SetBitNum(), I2C\_SetPrescalerClock(), I2C\_Init()
- 2) 転送設定:  
I2C\_ClearINTReq(), I2C\_Generatstart(), I2C\_Generatestop(), I2C\_SetSendData(),  
I2C\_GetReceiveData(), I2C\_SetINTReq(), I2C\_ClearINTOutput()
- 3) ステータスの取得:  
I2C\_GetState(), I2C\_GetSlaveAddrMatchState(), I2C\_GetINTStatus()
- 4) その他:  
I2C\_SWReset(), I2C\_SetFreeDataMode()

## 10.2.3 関数仕様

\*補足: 下記の全 API において、パラメータ "TSB\_I2C\_TypeDef\* **I2Cx**" は、以下のいずれかを指定してください。

TSB\_I2C0, TSB\_I2C1, TSB\_I2C2, TSB\_I2C3, TSB\_I2C4

### 10.2.3.1 I2C\_SetACK

ACK クロックの発生/停止

関数のプロトタイプ宣言:

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

引数:

**I2Cx**: I2C チャンネルを指定します。

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

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

機能:

ACK のためのクロックを発生する／発生しないを選択します。

戻り値:

なし

### 10.2.3.2 I2C\_Init

I2C の初期化

関数のプロトタイプ宣言:

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

引数:

**I2Cx**: I2C チャンネルを指定します。

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

機能:

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

戻り値:

なし

### 10.2.3.3 I2C\_SetBitNum

転送ビット数の設定

関数のプロトタイプ宣言:

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

**引数:**

**I2Cx**: I2C チャンネルを指定します。

**I2CBitNum**: 転送ビット数を選択します。

- **I2C\_DATA\_LEN\_8**: データ長は 8
- **I2C\_DATA\_LEN\_1**: データ長は 1
- **I2C\_DATA\_LEN\_2**: データ長は 2
- **I2C\_DATA\_LEN\_3**: データ長は 3
- **I2C\_DATA\_LEN\_4**: データ長は 4
- **I2C\_DATA\_LEN\_5**: データ長は 5
- **I2C\_DATA\_LEN\_6**: データ長は 6
- **I2C\_DATA\_LEN\_7**: データ長は 7

**機能:**

転送ビット数を選択します。

**戻り値:**

なし

## 10.2.3.4 I2C\_SWReset

ソフトウェアリセットの発生

**関数のプロトタイプ宣言:**

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

**引数:**

**I2Cx**: I2C チャンネルを指定します。

**機能:**

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

**戻り値:**

なし

## 10.2.3.5 I2C\_ClearINTReq

INTI2C 割込み要求の解除

**関数のプロトタイプ宣言:**

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

**引数:**

**I2Cx**: I2C チャンネルを指定します。



**機能:**

INTI2C 割込み要求を解除します。

**戻り値:**

なし

## 10.2.3.6 I2C\_GenerateStart

マスタモードの選択とスタートコンディションの発生

**関数のプロトタイプ宣言:**

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

**引数:**

**I2Cx**: I2C チャンネルを指定します。

**機能:**

マスタモードを選択し、スタートコンディションを発生します。

**戻り値:**

なし

## 10.2.3.7 I2C\_GenerateStop

マスタモードの選択とストップコンディションの発生

**関数のプロトタイプ宣言:**

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

**引数:**

**I2Cx**: I2C チャンネルを指定します。

**機能:**

マスタモードを選択し、ストップコンディションを発生します。

**戻り値:**

なし

## 10.2.3.8 I2C\_GetState

I2C バスステータスの取得

**関数のプロトタイプ宣言:**

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

**引数:**

**I2Cx**: I2C チャンネルを指定します。

**機能:**

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

**戻り値:**

I2C バスステータス

## 10.2.3.9 I2C\_SetSendData

送信データの設定と送信開始

**関数のプロトタイプ宣言:**

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

**引数:**

**I2Cx**: I2C チャンネルを指定します。

**Data**: 送信データを設定します。送信データの最大値は 0xFF です。

**機能:**

送信データの設定と送信を開始します。

**戻り値:**

なし

## 10.2.3.10 I2C\_GetReceiveData

受信データの取得

**関数のプロトタイプ宣言:**

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

**引数:**

**I2Cx**: I2C チャンネルを指定します。

**機能:**

受信データを取得します。

**戻り値:**

受信データ

## 10.2.3.11 I2C\_SetFreeDataMode

I2C フリーデータフォーマットの設定

**関数のプロトタイプ宣言:**

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

**引数:**

**I2Cx:** I2C チャンネルを指定します。

**NewState:** システムが IDLE モードの場合に以下の状態を選択します。

- **ENABLE:** スレーブアドレスを認識しない(フリーデータフォーマット)。
- **DISABLE:** スレーブアドレスを認識する。

**機能:**

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

**戻り値:**

なし

### 10.2.3.12 I2C\_GetSlaveAddrMatchState

スレーブアドレス一致検出およびゼネラルコール検出選択状態の取得

**関数のプロトタイプ宣言:**

FunctionalState

I2C\_GetSlaveAddrMatchState(TSB\_I2C\_TypeDef\* **I2Cx**)

**引数:**

**I2Cx:** I2C チャンネルを指定します。

**機能:**

スレーブアドレス一致検出およびゼネラルコール検出選択状態を取得します。

**戻り値:**

スレーブアドレス一致検出およびゼネラルコール検出選択状態:

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

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

### 10.2.3.13 I2C\_SetPrescalerClock

内部 SCL 出力クロックの周波数選択

**関数のプロトタイプ宣言:**

void

I2C\_SetPrescalerClock(TSB\_I2C\_TypeDef\* **I2Cx**,  
uint32\_t **PrescalerClock**)

**引数:**

**I2Cx:** I2C チャンネルを指定します。

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

- **I2C\_PRESCALER\_DIV\_1 ~ I2C\_PRESCALER\_DIV\_32**

**機能:**

内部 SCL 出力クロックの周波数を選択します。

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

戻り値:

なし

## 10.2.3.14 I2C\_SetINTReq

I2C 割込み出力の許可/禁止の設定

関数のプロトタイプ宣言:

```
void  
I2C_SetINTReq(TSB_I2C_TypeDef* I2Cx,  
              FunctionalState NewState)
```

引数:

**I2Cx**: I2C チャンネルを指定します。

**NewState**: I2C 割込み出力の許可/禁止を選択します。

- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:

I2C 割込み出力の許可/禁止を選択します。

戻り値:

なし

## 10.2.3.15 I2C\_GetINTStatus

I2C 割込み状態の取得

関数のプロトタイプ宣言:

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

引数:

**I2Cx**: I2C チャンネルを指定します。

機能:

I2C 割込み状態を取得します。

戻り値:

I2C 割込み状態:

**ENABLE**: 割込み発生

**DISABLE**: 割込みなし

## 10.2.3.16 I2C\_ClearINTOutput

I2C 割込みのクリア

**関数のプロトタイプ宣言:**

void  
I2C\_ClearINTOutput(TSB\_I2C\_TypeDef\* **I2Cx**)

**引数:**

**I2Cx**: I2C チャンネルを指定します。

**機能:**

I2C 割込み出力(INTI2C)をクリアします。

**戻り値:**

なし

## 10.2.4 データ構造

### 10.2.4.1 I2C\_InitTypeDef

**メンバ:**

uint32\_t

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

uint32\_t

**I2CDataLen** 送信ビット数を選択します。

- **I2C\_DATA\_LEN\_8**: データ長 8
- **I2C\_DATA\_LEN\_1**: データ長 1
- **I2C\_DATA\_LEN\_2**: データ長 2
- **I2C\_DATA\_LEN\_3**: データ長 3
- **I2C\_DATA\_LEN\_4**: データ長 4
- **I2C\_DATA\_LEN\_5**: データ長 5
- **I2C\_DATA\_LEN\_6**: データ長 6
- **I2C\_DATA\_LEN\_7**: データ長 7

uint32\_t

**I2CClkDiv**: プリスケールクロックの分周値を選択します。

- **I2C\_SCK\_CLK\_DIV\_20**: シリアルクロックは fprsck を 20 で割った商の値です。
- **I2C\_SCK\_CLK\_DIV\_24**: シリアルクロックは fprsck を 24 で割った商の値です。
- **I2C\_SCK\_CLK\_DIV\_32**: シリアルクロックは fprsck を 32 で割った商の値です。
- **I2C\_SCK\_CLK\_DIV\_48**: シリアルクロックは fprsck を 48 で割った商の値です。
- **I2C\_SCK\_CLK\_DIV\_80**: シリアルクロックは fprsck を 80 で割った商の値です。
- **I2C\_SCK\_CLK\_DIV\_144**: シリアルクロックは fprsck を 144 で割った商の値です。
- **I2C\_SCK\_CLK\_DIV\_272**: シリアルクロックは fprsck を 272 で割った商の値です。
- **I2C\_SCK\_CLK\_DIV\_528**: シリアルクロックは fprsck を 528 で割った商の値です。

uint32\_t

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

- **I2C\_PRESCALER\_DIV\_1**: fprsck は、fsys を 1 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_2**: fprsck は、fsys を 2 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_3**: fprsck は、fsys を 3 で割った商の値です。

- **I2C\_PRESCALER\_DIV\_4:** fprsck は、fsys を 4 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_5:** fprsck は、fsys を 5 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_6:** fprsck は、fsys を 6 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_7:** fprsck は、fsys を 7 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_8:** fprsck は、fsys を 8 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_9:** fprsck は、fsys を 9 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_10:** fprsck は、fsys を 10 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_11:** fprsck は、fsys を 11 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_12:** fprsck は、fsys を 12 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_13:** fprsck は、fsys を 13 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_14:** fprsck は、fsys を 14 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_15:** fprsck は、fsys を 15 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_16:** fprsck は、fsys を 16 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_17:** fprsck は、fsys を 17 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_18:** fprsck は、fsys を 18 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_19:** fprsck は、fsys を 19 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_20:** fprsck は、fsys を 20 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_21:** fprsck は、fsys を 21 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_22:** fprsck は、fsys を 22 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_23:** fprsck は、fsys を 23 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_24:** fprsck は、fsys を 24 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_25:** fprsck は、fsys を 25 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_26:** fprsck は、fsys を 26 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_27:** fprsck は、fsys を 27 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_28:** fprsck は、fsys を 28 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_29:** fprsck は、fsys を 29 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_30:** fprsck は、fsys を 30 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_31:** fprsck は、fsys を 31 で割った商の値です。
- **I2C\_PRESCALER\_DIV\_32:** fprsck は、fsys を 32 で割った商の値です。

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

## FunctionalState

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

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

### 10.2.4.2 I2C\_State

メンバ:

uint32\_t

All: すべての状態です。

ビットフィールド:

uint32\_t

**LastRxBit:** 最終受信ビットモニタ

uint32\_t

**GeneralCall:** ゼネラルコール検出モニタ

uint32\_t

**SlaveAddrMatch:** スレーブアドレス一致検出モニタ

uint32\_t

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

uint32\_t

**INTReq:** INTI2C 割込み要求状態モニタ

uint32\_t

**BusState:** I2C バス状態モニタ

uint32\_t

**TRx:** トランスミッタ/レシーバ選択状態モニタ

uint32\_t

**MasterSlave:** マスタ/スレーブ選択状態モニタ

## 11. IGBT

### 11.1 概要

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

IGBT モードには、次のような機能があります。

- 1) 16ビットプログラマブル矩形波出力モード(PPG、2 相)
- 2) 外部トリガスタート
- 3) 周期一致検出機能
- 4) 緊急停止機能
- 5) 同期スタートモード

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

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

/Libraries/TX04\_Periph\_Driver/src/tmpm462\_igbt.c  
/Libraries/TX04\_Periph\_Driver/inc/ tmpm462\_igbt.h

### 11.2 API 関数

#### 11.2.1 関数一覧

- ◆ void IGBT\_Enable(TSB\_MT\_TypeDef\* **IGBTx**)
- ◆ void IGBT\_Disable(TSB\_MT\_TypeDef\* **IGBTx**)
- ◆ void IGBT\_SetClkInCoreHalt(TSB\_MT\_TypeDef\* **IGBTx**, uint8\_t **ClkState**)
- ◆ void IGBT\_SetSWRunState(TSB\_MT\_TypeDef\* **IGBTx**, uint8\_t **Cmd**)
- ◆ uint16\_t IGBT\_GetCaptureValue(TSB\_MT\_TypeDef\* **IGBTx**, uint8\_t **CapReg**)
- ◆ void IGBT\_Init(TSB\_MT\_TypeDef\* **IGBTx**, IGBT\_InitTypeDef\* **InitStruct**)
- ◆ void IGBT\_Recount(TSB\_MT\_TypeDef\* **IGBTx**)
- ◆ void IGBT\_ChangeOutputActiveTiming(TSB\_MT\_TypeDef\* **IGBTx**, uint8\_t **Output**,  
uint16\_t **Timing**)
- ◆ void IGBT\_ChangeOutputInactiveTiming(TSB\_MT\_TypeDef\* **IGBTx**, uint8\_t **Output**,  
uint16\_t **Timing**)
- ◆ void IGBT\_ChangePeriod(TSB\_MT\_TypeDef\* **IGBTx**, uint16\_t **Period**)
- ◆ WorkState IGBT\_GetCntState(TSB\_MT\_TypeDef\* **IGBTx**)
- ◆ Result IGBT\_CancelEMGState(TSB\_MT\_TypeDef\* **IGBTx**)
- ◆ IGBT\_EMGStateTypeDef IGBT\_GetEMGState(TSB\_MT\_TypeDef \* **IGBTx**)
- ◆ void IGBT\_ChangeTrgValue(TSB\_MT\_TypeDef\* **IGBTx**, uint16\_t **uTrgCnt**)
- ◆ void IGBT\_CISynSlaveChCounter(TSB\_MT\_TypeDef \* **IGBTx**)

#### 11.2.2 関数の種類

関数は、主に以下の 5 種類に分かれています:

- 1) IGBT の初期化:  
IGBT\_Enable(), IGBT\_Disable(), IGBT\_Init()



- 2) カウンタ状態の取得と IGBT の設定:  
IGBT\_SetCkInCoreHalt(), IGBT\_SetSWRunState(), IGBT\_Recount(),  
IGBT\_GetCntState()
- 3) 動作パラメータの変更と IGBT 値の取得:  
IGBT\_GetCaptureValue(), IGBT\_ChangeOutputActiveTiming(),  
IGBT\_ChangeOutputInactiveTiming(), IGBT\_ChangePeriod()
- 4) EMG 保護状態の取得とキャンセル:  
IGBT\_GetEMGState(), IGBT\_CancelEMGState()
- 5) トリガ値の変更と同期クリア:  
IGBT\_ChangeTrgValue(), IGBT\_CISynSlaveChCounter()

## 11.2.3 関数仕様

\*補足: 下記の全 API において、パラメータ“TSB\_MT\_TypeDef\* **IGBTx**”は、以下のいずれかの値となります。:

**IGBT0, IGBT1.**

### 11.2.3.1 IGBT\_Enable

IGBT モードの許可

関数のプロトタイプ宣言:

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

引数:

**IGBTx**: IGBT モードでの MPT チャンネルを指定します。

機能:

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

戻り値:

なし

### 11.2.3.2 IGBT\_Disable

IGBT モードの禁止

関数のプロトタイプ宣言:

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

引数:

**IGBTx**: IGBT モードでの MPT チャンネルを指定します。

機能:

**IGBTx** により選択された指定 MPT チャンネルをディセーブルにします。

戻り値:

なし

## 11.2.3.3 IGBT\_SetClkInCoreHalt

IGBT モードにおけるコア Halt 時の動作指定

関数のプロトタイプ宣言:

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

引数:

**IGBTx**: IGBT モードでの MPT チャンネルを指定します。

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

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

機能:

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

戻り値:

なし

## 11.2.3.4 IGBT\_SetSWRunState

IGBT モードにおけるカウント動作制御

関数のプロトタイプ宣言:

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

引数:

**IGBTx**: IGBT モードでの MPT チャンネルを指定します。

**Cmd**: カウントの開始/停止を選択します。

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

機能:

カウント動作の開始/停止を選択します。

戻り値:

なし

補足:

- 1) 実際のカウンタ開始または停止のタイミングは、IGBT モードの設定により異なります。**IGBT\_CMD\_START** または **IGBT\_CMD\_START\_NO\_START\_INT** が **StartMode**(詳細はデータ構成説明を参照)として選択されている場合、

本 API は、カウンタを完全に制御することが可能です。**StartMode**として他の値が設定されている場合、トリガもカウンタを制御することができます。

- 2) **IGBT\_FALLING\_TRG\_START** または **IGBT\_RISING\_TRG\_START** が **StartMode**(詳細はデータ構成説明を参照)として選択されている場合、初期化と設定が完了すると、ソフトウェアはコマンドを発行します。  
**IGBT\_SetSWRunState(IGBTx, IGBT\_RUN)**は、トリガにより開始される前に発行してください。その後トリガによるカウンタの開始が可能になります。
- 3) EMG 入力 Low レベルで、EMG 割り込みが発生した場合、カウンタの停止には **IGBT\_SetSWRunState(IGBTx, IGBT\_STOP)** を使用してください。

### 11.2.3.5 IGBT\_GetCaptureValue

IGBT モードにおけるキャプチャカウンタの取得

関数のプロトタイプ宣言:

```
uint16_t  
IGBT_GetCaptureValue(TSB_MT_TypeDef* IGBTx,  
                     uint8_t CapReg)
```

引数:

**IGBTx**: IGBT モードでの MPT チャ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。

- **IGBT\_CAPTURE\_0**: キャプチャレジスタ 0
- **IGBT\_CAPTURE\_1**: キャプチャレジスタ 1

機能:

現在のアップカウンタの値をキャプチャすることができます。

戻り値:

アップカウンタの値

補足:

カウンタ値は、**StartMode**(詳細はデータ構成説明を参照)として

**IGBT\_CMD\_START** または **IGBT\_CMD\_START\_NO\_START\_INT** が選択されたときのみ、本関数の呼び出しにより、取得されます。最初の入力エッジのタイミングで取得された値は、キャプチャレジスタ 0 に格納されます。2 番目の入力エッジで取得された値は、キャプチャレジスタ 1 に格納されます。

### 11.2.3.6 IGBT\_Init

IGBT モードにおける MPT チャンネルの初期化と設定

関数のプロトタイプ宣言:

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

引数:

**IGBTx**: IGBT モードでの MPT チャンネルを指定します。

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

**機能:**

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

**戻り値:**

なし

**補足:**

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

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

### 11.2.3.7 IGBT\_Recount

カウントリスタート

**関数のプロトタイプ宣言:**

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

**引数:**

**IGBTx:** IGBT モードでの MPT チャンネルを指定します。

**機能:**

カウンタのクリアとリスタートを行います。

**戻り値:**

なし

### 11.2.3.8 IGBT\_ChangeOutputActiveTiming

IGBT 出力 0/1 のアクティブタイミングの変更

**関数のプロトタイプ宣言:**

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

**引数:**

**IGBTx:** IGBT モードでの MPT チャンネルを指定します。

**Output:** IGBT 出力ポートを選択します。

- **IGBT\_OUTPUT\_0:** IGBT 出力ポート 0
- **IGBT\_OUTPUT\_1:** IGBT 出力ポート 1

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

**機能:**

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

**戻り値:**

なし

### 11.2.3.9 IGBT\_ChangeOutputInactiveTiming

IGBT 出力 0/1 のインアクティブタイミングの変更。

**関数のプロトタイプ宣言:**

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

**引数:**

**IGBTx:** IGBT モードでの MPT チャンネルを指定します。

**Output:** IGBT 出力ポートを選択します。

- **IGBT\_OUTPUT\_0:** IGBT 出力ポート 0
- **IGBT\_OUTPUT\_1:** IGBT 出力ポート 1

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

**機能:**

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

**戻り値:**

なし

### 11.2.3.10 IGBT\_ChangePeriod

IGBT 出力周期の変更

**関数のプロトタイプ宣言:**

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

**引数:**

**IGBTx:** IGBT モードでの MPT チャンネルを指定します。

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

**機能:**

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

戻り値:

なし

## 11.2.3.11 IGBT\_GetCntState

カウンタ状態の取得

関数のプロトタイプ宣言:

WorkState

IGBT\_GetCntState(TSB\_MT\_TypeDef\* **IGBTx**)

引数:

**IGBTx**: IGBT モードでの MPT チャンネルを指定します。

機能:

カウンタ状態を取得します。

戻り値:

カウンタ状態は、以下のようになります。

**BUSY**: カウンタ動作中

**DONE**: カウンタ停止中

## 11.2.3.12 IGBT\_CancelEMGState

IGBT モードにおける EMG 状態のキャンセル

関数のプロトタイプ宣言:

Result

IGBT\_CancelEMGState(TSB\_MT\_TypeDef\* **IGBTx**)

引数:

**IGBTx**: IGBT モードでの MPT チャンネルを指定します。

機能:

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

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

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

戻り値:

EMG 状態キャンセルの結果を返却します。

**SUCCESS**: キャンセルの成功

**ERROR**: キャンセルの失敗

## 11.2.3.13 IGBT\_GetEMGState

IGBT の EMG 状態の取得

**関数のプロトタイプ宣言:**

IGBT\_EMGStateTypeDef

IGBT\_GetEMGState(TSB\_MT\_TypeDef \* **IGBTx**)

**引数:**

**IGBTx:** IGBT モードでの MPT チャンネルを指定します。

**機能:**

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

**戻り値:**

**IGBT\_EMGStateTypeDef:** EMG ステータス (詳細は、データ構造説明を参照)

## 11.2.3.14 IGBT\_ChangeTrgValue

IGBT 出力のタイマカウンタ値設定

**関数のプロトタイプ宣言:**

void

IGBT\_ChangeTrgValue(TSB\_MT\_TypeDef\***IGBTx**,uint16\_t **uTrgCnt**)

**引数:**

**IGBTx:** IGBT モードでの MPT チャンネルを指定します。

**uTrgCnt:** タイマカウンタ値を設定します。

**機能:**

IGBT 出力のタイマカウンタ値を設定します。

**戻り値:**

なし

## 11.2.3.15 IGBT\_CISynSlaveChCounter

同期動作のアップカウンタクリア

**関数のプロトタイプ宣言:**

void

IGBT\_CISynSlaveChCounter(TSB\_MT\_TypeDef \* **IGBTx**)

**引数:**

**IGBTx:** IGBT モードでの MPT チャンネルを指定します。

**機能:**

同期動作のアップカウンタをクリアします。

**戻り値:**

なし

## 11.2.4 データ構造

### 11.2.4.1 IGBT\_InitTypeDef

メンバ:

uint8\_t

**StartMode:** カウンタのスタートモードを選択します。

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

uint8\_t

**OperationMode:** IGBT 動作モードを選択します。

- **IGBT\_CONTINUOUS\_OUTPUT:** 連続動作
- **IGBT\_ONE\_TIME\_OUTPUT:** 単発動作

uint8\_t

**CntStopState:** カウンタ停止時の出力状態を指定します。

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

FunctionalState

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

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

uint8\_t

**INTPeriod:** 割り込み周期を選択します。

- **IGBT\_INT\_PERIOD\_1:** 1 周期毎
- **IGBT\_INT\_PERIOD\_2:** 2 周期毎



- **IGBT\_INT\_PERIOD\_4**: 3 周期毎

uint8\_t

**ClkDiv**: IGBT のソースクロックを選択します。

- **IGBT\_CLK\_DIV\_1**:  $\phi T0(\phi T0/1)$
- **IGBT\_CLK\_DIV\_2**:  $\phi T1(\phi T0/2)$
- **IGBT\_CLK\_DIV\_4**:  $\phi T2(\phi T0/4)$
- **IGBT\_CLK\_DIV\_8**:  $\phi T4(\phi T0/8)$

uint8\_t

**Output0Init**: IGBT 出力 0 の初期化をします。

- **IGBT\_OUTPUT\_DISABLE**: IGBT 出力をディセーブルにします
- **IGBT\_OUTPUT\_HIGH\_ACTIVE**: 初期出力は Low レベルです。High レベルはアクティブ出力です
- **IGBT\_OUTPUT\_LOW\_ACTIVE**: 初期出力は High レベルです。Low レベルはアクティブ出力です

uint8\_t

**Output1Init**: IGBT 出力 1 の初期化をします。

- **IGBT\_OUTPUT\_DISABLE**: IGBT 出力をディセーブルにします
- **IGBT\_OUTPUT\_HIGH\_ACTIVE**: 初期出力は Low レベルです。High レベルはアクティブ出力です
- **IGBT\_OUTPUT\_LOW\_ACTIVE**: 初期出力は High レベルです。Low レベルはアクティブ出力です

uint8\_t

**TrgDenoiseDiv**: IGBT モードでのトリガ入力ノイズ除去時間を選択します。

- **IGBT\_NO\_DENOISE**: ノイズフィルタを経由しません
- **IGBT\_DENOISE\_DIV\_16**: ノイズ除去時間 16 / fsys[s]
- **IGBT\_DENOISE\_DIV\_32**: ノイズ除去時間 32 / fsys[s]
- **IGBT\_DENOISE\_DIV\_48**: ノイズ除去時間 48 / fsys[s]
- **IGBT\_DENOISE\_DIV\_64**: ノイズ除去時間 64 / fsys[s]
- **IGBT\_DENOISE\_DIV\_80**: ノイズ除去時間 80 / fsys[s]
- **IGBT\_DENOISE\_DIV\_96**: ノイズ除去時間 96 / fsys[s]
- **IGBT\_DENOISE\_DIV\_112**: ノイズ除去時間 112 / fsys[s]
- **IGBT\_DENOISE\_DIV\_128**: ノイズ除去時間 128 / fsys[s]
- **IGBT\_DENOISE\_DIV\_144**: ノイズ除去時間 144 / fsys[s]
- **IGBT\_DENOISE\_DIV\_160**: ノイズ除去時間 160 / fsys[s]
- **IGBT\_DENOISE\_DIV\_176**: ノイズ除去時間 176 / fsys[s]
- **IGBT\_DENOISE\_DIV\_192**: ノイズ除去時間 192 / fsys[s]
- **IGBT\_DENOISE\_DIV\_208**: ノイズ除去時間 208 / fsys[s]
- **IGBT\_DENOISE\_DIV\_224**: ノイズ除去時間 224 / fsys[s]
- **IGBT\_DENOISE\_DIV\_240**: ノイズ除去時間 240 / fsys[s]

uint16\_t

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

uint16\_t

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

uint16\_t

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

uint16\_t

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

uint16\_t

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

uint16\_t

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

uint8\_t

**EMGFunction:** EMG 停止機能を指定します。

- **IGBT\_DISABLE\_EMG:** IGBT の EMG 停止機能をディセーブルにします。
- **IGBT\_EMG\_OUTPUT\_INACTIVE:** EMG 状態中 IGBT 出力をインアクティブレベルにします。
- **IGBT\_EMG\_OUTPUT\_HIZ:** EMG 状態中の IGBT 出力を Hi-z にします。

uint8\_t

**EMGDenoiseDiv:** IGBT モードでの EMG 入力用ノイズ除去分周時間を選択します。

- **IGBT\_NO\_DENOISE:** ノイズ除去時間なし
- **IGBT\_DENOISE\_DIV\_16:** ノイズ除去時間 16 / fsys[s]
- **IGBT\_DENOISE\_DIV\_32:** ノイズ除去時間 32 / fsys[s]
- **IGBT\_DENOISE\_DIV\_48:** ノイズ除去時間 48 / fsys[s]
- **IGBT\_DENOISE\_DIV\_64:** ノイズ除去時間 64 / fsys[s]
- **IGBT\_DENOISE\_DIV\_80:** ノイズ除去時間 80 / fsys[s]
- **IGBT\_DENOISE\_DIV\_96:** ノイズ除去時間 96 / fsys[s]
- **IGBT\_DENOISE\_DIV\_112:** ノイズ除去時間 112 / fsys[s]
- **IGBT\_DENOISE\_DIV\_128:** ノイズ除去時間 128 / fsys[s]
- **IGBT\_DENOISE\_DIV\_144:** ノイズ除去時間 144 / fsys[s]
- **IGBT\_DENOISE\_DIV\_160:** ノイズ除去時間 160 / fsys[s]
- **IGBT\_DENOISE\_DIV\_176:** ノイズ除去時間 176 / fsys[s]
- **IGBT\_DENOISE\_DIV\_192:** ノイズ除去時間 192 / fsys[s]
- **IGBT\_DENOISE\_DIV\_208:** ノイズ除去時間 208 / fsys[s]
- **IGBT\_DENOISE\_DIV\_224:** ノイズ除去時間 224 / fsys[s]
- **IGBT\_DENOISE\_DIV\_240:** ノイズ除去時間 240 / fsys[s]

**\*補足:**

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

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

## 11.2.4.2 IGBT\_EMGStateTypeDef

**メンバ:**

enum

**IGBT\_EMGInputState:** ノイズ除去後の EMG 入力端子のステータスを表示します。

- **IGBT\_EMG\_INPUT\_LOW:** ノイズ除去後 EMG 入力端子が Low。
- **IGBT\_EMG\_INPUT\_HIGH:** ノイズ除去後 EMG 入力端子が High。

enum

**IGBT\_EMGProtectState:** EMG 保護ステータスを表示します。

- **IGBT\_EMG\_NORMAL:** EMG 保護ステータスは、ノーマル動作。
- **IGBT\_EMG\_PROTECT:** EMG 保護ステータスは、保護動作中。

## 12. LVD

### 12.1 概要

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

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

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

/Libraries/TX04\_Periph\_Driver/src/tmpm462\_lvd.c

/Libraries/TX04\_Periph\_Driver/inc/tmpm462\_lvd.h

### 12.2 API 関数

#### 12.2.1 関数一覧

- ◆ void LVD\_EnableVD1(void)
- ◆ void LVD\_DisableVD1(void)
- ◆ void LVD\_SetVD1Level(uint32\_t **VDLevel**)
- ◆ LVD\_VDStatus LVD\_GetVD1Status(void)
- ◆ void LVD\_EnableVD2(void)
- ◆ void LVD\_DisableVD2(void)
- ◆ void LVD\_SetVD2Level(uint32\_t **VDLevel**)
- ◆ LVD\_VDStatus LVD\_GetVD2Status(void)
- ◆ void LVD\_SetVD1ResetOutput(FunctionalState **NewState**)
- ◆ void LVD\_SetVD1INTOutput(FunctionalState **NewState**)
- ◆ void LVD\_SetVD2ResetOutput(FunctionalState **NewState**)
- ◆ void LVD\_SetVD2INTOutput(FunctionalState **NewState**)

#### 12.2.2 関数の種類

関数は、主に以下の 2 種類に分かれています:

- 1) LVD 機能の設定:  
LVD\_EnableVD1(), LVD\_DisableVD1(), LVD\_SetVD1Level(), LVD\_EnableVD2(),  
LVD\_DisableVD2(), LVD\_SetVD2Level(), LVD\_SetVD1ResetOutput(),  
LVD\_SetVD1INTOutput(), LVD\_SetVD2ResetOutput(), LVD\_SetVD2INTOutput()
- 2) 電圧検出状態の確認:  
LVD\_GetVD1Status(), LVD\_GetVD2Status()

#### 12.2.3 関数仕様

##### 12.2.3.1 LVD\_EnableVD1

LVDLVL1 の許可

関数のプロトタイプ宣言:

void  
LVD\_EnableVD1(void)

引数:

なし。

機能:

LVDLVL1 を許可します。

戻り値:

なし

## 12.2.3.2 LVD\_DisableVD1

LVDLVL1 の禁止

関数のプロトタイプ宣言:

void

LVD\_DisableVD1(void)

引数:

なし。

機能:

LVDLVL1 を禁止します。

戻り値:

なし

## 12.2.3.3 LVD\_SetVD1Level

LVDLVL1 用電圧レベルの選択

関数のプロトタイプ宣言:

void

LVD\_SetVD1Level(uint32\_t **VDLevel**)

引数:

**VDLevel**: LVDLVL1 用の電圧レベルです。

- **LVD\_VDLVL1\_240**:  $2.40 \pm 0.1V$
- **LVD\_VDLVL1\_250**:  $2.50 \pm 0.1V$
- **LVD\_VDLVL1\_260**:  $2.60 \pm 0.1V$
- **LVD\_VDLVL1\_270**:  $2.70 \pm 0.1V$
- **LVD\_VDLVL1\_280**:  $2.80 \pm 0.1V$
- **LVD\_VDLVL1\_290**:  $2.90 \pm 0.1V$

機能:

LVDLVL1 用電圧レベルを設定します。

戻り値:

なし

## 12.2.3.4 LVD\_GetVD1Status

LVDLVL1 状態の取得

**関数のプロトタイプ宣言:**

LVD\_VDStatus

LVD\_GetVD1Status(void)

**引数:**

なし。

**機能:**

LVDLVL1 のステータスを取得します。

**戻り値:**

**LVD\_VDStatus:** LVDLVL1 のステータス。

➤ **LVD\_VD\_UPPER:** 電源電圧は検出電圧以上。

➤ **LVD\_VD\_LOWER:** 電源電圧は検出電圧以下。

## 12.2.3.5 LVD\_EnableVD2

LVDLVL2 の許可

**関数のプロトタイプ宣言:**

void

LVD\_EnableVD2(void)

**引数:**

なし。

**機能:**

LVDLVL2 を許可します。

**戻り値:**

なし

## 12.2.3.6 LVD\_DisableVD2

LVDLVL2 の禁止

**関数のプロトタイプ宣言:**

void

LVD\_DisableVD2(void)

**引数:**

なし。

**機能:**

LVDLVL2 を禁止します。

**戻り値:**

なし

## 12.2.3.7 LVD\_SetVD2Level

LVDLVL2 用電圧レベルの選択

関数のプロトタイプ宣言:

```
void  
LVD_SetVD2Level(uint32_t VDLevel)
```

引数:

**VDLevel**: LVDLVL2 用の電圧レベルです。

- **LVD\_VDLVL\_280**:  $2.80 \pm 0.1V$
- **LVD\_VDLVL\_285**:  $2.85 \pm 0.1V$
- **LVD\_VDLVL\_290**:  $2.90 \pm 0.1V$
- **LVD\_VDLVL\_295**:  $2.95 \pm 0.1V$
- **LVD\_VDLVL\_300**:  $3.00 \pm 0.1V$
- **LVD\_VDLVL\_305**:  $3.05 \pm 0.1V$
- **LVD\_VDLVL\_310**:  $3.10 \pm 0.1V$
- **LVD\_VDLVL\_315**:  $3.15 \pm 0.1V$

機能:

LVDLVL2 用電圧レベルを設定します。

戻り値:

なし

## 12.2.3.8 LVD\_GetVD2Status

LVDLVL2 状態の取得

関数のプロトタイプ宣言:

```
LVD_VDStatus  
LVD_GetVD2Status(void)
```

引数:

なし。

機能:

LVDLVL1 のステータスを取得します。

戻り値:

**LVD\_VDStatus**: LVDLVL2 のステータス。

- **LVD\_VD\_UPPER**: 電源電圧は検出電圧以上。
- **LVD\_VD\_LOWER**: 電源電圧は検出電圧以下。

## 12.2.3.9 LVD\_SetVD1ResetOutput

LVD1 の RESET 信号の出力

関数のプロトタイプ宣言:

```
void  
LVD_SetVD1ResetOutput(FunctionalState NewState)
```

引数:

**NewState:** LVDRST 信号の出力状態を選択します。

- **ENABLE:** 許可
- **DISABLE:** 禁止

**機能:**

LVDRST 信号の出力状態を選択します。

**戻り値:**

なし

## 12.2.3.10 LVD\_SetVD1INTOutput

LVD1 の INTLVD 信号の出力

**関数のプロトタイプ宣言:**

void

LVD\_SetVD1INTOutput(FunctionalState **NewState**)

**引数:**

**NewState:** INTLVD 信号の出力状態を選択します。

- **ENABLE:** 許可
- **DISABLE:** 禁止

**機能:**

INTLVD 信号の出力状態を選択します。

**戻り値:**

なし

## 12.2.3.11 LVD\_SetVD2ResetOutput

LVD2 の RESET 信号の出力

**関数のプロトタイプ宣言:**

void

LVD\_SetVD2ResetOutput(FunctionalState **NewState**)

**引数:**

**NewState:** LVDRST 信号の出力状態を選択します。

- **ENABLE:** 許可
- **DISABLE:** 禁止

**機能:**

LVDRST 信号の出力状態を選択します。

**戻り値:**

なし

## 12.2.3.12 LVD\_SetVD2INTOutput

LVD2 の INTLVD 信号の出力

**関数のプロトタイプ宣言:**

void

LVD\_SetVD2INTOutput(FunctionalState **NewState**)

**引数:**

**NewState:** INTLVD 信号の出力状態を選択します。

- **ENABLE:** 許可
- **DISABLE:** 禁止

**機能:**

INTLVD 信号の出力状態を選択します。

**戻り値:**

なし

## 12.2.4 データ構造

なし



## 13. OFD

### 13.1 概要

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

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

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

/Libraries/TX04\_Periph\_Driver/src/tmpm462\_ofd.c

/Libraries/TX04\_Periph\_Driver/inc/tmpm462\_ofd.h

### 13.2 API 関数

#### 13.2.1 関数一覧

- ◆ void OFD\_SetRegWriteMode(FunctionalState **NewState**);
- ◆ void OFD\_Enable(void);
- ◆ void OFD\_Disable(void);
- ◆ void OFD\_SetDetectionFrequency(OFD\_OSCSource **Source**,  
uint32\_t **HigherDetectionCount**,  
uint32\_t **LowerDetectionCount**);
- ◆ void OFD\_Reset(FunctionalState **NewState**);
- ◆ OFD\_Status OFD\_GetStatus(void);
- ◆ void OFD\_SetDetectionMonitor(OFD\_MonitorMode **Mode**);

#### 13.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています:

- 1) OFD 機能の設定:  
OFD\_SetRegWriteMode(), OFD\_SetDetectionFrequency(),  
OFD\_SetDetectionMonitor(), OFD\_Enable (), OFD\_Disable()
- 2) OFD 状態の取得:  
OFD\_GetStatus()
- 3) その他:  
OFD\_Reset()

#### 13.2.3 関数仕様

##### 13.2.3.1 OFD\_SetRegWriteMode

レジスタ書き込み制御

関数のプロトタイプ宣言:

void  
OFD\_SetRegWriteMode(FunctionalState **NewState**)

引数:

**NewState:** レジスタ書き込みを制御します。

- **ENABLE** : 許可。
- **DISABLE**: 禁止。

**機能:**

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

**戻り値:**

なし

### 13.2.3.2 OFD\_Enable

OFD 機能の許可

**関数のプロトタイプ宣言:**

void  
OFD\_Enable(void)

**引数:**

なし。

**機能:**

OFD 機能を許可します。

**戻り値:**

なし

### 13.2.3.3 OFD\_Disable

OFD 機能の禁止

**関数のプロトタイプ宣言:**

void  
OFD\_Disable(void)

**引数:**

なし。

**機能:**

OFD 機能を禁止します。

**戻り値:**

なし

### 13.2.3.4 OFD\_SetDetectionFrequency

検知周波数の設定

**関数のプロトタイプ宣言:**

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

引数:

**Source**: 検知する周波数の発振ソースを選択します。

- **OFD\_IHOSC**: 内部高速発信器
- **OFD\_EHOSC**: 外部高速発信器

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

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

機能:

外部または内部発振検知用のカウント値を設定します。

戻り値:

なし

### 13.2.3.5 OFD\_Reset

OFD リセット発生制御

関数のプロトタイプ宣言:

```
void  
OFD_Reset(FunctionalState NewState)
```

引数:

**NewState**: OFD リセット発生の許可/禁止を選択します。

- **ENABLE**: 許可。
- **DISABLE**: 禁止。

機能:

OFD リセットの許可/禁止を選択します。

戻り値:

なし

### 13.2.3.6 OFD\_GetStatus

OFD 動作状態、異常検知状態の取得

関数のプロトタイプ宣言:

```
OFD_Status  
OFD_GetStatus(void)
```

引数:

なし。

機能:

OFD 動作状態と異常検知状態を取得します。

戻り値:

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

### 13.2.3.7 OFD\_SetDetectionMonitor

検出対象クロックの選択

**関数のプロトタイプ宣言:**

void

OFD\_SetDetectionMonitor(OFD\_MonitorMode **Mode**)

**引数:**

**Mode:** 検出対象クロックを選択します。

- **OFD\_NORMAL** : 通常監視モード。
- **OFD\_MONITOR**: モニタモード。

**機能:**

検出対象のクロックを選択します。

**戻り値:**

なし

## 13.2.4 データ構造

### 13.2.4.1 OFD\_Status

**メンバ:**

uint32\_t

**All:** すべてのステータス

**ビットフィールド:**

uint32\_t

**FrequencyError:** 1 異常検知

uint32\_t

**OFDBusy:** 1 OFD 動作中

## 14. RMC

### 14.1 概要

本デバイスは搬送波が取り除かれたリモコン信号の受信を行います。

リモコン受信:

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

RMCドライバ API ではチャンネル毎の機能セットが提供されています。

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

/Libraries/TX04\_Periph\_Driver/src/tmpm462\_rmc.c  
/Libraries/TX04\_Periph\_Driver/inc/tmpm462\_rmc.h

### 14.2 API 関数

#### 14.2.1 関数一覧

- ◆ void RMC\_Enable(TSB\_RMC\_TypeDef \* **RMCx**)
- ◆ void RMC\_Disable(TSB\_RMC\_TypeDef \* **RMCx**)
- ◆ void RMC\_Init(TSB\_RMC\_TypeDef \* **RMCx**, RMC\_InitTypeDef \* **RMC\_InitStruct**)
- ◆ void RMC\_SetRxCtrl(TSB\_RMC\_TypeDef \* **RMCx**, FunctionalState **NewState**)
- ◆ RMC\_RxDataTypeDef RMC\_GetRxData(TSB\_RMC\_TypeDef \* **RMCx**)
- ◆ void RMC\_SetLeaderDetection(TSB\_RMC\_TypeDef \* **RMCx**,  
RMC\_LeaderParameterTypeDef **LeaderPara**)
- ◆ void RMC\_SetFallingEdgeINT(TSB\_RMC\_TypeDef \* **RMCx**,  
FunctionalState **NewState**)
- ◆ void RMC\_SetSignalRxMethod(TSB\_RMC\_TypeDef \* **RMCx**,  
RMC\_RxMethod **Method**)
- ◆ void RMC\_SetRxTrg(TSB\_RMC\_TypeDef \* **RMCx**, uint8\_t **LowWidth**,  
uint8\_t **MaxDataBitCycle**)
- ◆ void RMC\_SetThreshold(TSB\_RMC\_TypeDef \* **RMCx**, uint8\_t **LargerThreshold**,  
uint8\_t **SmallerThreshold**)
- ◆ void RMC\_SetInputSignalReversed(TSB\_RMC\_TypeDef \* **RMCx**,  
FunctionalState **NewState**)
- ◆ void RMC\_SetNoiseCancellation(TSB\_RMC\_TypeDef \* **RMCx**,  
uint8\_t **NoiseCancellationTime**)
- ◆ RMC\_INTFactor RMC\_GetINTFactor(TSB\_RMC\_TypeDef \* **RMCx**)
- ◆ RMC\_LeaderDetection RMC\_GetLeader(TSB\_RMC\_TypeDef \* **RMCx**)
- ◆ void RMC\_SetRxEndBitNum(TSB\_RMC\_TypeDef \* **RMCx**,  
RMC\_RxEndBitsReg **Reg\_x**, uint8\_t **BitNum**)
- ◆ void RMC\_SetSrcClk(TSB\_RMC\_TypeDef \* **RMCx**, RMC\_SrcClk **Clk**)

## 14.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています:

- 1) RMC の初期化と設定:  
RMC\_Enable(), RMC\_Disable(), RMC\_Init(), RMC\_SetRxCtrl()
- 2) RMC 基本状態の設定:  
RMC\_SetLeaderDetection(), SetFallingEdgeINT(), RMC\_SetSignalRxMethod(),  
RMC\_SetRxTrg(), RMC\_SetThreshold(), RMC\_SetInputSignalReversed(),  
RMC\_SetNoiseCancellation(), RMC\_SetRxEndBitNum(), RMC\_SetSrcClk()
- 3) その他:  
RMC\_GetINTFactor(), RMC\_GetLeader(), RMC\_GetRxData()

## 14.2.3 関数仕様

\*補足: 引数“TSB\_RMC\_TypeDef \* **RMCx**”は以下のいずれかを指定してください。  
**TSB\_RMC0, TSB\_RMC1**

### 14.2.3.1 RMC\_Enable

RMC 機能の許可

関数のプロトタイプ宣言:

```
void  
RMC_Enable(TSB_RMC_TypeDef * RMCx)
```

引数:

**RMCx** : RMC チャンネルを指定します。

機能:

RMC 機能を許可します。

戻り値:

なし

### 14.2.3.2 RMC\_Disable

RMC 機能の禁止

関数のプロトタイプ宣言:

```
void  
RMC_Disable(TSB_RMC_TypeDef * RMCx)
```

引数:

**RMCx** : RMC チャンネルを指定します。

機能:

RMC 機能を禁止します。

戻り値:

なし

## 14.2.3.3 RMC\_Init

RMC レジスタの初期化

関数のプロトタイプ宣言:

void  
RMC\_Init(TSB\_RMC\_TypeDef \* **RMCx**, RMC\_InitTypeDef \* **RMC\_InitStruct**)

引数:

**RMCx** : RMC チャンネルを指定します。

**RMC\_InitStruct** : RMC 動作の初期値です。(詳細は“データ構造説明”を参照)

機能:

RMC チャンネルの初期化を行います。

戻り値:

なし

## 14.2.3.4 RMC\_SetRxCtrl

受信動作の設定

関数のプロトタイプ宣言:

void  
RMC\_SetRxCtrl(TSB\_RMC\_TypeDef \* **RMCx**, FunctionalState **NewState**)

引数:

**RMCx** : RMC チャンネルを指定します。

**NewState**: RMC 機能の受信動作を指定します。

- **ENABLE** : 許可。
- **DISABLE**: 禁止。

機能:

RMC 判定機能動作の許可/禁止を選択します。

戻り値:

なし

## 14.2.3.5 RMC\_GetRxData

受信データの取得

関数のプロトタイプ宣言:

RMC\_RxDataTypeDef  
RMC\_GetRxData(TSB\_RMC\_TypeDef \* **RMCx**)

引数:

**RMCx** : RMC チャンネルを指定します。

機能:

受信データを取得します。

戻り値:

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

## 14.2.3.6 RMC\_SetLeaderDetection

リーダ検出の設定

関数のプロトタイプ宣言:

void

RMC\_SetLeaderDetection(TSB\_RMC\_TypeDef \* **RMCx**,  
RMC\_LeaderParameterTypeDef **LeaderPara**)

引数:

**RMCx**: RMC チャンネルを指定します。

**LeaderPara**: リーダ検出を設定します。(詳細は“データ構造説明”を参照)

機能:

RMC リーダ検出を設定します。

戻り値:

なし

## 14.2.3.7 RMC\_SetFallingEdgeINT

リモコン入力立下りエッジ割り込み発生の許可

関数のプロトタイプ宣言:

void

RMC\_SetFallingEdgeINT(TSB\_RMC\_TypeDef \* **RMCx**,  
FunctionalState **NewState**)

引数:

**RMCx**: RMC チャンネルを指定します。

**NewState**: リモコン入力立下りエッジ割り込み発生 of 許可/禁止を選択します。

- **ENABLE**: 許可。
- **DISABLE**: 禁止。

機能:

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

戻り値:

なし

## 14.2.3.8 RMC\_SetSignalRxMethod

位相方式のリモコン受信モード選択

関数のプロトタイプ宣言:

void

RMC\_SetSignalRxMethod(TSB\_RMC\_TypeDef \* **RMCx**,



RMC\_RxMethod **Method**)

引数:

**RMCx**: RMC チャンネルを指定します。

**Method**: 位相方式のリモコン受信モードを選択します。

➤ **RMC\_RX\_IN\_CYCLE\_METHOD**: 周期方式で受信。

➤ **RMC\_RX\_IN\_PHASE\_METHOD**: 位相方式で受信。

機能:

位相方式のリモコン受信モードを選択します。

戻り値:

なし

## 14.2.3.9 RMC\_SetRxTrg

受信終了/割り込み設定

関数のプロトタイプ宣言:

```
void  
RMC_SetRxTrg(TSB_RMC_TypeDef * RMCx,  
              uint8_t LowWidth,  
              uint8_t MaxDataBitCycle)
```

引数:

**RMCx**: RMC チャンネルを指定します。

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

**MaxDataBitCycle**: データビットの周期 MAX で受信終了/割り込みを設定します。

機能:

RMC チャンネルのトリガ設定を行います。

**LowWidth** を RMCRCR2<RMCLL7:0> に設定した場合は、Low 幅の検出による受信終了/割り込み発生のタイミングを設定します。Low 幅検出時に受信が完了し、割り込みが発生します。<RMCLL7:0> = 11111111b の時は検出しません。

計算式:  $RMCLLx1/fs[s]$

**MaxDataBitCycle** を RMCRCR2<RMCDMAX7:0> に設定した場合は、データ bit の周期 MAX 検出のしきい値を設定します。データ bit 周期の値がしきい値以上であれば検出となります。<RMCDMAX7:0> = 11111111b の時は検出しません。

計算式:  $RMCDMAX \times 1/fs[s]$

戻り値:

なし

## 14.2.3.10 RMC\_SetThreshold

位相方式のしきい値の設定

関数のプロトタイプ宣言:

```
void  
RMC_SetThreshold(TSB_RMC_TypeDef * RMCx,  
                  uint8_t LargerThreshold,
```

uint8\_t **SmallerThreshold**)

引数:

**RMCx**: RMC チャンネルを指定します。

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

しきい値計算式:  $RMCDATHx1/fs[s]$

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

**SmallerThreshold**: 2種類のしきい値の設定: データビットの0/1判定のしきい値および、位相方式のリモコン信号の3値判定の1Tと1.5Tのしきい値の設定をします。

データビットの0/1判定の場合、測定結果がしきい値以上でデータ“1”、しきい値未満でデータ“0”と判別します。

しきい値の計算式:  $RMCDATLx1/fs[s]$

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

データビットの0/1判定:  $RMCDATLx1/fs[s]$

RMCR3<RMCDATH0-6> <RMCDATL0-6> ビットで設定します。

しきい値下位は 0x80 以下となります。

機能:

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

戻り値:

なし

## 14.2.3.11 RMC\_SetInputSignalReversed

リモコン入力信号の極性設定

関数のプロトタイプ宣言:

void

RMC\_SetInputSignalReversed(TSB\_RMC\_TypeDef \* **RMCx**,  
FunctionalState **NewState**)

引数:

**RMCx**: RMC チャンネルを指定します。

**NewState**: リモコン入力信号の極性を選択します。

- **ENABLE**: 負極。
- **DISABLE**: 正極。

機能:

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

戻り値:

なし

## 14.2.3.12 RMC\_SetNoiseCancellation

ノイズ除去時間の設定

関数のプロトタイプ宣言:

```
void  
RMC_SetNoiseCancellation(TSB_RMC_TypeDef * RMCx,  
                          uint8_t NoiseCancellationTime)
```

引数:

**RMCx**: RMC チャンネルを指定します。

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

機能:

ノイズ除去時間を設定します。

<RMCNC3:0> = 0000b の場合は、ノイズを除去しません。

ノイズキャンセル時間の計算式:  $RMCNC \times 1/fs[s]$ .

戻り値:

なし

## 14.2.3.13 RMC\_GetINTFactor

割り込み要因の取得

関数のプロトタイプ宣言:

```
RMC_INTFactor  
RMC_GetINTFactor(TSB_RMC_TypeDef * RMCx)
```

引数:

**RMCx**: RMC チャ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:

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

## 14.2.3.14 RMC\_GetLeader

リーダー検出の取得

関数のプロトタイプ宣言:

```
RMC_LeaderDetection  
RMC_GetLeader(TSB_RMC_TypeDef * RMCx)
```

引数:

**RMCx**: RMC チャンネルを指定します。

機能:

リーダー検出を取得します。

戻り値:

RMC\_LeaderDetection: リーダ検出結果

- RMC\_LEADER\_DETECTED: リーダ検出あり
- RMC\_NO\_LEADER: リーダ検出なし

## 14.2.3.15 RMC\_SetRxEndBitNum

受信終了ビット数の設定

関数のプロトタイプ宣言:

```
void  
RMC_SetRxEndBitNum(TSB_RMC_TypeDef * RMCx,  
                    RMC_RxEndBitsReg Reg_x,  
                    uint8_t BitNum)
```

引数:

**RMCx**: RMC チャンネルを指定します。

**Reg\_x**: 受信終了ビット数レジスタを選択します。

- RMC\_RX\_END\_BITS\_REG\_1: RMCxEND1 レジスタ。
- RMC\_RX\_END\_BITS\_REG\_2: RMCxEND2 レジスタ。
- RMC\_RX\_END\_BITS\_REG\_3: RMCxEND3 レジスタ。

**BitNum**: 受信するデータのビット数を設定します。

機能:

受信終了ビット数を設定します。

戻り値:

なし

## 14.2.3.16 RMC\_SetSrcClk

RMC サンプリングクロックの選択

関数のプロトタイプ宣言:

```
void  
RMC_SetSrcClk(TSB_RMC_TypeDef * RMCx,  
              RMC_SrcClk Clk)
```

引数:

**RMCx**: RMC チャンネルを指定します。

**Clk**: RMC サンプリングクロックを選択します。

- RMC\_CLK\_LOW\_FREQUENCY: 低速クロック(32KHz)
- RMC\_CLK\_TB1OUT: タイマ出力(TB1OUT).

機能:

RMC サンプリングクロックを選択します。

戻り値:

なし

## 14.2.4 データ構造

### 14.2.4.1 RMC\_RxDataDef

メンバ:

uint8

**RxDataBits:** 受信データビット数

uint32\_t

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

uint32\_t

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

uint8\_t

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

### 14.2.4.2 RMC\_LeaderParameterTypeDef

メンバ:

FunctionalState

**LeaderDetectionState:** リーダ検出のあり/なしを選択します。

- **ENABLE:** リーダ検出あり。
- **DISABLE:** リーダ検出なし。

uint8\_t

**MaxCycle:** リーダ検出の周期期間の上限。

uint8\_t

**MinCycle:** リーダ検出の周期期間の下限。

uint8\_t

**MaxLowWidth:** リーダ検出の LOW 期間の上限。

uint8\_t

**MinLowWidth:** リーダ検出の LOW 期間の下限。

FunctionalState

**LeaderINTState:** リーダ検出割り込み発生 of 許可/禁止を選択します。

- **ENABLE:** 割り込み発生する。
- **DISABLE:** 割り込み発生しない。

### 14.2.4.3 RMC\_InitTypeDef

メンバ:

RMC\_LeaderParameterTypeDef

**LeaderPara:** リーダ検出設定

FunctionalState

**FallingEdgeINTState:** リモコン入力立ち下がリエッジ割り込みの有効/無効を選択します。

- **ENABLE:** 割り込み発生する。
- **DISABLE:** 割り込み発生しない。

RMC\_RxMethod

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

- RMC\_RX\_IN\_CYCLE\_METHOD: 周期方式で受信。
- RMC\_RX\_IN\_PHASE\_METHOD: 位相方式で受信。

FunctionalState

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

- ENABLE: 負極。
- DISABLE: 正極。

uint8\_t

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

uint8\_t

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

uint8\_t

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

uint8\_t

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

uint8\_t

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

#### 14.2.4.4 RMC\_INTFactor

メンバ:

uint32\_t

**All:** データ

ビットフィールド:

uint32\_t

**Reserved** : 12 未使用

uint32\_t

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

uint32\_t

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

uint32\_t

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

uint32\_t

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

## 15. RTC

### 15.1 概要

RTC の機能概略は以下です。

- 時計機能(時間, 分, 秒)
- カレンダー機能(日月, 週, うるう年)
- 24 時間計と 12 時間計 (am/ pm)のいずれかを選択可能
- +/- 30 秒補正機能 (ソフトウェアによる補正)
- アラーム機能 (アラーム出力)
- アラーム割り込み発生
- 1MHz クロック出力機能

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

本ドライバは、アプリで使用する API 定義を格納する以下のファイルで構成されています。

/Libraries/TX04\_Periph\_Driver/src/tmpm462\_rtc.c  
/Libraries/TX04\_Periph\_Driver/inc/tmpm462\_rtc.h

### 15.2 API 関数

#### 15.2.1 関数一覧

- ◆ void RTC\_SetSec(uint8\_t **Sec**);
- ◆ uint8\_t RTC\_GetSec(void);
- ◆ void RTC\_SetMin(RTC\_FuncMode **NewMode**, uint8\_t **Min**);
- ◆ uint8\_t RTC\_GetMin(RTC\_FuncMode **NewMode**);
- ◆ uint8\_t RTC\_GetAMPM(RTC\_FuncMode **NewMode**);
- ◆ void RTC\_SetHour24(RTC\_FuncMode **NewMode**, uint8\_t **Hour**);
- ◆ void RTC\_SetHour12(RTC\_FuncMode **NewMode**, uint8\_t **Hour**, uint8\_t **AmPm**);
- ◆ uint8\_t RTC\_GetHour(RTC\_FuncMode **NewMode**);
- ◆ void RTC\_SetDay(RTC\_FuncMode **NewMode**, uint8\_t **Day**);
- ◆ uint8\_t RTC\_GetDay(RTC\_FuncMode **NewMode**);
- ◆ void RTC\_SetDate(RTC\_FuncMode **NewMode**, uint8\_t **Date**);
- ◆ uint8\_t RTC\_GetDate(RTC\_FuncMode **NewMode**);
- ◆ void RTC\_SetMonth(uint8\_t **Month**);
- ◆ uint8\_t RTC\_GetMonth(void);
- ◆ void RTC\_SetYear(uint8\_t **Year**);
- ◆ uint8\_t RTC\_GetYear(void);
- ◆ void RTC\_SetHourMode(uint8\_t **HourMode**);
- ◆ uint8\_t RTC\_GetHourMode(void);
- ◆ void RTC\_SetLeapYear(uint8\_t **LeapYear**);
- ◆ uint8\_t RTC\_GetLeapYear(void);
- ◆ void RTC\_SetTimeAdjustReq(void);
- ◆ RTC\_ReqState RTC\_GetTimeAdjustReq(void);
- ◆ void RTC\_EnableClock(void);
- ◆ void RTC\_DisableClock(void);
- ◆ void RTC\_EnableAlarm(void);
- ◆ void RTC\_DisableAlarm(void);
- ◆ void RTC\_SetRTCIINT(FunctionalState **NewState**);
- ◆ void RTC\_SetAlarmOutput(uint8\_t **Output**);

- ◆ void RTC\_ResetAlarm(void) ;
- ◆ void RTC\_ResetClockSec(void);
- ◆ RTC\_ReqState RTC\_GetResetClockSecReq(void);
- ◆ void RTC\_SetDateValue(RTC\_DateTypeDef \* **DateStruct**);
- ◆ void RTC\_GetDateValue(RTC\_DateTypeDef \* **DateStruct**);
- ◆ void RTC\_SetTimeValue(RTC\_TimeTypeDef \* **TimeStruct**);
- ◆ void RTC\_GetTimeValue(RTC\_TimeTypeDef \* **TimeStruct**);
- ◆ void RTC\_SetClockValue(RTC\_DateTypeDef \* **DateStruct**, RTC\_TimeTypeDef \* **TimeStruct**);
- ◆ void RTC\_GetClockValue(RTC\_DateTypeDef \* **DateStruct**, RTC\_TimeTypeDef \* **TimeStruct**);
- ◆ void RTC\_SetAlarmValue(RTC\_AlarmTypeDef \* **AlarmStruct**);
- ◆ void RTC\_GetAlarmValue(RTC\_AlarmTypeDef \* **AlarmStruct**);
- ◆ void RTC\_SetProtectCtrl(FunctionalState **NewState**)
- ◆ void RTC\_EnableCorrection(void)
- ◆ void RTC\_DisableCorrection(void)
- ◆ void RTC\_SetCorrectionTime(uint8\_t **Time**)
- ◆ void RTC\_SetCorrectionValue(RTC\_CorrectionMode **Mode**,uint16\_t **Cnt**)

## 15.2.2 関数の種類

関数は、主に以下の 6 種類に分かれています:

- 1) RTC 機能の年月日の設定:  
RTC\_SetDay(), RTC\_GetDay(), RTC\_SetDate(), RTC\_GetDate(), RTC\_SetMonth(),  
RTC\_GetMonth(), RTC\_SetYear(), RTC\_GetYear(), RTC\_SetLeapYear(),  
RTC\_GetLeapYear(), RTC\_SetDateValue(), RTC\_GetDateValue()
- 2) RTC 機能の時間の設定:  
RTC\_SetSec(), RTC\_GetSec(), RTC\_SetMin(),RTC\_GetMin(),RTC\_SetHour24(),  
RTC\_SetHour12(), RTC\_GetHour(), RTC\_SetHourMode(), RTC\_GetHourMode(),  
RTC\_GetAMPM(), RTC\_SetTimeValue(), RTC\_GetTimeValue()
- 3) RTC(clock)の設定:  
RTC\_EnableClock(), RTC\_DisableClock(), RTC\_SetTimeAdjustReq(),  
RTC\_GetTimeAdjustReq(), RTC\_ResetClockSec(), RTC\_GetResetClockSecReq(),  
RTC\_SetClockValue(), RTC\_GetClockValue()
- 4) RTC(alarm)の設定:  
RTC\_EnableAlarm(), RTC\_DisableAlarm(), RTC\_SetAlarmValue(),  
RTC\_ResetAlarm(), RTC\_GetAlarmValue()
- 5) RTC 補正基準時間の設定:  
RTC\_EnableCorrection(), RTC\_DisableCorrection(), RTC\_SetCorrectionTime(),  
RTC\_SetCorrectionValue()
- 6) その他:  
RTC\_SetAlarmOutput(), RTC\_SetProtectCtrl(), RTC\_SetRTCINT()

## 15.2.3 関数仕様

### 15.2.3.1 RTC\_SetSec

時計の秒桁設定

関数のプロトタイプ宣言:

```
void  
RTC_SetSec(uint8_t Sec);
```

引数:

**Sec**:最大 59 までの秒桁設定の値。



**機能:**

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

**戻り値:**

なし

## 15.2.3.2 RTC\_GetSec

時計の秒桁設定

**関数のプロトタイプ宣言:**

```
uint8_t  
RTC_GetSec(void);
```

**引数:**

なし。

**機能:**

時計の秒桁の値を返します。

**戻り値:**

時計の秒桁:  
➤ 0 ~ 59

## 15.2.3.3 RTC\_SetMin

時計/アラームの分析設定

**関数のプロトタイプ宣言:**

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

**引数:**

**NewMode**: RTC モードを選択します。

- **RTC\_CLOCK\_MODE**: 時計機能
- **RTC\_ALARM\_MODE**: アラーム機能

**Min**: 最大 59 までの分析を設定します。

**機能:**

**NewMode** が **RTC\_CLOCK\_MODE** の場合、時計の分析を設定します。

**NewMode** が **RTC\_ALARM\_MODE** の場合、アラームの分析を設定します。

RTC レジスタは、INTRTC のタイミングに同期して書き換えられます。この関数を呼び出した後に、1Hz 割り込みが発生するのを待つ必要があります。

**戻り値:**

なし

## 15.2.3.4 RTC\_GetMin

時計/アラームの分析読み込み

関数のプロトタイプ宣言:

```
uint8_t  
RTC_GetMin(RTC_FuncMode NewMode);
```

引数:

**NewMode**: RTC モードを選択します。

- **RTC\_CLOCK\_MODE**: 時計機能
- **RTC\_ALARM\_MODE**: アラーム機能

機能:

**NewMode** が **RTC\_CLOCK\_MODE** の場合、時計の分析の値を返します。

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

戻り値:

分析:

- 0 ~ 59

## 15.2.3.5 RTC\_GetAMPM

12 時間モードの AM/PM 読み込み

関数のプロトタイプ宣言:

```
uint8_t  
RTC_GetAMPM(RTC_FuncMode NewMode);
```

引数:

**NewMode**: RTC モードを選択します。

- **RTC\_CLOCK\_MODE**: 時計機能
- **RTC\_ALARM\_MODE**: アラーム機能

機能:

時計/アラームの AM/PM を返します。

**NewMode** が **RTC\_CLOCK\_MODE** の場合、時計の AM/PM を返します。

**NewMode** が **RTC\_ALARM\_MODE** の場合、アラームの AM/PM を返します。

戻り値:

時計モード:

**RTC\_AM\_MODE**: AM

**RTC\_PM\_MODE**: PM

## 15.2.3.6 RTC\_SetHour24

24 時間モードの時計/アラーム時桁設定

関数のプロトタイプ宣言:

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

**引数:**

**NewMode:** RTC モードを選択します。

- **RTC\_CLOCK\_MODE:** 時計機能
- **RTC\_ALARM\_MODE:** アラーム機能

**Hour:** 最大 23 までの時桁を設定します。

**機能:**

24 時間モードの時計/アラームの時桁を設定します。

**NewMode** が **RTC\_CLOCK\_MODE** の場合、時計機能の時桁を設定し、

**NewMode** が **RTC\_ALARM\_MODE** の場合、アラームの時桁を設定します。

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

\*12 時間モードから 24 時間モードに変更する場合、本関数 **RTC\_SetHour24()** によって HOUERR レジスタを再設定してください。

**戻り値:**

なし

## 15.2.3.7 RTC\_SetHour12

12 時間モードの時計/アラーム時桁設定

**関数のプロトタイプ宣言:**

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

**引数:**

**NewMode:** RTC モードを選択します。

- **RTC\_CLOCK\_MODE:** 時計機能
- **RTC\_ALARM\_MODE:** アラーム機能

**Hour:** 最大 11 までの時桁を設定します。

**AmPm:** 以下から時間モードを選択します。

- **RTC\_AM\_MODE:** 12H モードの AM モード
- **RTC\_PM\_MODE:** 12H モードの PM モード

**機能:**

12 時間モードの時計/アラームの時桁を設定します。

**NewMode** が **RTC\_CLOCK\_MODE** の場合、時計機能の時桁を設定し、

**NewMode** が **RTC\_ALARM\_MODE** の場合、アラーム機能の時桁を設定します。

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

\*24 時間モードから 12 時間モードに変更する場合、本関数 **RTC\_SetHour12()** によって HOUERR レジスタを再度設定してください。

**戻り値:**

なし

## 15.2.3.8 RTC\_GetHour

時計/アラームの時桁読み込み

関数のプロトタイプ宣言:

```
uint8_t  
RTC_GetHour(RTC_FuncMode NewMode);
```

引数:

**NewMode**: RTC モードを選択します。

- **RTC\_CLOCK\_MODE**: 時計機能
- **RTC\_ALARM\_MODE**: アラーム機能

機能:

時計/アラームの時桁を返します。

**NewMode** が **RTC\_CLOCK\_MODE** の場合、時計機能の時桁の値を返し、  
**NewMode** が **RTC\_ALARM\_MODE** の場合、アラーム機能の時桁の値を返します。

戻り値:

24 時間モードでの時桁:

- 0 ~ 23

12H 時間モードでの時桁:

- 0 ~ 11

## 15.2.3.9 RTC\_SetDay

時計/アラームの曜日設定

関数のプロトタイプ宣言:

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

引数:

**NewMode**: RTC モードを選択します。

- **RTC\_CLOCK\_MODE**: 時計機能
- **RTC\_ALARM\_MODE**: アラーム機能

**Day**: 曜日を選択します。

- **RTC\_SUN**: 日曜日
- **RTC\_MON**: 月曜日
- **RTC\_TUE**: 火曜日
- **RTC\_WED**: 水曜日
- **RTC\_THU**: 木曜日
- **RTC\_FRI**: 金曜日
- **RTC\_SAT**: 土曜日

機能:

時計/アラームの曜日を設定します。

**NewMode** が **RTC\_CLOCK\_MODE** の場合、時計機能の曜日を設定します。

**NewMode** が **RTC\_ALARM\_MODE** の場合、アラーム機能の曜日を設定します。

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

戻り値:  
なし

## 15.2.3.10 RTC\_GetDay

時計/アラームの曜日の読み込み

関数のプロトタイプ宣言:

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

引数:

**NewMode**: RTC モードを選択します。

- **RTC\_CLOCK\_MODE**: 時計機能
- **RTC\_ALARM\_MODE**: アラーム機能

機能:

時計/アラームの曜日を返します。

**NewMode** が **RTC\_CLOCK\_MODE** の場合、時計機能の曜日を返し、

**NewMode** が **RTC\_ALARM\_MODE** の場合、アラーム機能の曜日を返します。

戻り値:

曜日の値:

- **RTC\_SUN**: 日曜日
- **RTC\_MON**: 月曜日
- **RTC\_TUE**: 火曜日
- **RTC\_WED**: 水曜日
- **RTC\_THU**: 木曜日
- **RTC\_FRI**: 金曜日
- **RTC\_SAT**: 土曜日

## 15.2.3.11 RTC\_SetDate

時計/アラームの日桁設定

関数のプロトタイプ宣言:

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

引数:

**NewMode**: RTC モードを選択します。

- **RTC\_CLOCK\_MODE**: 時計機能
- **RTC\_ALARM\_MODE**: アラーム機能

**Date**: 1 から 31 の日桁を設定します。

機能:

時計/アラームの日桁を設定します。

**NewMode** が **RTC\_CLOCK\_MODE** の場合は、時計機能の日桁を設定し、

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

戻り値:  
なし

## 15.2.3.12 RTC\_GetDate

時計/アラームの日桁読み込み

関数のプロトタイプ宣言:

```
uint8_t  
RTC_GetDate(RTC_FuncMode NewMode);
```

引数:

**NewMode**: RTC モードを選択します。

- **RTC\_CLOCK\_MODE**: 時計機能
- **RTC\_ALARM\_MODE**: アラーム機能

機能:

時計/アラームの日桁を返します。

**NewMode** が **RTC\_CLOCK\_MODE** の場合、時計機能の日桁の値を返し、  
**NewMode** が **RTC\_ALARM\_MODE** の場合、アラーム機能の日桁の値を返します。

戻り値:  
日桁:  
➤ 1 ~ 31

## 15.2.3.13 RTC\_SetMonth

時計の月桁設定

関数のプロトタイプ宣言:

```
void  
RTC_SetMonth(uint8_t Month);
```

引数:

**Month**: 1 から 12 の月桁を設定します。

機能:

時計の月桁を設定します。

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

戻り値:  
なし

## 15.2.3.14 RTC\_GetMonth

時計の月桁読み込み

**関数のプロトタイプ宣言:**

```
uint8_t  
RTC_GetMonth(void);
```

**引数:**

なし。

**機能:**

時計の月桁の値を返します。

**戻り値:**

月桁:

➤ 1 ~ 12

## 15.2.3.15 RTC\_SetYear

時計の年桁設定

**関数のプロトタイプ宣言:**

```
void  
RTC_SetYear(uint8_t Year);
```

**引数:**

*Year*: 最大 99 までの年の値

**機能:**

時計の年桁を設定します。

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

**戻り値:**

なし

## 15.2.3.16 RTC\_GetYear

時計の年桁の読み込み

**関数のプロトタイプ宣言:**

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

**引数:**

なし。

**機能:**

時計の年桁の値を返します。

**戻り値:**

年桁:

➤ 0 ~ 99

## 15.2.3.17 RTC\_SetHourMode

24 時間時計/12 時間時計の選択

**関数のプロトタイプ宣言:**

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

**引数:**

**HourMode:** 時間モードを選択します。

- **RTC\_12\_HOUR\_MODE:** 12 時間時計
- **RTC\_24\_HOUR\_MODE:** 24 時間時計

**機能:**

24 時間時計/12 時間時計を選択します。

**HourMode** が **RTC\_24\_HOUR\_MODE** の時、12 時間時計を選択し、  
**HourMode** が **RTC\_12\_HOUR\_MODE** の時、24 時間時計を選択します。

**補足:**

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

**戻り値:**

なし

## 15.2.3.18 RTC\_GetHourMode

時計モードの読み込み

**関数のプロトタイプ宣言:**

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

**引数:**

なし。

**機能:**

時計モードを読み込みます。

**戻り値:**

時計モード:

- **RTC\_24\_HOUR\_MODE:** 24 時間時計
- **RTC\_12\_HOUR\_MODE:** 12 時間時計

## 15.2.3.19 RTC\_SetLeapYear

うるう年の設定

**関数のプロトタイプ宣言:**

```
void  
RTC_SetLeapYear(uint8_t LeapYear);
```



**引数:**

**LeapYear.** 以下からうるう年を選択します。

- **RTC\_LEAP\_YEAR\_0:** 現在の年(今年)がうるう年
- **RTC\_LEAP\_YEAR\_1:** 現在がうるう年から 1 年目
- **RTC\_LEAP\_YEAR\_2:** 現在がうるう年から 2 年目
- **RTC\_LEAP\_YEAR\_3:** 現在がうるう年から 3 年目

**機能:**

うるう年を設定します。

**LeapYear** が **RTC\_LEAP\_YEAR\_0** の場合、現在の年(今年)がうるう年で、  
**LeapYear** が **RTC\_LEAP\_YEAR\_1** の場合、現在がうるう年から 1 年目で、  
**LeapYear** が **RTC\_LEAP\_YEAR\_2** の場合、現在がうるう年から 2 年目で、  
**LeapYear** が **RTC\_LEAP\_YEAR\_3** の場合、現在がうるう年から 3 年目になります。

**戻り値:**

なし

## 15.2.3.20 RTC\_GetLeapYear

うるう年の読み込み

**関数のプロトタイプ宣言:**

```
uint8_t  
RTC_GetLeapYear(void);
```

**引数:**

なし。

**機能:**

うるう年の状態を返します。

**戻り値:**

うるう年の状態を表す値

## 15.2.3.21 RTC\_SetTimeAdjustReq

+/- 30 秒の補正

**関数のプロトタイプ宣言:**

```
void  
RTC_SetTimeAdjustReq(void);
```

**引数:**

なし。

**機能:**

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

**戻り値:**

なし

## 15.2.3.22 RTC\_GetTimeAdjustReq

ADJUST 要求状態の読み込み

**関数のプロトタイプ宣言:**

RTC\_ReqState

RTC\_GetTimeAdjustReq(void);

**引数:**

なし。

**機能:**

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

**戻り値:**

ADJUST 要求状態を読み込みます。

- **RTC\_NO\_REQ**: ADJUST 要求なし
- **RTC\_REQ**: ADJUST 要求あり

## 15.2.3.23 RTC\_EnableClock

時計機能の起動

**関数のプロトタイプ宣言:**

void

RTC\_EnableClock(void);

**引数:**

なし。

**機能:**

時計機能を有効にします。

**戻り値:**

なし

## 15.2.3.24 RTC\_DisableClock

時計機能の終了

**関数のプロトタイプ宣言:**

void

RTC\_DisableClock(void);

**引数:**

なし。

**機能:**

時計機能を無効にします。

戻り値:

なし

## 15.2.3.25 RTC\_EnableAlarm

アラーム機能の起動

関数のプロトタイプ宣言:

```
void  
RTC_EnableAlarm(void);
```

引数:

なし。

機能:

アラーム機能を有効にします。

戻り値:

なし

## 15.2.3.26 RTC\_DisableAlarm

アラーム機能の終了

関数のプロトタイプ宣言:

```
void  
RTC_DisableAlarm(void);
```

引数:

なし。

機能:

アラーム機能を無効にします。

戻り値:

なし

## 15.2.3.27 RTC\_SetRTCINT

INTRTC 割り込みの有効/無効設定

関数のプロトタイプ宣言:

```
void  
RTC_SetRTCINT(FunctionalState NewState);
```

引数:

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

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

**機能:**

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

**戻り値:**

なし

## 15.2.3.28 RTC\_SetAlarmOutput

ALARM 端子の出力設定

**関数のプロトタイプ宣言:**

```
void  
RTC_SetAlarmOutput(uint8_t Output);
```

**引数:**

**Output**: 以下から、アラーム端子の出力を選択します。

- **RTC\_LOW\_LEVEL**: “0” パルス
- **RTC\_PULSE\_1\_HZ**: 1Hz 周期の “0” パルス
- **RTC\_PULSE\_16\_HZ**: 16Hz 周期の “0” パルス
- **RTC\_PULSE\_2\_HZ**: 2Hz 周期の “0” パルス
- **RTC\_PULSE\_4\_HZ**: 4Hz 周期の “0” パルス
- **RTC\_PULSE\_8\_HZ**: 8Hz 周期の “0” パルス

**機能:**

アラーム端子の出力を設定します。

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

**戻り値:**

なし

## 15.2.3.29 RTC\_ResetAlarm

アラームリセット

**関数のプロトタイプ宣言:**

```
void  
RTC_ResetAlarm(void);
```

**引数:**

なし

**機能:**

アラームレジスタ(分、時、日、週桁レジスタ)を初期化します。

**戻り値:**

なし

## 15.2.3.30 RTC\_ResetClockSec

時計秒カウンタのリセット

**関数のプロトタイプ宣言:**

```
void  
RTC_ResetClockSec(void);
```

**引数:**

なし。

**機能:**

時計秒カウンタをリセットします。

**戻り値:**

なし

## 15.2.3.31 RTC\_GetResetClockSecReq

時計秒カウンタのリセット要求状態の読み込み

**関数のプロトタイプ宣言:**

```
RTC_ReqState  
RTC_GetResetClockSecReq(void);
```

**引数:**

なし。

**機能:**

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

**戻り値:**

リセット要求状態

- **RTC\_NO\_REQ:** リセット要求なし
- **RTC\_REQ:** リセット要求あり

## 15.2.3.32 RTC\_SetDateValue

時計の日付設定

**関数のプロトタイプ宣言:**

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

**引数:**

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

**機能:**

時計の日付(うるう年、年、月、曜日、日)を読み込みます。

RTC\_SetLeapYear(), RTC\_SetYear(), RTC\_SetMonth(), RTC\_SetDate(),  
RTC\_Setday()を実行します。

戻り値:  
なし

### 15.2.3.33 RTC\_GetDateValue

時計の日付の読み込み

関数のプロトタイプ宣言:

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

引数:

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

機能:

時計のうるう年、年、月、曜日、日を読み込みます。

RTC\_GetLeapYear(), RTC\_GetYear(), RTC\_GetMonth(), RTC\_GetDate(),  
RTC\_Getday()を実行します。

戻り値:  
なし

### 15.2.3.34 RTC\_SetTimeValue

時計の時刻設定

関数のプロトタイプ宣言:

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

引数:

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

機能:

時間モード、時間、12 時間モードの AM/PM モード、分、秒を設定します。

RTC\_SetHourMode(), RTC\_SetHour12(), RTC\_SetHour24(), RTC\_SetMin(),  
RTC\_SetSec() を実行します。

戻り値:  
なし

### 15.2.3.35 RTC\_GetTimeValue

時計の時刻の読み込み

関数のプロトタイプ宣言:

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

**引数:**

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

**機能:**

時刻(時間モード、時間、12 時間モードの AM/PM モード、分、秒)を読み込みます。

**RTC\_GetHourMode()**, **RTC\_GetHour()**, **RTC\_GetAMPM()**, **RTC\_GetMin()**, **RTC\_GetSec()** が実行されます。

**戻り値:**

なし

## 15.2.3.36 RTC\_SetClockValue

時計の日時設定

**関数のプロトタイプ宣言:**

```
void  
RTC_SetClockValue(RTC_DateTypeDef * DateStruct,  
                  RTC_TimeTypeDef * TimeStruct);
```

**引数:**

**DateStruct:** うるう年、年、月、曜日、日を格納する構造体。

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

**機能:**

時計の日付(うるう年、年、月、曜日、日)、および、時刻(時間モード、時間、12 時間モードの AM/PM モード、分、秒)を設定します。

**RTC\_SetLeapYear()**, **RTC\_SetYear()**, **RTC\_SetMonth()**, **RTC\_SetDate()**, **RTC\_SetDay()**, **RTC\_SetHourMode()**, **RTC\_SetHour24()**, **RTC\_SetHour12()**, **RTC\_SetMin()**, **RTC\_SetSec()** を実行します。

**戻り値:**

なし

## 15.2.3.37 RTC\_GetClockValue

時計の日時の読み込み

**関数のプロトタイプ宣言:**

```
void  
RTC_GetClockValue(RTC_DateTypeDef * DateStruct,  
                  RTC_TimeTypeDef * TimeStruct);
```

**引数:**

**DateStruct:** うるう年、年、月、曜日、日を格納する構造体。

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

**機能:**

時計の日付(うるう年、年、月、曜日、日)、および、時刻(時間モード、時間、12 時間モードの AM/PM モード、分、秒)を設定します。

**RTC\_GetLeapYear(), RTC\_GetYear(), RTC\_GetMonth(), RTC\_GetDate(),  
RTC\_GetDay(), RTC\_GetHourMode(), RTC\_GetHour(), RTC\_GetAMP(),  
RTC\_GetMin(), RTC\_GetSec()** を実行します。

**戻り値:**

なし

## 15.2.3.38 RTC\_SetAlarmValue

アラームの日時設定

**関数のプロトタイプ宣言:**

```
void  
RTC_SetAlarmValue(RTC_AlarmTypeDef * AlarmStruct);
```

**引数:**

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

**機能:**

アラームの日時(日、曜日、時間、12 時間モードの AM/PM モード、秒を含む)を設定します。**RTC\_SetDate(), RTC\_SetDay(), RTC\_SetHour12(), RTC\_SetHour24(),  
RTC\_SetMin()**をコールします。

**戻り値:**

なし

## 15.2.3.39 RTC\_GetAlarmValue

アラームの日時の取得

**関数のプロトタイプ宣言:**

```
void  
RTC_GetAlarmValue(RTC_AlarmTypeDef * AlarmStruct);
```

**引数:**

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

**機能:**

アラームの日時(日、曜日、時間、12 時間モードの AM/PM モード、秒を含む)を読み込みます。

**RTC\_GetDate(), RTC\_GetDay(), RTC\_GetHour(), RTC\_GetAMP(),  
RTC\_GetMin()** をコールします。



戻り値:  
なし

## 15.2.3.40 RTC\_SetProtectCtrl

補正機能レジスタ書き込み制御

関数のプロトタイプ宣言:

```
void  
RTC_SetProtectCtrl(FunctionalState NewState);
```

引数:

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

- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:

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

戻り値:  
なし

## 15.2.3.41 RTC\_EnableCorrection

補正機能の許可

関数のプロトタイプ宣言:

```
void  
RTC_EnableCorrection(void);
```

引数:

なし。

機能:

補正機能を許可します。

戻り値:  
なし

## 15.2.3.42 RTC\_DisableCorrection

補正機能の禁止

関数のプロトタイプ宣言:

```
void  
RTC_DisableCorrection(void);
```

引数:

なし。

**機能:**

補正機能を禁止します。

**戻り値:**

なし

## 15.2.3.43 RTC\_SetCorrectionTime

補正基準時間の設定

**関数のプロトタイプ宣言:**

```
void  
RTC_SetCorrectionTime(uint8_t Time);
```

**引数:**

**Time:** 補正基準時間を選択します。

- **RTC\_ADJ\_TIME\_1\_SEC**: 1 秒
- **RTC\_ADJ\_TIME\_10\_SEC**: 10 秒
- **RTC\_ADJ\_TIME\_20\_SEC**: 20 秒
- **RTC\_ADJ\_TIME\_30\_SEC**: 30 秒
- **RTC\_ADJ\_TIME\_1\_MIN**: 1 分

**機能:**

補正基準時間を設定します。

**戻り値:**

なし

## 15.2.3.44 RTC\_SetCorrectionValue

補正値の設定

**関数のプロトタイプ宣言:**

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

**引数:**

**Mode:** 補正符号を選択します。

- **RTC\_CORRECTION\_PLUS**: プラス補正
- **RTC\_CORRECTION\_MINUS**: マイナス補正

**Cnt:** 1 秒に対する補正値を選択します。

- **RTC\_CORRECTION\_PLUS** の場合、0~255 を選択できます。
- **RTC\_CORRECTION\_MINUS** の場合、1~256 を選択できます。

**機能:**

補正値を選択します。

**戻り値:**

なし

## 15.2.4 データ構造

### 15.2.4.1 RTC\_DateTypeDef

メンバ:

uint8\_t

**LeapYear**: うるう年を設定します:

- **RTC\_LEAP\_YEAR\_0**: 現在の年(今年)がうるう年
- **RTC\_LEAP\_YEAR\_1**: 現在がうるう年から 1 年目
- **RTC\_LEAP\_YEAR\_2**: 現在がうるう年から 2 年目
- **RTC\_LEAP\_YEAR\_3**: 現在がうるう年から 3 年目

uint8\_t

**Year** 年桁の値(0～99)。

uint8\_t

**Month** 月桁の値(1～12)。

uint8\_t

**Date** 日桁の値(1～31)。

uint8\_t

**Day** 週の値を設定します。

- **RTC\_SUN**: 日曜日
- **RTC\_MON**: 月曜日
- **RTC\_TUE**: 火曜日
- **RTC\_WED**: 水曜日
- **RTC\_THU**: 木曜日
- **RTC\_FRI**: 金曜日
- **RTC\_SAT**: 土曜日

### 15.2.4.2 RTC\_TimeTypeDef

メンバ:

uint8\_t

**HourMode** 24 時間時計、12 時間時計のモード選択の値:

- **RTC\_12\_HOUR\_MODE**: 12 時間モード
- **RTC\_24\_HOUR\_MODE**: 24 時間モード

uint8\_t

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

uint8\_t

**AmPm** 12 時間モード時の AM/PM の値:

- **RTC\_AM\_MODE**: AM モード
- **RTC\_PM\_MODE**: PM モード
- **RTC\_AMPM\_INVALID**: 24 時間モード

uint8\_t

**Min** 0～59 までの分桁の値。

uint8\_t

**Sec** 0～59 までの秒桁の値。

## 15.2.4.3 RTC\_AlarmTypeDef

メンバ:

uint8\_t

**Date** アラーム機能有効時の日桁の値(1～31)。

uint8\_t

**Day** アラーム機能有効時の週桁の値。

- **RTC\_SUN**: 日曜日
- **RTC\_MON**: 月曜日
- **RTC\_TUE**: 火曜日
- **RTC\_WED**: 水曜日
- **RTC\_THU**: 木曜日
- **RTC\_FRI**: 金曜日
- **RTC\_SAT**: 土曜日

uint8\_t

**Hour** アラーム機能有効時の時間桁の値。

uint8\_t

**AmPm** アラーム機能有効時の AM/PM 選択の値:

- **RTC\_AM\_MODE**: AM モード
- **RTC\_PM\_MODE**: PM モード
- **RTC\_AMPM\_INVALID**: 24 時間モード

uint8\_t

**Min** アラーム機能有効時の分桁の値(0～59)。

## 16. SSP

### 16.1 概要

本デバイスは、同期式シリアルインタフェースを (SSP: Synchronous Serial Port) を 3 チャンネル内蔵しています。(SSP0, SSP1, SSP2)

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

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

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

/Libraries/TX04\_Periph\_Driver/src/tmpm462\_ssp.c  
/Libraries/TX04\_Periph\_Driver/inc/tmpm462\_ssp.h

### 16.2 API 関数

#### 16.2.1 関数一覧

- ◆ void SSP\_Enable(TSB\_SSP\_TypeDef \* **SSPx**);
- ◆ void SSP\_Disable(TSB\_SSP\_TypeDef \* **SSPx**);
- ◆ void SSP\_Init(TSB\_SSP\_TypeDef \* **SSPx**, SSP\_InitTypeDef \* **InitStruct**);
- ◆ void SSP\_SetClkPreScale(TSB\_SSP\_TypeDef \* **SSPx**, uint8\_t **PreScale**,  
uint8\_t **ClkRate**);
- ◆ void SSP\_SetFrameFormat(TSB\_SSP\_TypeDef \* **SSPx**,  
SSP\_FrameFormat **FrameFormat**);
- ◆ void SSP\_SetClkPolarity(TSB\_SSP\_TypeDef \* **SSPx**, SSP\_ClkPolarity **ClkPolarity**);
- ◆ void SSP\_SetClkPhase(TSB\_SSP\_TypeDef \* **SSPx**, SSP\_ClkPhase **ClkPhase**);
- ◆ void SSP\_SetDataSize(TSB\_SSP\_TypeDef \* **SSPx**, uint8\_t **DataSize**);
- ◆ void SSP\_SetSlaveOutputCtrl(TSB\_SSP\_TypeDef \* **SSPx**,  
FunctionalState **NewState**);
- ◆ void SSP\_SetMSMode(TSB\_SSP\_TypeDef \* **SSPx**, SSP\_MS\_Mode **Mode**);
- ◆ void SSP\_SetLoopBackMode(TSB\_SSP\_TypeDef \* **SSPx**,  
FunctionalState **NewState**);
- ◆ void SSP\_SetTxData(TSB\_SSP\_TypeDef \* **SSPx**, uint16\_t **Data**);
- ◆ uint16\_t SSP\_GetRxData(TSB\_SSP\_TypeDef \* **SSPx**);
- ◆ WorkState SSP\_GetWorkState(TSB\_SSP\_TypeDef \* **SSPx**);
- ◆ SSP\_FIFOState SSP\_GetFIFOState(TSB\_SSP\_TypeDef \* **SSPx**,  
SSP\_Direction **Direction**);
- ◆ void SSP\_SetINTConfig(TSB\_SSP\_TypeDef \* **SSPx**, uint32\_t **IntSrc**);
- ◆ SSP\_INTState SSP\_GetINTConfig(TSB\_SSP\_TypeDef \* **SSPx**);
- ◆ SSP\_INTState SSP\_GetPreEnableINTState(TSB\_SSP\_TypeDef \* **SSPx**);
- ◆ SSP\_INTState SSP\_GetPostEnableINTState(TSB\_SSP\_TypeDef \* **SSPx**);
- ◆ void SSP\_ClearINTFlag(TSB\_SSP\_TypeDef \* **SSPx**, uint32\_t **IntSrc**);
- ◆ void SSP\_SetDMACtrl(TSB\_SSP\_TypeDef \* **SSPx**, SSP\_Direction **Direction**,

FunctionalState **NewState**);

## 16.2.2 関数の種類

関数は、主に以下の 6 種類に分かれています:

- 1) 共通関数:  
SSP\_Init(), SSP\_SetClkPreScale(), SSP\_SetFrameFormat(), SSP\_SetClkPolarity(),  
SSP\_SetClkPhase(), SSP\_SetDataSize(), SSP\_SetMSMode()
- 2) データ送受信:  
SSP\_SetTxData(), SSP\_GetRxData()
- 3) SSP 割り込み関連:  
SSP\_SetIntConfig(), SSP\_GetIntConfig(), SSP\_GetPreEnableINTState(),  
SSP\_GetPostEnableINTState(), SSP\_ClearINTFlag()
- 4) 状態の取得:  
SSP\_GetWorkState(), SSP\_GetFIFOState()
- 5) モジュールの有効/無効設定:  
SSP\_Enable(), SSP\_Disable()
- 6) その他:  
SSP\_SetSlaveOutputCtrl(), SSP\_SetLoopBackMode(), SSP\_SetDMACtrl()

## 16.2.3 関数仕様

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

### 16.2.3.1 SSP\_Enable

同期式シリアルインタフェース動作の許可

関数のプロトタイプ宣言:

```
void  
SSP_Enable(TSB_SSP_TypeDef * SSPx)
```

引数:

**SSPx**: SSP チャンネルを指定します。

機能:

SSP 動作を有効にします。

戻り値:

なし

### 16.2.3.2 SSP\_Disable

同期式シリアルインタフェース動作の禁止

関数のプロトタイプ宣言:

```
void  
SSP_Disable(TSB_SSP_TypeDef * SSPx)
```

引数:

**SSPx**: SSP チャンネルを指定します。

**機能:**

SSP 動作を無効にします。

**戻り値:**

なし

### 16.2.3.3 SSP\_Init

SSP 通信の初期化

**関数のプロトタイプ宣言:**

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

**引数:**

**SSPx**: SSP チャンネルを指定します。

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

**機能:**

SSP 通信の初期化を行います。

本 API がコールする API は以下の通りです。

```
    SSP_SetFrameFormat(),  
    SSP_SetClkPreScale(),  
    SSP_SetClkPolarity(),  
    SSP_SetClkPhase(),  
    SSP_SetDataSize(),  
    SSP_SetMSMode().
```

**戻り値:**

なし

### 16.2.3.4 SSP\_SetClkPreScale

送受信のビットレート設定

**関数のプロトタイプ宣言:**

```
void  
SSP_SetClkPreScale(TSB_SSP_TypeDef * SSPx,  
                   uint8_t PreScale,  
                   uint8_t ClkRate)
```

**引数:**

**SSPx**: SSP チャンネルを指定します。

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

**ClkRate**: シリアルクロックレートを 0~255 の間で設定します。

**機能:**

送受信のビットレートを設定します。**SSP\_Init()** によりコールされます。

Tx と Rx 用の本ビットレートは下記計算式で求めることができます。

$$\text{BitRate} = \text{fSYS} / (\text{PreScale} \times (1 + \text{ClkRate}))$$

fSYS はシステム周波数

戻り値:

なし

## 16.2.3.5 SSP\_SetFrameFormat

フレームフォーマットの選択

関数のプロトタイプ宣言:

void

SSP\_SetFrameFormat(TSB\_SSP\_TypeDef \* **SSPx**,  
SSP\_FrameFormat **FrameFormat**)

引数:

**SSPx**: SSP チャンネルを指定します。

**FrameFormat**: フレームフォーマットを選択します。

- **SSP\_FORMAT\_SPI**: SPI フレームフォーマット
- **SSP\_FORMAT\_SSI**: SSI シリアルフレームフォーマット
- **SSP\_FORMAT\_MICROWIRE**: Microwire フレームフォーマット

機能:

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

戻り値:

なし

## 16.2.3.6 SSP\_SetClkPolarity

SPxCLK 極性の選択

関数のプロトタイプ宣言:

void

SSP\_SetClkPolarity(TSB\_SSP\_TypeDef \* **SSPx**,  
SSP\_ClkPolarity **ClkPolarity**)

引数:

**SSPx**: SSP チャンネルを指定します。

**ClkPolarity**: SPxCLK 極性を選択します。

- **SSP\_POLARITY\_LOW**: SPxCLK は Low 状態。
- **SSP\_POLARITY\_HIGH**: SPxCLK は High 状態。

機能:

SPxCLK 極性を選択します。**SSP\_Init()** からコールされます。

戻り値:

なし

## 16.2.3.7 SSP\_SetClkPhase

SPxCLK フェーズの選択



## 関数のプロトタイプ宣言:

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

## 引数:

**SSPx**: SSP チャンネルを指定します。

**ClkPhase**: SPxCLK フェーズを選択します。

- **SSP\_PHASE\_FIRST\_EDGE**: 1st クロックエッジでデータを取り込み
- **SSP\_PHASE\_SECOND\_EDGE**: 2nd クロックエッジでデータを取り込み

## 機能:

SPxCLK フェーズを選択します。**SSP\_Init()** からコールされます。

## 戻り値:

なし

### 16.2.3.8 SSP\_SetDataSize

データサイズの選択

## 関数のプロトタイプ宣言:

```
void  
SSP_SetDataSize(TSB_SSP_TypeDef * SSPx,  
                uint8_t DataSize)
```

## 引数:

**SSPx**: SSP チャンネルを指定します。

**DataSize**: データサイズを 4~16 の間で選択します。

## 機能:

データサイズを選択します。**SSP\_Init()** からコールれます。

## 戻り値:

なし

### 16.2.3.9 SSP\_SetSlaveOutputCtrl

スレーブモード SPxDO 出力の制御

## 関数のプロトタイプ宣言:

```
void  
SSP_SetSlaveOutputCtrl(TSB_SSP_TypeDef * SSPx,  
                       FunctionalState NewState)
```

## 引数:

**SSPx**: SSP チャンネルを指定します。

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

- **ENABLE**: 許可。
- **DISABLE**: 禁止。

**機能:**

スレーブモード SPxDO 出力の許可/禁止を選択します。

**戻り値:**

なし

## 16.2.3.10 SSP\_SetMSMode

マスタ/スレーブモードの選択

**関数のプロトタイプ宣言:**

```
void  
SSP_SetMSMode(TSB_SSP_TypeDef * SSPx,  
               SSP_MS_Mode Mode)
```

**引数:**

**SSPx**: SSP チャンネルを指定します。

**Mode**: マスタ/スレーブモードを選択します。

- **SSP\_MASTER**: デバイスがマスタ。
- **SSP\_SLAVE**: デバイスがスレーブ。

**機能:**

マスタ/スレーブモードを選択します。

**戻り値:**

なし

## 16.2.3.11 SSP\_SetLoopBackMode

ループバックモードの制御

**関数のプロトタイプ宣言:**

```
void  
SSP_SetLoopBackMode(TSB_SSP_TypeDef * SSPx,  
                     FunctionalState NewState)
```

**引数:**

**SSPx**: SSP チャンネルを指定します。

**NewState**: ループバックモードの許可/禁止を選択します。

- **ENABLE**: 許可。
- **DISABLE**: 禁止。

**機能:**

ループバックモードを設定します。

例えば、ループバックモードが有効の場合、送受信間にセルフテストを行います。

**戻り値:**

なし

## 16.2.3.12 SSP\_SetTxData

送信 FIFO のデータ設定

関数のプロトタイプ宣言:

```
void  
SSP_SetTxData(TSB_SSP_TypeDef * SSPx,  
               uint16_t Data)
```

引数:

**SSPx**: SSP チャンネルを指定します。

**Data**: 送信データを 4～16 ビットの間で設定します。

機能:

送信 FIFO にデータを設定します。

戻り値:

なし

## 16.2.3.13 SSP\_GetRxData

受信 FIFO からのデータ読み込み

関数のプロトタイプ宣言:

```
uint16_t  
SSP_GetRxData(TSB_SSP_TypeDef * SSPx)
```

引数:

**SSPx**: SSP チャンネルを指定します。

機能:

受信 FIFO から受信データを読み込みます。

戻り値:

受信データ

## 16.2.3.14 SSP\_GetWorkState

ビジーフラグの読み込み

関数のプロトタイプ宣言:

```
WorkState  
SSP_GetWorkState(TSB_SSP_TypeDef * SSPx)
```

引数:

**SSPx**: SSP チャンネルを指定します。

機能:

ビジーフラグを読み込みます。

戻り値:

ビジーフラグ

BUSY: ビジー  
DONE: アイドル

## 16.2.3.15 SSP\_GetFIFOState

送受信 FIFO の読み込み

関数のプロトタイプ宣言:

```
SSP_FIFOState  
SSP_GetFIFOState(TSB_SSP_TypeDef * SSPx  
                  SSP_Direction Direction)
```

引数:

**SSPx**: SSP チャンネルを指定します。

**Direction**: 送受信方向を選択します。

- **SSP\_RX**: 受信 FIFO
- **SSP\_TX**: 送信 FIFO

機能:

送受信 FIFO の状態を読み込みます。

例えば、送信 FIFO の状態を判断した後でのデータ送信処理は次の通り。

```
SSP_FIFOState fifoState;  
  
fifoState = SSP_GetFIFOState(TSB_SSP0, SSP_TX);  
if ((fifoState == SSP_FIFO_EMPTY) || (fifoState == SSP_FIFO_NORMAL))  
{ SSP_SetTxData(SSP0, data_to_be_sent ); }
```

戻り値:

送受信 FIFO の状態:

**SSP\_FIFO\_EMPTY**: FIFO が空の状態。

**SSP\_FIFO\_NORMAL**: FIFO がフル、かつ空ではない状態。

**SSP\_FIFO\_INVALID**: FIFO が無効の状態。

**SSP\_FIFO\_FULL**: FIFO がフルの状態。

## 16.2.3.16 SSP\_SetINTConfig

割り込みの制御

関数のプロトタイプ宣言:

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

引数:

**SSPx**: SSP チャンネルを指定します。

**IntSrc**: 割り込みの許可/禁止を選択します。

- **SSP\_INTCFG\_NONE**: すべて禁止。
- **SSP\_INTCFG\_ALL**: すべて許可。

任意の割り込みを“|”で選択します。

- **SSP\_INTCFG\_RX\_OVERRUN**: 受信オーバーラン割り込み。
- **SSP\_INTCFG\_RX\_TIMEOUT**: 受信タイムアウト割り込み。

- **SSP\_INTCFG\_RX:** 受信 FIFO 割り込み(受信 FIFO の半分以上がフル)
- **SSP\_INTCFG\_TX:** 送信 FIFO 割り込み(送信 FIFO の半分以上がフル)

**機能:**

割り込みの許可/ 禁止を選択します。

例えば、送受信割り込みを設定する処理は次の通り。

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

**戻り値:**

なし

## 16.2.3.17 SSP\_GetINTConfig

割り込み制御の読み込み

**関数のプロトタイプ宣言:**

SSP\_INTState

SSP\_GetINTConfig(TSB\_SSP\_TypeDef \* **SSPx**)

**引数:**

**SSPx:** SSP チャンネルを指定します。

**機能:**

割り込みの許可/禁止状態を取得します。

例えば、SSP\_SetINTConfig() で許可または禁止した割り込みソースを確認することができます。

**戻り値:**

SSP\_INTState: 割り込み設定状態。詳細は"データ構造"を参照。

## 16.2.3.18 SSP\_GetPreEnableINTState

許可前の割り込み状態の読み込み

**関数のプロトタイプ宣言:**

SSP\_INTState

SSP\_GetPreEnableINTState(TSB\_SSP\_TypeDef \* **SSPx**)

**引数:**

**SSPx:** SSP チャンネルを指定します。

**機能:**

許可前の割り込み状態を読み込みます。

**戻り値:**

SSP\_INTState: 許可前の割り込み状態。詳細は"データ構造"を参照。

## 16.2.3.19 SSP\_GetPostEnableINTState

許可後の割り込み状態の読み込み

**関数のプロトタイプ宣言:**

SSP\_INTState

SSP\_GetPostEnableINTState(TSB\_SSP\_TypeDef \* **SSPx**)

**引数:**

**SSPx**: SSP チャンネルを指定します。

**機能:**

禁止前の割り込み状態を読み込みます。

**戻り値:**

SSP\_INTState: 許可前の割り込み状態。詳細は"データ構造"を参照。

## 16.2.3.20 SSP\_ClearINTFlag

割り込みフラグのクリア

**関数のプロトタイプ宣言:**

void

SSP\_ClearINTFlag(TSB\_SSP\_TypeDef \* **SSPx**,  
uint32\_t **IntSrc**)

**引数:**

**SSPx**: SSP チャンネルを指定します。

**IntSrc**: クリアする割り込みフラグを選択します。

- **SSP\_INTCFG\_RX\_OVERRUN**: 受信オーバーラン割り込みフラグ。
- **SSP\_INTCFG\_RX\_TIMEOUT**: 受信タイムアウト割り込みフラグ
- **SSP\_INTCFG\_ALL**: すべての割り込みフラグ。

**機能:**

割り込みフラグをクリアします。

**戻り値:**

なし

## 16.2.3.21 SSP\_SetDMACtrl

送受信 FIFO の DMA 制御

**関数のプロトタイプ宣言:**

void

SSP\_SetDMACtrl(TSB\_SSP\_TypeDef \* **SSPx**,  
SSP\_Direction **Direction**,  
FunctionalState **NewState**)

**引数:**

**SSPx**: SSP チャンネルを指定します。

**Direction**: 送受信方向を選択します。

- **SSP\_RX**: 受信。
- **SSP\_TX**: 送信。

**NewState**: DMA FIFO の状態。

- **ENABLE:** 許可。
- **DISABLE:** 禁止。

**機能:**

送受信 FIFO の DMA 許可/禁止を選択します。

**戻り値:**

なし

## 16.2.4 データ構造

### 16.2.4.1 SSP\_InitTypeDef

**メンバ:**

SSP\_FrameFormat

**FrameFormat:** フレームフォーマットを選択します。

- **SSP\_FORMAT\_SPI:** SPI フレームフォーマット
- **SSP\_FORMAT\_SSI:** SSI フレームフォーマット
- **SSP\_FORMAT\_MICROWIRE:** Microwire フレームフォーマット

uint8\_t

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

SSP\_ClkPolarity

**ClkPolarity:** SPxCLK 極性を選択します。

- **SSP\_POLARITY\_LOW:** SPxCLK 極性は Low 状態。
- **SSP\_POLARITY\_HIGH:** SPxCLK 極性は High 状態。

SSP\_ClkPhase

**ClkPhase:** SPxCLK フェーズを設定します。

- **SSP\_PHASE\_FIRST\_EDGE:** 1st クロックエッジでデータを取り込み
- **SSP\_PHASE\_SECOND\_EDGE:** 2nd クロックエッジでデータを取り込み

uint8\_t

**DataSize:** データを 4～16 ビットの間で設定します。

SSP\_MS\_Mode

**Mode:** マスタ/スレーブモードを選択します。

- **SSP\_MASTER:** デバイスがマスタ
- **SSP\_SLAVE:** デバイスがスレーブ

### 16.2.4.2 SSP\_INTState

**メンバ:**

uint32\_t

**All:** 割り込み要因

**Bit**

uint32\_t

**OverRun:** 1 オーバーラン割り込み

uint32\_t

**TimeOut:** 1 受信タイムアウト

uint32\_t

<b>Rx:</b>	1	受信
uint32_t		
<b>Tx:</b>	1	送信
uint32_t		
<b>Reserved:</b>	28	未使用



## 17. TMRB

### 17.1 概要

本デバイスは、16 チャンネルの多機能 16 ビットタイマ/ イベントカウンタ (TMRB0 ~ TMRBF)を内蔵しています。各チャンネルは下記モードで動作します。

- 16 ビットインタバルタイマモード
- 16 ビットイベントカウンタモード
- 16 ビットプログラマブル矩形波出力 (PPG) モード
- タイマ同期モード(各 4 チャンネルの出力設定可能)

また、キャプチャ機能を利用することで、次のような用途に使用することができます。

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

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

本ドライバは、クロック分割、サイクル、デューティ期間、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/TX04\_Periph\_Driver/src/tmpm462\_tmr.c  
/Libraries/TX04\_Periph\_Driver/inc/tmpm462\_tmr.h

### 17.2 API 関数

#### 17.2.1 関数一覧

- ◆ void TMRB\_Enable(TSB\_TB\_TypeDef \* **TBx**)
- ◆ void TMRB\_Disable(TSB\_TB\_TypeDef \* **TBx**)
- ◆ void TMRB\_SetRunState(TSB\_TB\_TypeDef \* **TBx**, uint32\_t **Cmd**)
- ◆ void TMRB\_Init(TSB\_TB\_TypeDef \* **TBx**, TMRB\_InitTypeDef \* **InitStruct**)
- ◆ void TMRB\_SetCaptureTiming(TSB\_TB\_TypeDef \* **TBx**, uint32\_t **CaptureTiming**)
- ◆ void TMRB\_SetFlipFlop(TSB\_TB\_TypeDef \* **TBx**,  
TMRB\_FFOutputTypeDef \* **FFStruct**)
- ◆ TMRB\_INTFactor TMRB\_GetINTFactor(TSB\_TB\_TypeDef \* **TBx**)
- ◆ void TMRB\_SetINTMask(TSB\_TB\_TypeDef \* **TBx**, uint32\_t **INTMask**)
- ◆ void TMRB\_ChangeLeadingTiming(TSB\_TB\_TypeDef \* **TBx**,  
uint32\_t **LeadingTiming**)
- ◆ void TMRB\_ChangeTrailingTiming(TSB\_TB\_TypeDef \* **TBx**,  
uint32\_t **TrailingTiming**)
- ◆ uint16\_t TMRB\_GetUpCntValue(TSB\_TB\_TypeDef \* **TBx**)
- ◆ uint16\_t TMRB\_GetCaptureValue(TSB\_TB\_TypeDef \* **TBx**, uint8\_t **CapReg**)
- ◆ void TMRB\_ExecuteSWCapture(TSB\_TB\_TypeDef \* **TBx**)
- ◆ void TMRB\_SetIdleMode(TSB\_TB\_TypeDef \* **TBx**, FunctionalState **NewState**)
- ◆ void TMRB\_SetSyncMode(TSB\_TB\_TypeDef \* **TBx**, FunctionalState **NewState**)
- ◆ void TMRB\_SetDoubleBuf(TSB\_TB\_TypeDef \* **TBx**, FunctionalState **NewState**,

uint8\_t *WriteRegMode*)

- ◆ void TMRB\_SetExtStartTrg(TSB\_TB\_TypeDef \* *TBx*, FunctionalState *NewState*, uint8\_t *TrgMode*)
- ◆ void TMRB\_SetClkInCoreHalt(TSB\_TB\_TypeDef \* *TBx*, uint8\_t *ClkState*)

## 17.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています:

- 1) 各タイマの設定:  
TMRB\_Enable(), TMRB\_Disable(), TMRB\_Init(), TMRB\_SetRunState(),  
TMRB\_ChangeLeadingTiming(), TMRB\_ChangeTrailingTiming()
- 2) キャプチャ機能の設定:  
TMRB\_SetCaptureTiming(), TMRB\_ExecuteSWCapture()
- 3) ステータスの確認:  
TMRB\_GetINTFactor(), TMRB\_GetUpCntValue(), TMRB\_GetCaptureValue()
- 4) その他:  
TMRB\_SetFlipFlop(), TMRB\_SetINTMask(), TMRB\_SetIdleMode(),  
TMRB\_SetSyncMode(), TMRB\_SetDoubleBuf(), TMRB\_SetExtStartTrg(),  
TMRB\_SetClkInCoreHalt()

## 17.2.3 関数仕様

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

TSB\_TB0, TSB\_TB1, TSB\_TB2, TSB\_TB3, TSB\_TB4, TSB\_TB5, TSB\_TB6,  
TSB\_TB7, TSB\_TB8, TSB\_TB9, TSB\_TBA, TSB\_TBB, TSB\_TBC, TSB\_TBD,  
TSB\_TBE, TSB\_TBF, TSB\_TB\_MPT0, TSB\_TB\_MPT1.

### 17.2.3.1 TMRB\_Enable

TMRB 動作の許可

関数のプロトタイプ宣言:

void  
TMRB\_Enable(TSB\_TB\_TypeDef\* *TBx*)

引数:

*TBx*: TMRB チャンネルを指定します。

機能:

TMRB 動作を有効にします。

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

戻り値:

なし

### 17.2.3.2 TMRB\_Disable

TMRB 動作の禁止

関数のプロトタイプ宣言:

void  
TMRB\_Disable(TSB\_TB\_TypeDef\* *TBx*)

引数:

**TBx**: TMRB チャ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

### 17.2.3.3 TMRB\_SetRunState

カウンタ動作の設定

関数のプロトタイプ宣言:

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

引数:

**TBx**: TMRB チャ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

### 17.2.3.4 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、デューティー期間の初期設定を行います。

戻り値:

なし

## 17.2.3.5 TMRB\_SetCaptureTiming

キャプチャタイミングの設定

関数のプロトタイプ宣言:

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

引数:

**TBx**: TMRB チャンネルを指定します。

**CaptureTiming**: キャプチャタイミングを選択します。

TBx = TSB\_TB\_MPT0 または TSB\_TB\_MPT1 の場合:

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

TBx = TSB\_TB0 ~ TSB\_TBF の場合:

- **TMRB\_DISABLE\_CAPTURE**: キャプチャ禁止
- **TMRB\_CAPTURE\_TBIN0\_TBIN1\_RISING**: TBxIN0 端子入力の立ち上がりでキャプチャレジスタ 0 (TBxCP0) にカウント値を取り込み、TBxIN1 端子入力の立ち上がりでキャプチャレジスタ 1 (TBxCP1) にカウント値を取り込みます。  
(TSB\_TB4 ~ TSB\_TBF のみ **TMRB\_CAPTURE\_TBIN0\_TBIN1\_RISING** を選択可能です)
- **TMRB\_CAPTURE\_TBIN0\_RISING\_FALLING**: TBxIN0 端子入力の立ち上がりでキャプチャレジスタ 0 (TBxCP0) にカウント値を取り込み、TBxIN0 端子入力の立ち下がりでキャプチャレジスタ 1 (TBxCP1) にカウント値を取り込みます。
- **TMRB\_CAPTURE\_TBFF0\_EDGE**: TBxFF0 の立ち上がりでキャプチャレジスタ 0 (TBxCP0) にカウント値を取り込み、TBxFF0 の立ち下がりでキャプチャレジスタ 1 (TBxCP1) にカウント値を取り込みます。
- **TMRB\_CLEAR\_TBIN1\_RISING**: TBxIN1↑でアップカウンタをクリアします。  
(TSB\_TB4 ~ TSB\_TBF のみ **TMRB\_CLEAR\_TBIN1\_RISING** を選択可能です)
- **TMRB\_CAPTURE\_TBIN0\_RISING\_CLEAR\_TBIN1\_RISING**: TBxIN0↑でキャプチャレジスタ 0 (TBxCP0) にカウント値を取り込み、TBxIN1↑でアップカウンタをクリアします。キャプチャタイミングとアップカウンタのクリアタイミングが同時だった場合、キャプチャが実行された後にアップカウンタのクリアが実行されます。  
(TSB\_TB4~TSB\_TBF のみ **TMRB\_CAPTURE\_TBIN0\_RISING\_CLEAR\_TBIN1\_RISING** を選択可能です)

機能:

キャプチャタイミングとアップカウンタのクリアタイミングを設定します。

戻り値:

なし

## 17.2.3.6 TMRB\_SetFlipFlop

フリップフロップ機能の設定

**関数のプロトタイプ宣言:**

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

**引数:**

**TBx:** TMRB チャ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

## 17.2.3.7 TMRB\_GetINTFactor

割り込み要因の取得

**関数のプロトタイプ宣言:**

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

**引数:**

**TBx:** TMRB チャ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:

**MatchLeadingTiming** (Bit0): 一致フラグ(TBxRG0)

**MatchTrailingTiming** (Bit1): 一致フラグ(TBxRG1)

**OverFlow** (Bit2): オーバーフローフラグ

**補足:**

異なる割り込み要因を処理する場合は、以下のように記述してください。

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

```
}
```

## 17.2.3.8 TMRB\_SetINTMask

割り込みマスクの設定

関数のプロトタイプ宣言:

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

引数:

**TBx:** TMRB チャ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\_TRAILING\_INT:** 一致 (TBxRG0) 割り込み
- **TMRB\_MASK\_MATCH\_LEADING\_INT:** 一致 (TBxRG1) 割り込み
- **TMRB\_MASK\_OVERFLOW\_INT:** オーバーフロー割り込み
- **TMRB\_NO\_INT\_MASK:** マスクしない
- **TMRB\_MASK\_MATCH\_LEADING\_INT |**  
**TMRB\_MASK\_MATCH\_TRAILING\_INT:** 一致 (TBxRG0) 割り込み、または一  
致 (TBxRG1) 割り込み
- **TMRB\_MASK\_MATCH\_LEADING\_INT | TMRB\_MASK\_OVERFLOW\_INT:**  
一致 (TBxRG1) 割り込み、またはオーバーフロー割り込み
- **TMRB\_MASK\_MATCH\_TRAILING\_INT |**  
**TMRB\_MASK\_OVERFLOW\_INT:** 一致 (TBxRG0) 割り込み、またはオーバ  
ーフロー割り込み
- **TMRB\_MASK\_MATCH\_LEADING\_INT |**  
**TMRB\_MASK\_MATCH\_TRAILING\_INT |**  
**TMRB\_MASK\_OVERFLOW\_INT:** 一致 (TBxRG1) 割り込み、または一致  
(TBxRG0) 割り込み、またはオーバーフロー割り込み

機能:

**TMRB\_MASK\_MATCH\_TRAILING\_INT** 選択時、アップカウンタ値と TBxRG1 が一致した場合、割り込みは発生しません。

**TMRB\_MASK\_MATCH\_LEADING\_INT** 選択時、アップカウンタ値と TBxRG0 が一致した場合、割り込みは発生しません。

**TMRB\_MASK\_OVERFLOW\_INT** 選択時、オーバーフロー発生時の割り込みは発生しません。

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

**TMRB\_MASK\_MATCH\_TRAILING\_INT** と

**TMRB\_MASK\_MATCH\_LEADING\_INT** と **TMRB\_MASK\_OVERFLOW\_INT** 選択時、どの条件が成立しても割り込みは発生しません。

戻り値:

なし

## 17.2.3.9 TMRB\_ChangeLeadingTiming

デューティの設定

関数のプロトタイプ宣言:

```
void
```

TMRB\_ChangeLeadingTiming(TSB\_TB\_TypeDef\* **TBx**,  
uint32\_t **LeadingTiming**)

引数:

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

戻り値:

なし。

補足:

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

## 17.2.3.10 TMRB\_ChangeTrailingTiming

周期の設定

関数のプロトタイプ宣言:

void  
TMRB\_ChangeTrailingTiming(TSB\_TB\_TypeDef\* **TBx**,  
uint32\_t **TrailingTiming**)

引数:

**TBx**: TMRB チャ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

補足:

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

## 17.2.3.11 TMRB\_GetUpCntValue

アップカウンタ値の読み込み

関数のプロトタイプ宣言:

uint16\_t  
TMRB\_GetUpCntValue(TSB\_TB\_TypeDef\* **TBx**)

引数:

**TBx**: TMRB チャ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

## 17.2.3.12 TMRB\_GetCaptureValue

キャプチャレジスタの読み込み

**関数のプロトタイプ宣言:**

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

**引数:**

**TBx**: TMRB チャ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

**機能:**

**CapReg** が **TMRB\_CAPTURE\_0** の場合、キャプチャレジスタ 0 の値を読み込み、**CapReg** が **TMRB\_CAPTURE\_1** の場合、キャプチャレジスタ 1 の値を読み込みます。

**戻り値:**

キャプチャレジスタの値

## 17.2.3.13 TMRB\_ExecuteSWCapture

ソフトウェアキャプチャの実行

**関数のプロトタイプ宣言:**

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

**引数:**

**TBx**: TMRB チャ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

## 17.2.3.14 TMRB\_SetIdleMode

IDLE 時の動作設定

**関数のプロトタイプ宣言:**



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

**引数:**

**TBx:** TMRB チャ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

## 17.2.3.15 TMRB\_SetSyncMode

同期モードの切り替え

**関数のプロトタイプ宣言:**

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

**引数:**

**TBx:** TMRB チャンネルを以下から選択します。

**TSB\_TB1, TSB\_TB2, TSB\_TB3, TSB\_TB5, TSB\_TB6, TSB\_TB7,  
TSB\_TB9, TSB\_TBA, TSB\_TBB**

**NewState:** 同期モードを切り替えます。

- **ENABLE:** 同期動作
- **DISABLE:** 個別動作(チャンネル毎)

**機能:**

TMRB1～TMRB3 を同期モードに設定すると、TMRB0 のスタートに同期して動作がスタートし、TMRB5 ～TMRB7 を同期モードに設定すると、TMRB4 のスタートに同期して動作がスタートし、TMRB9 ～TMRBB を同期モードに設定すると、TMRB8 のスタートに同期して動作がスタートします。

**戻り値:**

なし

**補足:**

同期モードを使用するために、TMRB0, TMRB4, TMRB8 のカウントを開始する前に、**TMRB\_SetRunState()** によって TMRB1 ～ TMRB3、TMRB5 ～ TMRB7、TMRB9 ～ TMRBB をスタートしてください。

## 17.2.3.16 TMRB\_SetDoubleBuf

ダブルバッファ動作の制御

## 関数のプロトタイプ宣言:

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

## 引数:

**TBx**: TMRB チャ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**: 有効。
- **DISABLE**: 無効。

**WriteRegMode**: ダブルバッファがイネーブルの場合のタイマレジスタ 0 および 1 への書き込みタイミングを指定します。

- **TMRB\_WRITE\_REG\_SEPARATE**: タイマレジスタ 0 および 1 は個別に書き込みが可能です。一方のレジスタのみ書き込み準備が完了した場合も同様です。
- **TMRB\_WRITE\_REG\_SIMULTANEOUS**: 両方のレジスタの書き込み準備が完了していない場合、タイマレジスタ 0 および 1 への書き込みはできません。

## 機能:

TBxRG0 レジスタ(**LeadingTiming**)と TBxRG1 (**TrailingTiming**)およびこれらのバッファは、同一アドレスへ割り付けられます。ダブルバッファがディセーブルの場合、同一の値はレジスタとそのバッファに書き込まれます。

ダブルバッファがイネーブルの場合、その値は各レジスタのバッファのみに書き込まれます。そのため初期値をレジスタ(TBxRG0 (**LeadingTiming**) および TBxRG1 (**TrailingTiming**))へ書き込むためには、ダブルバッファは **DISABLE** に設定してください。その後、イネーブルのダブルバッファには、レジスタへ書き込む次のデータが書き込まれます。データは対応する割り込みが発生した場合に自動的にロードされます。

## 戻り値:

なし

## 補足:

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

## 17.2.3.17 TMRB\_SetExtStartTrg

外部トリガの設定

## 関数のプロトタイプ宣言:

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

## 引数:

**TBx**: TMRB チャ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

## 17.2.3.18 TMRB\_SetClkInCoreHalt

デバッグ HALT 中のクロック動作

**関数のプロトタイプ宣言:**

void

TMRB\_SetClkInCoreHalt (TSB\_TB\_TypeDef\* **TBx**, uint8\_t **ClkState**)

**引数:**

**TBx:** TMRB チャ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 モードに移移した場合、TMRB クロック動作/停止の設定を行ないます。

**戻り値:**

なし

## 17.2.4 データ構造

### 17.2.4.1 TMRB\_InitTypeDef

**メンバ:**

uint32\_t

**Mode:** タイマモードを選択します。

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

uint32\_t

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

- **TMRB\_CLK\_DIV\_2:** fperiph / 2
- **TMRB\_CLK\_DIV\_8:** fperiph / 8
- **TMRB\_CLK\_DIV\_32:** fperiph / 32
- **TMRB\_CLK\_DIV\_64:** fperiph / 64 (TMRB0~TMRB7 のみ)
- **TMRB\_CLK\_DIV\_128:** fperiph / 128 (TMRB0~TMRB7 のみ)
- **TMRB\_CLK\_DIV\_256:** fperiph / 256 (TMRB0~TMRB7 のみ)
- **TMRB\_CLK\_DIV\_512:** fperiph / 512 (TMRB0~TMRB7 のみ)

uint32\_t

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

uint32\_t

**UpCntCtrl:** アップカウンタの動作を選択します。

- **TMRB\_FREE\_RUN:** 周期が一致した後も、0xFFFF になるまでアップカウンタは停止しません。その後、カウンタがクリアされ、0 からカウントを開始します。(TMRB0~TMRBF のみ)
- **TMRB\_AUTO\_CLEAR:** **TrailingTiming** と一致したときに、0 クリアされ、再スタートします。(TMRB0~TMRBF のみ)
- **MPT\_FREE\_RUN:** 周期が一致した後も、0xFFFF になるまでアップカウンタは停止しません。その後、カウンタがクリアされ、0 からカウントを開始します。(MPT0~MPT1 のみ)
- **MPT\_AUTO\_CLEAR:** **TrailingTiming** と一致したときに、0 クリアされ、再スタートします。(MPT0~MPT1 のみ)

uint32\_t

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

## 17.2.4.2 TMRB\_FFOutputTypeDef

メンバ:

uint32\_t

**FlipflopCtrl:** フリップフロップのレベルを選択します。

- **TMRB\_FLIPFLOP\_INVERT:** TBxFF0 の値を反転(ソフト反転)します。
- **TMRB\_FLIPFLOP\_SET:** TBxFF0 を"1"にセットします。
- **TMRB\_FLIPFLOP\_CLEAR:** TBxFF0 を"0"にクリアします。

uint32\_t

**FlipflopReverseTrg:** 以下から、フリップフロップの反転トリガを選択します。

- **TMRB\_DISALBE\_FLIPFLOP:** 反転トリガを無効にします。
- **TMRB\_FLIPFLOP\_TAKE\_CATPURE\_0:** アップカウンタの値がキャプチャレジスタ 0 に取り込まれた時にタイマフリップフロップを反転します。
- **TMRB\_FLIPFLOP\_TAKE\_CATPURE\_1:** アップカウンタの値がキャプチャレジスタ 1 に取り込まれた時にタイマフリップフロップを反転します。
- **TMRB\_FLIPFLOP\_MATCH\_TRAILING:** アップカウンタと周期との一致時にタイマフリップフロップを反転します。
- **TMRB\_FLIPFLOP\_MATCH\_LEADING:** アップカウンタとデューティとの一致時にタイマフリップフロップを反転します。

## 17.2.4.3 TMRB\_INTFactor

メンバ:

uint32\_t

**All:** TMRB 割り込み要因

ビットフィールド:

uint32\_t

**MatchLeadingTiming:** 1 デューティとの一致検出

uint32\_t

**MatchTrailingTiming:** 1周期との一致検出

uint32\_t

*OverFlow* : 1      オーバーフロー  
uint32\_t  
*Reserverd* : 29      -

## 18. SIO/UART

### 18.1 概要

本デバイスのシリアル I/O チャンネルは、I/O インタフェースモード(同期通信モード)と 7, 8, 9 ビット長の UART モード(非同期通信)を実装しています。

9 ビット UART モードでは、シリアルリンク(マルチコントローラ・システム) でマスタコントローラがスレーブコントローラを起動するときにウェイクアップ機能が使用されます。

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

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

/Libraries/TX04\_Periph\_Driver/src/tmpm462\_uart.c

/Libraries/TX04\_Periph\_Driver/inc/tmpm462\_uart.h

### 18.2 API 関数

#### 18.2.1 関数一覧

- ◆ void UART\_Enable(TSB\_SC\_TypeDef\* **UARTx**)
- ◆ void UART\_Disable(TSB\_SC\_TypeDef\* **UARTx**)
- ◆ WorkState UART\_GetBufState(TSB\_SC\_TypeDef\* **UARTx**, uint8\_t **Direction**)
- ◆ void UART\_SWReset(TSB\_SC\_TypeDef\* **UARTx**)
- ◆ void UART\_Init(TSB\_SC\_TypeDef\* **UARTx**, UART\_InitTypeDef\* **InitStruct**)
- ◆ uint32\_t UART\_GetRxData(TSB\_SC\_TypeDef\* **UARTx**)
- ◆ void UART\_SetTxData(TSB\_SC\_TypeDef\* **UARTx**, uint32\_t **Data**)
- ◆ void UART\_DefaultConfig(TSB\_SC\_TypeDef\* **UARTx**)
- ◆ UART\_Err UART\_GetErrState(TSB\_SC\_TypeDef\* **UARTx**)
- ◆ void UART\_SetWakeUpFunc(TSB\_SC\_TypeDef\* **UARTx**,  
FunctionalState **NewState**)
  
- ◆ void UART\_SetIdleMode(TSB\_SC\_TypeDef\* **UARTx**, FunctionalState **NewState**)
- ◆ void UART\_FIFOConfig(TSB\_SC\_TypeDef \* **UARTx**, FunctionalState **NewState**);
- ◆ void UART\_SetFIFOTransferMode(TSB\_SC\_TypeDef \* **UARTx**,  
uint32\_t **TransferMode**);
  
- ◆ void UART\_TRxAutoDisable(TSB\_SC\_TypeDef \* **UARTx**,  
UART\_TRxAutoDisable **TRxAutoDisable**);
  
- ◆ void UART\_RxFIFOINTCtrl(TSB\_SC\_TypeDef \* **UARTx**, FunctionalState **NewState**);
- ◆ void UART\_TxFIFOINTCtrl(TSB\_SC\_TypeDef \* **UARTx**, FunctionalState **NewState**);
- ◆ void UART\_RxFIFOByteSel(TSB\_SC\_TypeDef \* **UARTx**, uint32\_t **BytesUsed**);
- ◆ void UART\_RxFIFOFillLevel(TSB\_SC\_TypeDef \* **UARTx**, uint32\_t **RxFIFOLevel**);
- ◆ void UART\_RxFIFOINTSel(TSB\_SC\_TypeDef \* **UARTx**, uint32\_t **RxINTCondition**);
- ◆ void UART\_RxFIFOClear(TSB\_SC\_TypeDef \* **UARTx**);
- ◆ void UART\_TxFIFOFillLevel(TSB\_SC\_TypeDef \* **UARTx**, uint32\_t **TxFIFOLevel**);
- ◆ void UART\_TxFIFOINTSel(TSB\_SC\_TypeDef \* **UARTx**, uint32\_t **TxINTCondition**);
- ◆ void UART\_TxFIFOClear(TSB\_SC\_TypeDef \* **UARTx**);
- ◆ void UART\_TxBufferClear(TSB\_SC\_TypeDef \* **UARTx**);
- ◆ uint32\_t UART\_GetRxFIFOFillLevelStatus(TSB\_SC\_TypeDef \* **UARTx**);
- ◆ uint32\_t UART\_GetRxFIFOOverRunStatus(TSB\_SC\_TypeDef \* **UARTx**);
- ◆ uint32\_t UART\_GetTxFIFOFillLevelStatus(TSB\_SC\_TypeDef \* **UARTx**);
- ◆ uint32\_t UART\_GetTxFIFOUnderRunStatus(TSB\_SC\_TypeDef \* **UARTx**);
- ◆ void UART\_SetInputClock(TSB\_SC\_TypeDef \* **UARTx**, uint32\_t **clock**)

- ◆ void SIO\_SetInputClock(TSB\_SC\_TypeDef \* **SIOx**, uint32\_t **Clock**)
- ◆ void SIO\_Enable(TSB\_SC\_TypeDef\* **SIOx**)
- ◆ void SIO\_Disable(TSB\_SC\_TypeDef\* **SIOx**)
- ◆ void SIO\_Init(TSB\_SC\_TypeDef\* **SIOx**,  
                  uint32\_t **IOClkSel**,  
                  UART\_InitTypeDef\* **InitStruct**)
- ◆ uint8\_t SIO\_GetRxData(TSB\_SC\_TypeDef\* **SIOx**)
- ◆ void SIO\_SetTxData(TSB\_SC\_TypeDef\* **SIOx**, uint8\_t **Data**)

## 18.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています:

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

## 18.2.3 関数仕様

補足: 引数に記述している“TSB\_SC\_TypeDef\* **UARTx**” は、以下から選択してください。

**UART0, UART1, UART2, UART3, UART4, UART5, UART6, UART7, UART8, UART9**

引数に記述している“TSB\_SC\_TypeDef\* **SIOx**” は、以下から選択してください。

**SIO0, SIO1, SIO2, SIO3, SIO4, SIO5, SIO6, SIO7, SIO8, SIO9**

### 18.2.3.1 UART\_Enable

UART 動作の許可

関数のプロトタイプ宣言:

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

引数:

**UARTx**: UART チャ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

## 18.2.3.2 UART\_Disable

UART 動作の禁止

関数のプロトタイプ宣言:

```
void  
UART_Disable(TSB_SC_TypeDef* UARTx)
```

引数:

**UARTx**: UART チャ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

## 18.2.3.3 UART\_GetBufState

送受信バッファ状態の読み込み

関数のプロトタイプ宣言:

```
WorkState  
UART_GetBufState(TSB_SC_TypeDef* UARTx,  
uint8_t Direction)
```

引数:

**UARTx**: UART チャ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

## 18.2.3.4 UART\_SWReset

ソフトウェアリセット

関数のプロトタイプ宣言:

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



**引数:**

**UARTx:** UART チャンネルを指定します。

**機能:**

ソフトウェアリセットが発生します。

**戻り値:**

なし

## 18.2.3.5 UART\_Init

UART チャンネルの初期化

**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

## 18.2.3.6 UART\_GetRxData

受信データの読み込み

**関数のプロトタイプ宣言:**

```
uint32_t  
UART_GetRxData(TSB_SC_TypeDef* UARTx)
```

**引数:**

**UARTx:** UART チャ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 (シリアルチャネル) 割り込み関数の中で実行してください。

**戻り値:**

受信データです。データ範囲は 0x00~0x1FF です

## 18.2.3.7 UART\_SetTxData

送信データの設定

**関数のプロトタイプ宣言:**

```
void  
UART_SetTxData(TSB_SC_TypeDef* UARTx,  
               uint32_t Data)
```

**引数:**

**UARTx**: UART チャ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

## 18.2.3.8 UART\_DefaultConfig

デフォルト構成での初期化

**関数のプロトタイプ宣言:**

```
void  
UART_DefaultConfig(TSB_SC_TypeDef* UARTx)
```

**引数:**

**UARTx**: UART チャ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

## 18.2.3.9 UART\_GetErrState

転送エラーフラグの読み出し

**関数のプロトタイプ宣言:**

```
UART_Err  
UART_GetErrState(TSB_SC_TypeDef* UARTx)
```

**引数:**

**UARTx**: UART チャ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

## 18.2.3.10 UART\_SetWakeUpFunc

9 ビットモード時のウェイクアップ機能の設定

関数のプロトタイプ宣言:

void

UART\_SetWakeUpFunc(TSB\_SC\_TypeDef\* **UARTx**,  
FunctionalState **NewState**)

引数:

**UARTx**: UART チャ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

## 18.2.3.11 UART\_SetInputClock

プリスケアラの入カクロック選択

関数のプロトタイプ宣言:

void

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

引数:

**UARTx**: UART チャンネルを指定します。

**Clock**: プリスケアラの入カクロックを選択します。

➤ **0**: PhiT0/2

➤ **1**: PhiT0

機能:

プリスケアラの入カクロックを選択します。

戻り値:

なし

## 18.2.3.12 UART\_SetIdleMode

IDLE 時の動作

関数のプロトタイプ宣言:

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

引数:

**UARTx**: UART チャ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:

IDLE 時の動作を選択します。

**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

## 18.2.3.13 UART\_FIFOConfig

FIFO の許可

関数のプロトタイプ宣言:

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

引数:

**UARTx**: UART チャ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

## 18.2.3.14 UART\_SetFIFOTransferMode

転送モードの選択

関数のプロトタイプ宣言:

```
void
```

UART\_SetFIFOTransferMode(TSB\_SC\_TypeDef \* **UARTx**,  
uint32\_t **TransferMode**)

引数:

**UARTx**: UART チャ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

### 18.2.3.15 UART\_TRxAutoDisable

送信/受信の自動禁止

関数のプロトタイプ宣言:

void  
UART\_TRxAutoDisable (TSB\_SC\_TypeDef \* **UARTx**,  
UART\_TRxDisable **TRxAutoDisable**)

引数:

**UARTx**: UART チャ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

### 18.2.3.16 UART\_RxFIFOINTCtrl

受信 FIFO 使用時の受信割り込み許可

関数のプロトタイプ宣言:

void  
UART\_RxFIFOINTCtrl (TSB\_SC\_TypeDef \* **UARTx**,  
FunctionalState **NewState**)

引数:

**UARTx**: UART チャ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

## 18.2.3.17 UART\_TxFIFOINTCtrl

送信 FIFO 使用時の送信割り込み許可

**関数のプロトタイプ宣言:**

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

**引数:**

**UARTx:** UART チャ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

## 18.2.3.18 UART\_RxFIFOByteSel

受信 FIFO 使用バイト数

**関数のプロトタイプ宣言:**

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

**引数:**

**UARTx:** UART チャ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

## 18.2.3.19 UART\_RxFIFOFillLevel

受信割り込みが発生する受信 FIFO の fill レベルの設定

関数のプロトタイプ宣言:

```
void
UART_RxFIFOFillLevel (TSB_SC_TypeDef * UARTx,
                      uint32_t RxFIFOLevel)
```

引数:

**UARTx**: UART チャ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
UART_RXFIFO4B_FLEVLE_4_2B	4 バイト	2 バイト
UART_RXFIFO4B_FLEVLE_1_1B	1 バイト	1 バイト
UART_RXFIFO4B_FLEVLE_2_2B	2 バイト	2 バイト
UART_RXFIFO4B_FLEVLE_3_1B	3 バイト	1 バイト

機能:

受信割り込みが発生する受信 FIFO の fill レベルを選択します。

戻り値:

なし

## 18.2.3.20 UART\_RxFIFOINTSel

受信割り込み発生条件の選択

関数のプロトタイプ宣言:

```
void
UART_RxFIFOINTSel (TSB_SC_TypeDef * UARTx,
                   uint32_t RxINTCondition)
```

引数:

**UARTx**: UART チャ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

## 18.2.3.21 UART\_RxFIFOClear

受信 FIFO クリア

関数のプロトタイプ宣言:

```
void
UART_RxFIFOClear (TSB_SC_TypeDef * UARTx)
```

引数:

**UARTx**: UART チャ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

## 18.2.3.22 UART\_TxFIFOFillLevel

送信割り込みが発生する送信 FIFO の fill レベルの設定

関数のプロトタイプ宣言:

```
void
UART_TxFIFOFillLevel (TSB_SC_TypeDef * UARTx,
                      uint32_t TxFIFOLevel)
```

引数:

**UARTx**: UART チャ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。

戻り値:

なし

## 18.2.3.23 UART\_TxFIFOINTSel

送信割り込み発生条件の選択

関数のプロトタイプ宣言:

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

引数:

**UARTx**: UART チャ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。

**戻り値:**

なし

## 18.2.3.24 UART\_TxFIFOClear

送信 FIFO クリア

**関数のプロトタイプ宣言:**

void

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

**引数:**

**UARTx**: UART チャ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

## 18.2.3.25 UART\_TxBufferClear

送信バッファクリア

**関数のプロトタイプ宣言:**

void

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

**引数:**

**UARTx**: UART チャ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

## 18.2.3.26 UART\_GetRxFIFOFillLevelStatus

受信 FIFO の fill レベルの取得

**関数のプロトタイプ宣言:**

uint32\_t

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

**引数:**

**UARTx**: UART チャ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

### 18.2.3.27 UART\_GetRxFIFOOverRunStatus

受信 FIFO オーバーラン状態の取得

**関数のプロトタイプ宣言:**

uint32\_t

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

**引数:**

**UARTx:** UART チャ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
- **0:** オーバーランは発生していない

### 18.2.3.28 UART\_GetTxFIFOFillLevelStatus

送信 FIFO の fill レベルの取得

**関数のプロトタイプ宣言:**

uint32\_t

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

**引数:**

**UARTx:** UART チャ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

## 18.2.3.29 UART\_GetTxFIFOUnderRunStatus

送信 FIFO アンダーラン状態の取得

**関数のプロトタイプ宣言:**

uint32\_t

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

**引数:**

**UARTx:** UART チャンネルを指定します。

**機能:**

送信 FIFO アンダーラン状態を取得します。

**戻り値:**

- **UART\_TXFIFO\_UNDERRUN:** アンダーラン発生
- **0:** アンダーランは発生していない

## 18.2.3.30 SIO\_SetInputClock

プリスケアラの入カクロック選択

**関数のプロトタイプ宣言:**

void

SIO\_SetInputClock (TSB\_SC\_TypeDef \* SIOx,  
uint32\_t Clock)

**引数:**

**SIOx:** SIO チャンネルを指定します。

**Clock:** プリスケアラの入カクロックを選択します。

- **SIO\_CLOCK\_T0\_HALF:** PhiT0/2
- **SIO\_CLOCK\_T0:** PhiT0

**機能:**

プリスケアラの入カクロックを選択します。

**戻り値:**

なし

## 18.2.3.31 SIO\_Enable

SIO 動作の許可

**関数のプロトタイプ宣言:**

void

SIO\_Enable (TSB\_SC\_TypeDef\* **SIOx**)

**引数:**

**SIOx:** SIO チャ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

## 18.2.3.32 SIO\_Disable

SIO 動作の禁止

関数のプロトタイプ宣言:  
void  
SIO\_Disable(TSB\_SC\_TypeDef\* **SIOx**)

引数:  
**SIOx**: SIO チャ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

## 18.2.3.33 SIO\_GetRxData

受信用バッファ

関数のプロトタイプ宣言:  
uint32\_t  
SIO\_GetRxData(TSB\_SC\_TypeDef\* **SIOx**)

引数:  
**SIOx**: SIO チャ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)

## 18.2.3.34 SIO\_SetTxData

送信用バッファ

関数のプロトタイプ宣言:  
void  
SIO\_SetTxData(TSB\_SC\_TypeDef\* **SIOx**,  
uint8\_t **Data**)

引数:  
**SIOx**: SIO チャ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

## 18.2.3.35 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**: ポ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:

ポーレート、転送方向、転送モードなどの初期設定を行います。

戻り値:

なし

## 18.2.4 データ構造

### 18.2.4.1 UART\_InitTypeDef

メンバ:

uint32\_t

**BaudRate**: UART 通信ポーレートを 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

## 18.2.4.2 SIO\_InitTypeDef

メンバ:

uint32\_t

**InputClkEdge**: 入力クロックエッジを選択します。

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

uint32\_t

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

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

uint32\_t

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

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

uint32\_t

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

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

uint32\_t

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

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

uint32\_t

**TransferMode**: 転送モードを選択します。

- **SIO\_TRANSFER\_PROHIBIT**: 転送禁止

- SIO\_TRANSFER\_HALFDPX\_RX: 半二重(受信)
- SIO\_TRANSFER\_HALFDPX\_TX: 半二重(送信)
- SIO\_TRANSFER\_FULDPX: 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

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

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

## 19. uDMAC

### 19.1 概要

本デバイスには、マイクロ DMAC (uDMAC)モジュールを内蔵しています。

主な機能は以下の通りです。

Functions	Features		Descriptions
Channels	32 channels		-
Start trigger	Start by Hardware		DMA requests from peripheral functions
	Start by Software		Specified by DMAxChnlSwRequest register
Priority	Between channels	ch0 (high priority) > ... > ch31 (high priority) > ch0 (Normal priority) > ... > ch31 (Normal priority)	High-priority can be configured by DMAxChnlPriority-Set register
Transfer data size	8/16/32bit		Can be specified source and destination independently
The number of transfer	1 to 4095 times		-
Address	Transfer source address	Increment / fixed	Transfer source address and destination address can be selected to increment or fixed.
	transfer destination address	Increment / fixed	
Endian	Little Endian		-
Transfer type	Peripheral (register) → memory Memory → peripheral (register) Memory → memory		If you select memory to memory, hardware start for DMA start up is not supported. Refer to the DMAxConfiguration register for more information.
Interrupt function	Transfer end interrupt Error interrupt		Output for each unit
Transfer mode	Basic mode Automatic request mode Ping-pong mode Memory scatter / gather mode Peripheral scatter / gather mode		-

μDMAC API は、MCU の μDMAC モジュールを使用するための機能セットを提供します。本 API には、μDMAC 転送タイプセット、チャンネルセット、マスクセット、初期/代替データエリアセット、チャンネル優先、初期化データ設定などが含まれます。

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

/Libraries/TX04\_Periph\_Driver/src\tmpm462\_udmac.c

/Libraries/TX04\_Periph\_Driver/inc\tmpm462\_udmac.h

\*補足: 本ドキュメントでは、DMAC は μDMAC を意味します。

### 19.2 API 関数

#### 19.2.1 関数一覧

- ◆ FunctionalState DMAC\_GetDMACState(TSB\_DMA\_TypeDef \* **DMACx**)
- ◆ void DMAC\_Enable(TSB\_DMA\_TypeDef \* **DMACx**)
- ◆ void DMAC\_Disable(TSB\_DMA\_TypeDef \* **DMACx**)
- ◆ void DMAC\_SetPrimaryBaseAddr(TSB\_DMA\_TypeDef \* **DMACx**, uint32\_t **Addr**)
- ◆ uint32\_t DMAC\_GetBaseAddr(TSB\_DMA\_TypeDef \* **DMACx**,



- ◆ void DMAC\_SetSWReq(TSB\_DMA\_TypeDef \* **DMACx** ,  
uint8\_t **Channel**)
- ◆ void DMACA\_SetTransferType(DMACA\_Channel **Channel**,  
DMAC\_TransferType **Type**)
- ◆ DMAC\_TransferType DMACA\_GetTransferType( DMACA\_Channel **Channel**)
- ◆ void DMAC\_SetMask(TSB\_DMA\_TypeDef \* **DMACx** ,  
uint8\_t **Channel** ,  
FunctionalState **NewState**)
- ◆ FunctionalState DMAC\_GetMask(TSB\_DMA\_TypeDef \* **DMACx** ,  
uint8\_t **Channel** )
- ◆ void DMAC\_SetChannel(TSB\_DMA\_TypeDef \* **DMACx** ,  
uint8\_t **Channel** ,  
FunctionalState **NewState**)
- ◆ FunctionalState DMAC\_GetChannelState(TSB\_DMA\_TypeDef \* **DMACx** ,  
uint8\_t **Channel** )
- ◆ void DMAC\_SetPrimaryAlt(TSB\_DMA\_TypeDef \* **DMACx** ,  
uint8\_t **Channel** ,  
DMAC\_PrimaryAlt **PriAlt**)
- ◆ DMAC\_PrimaryAlt DMAC\_GetPrimaryAlt(TSB\_DMA\_TypeDef \* **DMACx** ,  
uint8\_t **Channel**)
- ◆ void DMAC\_SetChannelPriority(TSB\_DMA\_TypeDef \* **DMACx** ,  
uint8\_t **Channel** ,  
DMAC\_Priority **Priority**)
- ◆ DMAC\_Priority DMAC\_GetChannelPriority(TSB\_DMA\_TypeDef \* **DMACx** ,  
uint8\_t **Channel**)
- ◆ void DMAC\_ClearBusErr(TSB\_DMA\_TypeDef \* **DMACx**)
- ◆ Result DMAC\_GetBusErrState(TSB\_DMA\_TypeDef \* **DMACx**)
- ◆ void DMAC\_FillInitData(TSB\_DMA\_TypeDef \* **DMACx** ,  
uint8\_t **Channel** ,  
DMAC\_InitTypeDef \* **InitStruct**)
- ◆ DMACA\_Flag DMACA\_GetINTFlag(void)
- ◆ DMACB\_Flag DMACB\_GetINTFlag(void)
- ◆ DMACC\_Flag DMACC\_GetINTFlag(void)

## 19.2.2 関数の種類

関数は、主に以下の 6 種類に分かれています:

- 1) DMAC の設定:  
DMACA\_SetTransferType(), DMACA\_GetTransferType(), DMAC\_SetMask(),  
DMAC\_GetMask(), DMAC\_SetChannel(), DMAC\_GetChannelState(),  
DMAC\_SetPrimaryAlt(), DMAC\_GetPrimaryAlt(), DMAC\_SetChannelPriority(),  
DMAC\_GetChannelPriority()
- 2) DMAC の許可/禁止:  
DMAC\_GetDMACState(), DMAC\_Enable(), DMAC\_Disable()
- 3) ソフトウェアトリガ制御:  
DMAC\_SetSWReq()
- 4) バスエラー監視:  
DMAC\_ClearBusErr(), DMAC\_GetBusErrState()
- 5) 制御データエリア設定:  
DMAC\_FillInitData(), DMAC\_SetPrimaryBaseAddr(), DMAC\_GetBaseAddr()
- 6) DMA 要因の取得:  
DMACA\_GetINTFlag(), DMACB\_GetINTFlag(), DMACC\_GetINTFlag()

## 19.2.3 関数仕様

補足: 引数に記述している“DMACx”および“Channel”は、特に断りのない限り、以下から選択してください。

**DMACx:** ユニット選択です。

- **DMAC\_UNIT\_A:** DMAC ユニット A
- **DMAC\_UNIT\_B:** DMAC ユニット B
- **DMAC\_UNIT\_C:** DMAC ユニット C

**Channel:** チャネル選択です。

[DMAC\_UNIT\_A の場合]

- **DMACA\_ADC\_COMPLETION:** ADC 変換終了
- **DMACA\_SSP0\_RX:** SSP0 受信
- **DMACA\_SSP0\_TX:** SSP0 送信
- **DMACA\_SSP1\_RX:** SSP1 受信
- **DMACA\_SSP1\_TX:** SSP1 送信
- **DMACA\_SSP2\_RX:** SSP2 受信
- **DMACA\_SSP2\_TX:** SSP2 送信
- **DMACA\_UART0\_RX:** UART0 受信
- **DMACA\_UART0\_TX:** UART0 送信
- **DMACA\_UART1\_RX:** UART1 受信
- **DMACA\_UART1\_TX:** UART1 送信
- **DMACA\_I2C0\_TX\_RX:** I2C0 送受信
- **DMACA\_I2C1\_TX\_RX:** I2C1 送受信
- **DMACA\_I2C2\_TX\_RX:** I2C2 送受信
- **DMACA\_I2C3\_TX\_RX:** I2C3 送受信
- **DMACA\_I2C4\_TX\_RX:** I2C4 送受信
- **DMACA\_DMAREQA:** DMA ユニット A リクエストピン MAREQA

[DMAC\_UNIT\_B の場合]

- **DMACB\_SIO0\_UART0\_RX:** SIO/UART0 受信
- **DMACB\_SIO0\_UART0\_TX:** SIO/UART0 送信
- **DMACB\_SIO1\_UART1\_RX:** SIO/UART1 受信
- **DMACB\_SIO1\_UART1\_TX:** SIO/UART1 送信
- **DMACB\_SIO2\_UART2\_RX:** SIO/UART2 受信
- **DMACB\_SIO2\_UART2\_TX:** SIO/UART2 送信
- **DMACB\_SIO6\_UART6\_RX:** SIO/UART6 受信
- **DMACB\_SIO6\_UART6\_TX:** SIO/UART6 送信
- **DMACB\_SIO7\_UART7\_RX:** SIO/UART7 受信
- **DMACB\_SIO7\_UART7\_TX:** SIO/UART7 送信
- **DMACB\_TMRB0\_CMP\_MATCH:** TMRB0 コンペア一致
- **DMACB\_TMRB1\_CMP\_MATCH:** TMRB1 コンペア一致
- **DMACB\_TMRB2\_CMP\_MATCH:** TMRB2 コンペア一致
- **DMACB\_TMRB3\_CMP\_MATCH:** TMRB3 コンペア一致
- **DMACB\_TMRB4\_CMP\_MATCH:** TMRB4 コンペア一致
- **DMACB\_TMRB5\_CMP\_MATCH:** TMRB5 コンペア一致
- **DMACB\_TMRB6\_CMP\_MATCH:** TMRB6 コンペア一致
- **DMACB\_TMRB7\_CMP\_MATCH:** TMRB7 コンペア一致
- **DMACB\_TMRBF\_CMP\_MATCH:** TMRBF コンペア一致
- **DMACB\_TMRB0\_INPUT\_CAP0:** TMRB0 インพุットキャプチャ 0
- **DMACB\_TMRB0\_INPUT\_CAP1:** TMRB0 インพุットキャプチャ 1

- **DMACB\_TMRB1\_INPUT\_CAP0** : TMRB1 インพุットキャプチャ 0
- **DMACB\_TMRB1\_INPUT\_CAP1** : TMRB1 インพุットキャプチャ 1
- **DMACB\_TMRB2\_INPUT\_CAP0** : TMRB2 インพุットキャプチャ 0
- **DMACB\_TMRB2\_INPUT\_CAP1** : TMRB2 インพุットキャプチャ 1
- **DMACB\_TMRB3\_INPUT\_CAP0** : TMRB3 インพุットキャプチャ 0
- **DMACB\_TMRB3\_INPUT\_CAP1** : TMRB3 インพุットキャプチャ 1
- **DMACB\_TMRB4\_INPUT\_CAP0** : TMRB4 インพุットキャプチャ 0
- **DMACB\_TMRB4\_INPUT\_CAP1** : TMRB4 インพุットキャプチャ 1
- **DMACB\_TMRB5\_INPUT\_CAP0** : TMRB5 インพุットキャプチャ 0
- **DMACB\_TMRB5\_INPUT\_CAP1** : TMRB5 インพุットキャプチャ 1
- **DMACB\_DMAREQB** : DMA ユニット B リクエストピン DMAREQB

[DMAC\_UNIT\_C の場合]

- **DMACC\_SIO3\_UART3\_RX** : SIO/UART3 受信
- **DMACC\_SIO3\_UART3\_TX** : SIO/UART3 送信
- **DMACC\_SIO4\_UART4\_RX** : SIO/UART4 受信
- **DMACC\_SIO4\_UART4\_TX** : SIO/UART4 送信
- **DMACC\_SIO5\_UART5\_RX** : SIO/UART5 受信
- **DMACC\_SIO5\_UART5\_TX** : SIO/UART5 送信
- **DMACC\_SIO8\_UART8\_RX** : SIO/UART8 受信
- **DMACC\_SIO8\_UART8\_TX** : SIO/UART8 送信
- **DMACC\_SIO9\_UART9\_RX** : SIO/UART9 受信
- **DMACC\_SIO9\_UART9\_TX** : SIO/UART9 送信
- **DMACC\_TMRB8\_CMP\_MATCH** : TMRB8 コンペアー致
- **DMACC\_TMRB9\_CMP\_MATCH** : TMRB9 コンペアー致
- **DMACC\_TMRBA\_CMP\_MATCH** : TMRBA コンペアー致
- **DMACC\_TMRBB\_CMP\_MATCH** : TMRBB コンペアー致
- **DMACC\_TMRBC\_CMP\_MATCH** : TMRBC コンペアー致
- **DMACC\_TMRBD\_CMP\_MATCH** : TMRBD コンペアー致
- **DMACC\_TMRBE\_CMP\_MATCH** : TMRBE コンペアー致
- **DMACC\_TMRB7\_CMP\_MATCH** : TMRB7 コンペアー致
- **DMACC\_TMRBF\_CMP\_MATCH** : TMRBF コンペアー致
- **DMACC\_TMRB8\_INPUT\_CAP0** : TMRB8 インพุットキャプチャ 0
- **DMACC\_TMRB8\_INPUT\_CAP1** : TMRB8 インพุットキャプチャ 1
- **DMACC\_TMRB9\_INPUT\_CAP0** : TMRB9 インพุットキャプチャ 0
- **DMACC\_TMRB9\_INPUT\_CAP1** : TMRB9 インพุットキャプチャ 1
- **DMACC\_TMRBA\_INPUT\_CAP0** : TMRBA インพุットキャプチャ 0
- **DMACC\_TMRBA\_INPUT\_CAP1** : TMRBA インพุットキャプチャ 1
- **DMACC\_TMRBB\_INPUT\_CAP0** : TMRBB インพุットキャプチャ 0
- **DMACC\_TMRBB\_INPUT\_CAP1** : TMRBB インพุットキャプチャ 1
- **DMACC\_TMRBC\_INPUT\_CAP0** : TMRBC インพุットキャプチャ 0
- **DMACC\_TMRBC\_INPUT\_CAP1** : TMRBC インพุットキャプチャ 1
- **DMACC\_TMRBD\_INPUT\_CAP0** : TMRBD インพุットキャプチャ 0
- **DMACC\_TMRBD\_INPUT\_CAP1** : TMRBD インพุットキャプチャ 1
- **DMACC\_DMAREQC** : DMA ユニット C リクエストピン DMAREQC

### 19.2.3.1 DMAC\_GetDMACState

DMAC ユニットの許可/禁止状態の読み出し

関数のプロトタイプ宣言:

FunctionalState

DMAC\_GetDMACState(TSB\_DMA\_TypeDef \* **DMACx**)

引数:

**DMACx**: DMAC ユニットを選択します。

機能:

DMAC ユニットの許可/禁止状態を読み出します。

戻り値:

- **DISABLE**: 禁止状態
- **ENABLE**: 許可状態

### 19.2.3.2 DMAC\_Enable

DMAC ユニット動作の許可

関数のプロトタイプ宣言:

void  
DMAC\_Enable(TSB\_DMA\_TypeDef \* **DMACx**)

引数:

**DMACx**: DMAC ユニットを選択します。

機能:

DMAC ユニット動作を許可します。

戻り値:

なし

### 19.2.3.3 DMAC\_Disable

DMAC ユニット動作の禁止

関数のプロトタイプ宣言:

void  
DMAC\_Disable(TSB\_DMA\_TypeDef \* **DMACx**)

引数:

**DMACx**: DMAC ユニットを選択します。

機能:

DMAC ユニット動作を禁止します。

戻り値:

なし

### 19.2.3.4 DMAC\_SetPrimaryBaseAddr

DMAC ユニットの一次データのベースアドレスの設定

関数のプロトタイプ宣言:

```
void  
DMAC_SetPrimaryBaseAddr(TSB_DMA_TypeDef * DMACx,  
                        uint32_t Addr)
```

**引数:**

**DMACx:** DMAC ユニットを選択します。

**Addr:** 一次データのベースアドレスを指定します。ビット 0 ~9 は 0 に設定してください。

**機能:**

DMAC ユニットの一次データのベースアドレスを設定します。

**戻り値:**

なし

## 19.2.3.5 DMAC\_GetBaseAddr

DMAC ユニットの一次/代替ベースアドレスの取得

**関数のプロトタイプ宣言:**

```
uint32_t  
DMAC_GetBaseAddr(TSB_DMA_TypeDef * DMACx,  
                 DMAC_PrimaryAlt PriAlt)
```

**引数:**

**DMACx:** DMAC ユニットを選択します。

**PriAlt:** ベースアドレスタイプを選択します。

- **DMAC\_PRIMARY:** 一次ベースアドレス
- **DMAC\_ALTERNATE:** 代替ベースアドレス

**機能:**

DMAC ユニットの初期/代替ベースアドレスを取得します。

**戻り値:**

初期/代替データのベースアドレス

## 19.2.3.6 DMAC\_SetSWReq

ソフトウェア転送要求の設定

**関数のプロトタイプ宣言:**

```
void  
DMAC_SetSWReq(TSB_DMA_TypeDef * DMACx,  
              DMAC_Channel Channel)
```

**引数:**

**DMACx:** DMAC ユニットを選択します。

**Channel:** チャンネルを選択します。

**機能:**

ソフトウェア転送要求を設定します。

戻り値:  
なし

## 19.2.3.7 DMACA\_SetTransferType

DMAC ユニット A の転送タイプの設定

関数のプロトタイプ宣言:

```
void  
DMACA_SetTransferType(uint8_t Channel,  
                      DMAC_TransferType Type)
```

引数:

**Channel**: ユニット A のチャネルを選択します。

**Type** が DMAC\_BURST の場合:

- DMACA\_ADC\_COMPLETION : ADC 変換終了
- DMACA\_SSP0\_RX : SSP0 受信
- DMACA\_SSP0\_TX : SSP0 送信
- DMACA\_SSP1\_RX : SSP1 受信
- DMACA\_SSP1\_TX : SSP1 送信
- DMACA\_SSP2\_RX : SSP2 受信
- DMACA\_SSP2\_TX : SSP2 送信
- DMACA\_UART0\_RX : UART0 受信
- DMACA\_UART0\_TX : UART0 送信
- DMACA\_UART1\_RX : UART1 受信
- DMACA\_UART1\_TX : UART1 送信
- DMACA\_I2C0\_TX\_RX : I2C0 送受信
- DMACA\_I2C1\_TX\_RX : I2C1 送受信
- DMACA\_I2C2\_TX\_RX : I2C2 送受信
- DMACA\_I2C3\_TX\_RX : I2C3 送受信
- DMACA\_I2C4\_TX\_RX : I2C4 送受信
- DMACA\_DMAREQA : DMA ユニット A リクエストピン DMAREQA

**Type** が DMAC\_SINGLE の場合:

- DMACA\_SSP0\_RX : SSP0 受信
- DMACA\_SSP0\_TX : SSP0 送信
- DMACA\_SSP1\_RX : SSP1 受信
- DMACA\_SSP1\_TX : SSP1 送信
- DMACA\_SSP2\_RX : SSP2 受信
- DMACA\_SSP2\_TX : SSP2 送信
- DMACA\_UART0\_RX : UART0 受信
- DMACA\_UART0\_TX : UART0 送信
- DMACA\_UART1\_RX : UART1 受信
- DMACA\_UART1\_TX : UART1 送信

**Type**: 転送タイプを選択します。

- DMAC\_BURST : シングル転送が禁止され、バースト転送要求のみが有効になります。
- DMAC\_SINGLE : シングル転送。

機能:

転送タイプを設定します。

戻り値:  
なし

## 19.2.3.8 DMACA\_GetTransferType

ユニット A の転送タイプの読み出し

関数のプロトタイプ宣言:

```
DMAC_TransferType  
DMACA_GetTransferType( uint8_t Channel)
```

引数:

**Channel:** ユニット A のチャンネルを選択します。

- **DMACA\_ADC\_COMPLETION** : ADC 変換終了
- **DMACA\_SSP0\_RX** : SSP0 受信
- **DMACA\_SSP0\_TX** : SSP0 送信
- **DMACA\_SSP1\_RX** : SSP1 受信
- **DMACA\_SSP1\_TX** : SSP1 送信
- **DMACA\_SSP2\_RX** : SSP2 受信
- **DMACA\_SSP2\_TX** : SSP2 送信
- **DMACA\_UART0\_RX** : UART0 受信
- **DMACA\_UART0\_TX** : UART0 送信
- **DMACA\_UART1\_RX** : UART1 受信
- **DMACA\_UART1\_TX** : UART1 送信
- **DMACA\_I2C0\_TX\_RX** : I2C0 送受信
- **DMACA\_I2C1\_TX\_RX** : I2C1 送受信
- **DMACA\_I2C2\_TX\_RX** : I2C2 送受信
- **DMACA\_I2C3\_TX\_RX** : I2C3 送受信
- **DMACA\_I2C4\_TX\_RX** : I2C4 送受信
- **DMACA\_DMAREQA** : DMA ユニット A リクエストピン DMAREQA

機能:

転送タイプを読み出します。

戻り値:

転送タイプ:

- **DMAC\_BURST** : シングル転送が禁止され、バースト転送要求のみが有効
- **DMAC\_SINGLE** : シングル転送

## 19.2.3.9 DMAC\_SetMask

DMA 要求のマスク設定/クリア制御

関数のプロトタイプ宣言:

```
void  
DMAC_SetMask(TSB_DMA_TypeDef * DMACx,  
              DMAC_Channel Channel,  
              FunctionalState NewState)
```

引数:

**DMACx:** DMAC ユニットを選択します。

**Channel:** チャンネルを選択します。

**NewState:** DMA 要求のマスク設定/クリアを選択します。

- **ENABLE:** DMA 要求マスクのクリア。
- **DISABLE:** DMA 要求のマスク設定

**機能:**

DMA 要求のマスク設定/クリアを選択します。

**戻り値:**

なし

## 19.2.3.10 DMAC\_GetMask

DMA 要求のマスク状態の読み出し

**関数のプロトタイプ宣言:**

FunctionalState

DMAC\_GetMask(TSB\_DMA\_TypeDef \* **DMACx**,  
DMAC\_Channel **Channel**)

**引数:**

**DMACx:** DMAC ユニットを選択します。

**Channel:** チャンネルを選択します。

**機能:**

DMA 要求のマスク状態を読み出します。

**戻り値:**

DMA 要求のマスク状態:

- **ENABLE:** DMA 要求のマスク設定なし
- **DISABLE:** DMA 要求のマスク設定あり

## 19.2.3.11 DMAC\_SetChannel

DMA チャンネルの有効/無効設定

**関数のプロトタイプ宣言:**

void

DMAC\_SetChannel(TSB\_DMA\_TypeDef \* **DMACx**,  
DMAC\_Channel **Channel**,  
FunctionalState **NewState**)

**引数:**

**DMACx:** DMAC ユニットを選択します。

**Channel:** チャンネルを選択します。

**NewState:** DMA チャンネルの有効/無効を選択します。

- **ENABLE:** 有効
- **DISABLE:** 無効

**機能:**



DMA チャンネルの有効/無効を設定します。

戻り値:

なし

## 19.2.3.12 DMAC\_GetChannelState

DMA チャンネルの有効/無効状態の取得

関数のプロトタイプ宣言:

FunctionalState

DMAC\_GetChannelState(TSB\_DMA\_TypeDef \* **DMACx**,  
DMAC\_Channel **Channel**)

引数:

**DMACx**: DMAC ユニットを選択します。

**Channel**: チャンネルを選択します。

機能:

DMA チャンネルの有効/無効状態を取得します。

戻り値:

DMA チャンネルの有効/無効状態:

- **ENABLE**: 有効
- **DISABLE**: 無効

## 19.2.3.13 DMAC\_SetPrimaryAlt

一次データあるいは代替データの選択

関数のプロトタイプ宣言:

void

DMAC\_SetPrimaryAlt(TSB\_DMA\_TypeDef \* **DMACx**,  
DMAC\_Channel **Channel**,  
DMAC\_PrimaryAlt **PriAlt**)

引数:

**DMACx**: DMAC ユニットを選択します。

**Channel**: チャンネルを選択します。

**PriAlt**: 一次データあるいは代替データを選択します。

- **DMAC\_PRIMARY**: 一次データ使用
- **DMAC\_ALTERNATE**: 代替データ使用

機能:

一次データあるいは代替データの使用有無を設定します。

戻り値:

なし

## 19.2.3.14 DMAC\_GetPrimaryAlt

一次データあるは代替データの選択状態の読み出し

関数のプロトタイプ宣言:

```
DMAC_PrimaryAlt  
DMAC_GetPrimaryAlt(TSB_DMA_TypeDef * DMACx,  
                   DMAC_Channel Channel)
```

引数:

**DMACx**: DMAC ユニットを選択します。

**Channel**: チャンネルを選択します。

機能:

一次データあるは代替データの選択状態を読み出します。

戻り値:

一次データあるは代替データの選択状態:

- **DMAC\_PRIMARY**: 一次データ
- **DMAC\_ALTERNATE**: 代替データ

## 19.2.3.15 DMAC\_SetChannelPriority

優先度の設定

関数のプロトタイプ宣言:

```
void  
DMAC_SetChannelPriority(TSB_DMA_TypeDef * DMACx,  
                      DMAC_Channel Channel,  
                      DMAC_Priority Priority)
```

引数:

**DMACx**: DMAC ユニットを選択します。

**Channel**: チャンネルを選択します。

**Priority**: 優先順位を選択します。

- **DMAC\_PRIOTIRY\_NORMAL**: 通常優先度
- **DMAC\_PRIOTIRY\_HIGH**: 高優先度

機能:

優先度を設定します。

戻り値:

なし

## 19.2.3.16 DMAC\_GetChannelPriority

優先度の読み出し

関数のプロトタイプ宣言:

```
DMAC_Priority  
DMAC_GetChannelPriority(TSB_DMA_TypeDef * DMACx,  
                      DMAC_Channel Channel)
```

**引数:**

**DMACx:** DMAC ユニットを選択します。

**Channel:** チャンネルを選択します。

**機能:**

優先度の設定状態を読み出します。

**戻り値:**

優先度の設定状態:

- **DMAC\_PRIOTIRY\_NORMAL:** 通常優先度
- **DMAC\_PRIOTIRY\_HIGH:** 高優先度

## 19.2.3.17 DMAC\_ClearBusErr

バスエラー解除

**関数のプロトタイプ宣言:**

void

DMAC\_ClearBusErr(TSB\_DMA\_TypeDef \* **DMACx**)

**引数:**

**DMACx:** DMAC ユニットを選択します。

**機能:**

バスエラーを解除します。

**戻り値:**

なし

## 19.2.3.18 DMAC\_GetBusErrState

バスエラー状態の読み出し

**関数のプロトタイプ宣言:**

Result

DMAC\_GetBusErrState(TSB\_DMA\_TypeDef \* **DMACx**)

**引数:**

**DMACx:** DMAC ユニットを選択します。

**機能:**

バスエラー状態を読み出します。

**戻り値:**

バスエラー状態:

- **SUCCESS:** バスエラーなし
- **ERROR:** バスエラー状態

## 19.2.3.19 DMAC\_FillInitData

DMA 設定状態の読み出し

**関数のプロトタイプ宣言:**

```
void  
DMAC_FillInitData(TSB_DMA_TypeDef * DMACx,  
                  DMAC_Channel Channel,  
                  DMAC_InitTypeDef * InitStruct)
```

**引数:**

**DMACx:** DMAC ユニットを選択します。

**Channel:** チャンネルを選択します。

**InitStruct:** DMA 設定に関する構造体です。

**機能:**

DMA 設定状態を読み出します。

**戻り値:**

なし

## 19.2.3.20 DMACA\_GetINTFlag

ユニット A の DMA 要因の取得

**関数のプロトタイプ宣言:**

```
DMACA_Flag  
DMACA_GetINTFlag(void)
```

**引数:**

なし

**機能:**

ユニット A の DMA 要因を取得します。

**戻り値:**

ユニット A の DMA 要因の構造体 (詳細は、"データ構造"を参照してください)

## 19.2.3.21 DMACB\_GetINTFlag

ユニット B の DMA 要因の取得

**関数のプロトタイプ宣言:**

```
DMACB_Flag  
DMACB_GetINTFlag(void)
```

**引数:**

なし

**機能:**

ユニット B の DMA 要因を取得します。

**戻り値:**

ユニット B の DMA 要因の構造体 (詳細は、"データ構造"を参照してください)

## 19.2.3.22 DMACC\_GetINTFlag

ユニット C の DMA 要因の取得

関数のプロトタイプ宣言:

DMACC\_Flag

DMACC\_GetINTFlag(void)

引数:

なし

機能:

ユニット C の DMA 要因を取得します。

戻り値:

ユニット C の DMA 要因の構造体 (詳細は、"データ構造"を参照してください)

## 19.2.4 データ構造

### 19.2.4.1 DMAC\_InitTypeDef

メンバ:

uint32\_t

**SrcEndPoint**: データ送信元最終アドレス

uint32\_t

**DstEndPoint**: データ送信先最終アドレス

DMAC\_CycleCtrl

**Mode**: 動作モード

- **DMAC\_INVALID**: 無効。DMA は動作を停止します。
- **DMAC\_BASIC**: 基本モード
- **DMAC\_AUTOMATIC**: 自動要求モード
- **DMAC\_PINGPONG**: ピンポンモード
- **DMAC\_MEM\_SCATTER\_GATHER\_PRI**: メモリスキャッターギャザーモード (一次データ)
- **DMAC\_MEM\_SCATTER\_GATHER\_ALT**: メモリスキャッターギャザーモード (代替データ)
- **DMAC\_PERI\_SCATTER\_GATHER\_PRI**: 周辺スキャッターギャザーモード (一次データ)
- **DMAC\_PERI\_SCATTER\_GATHER\_ALT**: 周辺スキャッターギャザーモード (代替データ)

DMAC\_Next\_UseBurst

**NextUseBurst**: 周辺スキャッターギャザーモードで代替データを用いた DMA 転送終了時に<chnl\_useburst\_set>ビットに"1"を設定するかどうかを指定します。

- **DMAC\_NEXT\_NOT\_USE\_BURST**: <chnl\_useburst\_set>の値を変更しない。
- **DMAC\_NEXT\_USE\_BURST**: <chnl\_useburst\_set> に"1"を設定する。

uint32\_t

**TxNum:** 転送数(最大値は 1024 回)

DMAC\_Arbitration

**ArbitrationMoment:** アービトレーションを選択します。設定した回数の転送後に転送要求を確認し、優先度の高い要求があれば制御が高優先度のチャンネルに切り替わります。

DMAC\_BitWidth

**SrcWidth:** 転送元データサイズ

- **DMAC\_BYTE:** 1 バイト
- **DMAC\_HALF\_WORD:** 2 バイト
- **DMAC\_WORD:** 4 バイト

DMAC\_IncWidth

**SrcInc:** 転送元アドレスのインクリメント

- **DMAC\_INC\_1B:** 1 バイト
- **DMAC\_INC\_2B:** 2 バイト
- **DMAC\_INC\_4B:** 4 バイト
- **DMAC\_INC\_0B:** インクリメントなし

DMAC\_BitWidth

**DstWidth:** 転送先データサイズ

- **DMAC\_BYTE:** 1 バイト
- **DMAC\_HALF\_WORD:** 2 バイト
- **DMAC\_WORD:** 4 バイト

DMAC\_IncWidth

**DstInc:** 転送先アドレスのインクリメント

- **DMAC\_INC\_1B:** 1 バイト
- **DMAC\_INC\_2B:** 2 バイト
- **DMAC\_INC\_4B:** 4 バイト
- **DMAC\_INC\_0B:** インクリメントなし

## 19.2.4.2 DMACA\_Flag

メンバ:

uint32\_t

**All** ユニット A のすべての DMA 要因

ビットフィールド: 各ビットの値の意味は以下の通りです。

‘0’: 終了割り込みは発生していない

‘1’: 終了割り込みが発生

uint32\_t

**ADCCompletion** (Bit 0) AD 変換終了による終了割り込み

uint32\_t

**SSP0Reception** (Bit 1) SSP0 受信による終了割り込み

uint32\_t

**SSP0Transmission** (Bit 2) SSP0 送信による終了割り込み

uint32\_t

**SSP1Reception** (Bit 3) SSP1 受信による終了割り込み

```

uint32_t
SSP1Transmission (Bit 4)    SSP1 送信による終了割り込み
uint32_t
SSP2Reception (Bit 5)    SSP2 受信による終了割り込み
uint32_t
SSP2Transmission (Bit 6)    SSP2 送信による終了割り込み
uint32_t
UART0Reception (Bit 7)    UART0 受信による終了割り込み
uint32_t
UART0Transmission (Bit 8)    UART0 送信による終了割り込み
uint32_t
UART1Reception (Bit 9)    UART1 受信による終了割り込み
uint32_t
UART1Transmission (Bit 10)    UART1 送信による終了割り込み
uint32_t
I2C0RxorTx (Bit 11)        I2C0 送受信による終了割り込み
uint32_t
I2C1RxorTx (Bit 12)        I2C1 送受信による終了割り込み
uint32_t
I2C2RxorTx (Bit 13)        I2C2 送受信による終了割り込み
uint32_t
I2C3RxorTx (Bit 14)        I2C3 送受信による終了割り込み
uint32_t
I2C4RxorTx (Bit 15)        I2C4 送受信による終了割り込み
uint32_t
Reserved (Bit 16 ~ Bit 30)    未使用
uint32_t
DMAREQA (Bit 31)            DMA ユニット A リクエストピン
DMAREQA による終了割り込み

```

## 19.2.4.3 DMACB\_Flag

メンバ:

```
uint32_t
```

**All** DMA UINTB interrupt.

ビットフィールド: 各ビットの値の意味は以下の通りです。

‘0’: 終了割り込みは発生していない

‘1’: 終了割り込みが発生

```

uint32_t
SIO_UART0Reception (Bit 0) SIO/UART0 受信による終了割り込み
uint32_t
SIO_UART0Transmission (Bit 1) SIO/UART0 送信による終了割り込み
uint32_t
SIO_UART1Reception (Bit 2) SIO/UART1 受信による終了割り込み
uint32_t
SIO_UART1Transmission (Bit 3) SIO/UART1 送信による終了割り込み
uint32_t
SIO_UART2Reception (Bit 4) SIO/UART2 受信による終了割り込み
uint32_t
SIO_UART2Transmission (Bit 5) SIO/UART2 送信による終了割り込み
uint32_t
SIO_UART6Reception (Bit 6) SIO/UART6 受信による終了割り込み

```

uint32\_t  
**SIO\_UART6Transmission** (Bit 7) SIO/UART6 送信による終了割り込み

uint32\_t  
**SIO\_UART7Reception** (Bit 8) SIO/UART7 受信による終了割り込み

uint32\_t  
**SIO\_UART7Transmission** (Bit 9) SIO/UART7 送信による終了割り込み

uint32\_t  
**TMRB0CompareMatch** (Bit 10) TMRB0 コンペア一致による終了割り込み

uint32\_t  
**TMRB1CompareMatch** (Bit 11) TMRB1 コンペア一致による終了割り込み

uint32\_t  
**TMRB2CompareMatch** (Bit 12) TMRB2 コンペア一致による終了割り込み

uint32\_t  
**TMRB3CompareMatch** (Bit 13) TMRB3 コンペア一致による終了割り込み

uint32\_t  
**TMRB4CompareMatch** (Bit 14) TMRB4 コンペア一致による終了割り込み

uint32\_t  
**TMRB5CompareMatch** (Bit 15) TMRB5 コンペア一致による終了割り込み

uint32\_t  
**TMRB6CompareMatch** (Bit 16) TMRB6 コンペア一致による終了割り込み

uint32\_t  
**TMRB7CompareMatch** (Bit 17) TMRB7 コンペア一致による終了割り込み

uint32\_t  
**TMRBFCompareMatch** (Bit 18) TMRBF コンペア一致による終了割り込み

uint32\_t  
**TMRB0InputCapture0** (Bit 19) TMRB0 インพุットキャプチャ 0 による終了割り込み

uint32\_t  
**TMRB0InputCapture1** (Bit 20) TMRB0 インพุットキャプチャ 1 による終了割り込み

uint32\_t  
**TMRB1InputCapture0** (Bit 21) TMRB1 インพุットキャプチャ 0 による終了割り込み

uint32\_t  
**TMRB1InputCapture1** (Bit 22) TMRB1 インพุットキャプチャ 1 による終了割り込み

uint32\_t  
**TMRB2InputCapture0** (Bit 23) TMRB2 インพุットキャプチャ 0 による終了割り込み

uint32\_t  
**TMRB2InputCapture1** (Bit 24) TMRB2 インพุットキャプチャ 1 による終了割り込み

uint32\_t



**TMRB3InputCapture0** (Bit 25) TMRB3 インพุットキャプチャ 0 による終了割り込み

uint32\_t

**TMRB3InputCapture1** (Bit 26) TMRB3 インพุットキャプチャ 1 による終了割り込み

uint32\_t

**TMRB4InputCapture0** (Bit 27) TMRB4 インพุットキャプチャ 0 による終了割り込み

uint32\_t

**TMRB4InputCapture1** (Bit 28) TMRB4 インพุットキャプチャ 1 による終了割り込み

uint32\_t

**TMRB5InputCapture0** (Bit 29) TMRB5 インพุットキャプチャ 0 による終了割り込み

uint32\_t

**TMRB5InputCapture1** (Bit 30) TMRB5 インพุットキャプチャ 1 による終了割り込み

uint32\_t

**DMAREQB** (Bit 31) DMA ユニット B リクエストピン

DMAREQB による終了割り込み

## 19.2.4.4 DMACC\_Flag

メンバ:

uint32\_t

**All** DMA UINTC interrupt.

ビットフィールド: 各ビットの値の意味は以下の通りです。

‘0’: 終了割り込みは発生していない

‘1’: 終了割り込みが発生

uint32\_t

**SIO\_UART3Reception** (Bit 0) SIO/UART3 受信による終了割り込み

uint32\_t

**SIO\_UART3Transmission** (Bit 1) SIO/UART3 送信による終了割り込み

uint32\_t

**SIO\_UART4Reception** (Bit 2) SIO/UART4 受信による終了割り込み

uint32\_t

**SIO\_UART4Transmission** (Bit 3) SIO/UART4 送信による終了割り込み

uint32\_t

**SIO\_UART5Reception** (Bit 4) SIO/UART5 受信による終了割り込み

uint32\_t

**SIO\_UART5Transmission** (Bit 5) SIO/UART5 送信による終了割り込み

uint32\_t

**SIO\_UART8Reception** (Bit 6) SIO/UART8 受信による終了割り込み

uint32\_t

**SIO\_UART8Transmission** (Bit 7) SIO/UART8 送信による終了割り込み

uint32\_t

**SIO\_UART9Reception** (Bit 8) SIO/UART9 受信による終了割り込み

uint32\_t

**SIO\_UART9Transmission** (Bit9) SIO/UART9 送信による終了割り込み

uint32\_t

**TMRB8CompareMatch** (Bit 10) TMRB8 コンペア一致による終了割り込み

uint32\_t

**TMRB9CompareMatch** (Bit 11) TMRB9 コンペア一致による終了割り込み

uint32\_t

**TMRBACompareMatch** (Bit 12) TMRBA コンペア一致による終了割り込み

uint32\_t

**TMRBBCompareMatch** (Bit 13) TMRBB コンペア一致による終了割り込み

uint32\_t

**TMRBCCompareMatch** (Bit 14) TMRBC コンペア一致による終了割り込み

uint32\_t

**TMRBDCompareMatch** (Bit 15) TMRBD コンペア一致による終了割り込み

uint32\_t

**TMRBECompareMatch** (Bit 16) TMRBE コンペア一致による終了割り込み

uint32\_t

**TMRB7CompareMatch** (Bit 17) TMRB7 コンペア一致による終了割り込み

uint32\_t

**TMRBFCompareMatch** (Bit 18) TMRBF コンペア一致による終了割り込み

uint32\_t

**TMRB8InputCapture0** (Bit 19) TMRB8 インพุットキャプチャ 0 による終了割り込み

uint32\_t

**TMRB8InputCapture1** (Bit 20) TMRB8 インพุットキャプチャ 1 による終了割り込み

uint32\_t

**TMRB9InputCapture0** (Bit 21) TMRB9 インพุットキャプチャ 0 による終了割り込み

uint32\_t

**TMRB9InputCapture1** (Bit 22) TMRB9 インพุットキャプチャ 1 による終了割り込み

uint32\_t

**TMRBAInputCapture0** (Bit 23) TMRBA インพุットキャプチャ 0 による終了割り込み

uint32\_t

**TMRBAInputCapture1** (Bit 24) TMRBA インพุットキャプチャ 1 による終了割り込み

uint32\_t

**TMRBBInputCapture0** (Bit 25) TMRBB インพุットキャプチャ 0 による終了割り込み

uint32\_t

**TMRBBInputCapture1** (Bit 26) TMRBB インพุットキャプチャ 1 による終了割り込み

uint32\_t

***TMRBCInputCapture0*** (Bit 27) TMRBC インพุットキャプチャ 0 による終了割り込み

uint32\_t

***TMRBCInputCapture1*** (Bit 28) TMRBC インพุットキャプチャ 1 による終了割り込み

uint32\_t

***TMRBDInputCapture0*** (Bit 29) TMRBD インพุットキャプチャ 0 による終了割り込み

uint32\_t

***TMRBDInputCapture1*** (Bit 30) TMRBD インพุットキャプチャ 1 による終了割り込み

uint32\_t

***DMAREQC*** (Bit 31) DMA ユニット C リクエストピン

DMAREQC による終了割り込み

## 20. WDT

### 20.1 概要

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

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

本ドライバは、以下のファイルで構成されています。  
\\Libraries\\TX04\_Periph\_Driver\\src\\tmpm462\_wdt.c  
\\Libraries\\TX04\_Periph\_Driver\\inc\\tmpm462\_wdt.h

### 20.2 API 関数

#### 20.2.1 関数一覧

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

#### 20.2.2 関数の種類

関数は、主に以下の 2 種類に分かれています:

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

#### 20.2.3 関数仕様

##### 20.2.3.1 WDT\_SetDetectTime

検出時間の設定

関数のプロトタイプ宣言:

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

引数:

**DetectTime**: 検出時間を選択します。

- WDT\_DETECT\_TIME\_EXP\_15: **DetectTime** is 2<sup>15</sup>/fsys
- WDT\_DETECT\_TIME\_EXP\_17: **DetectTime** is 2<sup>17</sup>/fsys
- WDT\_DETECT\_TIME\_EXP\_19: **DetectTime** is 2<sup>19</sup>/fsys

- WDT\_DETECT\_TIME\_EXP\_21: *DetectTime* is  $2^{21}/fsys$
- WDT\_DETECT\_TIME\_EXP\_23: *DetectTime* is  $2^{23}/fsys$
- WDT\_DETECT\_TIME\_EXP\_25: *DetectTime* is  $2^{25}/fsys$

**機能:**

WDT の検出時間を設定します。

**戻り値:**

なし

## 20.2.3.2 WDT\_SetIdleMode

IDLE モード時の動作

**関数のプロトタイプ宣言:**

```
void  
WDT_SetIdleMode(FunctionalState NewState)
```

**引数:**

**NewState**: IDLE 時の動作の有効/無効を選択します。

- **ENABLE**: 動作
- **DISABLE**: 停止

**機能:**

本関数は、IDLE モード時の WDT カウンタの動作を設定します。

**NewState** が **ENABLE** の時は WDT カウンタ動作

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

**補足:**

CPU が IDLE モードに入る前に、設定してください。

**戻り値:**

なし

## 20.2.3.3 WDT\_SetOverflowOutput

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

**関数のプロトタイプ宣言:**

```
void  
WDT_SetOverflowOutput(uint32_t OverflowOutput)
```

**引数:**

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

- **WDT\_NMIINT**: NMI 割り込み発生
- **WDT\_WDOUT**: リセット

**機能:**

カウンタオーバーフロー時の NMI 割り込み/リセットの設定を行います。

**OverflowOutput** が **WDT\_NMIINT** の時、カウンタオーバーフローが発生すると NMI 割り込みが発生します。

戻り値:  
なし

## 20.2.3.4 WDT\_Init

WDT の初期化

関数のプロトタイプ宣言:

void  
WDT\_Init (WDT\_InitTypeDef\* **InitStruct**)

引数:

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

機能:

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

戻り値:  
なし

## 20.2.3.5 WDT\_Enable

WDT 動作の許可

関数のプロトタイプ宣言:

void  
WDT\_Enable(void)

引数:

なし。

機能:

WDT 動作を許可します。

戻り値:  
なし

## 20.2.3.6 WDT\_Disable

WDT 動作の禁止

関数のプロトタイプ宣言:

void  
WDT\_Disable(void)

引数:

なし。

**機能:**

WDT 動作を禁止します。

**戻り値:**

なし

## 20.2.3.7 WDT\_WriteClearCode

クリアコードの書き込み

**関数のプロトタイプ宣言:**

void  
WDT\_WriteClearCode (void)

**引数:**

なし。

**機能:**

WDT カウンタにクリアコードを書き込みます。

**戻り値:**

なし

## 20.2.4 データ構造

### 20.2.4.1 WDT\_InitTypeDef

**メンバ:**

uint32\_t

**DetectTime** 検出時間を選択します。

- WDT\_DETECT\_TIME\_EXP\_15: 2<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:** カウンタオーバーフロー時の設定を選択します。

- WDT\_WDOUT: リセット
- WDT\_NMIINT: NMI 割り込み