TOSHIBA

TX03 ペリフェラルドライバ
ユーザーガイド
(TMPM311)

第 1.000 版
2017 年 9 月

東芝デバイス＆ストレージ株式会社

CMDR-M311UG-01xJ

本製品取り扱い上のお願い

- ソフトウエア使用権許諾契約書の同意無しに使用しないで下さい。

目次

1 はじめに

本ペリフェラルドライバセットは、東芝製マイコンTMPM311用です。

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

TMPM311 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 TX03 ペリフェラルドライバの構成

/TMPM311/Libraries
TX03 CMSIS ファイルと TMPM011 ペリフェラルドライバが格納されています。

/TMPM311/Libraries/TX03_CMSIS
このフォルダには TMPM311 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。

/TMPM311/Libraries/TX03_Periph_Driver
TMPM311 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。

/TMPM311/Libraries/TX03_Periph_Driver/inc
TMPM311 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。

/TMPM311/Libraries/TX03_Periph_Driver/src
TMPM311 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。

/TMPM311/Project
TMPM311 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。

/TMPM311/Project/Template
TMPM311 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。

/TMPM311/Project/Examples
TMPM311 ペリフェラルドライバの使用例が格納されています。

/TMPM311/Project/Examples/Utilities/TMPM311-EVAL
TMPM311 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 CG

3.1 概要

クロック/モード制御ブロックでは、クロックギアやプリスケーラクロックの選択等を設定することが可能です。クロックに関連する機能としては以下のようなものがあります。

- システムクロックの制御
- プリスケーラクロックの制御
- ウォーミングアップタイマの制御
- 割り込み要求のクリア

全ドライバ API は、アプリで使用する API 定義、マクロ、データタイプ、構造を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver¥src¥tmpM311_cg.c
/Libraries/TX03_Periph_Driver¥inc¥tmpM311_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。

EHOSC： 外部高速発振クロック、X1、X2 端子に接続する発振子より入力されるクロック
EHCLKIN： 外部高速クロック、X1 から入力する高速クロック
IHOSC： 内部高速発振クロック、内部高速発振器より入力されるクロック
fosc : EHOSC または IHOSC のどちらか選択されたクロック
fc : CG0OSCSEL<EHOSCEN[1:0]>で選択されたクロック(高速クロック)
fgear : CG0CLKCR<GEAR[2:0]>で選択された分周クロック
fsys : CG0CLKCR<GEAR[2:0]>で選択されたクロック(システムクロック)
fperiph : CG0CLKCR<FPSEL>で選択されたプリスケーラ用クロック
φT0 : CG0CLKCR<PRCK[2:0]>で選択されたクロック (プリスケーラクロック)

3.2 API 関数

3.2.1 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)
- void CG_SetSysTickSrc(CG_SysTickSrc ***SysTickSrc)***
- CG_SysTickSrc CG_GetSysTickSrc(void)
- Result CG_SetPhiT0Level(CG_DivideLevel ***DividePhiT0FromFc***)
- CG_DivideLevel CG_GetPhiT0Level(void)
- void CG_SetWarmUpTime(CG_WarmUpSrc ***Source***, uint16_t ***Time***)
- void CG_StartWarmUp(void)
- WorkState CG_GetWarmUpState(void)
- Result CG_SetFosc(CG_FoscSrc Source, FunctionalState ***NewState***)
- void CG_SetFoscSrc(CG_FoscSrc ***Source***)
- CG_FoscSrc CG_GetFoscSrc(void)
- FunctionalState CG_GetFoscState(CG_FoscSrc ***Source***)

- Result CG_SetFcSrc(CG_FcSrc ***Source***)
- CG_FcSrc CG_GetFcSrc(void)
- void CG_SetProtectCtrl(FunctionalState ***NewState***)
- void CG_SetSTBYReleaseINTSrc(CG_INTSrc ***INTSource***,
  CG_INTActiveState ***ActiveState***)
- CG_INTActiveState CG_GetSTBYReleaseINTState(CG_INTSrc ***INTSource***)
- void CG_ClearINTReq(CG_INTSrc ***INTSource***)
- CG_ResetFlag CG_GetResetFlag(void)

3.2.2 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) クロックの選択:
   CG_SetFgearLevel(), CG_GetFgearLevel(), CG_SetPhiT0Src(), CG_GetPhiT0Src(), CG_SetSysTickSrc(), CG_GetSysTickSrc(), CG_SetPhiT0Level(), CG_GetPhiT0Level(), CG_SetWarmUpTime(), CG_StartWarmUp(), CG_GetWarmUpState(), CG_SetFosc(), CG_SetFoscSrc(), CG_GetFoscSrc(), CG_GetFoscState(), CG_SetFcSrc(), CG_GetFcSrc(), CG_SetProtectCtrl()
2) 割り込みの設定:
   CG_SetSTBYReleaseINTSrc(), CG_GetSTBYReleaseINTState(), CG_ClearINTReq(), CG_GetResetFlag()

3.2.3 関数仕様

3.2.3.1 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

3.2.3.2 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

3.2.3.3 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

3.2.3.4 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

3.2.3.5 CG_SetSysTickSrc

SysTick リファレンスソースクロックの設定

関数のプロトタイプ宣言:
void
CG_SetSysTickSrc (CG_SysTickSrc ***SysTickSrc)***

引数:
SysTickSrc: SysTick ソースクロックを選択します。
- **CG_STICK_SRC_IHOSC:** IHOSC
- **CG_STICK_SRC_FOSC:** fosc

機能:
SysTick のリファレンスソースクロックを選択します。

戻り値:
なし

3.2.3.6 CG_GetSysTickSrc

SysTick リファレンスソースクロックの取得

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

引数:
なし

機能:
SysTick のりふぁフェンスソースクロックを取得します。

戻り値:
SysTick ソースクロックを選択します。
- **CG_STICK_SRC_IHOSC:** IHOSC
- **CG_STICK_SRC_FOSC:** fosc

3.2.3.7 CG_SetPhiT0Level

PhiT0 (ΦT0) と fc 間の分周レベルの設定

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

引数:
DividePhiT0FromFc: PhiT0 (ΦT0) と fc 間の分周レベルを下記の値から設定します。
- **CG_DIVIDE_1**: ΦT0 = fc
- **CG_DIVIDE_2**: ΦT0 = fc/2

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

機能:
プリスケーラークロックの分周レベルを設定します。

戻り値:
SUCCESS: 設定成功
ERROR:エラー

3.2.3.8 CG_GetPhiT0Level

PhiT0 (ΦT0) と fc 間の分周レベルの取得

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

引数:
なし

機能:
PhiT0 (ΦT0) と fc 間の分周レベルを取得します。
取得した値が"Reserved"の場合は、**CG_DIVIDE_UNKNOWN** を返します。

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

3.2.3.9 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**: IHOSC に設定
- **CG_WARM_UP_SRC_OSC_EXT_HIGH**: EHOSC に設定

Time:
クロックは 0U から 0x1000U の範囲で設定してください。

機能:
ウォームアップ時間とウォームアップカウンタを設定します。計算式は下記になります。
Setting_value = ((warm-up time) / (input cycle time by frequency))/16

ウォームアップ時間の計算レジスタ値例:
/* When using high-speed oscillator 10MHz, and set warm-up time 5ms. */
value = (warm-up time to set) / (input frequency cycle(s)) = 5ms / (1/10MHz) = 50000cycle = 0xC350
下位 4 ビット右シフトして 0xC35 を CG0WUHCR<WUPT[11:0]>に設定します。

戻り値:
なし

3.2.3.10 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

3.2.3.11 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。

ウォームアップ時間の使用例:

```
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動作状態:

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

3.2.3.12 CG_SetFosc

高速発振器の動作選択 (fosc)

関数のプロトタイプ宣言:

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

引数:

Source: 以下から、高速発振器を選択します。

- **CG_FOSC_OSC_EXT**: EHOSC
- **CG_FOSC_OSC_INT**: IHOSC

NewState: 以下から高速発振器の動作を選択します。

- **ENABLE**: 発振
- **DISABLE**: 停止

機能:

高速発振器の動作を選択します。

システムクロック(fsys)として fgear を選択した場合、高速発振器は選択できず **ERROR** を返します。

戻り値:

SUCCESS: 成功

ERROR: 失敗

3.2.3.13 CG_SetFoscSrc

高速発振器のソース選択

関数のプロトタイプ宣言:

```
void
CG_SetFoscSrc(CG_FoscSrc Source)
```

引数:

Source: 以下から高速発振器のソースを選択します。

- **CG_FOSC_OSC_EXT**: 外部発振器
- **CG_FOSC_CLKIN_EXT**: 外部クロック

機能:
高速発振器のソースを選択します。

戻り値:
なし

3.2.3.14 CG_GetFoscSrc

高速発振器のソース選択状態の取得

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

引数:
なし

機能:
高速発振器のソース選択状態を取得します。

戻り値:
高速発振器のソース
CG_FOSC_OSC_EXT:　外部発振器
CG_FOSC_CLKIN_EXT: 外部クロック
CG_FOSC_UNKNOWN: 無効

3.2.3.15 CG_GetFoscState

高速発振器の動作選択状態の取得

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

引数:
Source: 以下から高速発振器を選択します。
- **CG_FOSC_OSC_EXT**: 外部高速発振器
- **CG_FOSC_OSC_INT**: 内部高速発振器

機能:
高速発振器の動作選択状態を取得します。

戻り値:
高速発振器の動作選択状態
ENABLE: 発振
DISABLE: 停止

3.2.3.16 CG_SetFcSrc

fc のソースクロックの設定

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

引数:
Source: fc のソースクロックの設定
- **CG_FC_SRC_FOSC:** fos
- **CG_FC_SRC_IHOSC:** IHOSC

機能:
fc のソースクロックを設定します。

戻り値:
SUCCESS: 成功
ERROR: 失敗

3.2.3.17 CG_GetFcSrc

fc のソースクロックの取得

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

引数:
なし

機能:
fc のソースクロックを取得します。

戻り値:
fc のソースクロック:
CG_FC_SRC_FOSC: fosc
CG_FC_SRC_IHOSC: IHOSC

3.2.3.18 CG_SetProtectCtrl

CG レジスタの書き込み制御

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

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

機能:
CG レジスタの書き込み許可/禁止を設定します。

戻り値:

なし

3.2.3.19 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***)

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

ActiveState: 解除トリガのアクティブ状態を選択します。
- **CG_INT_ACTIVE_STATE_L**: "Low"レベル
- **CG_INT_ACTIVE_STATE_H**: "High"レベル
- **CG_INT_ACTIVE_STATE_FALLING**: ↓エッジ
- **CG_INT_ACTIVE_STATE_RISGING**: ↑エッジ
- **CG_INT_ACTIVE_STATE_BOTH_EDGES**: 両エッジ

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

3.2.3.20 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** : INT0
- **CG_INT_SRC_1** : INT1

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

3.2.3.21 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** : INT0
- **CG_INT_SRC_1** : INT1

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

3.2.3.22 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:
- **PinReset** (Bit0) RESET 端子によるリセット
- **WDTReset** (Bit 3) WDT によるリセット
- **DebugReset** (Bit 4) <SYSRESETREQ>によるリセット

3.2.4 データ構造

3.2.4.1 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
PinReset(Bit0)　　RESET 端子によるリセット
uint32_t
Reserved1 (Bit1~bit2)　　未使用
uint32_t

WDTReset(Bit3)　　WDT によるリセット
uint32_t
DebugReset(Bit4)　　<SYSRESETREQ>によるリセット
uint32_t
Reserved2 (Bit5~bit31)　　未使用

4 DSADC

4.1 概要

TMPM311CHDUG は3 ユニットの24bitΔΣADコンバータ(DSADC)を内蔵しています。
DSADC の同期スタート機能における、マスタユニットとスレーブユニットの割り当ては以下の通りです。

マスタ/スレーブ割り当て

マスタ	スレーブ
ユニットA	ユニットB ユニットC ユニットD

DSADC の基準電圧回路(BGR)は温度センサと共通に使用しており、使用するためには温度センサの制御レジスタ(TEPEN)の設定も必要です。

特徴
DSADC には、以下のような特徴があります。
・変換スタート
- ソフトウエアによる変換スタート
- ハードウェアトリガによる変換スタート
・変換モード
- シングル変換
- リピート変換
・ステータスフラグ
- 変換結果格納フラグ
- オーバーランフラグ
- 変換終了フラグ
- 変換中フラグ
・変換クロックを分周可能
fc/1、fc/2、fc/4、fc/8
・変換終了割り込みを出力
・変換開始補正機能
・ユニット間同時スタート機能
・変換結果信号出力

DSADC を使用する場合、下記のとおり端子処理を行ってください。
・VREFINx に基準電源の接続はしない
・AGNDREFx はDVSS に接続
・VREFINx とAGNDREFx の間に1μF のコンデンサを接続

DSADC を使用しない場合、下記のとおり端子処理を行ってください。
・VREFINx はDVDD3 に接続
・AGNDREFx はDVSS に接続

また、温度センサも使用しない場合、基準電圧回路に関し下記のとおり端子処理を行ってください。
・DSRVDD3、SRVDD はDVDD3 に接続
・DSRVSS はDVSS に接続

DSADC ペリフェラルドライバ API は DSADC 回路の設定を行なう関数セットです。本関数セットは DSADC 変換クロックの設定、モード設定、変換開始設定、補正機能の設定、DSADC ステータスの取得、DSADC 変換結果の取得などの機能があります。

\Libraries\TX03_Periph_Driver\src\tmpm311_dsad.c
\Libraries/TX03_Periph_Driver\inc\tmpm311_dsad.h

4.2 API 関数

4.2.1 関数一覧

- void DSADC_SetClk(TSB_DSAD_TypeDef * ***DSADCx***, uint32_t ***Clk***)
- void DSADC_SWReset(TSB_DSAD_TypeDef * ***DSADCx***)
- void DSADC_Start(TSB_DSAD_TypeDef * ***DSADCx***)
- void DSADC_ChangeMode(TSB_DSAD_TypeDef * ***DSADCx***, uint32_t ***SyncMode***, uint32_t ***ConvMode***)
- void DSADC_SetHWStartup(TSB_DSAD_TypeDef * ***DSADCx***, FunctionalState ***NewState***)
- void DSADC_SetHWStartupFactor(TSB_DSAD_TypeDef * ***DSADCx***, uint32_t ***StartupFactor***)
- void DSADC_SetAmplifier(TSB_DSAD_TypeDef * ***DSADCx***, uint32_t ***Amplifier***)
- void DSADC_SetAnalogInput(TSB_DSAD_TypeDef * ***DSADCx***, uint32_t ***AnalogInput***)
- uint32_t DSADC_GetConvertResult(TSB_DSAD_TypeDef * ***DSADCx***)
- void DSADC_Init(TSB_DSAD_TypeDef * ***DSADCx***, DSADC_InitTypeDef * ***InitStruct***)
- DSAD_status DSADC_GetStatus(TSB_DSAD_TypeDef * ***DSADCx***)

4.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。

1) AD 変換設定:
   DSADC_SetClk(), DSADC_ChangeMode(), DSADC_Init (), DSADC_SetAmplifier(), DSADC_SetAnalogInput()
2) AD 変換の開始:
   DSADC_Start(), DSADC_SetHWStartup(), DSADC_SetHWStartupFactor()
3) AD 変換ステータス/結果の読み出し:
   DSADC_GetConvertResult(), DSADC_GetStatus()
4) その他:
   DSADC_SWReset()

4.2.3 関数仕様

4.2.3.1 DSADC_SetClk

AD 変換クロックの選択

関数のプロトタイプ宣言:

```
void
DSADC_SetClk(TSB_DSAD_TypeDef *DSADCx,uint32_t Clk)
```

引数:

DSADCx: 以下から DSADC ユニットを選択します。

- **TSB_DSADA:** ユニット A
- **TSB_DSADB:** ユニット B
- **TSB_DSADC:** ユニット C
- **TSB_DSADD:** ユニット D

Clk: 以下から AD 変換クロックを選択します。

- **DSADC_FC_DIVIDE_LEVEL_1:** fc / 1
- **DSADC_FC_DIVIDE_LEVEL_2:** fc / 4
- **DSADC_FC_DIVIDE_LEVEL_4:** fc / 4
- **DSADC_FC_DIVIDE_LEVEL_8:** fc / 8

機能:
AD 変換クロックを選択します。

補足:
AD変換中に本APIをコールしないでください。
本APIをコールする前に**DSADC_GetStatus()**をコールしてDSAD変換状態が**BUSY**でないことを確認してください。

戻り値:
なし

4.2.3.2 DSADC_SWReset

ソフトウェアリセット

関数のプロトタイプ宣言:
void
DSADC_SWReset(TSB_DSAD_TypeDef * ***DSADCx***)

引数:
DSADCx: 以下から DSADC ユニットを選択します。

- **TSB_DSADA:** ユニット A
- **TSB_DSADB:** ユニット B
- **TSB_DSADC:** ユニット C
- **TSB_DSADD:** ユニット D

機能:
ソフトウェアリセットを行います。

補足:
DSADCLK<ADCLK>以外のレジスタを初期化します。.

戻り値:
なし

4.2.3.3 DSADC_Start

変換開始

関数のプロトタイプ宣言:
void

DSADC_Start(TSB_DSAD_TypeDef * ***DSADCx***)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSADA:** ユニット A
- **TSB_DSADB:** ユニット B
- **TSB_DSADC:** ユニット C
- **TSB_DSADD:** ユニット D

機能:
変換を開始します。

補足:
本 API をコールする前に、以下のいずれかのモードを選択してください。
シングル変換モード
連続変換モード
詳細は **DSADC_ChangeMode()**を参照してください。

補足:
AD 変換を開始する前に、DSADC の設定手順があります。詳細は、データシートの DSADC“起動および停止手順”を参照してください。

戻り値:
なし

4.2.3.4 DSADC_ChangeMode

同期モードと変換モードの選択

関数のプロトタイプ宣言:
void
DSADC_ChangeMode(TSB_DSAD_TypeDef * ***DSADCx***,uint32_t ***SyncMode***,uint32_t ***ConvMode***)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSADA:** ユニット A
- **TSB_DSADB:** ユニット B
- **TSB_DSADC:** ユニット C
- **TSB_DSADD:** ユニット D

SyncMode: 以下から同期モードを選択します。
- **DSADC_A_SYNC_MODE**: 個別モード
- **DSADC_SYNC_MODE**: 同期モード

ConvMode: 以下から変換モードを選択します。
- **DSADC_SINGLE_MODE**: シングル変換
- **DSADC_REPEAT_MODE**: 連続変換

機能:
同期モードと変換モードを選択します。

戻り値:

なし

4.2.3.5 DSADC_SetHWStartup

ハードウェアトリガの許可/禁止

関数のプロトタイプ宣言:
void
DSADC_SetHWStartup(TSB_DSAD_TypeDef * ***DSADCx***, FunctionalState ***NewState***)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSADA:** ユニット A
- **TSB_DSADB:** ユニット B
- **TSB_DSADC:** ユニット C
- **TSB_DSADD:** ユニット D

NewState: ハードウェアトリガの許可/禁止を選択します。
- **ENABLE:** 許可
- **DISABLE:** 禁止

機能:
ハードウェアトリガの許可/禁止を選択します。

戻り値:
なし

4.2.3.6 DSADC_SetHWStartupFactor

ハードウェア起動の許可/禁止

関数のプロトタイプ宣言:
void
void DSADC_SetHWStartupFactor(TSB_DSAD_TypeDef * ***DSADCx***, uint32_t ***StartupFactor***)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSADA:** ユニット A
- **TSB_DSADB:** ユニット B
- **TSB_DSADC:** ユニット C
- **TSB_DSADD:** ユニット D

StartupFactor: ハードウェア起動の許可/禁止を選択します。
- **DSADC_HARDWARE_TRIGGER_EXT:** 許可(ハードウェアトリガが選択されます)
- **DSADC_HARDWARE_TRIGGER_INT:** 禁止(ソフトウェアトリガが選択されます)

機能:

ハードウェア起動の許可/禁止を選択します。

戻り値:
なし

4.2.3.7 DSADC_SetAmplifier

ゲインアンプの設定

関数のプロトタイプ宣言:
void
DSADC_SetAmplifier(TSB_DSAD_TypeDef * ***DSADCx***,uint32_t ***Amplifier***)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSADA:** ユニット A
- **TSB_DSADB:** ユニット B
- **TSB_DSADC:** ユニット C
- **TSB_DSADD:** ユニット D

Gain: ゲインアンプの設定値を選択します。
- **DSADC_GAIN_1x:** 1
- **DSADC_GAIN_2x:** 2
- **DSADC_GAIN_4x:** 4
- **DSADC_GAIN_8x:** 8
- **DSADC_GAIN_16x:** 16

機能:
ゲインアンプの設定を行います。

戻り値:
なし

4.2.3.8 DSADC_SetAnalogInput

アナログ入力信号の指定

関数のプロトタイプ宣言:
void
DSADC_SetAnalogInput(TSB_DSAD_TypeDef * ***DSADCx***, uint32_t ***AnalogInput***)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSADA:** ユニット A
- **TSB_DSADB:** ユニット B
- **TSB_DSADC:** ユニット C
- **TSB_DSADD:** ユニット D

AnalogInput: アナログ入力信号を選択します。
[ユニット A、B、C の場合]
- **DSADC_ANALOG_INPUT_DAIN:** DAINx (+/-)

[ユニット D]
- **DSADC_ANALOG_INPUT_DAIN:** DAINx (+/-)
- **DSADC_ANALOG_INPUT_INT:** 内部アナログ入力(+/-)

機能:
アナログ入力信号を指定します。

戻り値:
なし

4.2.3.9 DSADC_GetConvertResult

変換結果の取得

関数のプロトタイプ宣言:
uint32_t
DSADC_GetConvertResult(TSB_DSAD_TypeDef * ***DSADCx***)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSADA:** ユニット A
- **TSB_DSADB:** ユニット B
- **TSB_DSADC:** ユニット C
- **TSB_DSADD:** ユニット D

機能:
変換結果を取得します。

戻り値:
変換結果

4.2.3.10 DSADC_Init

DSADC のコンフィグレーション

関数のプロトタイプ宣言:
void
DSADC_Init(TSB_DSAD_TypeDef * ***DSADCx***,DSADC_InitTypeDef * ***InitStruct***)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSADA:** ユニット A
- **TSB_DSADB:** ユニット B
- **TSB_DSADC:** ユニット C
- **TSB_DSADD:** ユニット D

InitStruct: DSAD 変換のコンフィグレーションを行う構造体を指定します。(詳細はデータ構造を参照)

機能:
DSADC のコンフィグレーションを行います。

戻り値:
なし

4.2.3.11 DSADC_GetStatus

DSAD 変換フラグの取得

関数のプロトタイプ宣言:
DSAD_status
DSADC_GetStatus(TSB_DSAD_TypeDef * ***DSADCx***)

引数:
DSADCx: 以下から DSADC ユニットを選択します。
- **TSB_DSADA:** ユニット A
- **TSB_DSADB:** ユニット B
- **TSB_DSADC:** ユニット C
- **TSB_DSADD:** ユニット D

機能:
DSAD 変換フラグを取得します。

戻り値:
DSAD 変換フラグ

4.2.4 データ構造

4.2.4.1DSADC_InitTypeDef

メンバ:

uint32_t
Clk AD 変換クロックを選択します。
- **DSADC_FC_DIVIDE_LEVEL_1:** fc / 1
- **DSADC_FC_DIVIDE_LEVEL_2:** fc / 4
- **DSADC_FC_DIVIDE_LEVEL_4:** fc / 4
- **DSADC_FC_DIVIDE_LEVEL_8:** fc / 8

uint32_t
BiasE バイアス制御を行います。
- **0:** 停止
- **1:** 動作

uint32_t
ModulatorEn モジュレータ制御を行います。
- **0:** 停止
- **1:** 動作

uint32_t
HardwareFactor ハードウェア起動要因を選択します。
- **DSADC_HARDWARE_TRIGGER_EXT:** 外部トリガ
- **DSADC_HARDWARE_TRIGGER_INT:** 内部トリガ

FunctionalState
HardwareEn ハードウェア起動制御を行います。
- **ENABLE:** 有効
- **DISABLE:** 無効

uint32_t
SyncMode 同期モードを選択します。
- **DSADC_A_SYNC_MODE:** 個別動作
- **DSADC_SYNC_MODE:** 同期動作

uint32_t
Repeatmode 変換モードを選択します。
- **DSADC_SINGLE_MODE:** シングル変換
- **DSADC_REPEAT_MODE:** リピート変換

uint32_t
Amplifier ゲインアンプに設定する値を選択します。
- **DSADC_GAIN_1x:** 1
- **DSADC_GAIN_2x:** 2
- **DSADC_GAIN_4x:** 4
- **DSADC_GAIN_8x:** 8
- **DSADC_GAIN_16x:** 16

uint32_t
AnalogInput アナログ入力信号を選択します。
[ユニット A、B、C]
- **DSADC_ANALOG_INPUT_DAIN:** DAINx (+/-)

[ユニット D]
- **DSADC_ANALOG_INPUT_DAIN:** DAINx (+/-)
- **DSADC_ANALOG_INPUT_INT:** 内部アナログ入力信号(+/-)

uint16_t
Offset (Bit 7) 変換開始補正時間を指定します。

uint32_t
CorrectEn 変換開始補正("1"を選択すると、***Offset*** で設定した遅延後に変換を開始します。
- **0:** 補正しない
- **1:** 補正する

4.2.4.2 DSAD_status

メンバ:
uint32_t
All すべての AD 変換結果

uint32_t
F_ResultStore (Bit 0) 変換結果格納フラグ(1: 格納された、0: 格納されていない)
uint32_t

F_Overrun (Bit 1)	オーバーランフラグ(1: オーバーラン発生、0: 発生していない)
uint32_t ***F_Convert*** (Bit 2)	変換中フラグ(1: 変換中、0: 変換していない)
uint32_t ***F_ConvertEnd*** (Bit 3)	変換終了フラグ(1: 変換終了、0: 変換終了していない)
uint32_t ***Reserved*** (Bit4 ~ Bit7)	未使用
uint32_t ***ConversionResult*** (Bit8 ~ Bit31)	変換結果

5 GPIO

5.1 概要

TMPM311 は 4 つの汎用ポート (A, B, C, D) があり、入出力を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できます。
GPIO ドライバ API は各ポートの設定機能を持ち、入出力、プルアップ、プルダウン、オープンドレイン、CMOS などを設定します。
全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm311_gpio.c
/Libraries/TX03_Periph_Driver/inc/tmpm311_gpio.h

5.2 API 関数

5.2.1 関数一覧

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

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) 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_Init()
3) その他:
   GPIO_EnableFuncReg(), GPIO_DisableFuncReg()

5.2.3 関数仕様

5.2.3.1 GPIO_ReadData

DATA データレジスタの読み込み

関数のプロトタイプ宣言:
uint8_t
GPIO_ReadData(GPIO_Port ***GPIO_x***)

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

機能:
DATA レジスタを読み込みます。

戻り値:
DATA レジスタの値

5.2.3.2 GPIO_ReadDataBit

ビット単位での DATA レジスタの読み込み

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

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

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

機能:
ビット単位で DATA データレジスタを読み込みます。

戻り値:
GPIO 端子値
- **GPIO_BIT_VALUE_0**: 0
- **GPIO_BIT_VALUE_1**: 1

5.2.3.3 GPIO_WriteData

DATA レジスタへの書き込み

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

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

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

5.2.3.4 GPIO_WriteDataBit

ビット単位での DATA レジスタの書き込み

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

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

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

BitValue: 設定ビットを指定します。
- **GPIO_BIT_VALUE_0**: 0

- **GPIO_BIT_VALUE_1**: 1

機能:
ビット単位で DATA データレジスタを書き込みます。

戻り値:
なし

5.2.3.5 GPIO_Init

GPIO ポートの初期設定

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

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

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

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()**を実行します。

戻り値:
なし

5.2.3.6 GPIO_SetOutput

出力ポートの設定

関数のプロトタイプ宣言:
void
GPIO_SetOutput(GPIO_Port ***GPIO_x***,

uint8_t ***Bit_x***);

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

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

機能:
出力ポートに設定します。

戻り値:
なし

5.2.3.7 GPIO_SetInput

入力ポートの設定

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

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

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

機能:

入力ポートに設定します。

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

5.2.3.8 GPIO_SetOutputEnableReg

出力ポートの許可/禁止設定

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

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

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

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

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

戻り値:
なし

5.2.3.9 GPIO_SetInputEnableReg

入力ポートの許可/禁止設定

関数のプロトタイプ宣言:
void

GPIO_SetInputEnableReg(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***,
FunctionalState ***NewState***)

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

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

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

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

戻り値:
なし

5.2.3.10 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***,
uint8_t ***Bit_x***,
FunctionalState ***NewState***)

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

Bit_x: GPIO 端子を選択します。有効ビットの組み合わせが可能です。
- **GPIO_BIT_0:** GPIO pin 0
- **GPIO_BIT_1:** GPIO pin 1

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

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

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

5.2.3.11 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,
                 uint8_t Bit_x,
                   FunctionalState NewState)
```

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

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_3:** GPIO pin 3
- **GPIO_BIT_ALL:** GPIO pin[0:7]

NewState:
- **ENABLE**: プルダウン有効
- **DISABLE**: 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

5.2.3.12 GPIO_EnableFuncReg

機能ポートの有効設定

関数のプロトタイプ宣言:

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

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

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

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

機能:
GPIO 端子の機能を有効に設定します。

戻り値:
なし

5.2.3.13 GPIO_DisableFuncReg

機能ポートの無効設定

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

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

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

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

- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7
- **GPIO_BIT_ALL:** GPIO pin[0:7]

機能:
GPIO 端子の機能を無効に設定します。

戻り値:
なし

5.2.4 データ構造

5.2.4.1 GPIO_InitTypeDef

メンバ:
uint8_t
IOMode ポートの入出力を選択します。
- **GPIO_INPUT:** 入力ポートに設定します。
- **GPIO_OUTPUT:** 出力ポートに設定します。
- **GPIO_IO_MODE_NONE:** 入出力モードを変更しません。

uint8_t
PullUp 内蔵プルアップの有効/無効を選択します。
- **GPIO_PULLUP_ENABLE**: 内蔵プルアップを有効にします。
- **GPIO_PULLUP_DISABLE**: 内蔵プルアップを無効にします。
- **GPIO_PULLUP_NONE**: 内蔵プルアップ機能がない、または設定変更しません。

uint8_t
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。

5.2.4.2 GPIO_RegTypeDef

メンバ:
uint8_t
PinDATA DATA レジスタのマスク値

uint8_t
PinCR CR レジスタのマスク値
- **"0":** 出力禁止
- **"1":** 出力許可

uint8_t
PinFR[FRMAX] FR レジスタのマスク値

uint8_t
PinPUP PUP レジスタのマスク値
- **"0":** プルアップ禁止

- **”1”:** プルアップ許可

uint8_t
PinPDN PDN レジスタのマスク値
- **”0”:** プロダウン禁止
- **”1”:** プロダウン許可

uint8_t
PinPIE IE レジスタのマスク値
- **”0”:** 入力禁止
- **”1”:** 入力許可

5.2.4.3 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] 未定義

__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] 未定義

__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。

6 SSP

6.1 概要

TMPM311 は、同期式シリアルインタフェースを (SSP：Synchronous Serial Port) を 1 チャネル内蔵しています。 (SSP0)

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

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

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm311_ssp.c
/Libraries/TX03_Periph_Driver/inc/tmpm311_ssp.h

6.2 API 関数

6.2.1 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***);

6.2.2 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()

6.2.3 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

6.2.3.1 SSP_Enable

同期式シリアルインタフェース動作の許可

関数のプロトタイプ宣言:

void
SSP_Enable(TSB_SSP_TypeDef * ***SSPx***)

引数:

SSPx*:* SSP チャネルを指定します。

機能:

SSP 動作を有効にします。

戻り値:

なし

6.2.3.2 SSP_Disable

同期式シリアルインタフェース動作の禁止

関数のプロトタイプ宣言:

void
SSP_Disable(TSB_SSP_TypeDef * ***SSPx***)

引数:

SSPx*:* SSP チャネルを指定します。

機能:
SSP 動作を無効にします。

戻り値:
なし

6.2.3.3 SSP_Init

SSP 通信の初期化

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

引数:
SSPx*:* SSP チャネルを指定します。
InitStruct: SSP に関する構造体です。(詳細は"データ構造"を参照)

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

戻り値:
なし

6.2.3.4 SSP_SetClkPreScale

送受信のビットレート設定

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

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

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

fSYS はシステム周波数

戻り値:
なし

6.2.3.5 SSP_SetFrameFormat

フレームフォーマットの選択

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

引数:
SSPx*:* SSP チャネルを指定します。
FrameFormat: フレームフォーマットを選択します。

- **SSP_FORMAT_SPI:** SPI フレームフォーマット
- **SSP_FORMAT_SSI:** SSI シリアルフレームフォーマット
- **SSP_FORMAT_MICROWIRE:** Microwire フレームフォーマット

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

戻り値:
なし

6.2.3.6 SSP_SetClkPolarity

SPxCLK 極性の選択

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

引数:
SSPx*:* SSP チャネルを指定します。
ClkPolarity: SPxCLK 極性を選択します。

- **SSP_POLARITY_LOW:** SPxCLK は Low 状態。
- **SSP_POLARITY_HIGH:** SPxCLK は High 状態。

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

戻り値:
なし

6.2.3.7 SSP_SetClkPhase

SPxCLK フェーズの選択

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

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

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

戻り値:
なし

6.2.3.8 SSP_SetDataSize

データサイズの選択

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

引数:
SSPx*:* SSP チャネルを指定します。
DataSize: データサイズを 4～16 の間で選択します。

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

戻り値:
なし

6.2.3.9 SSP_SetSlaveOutputCtrl

スレーブモード SPxDO 出力の制御

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

引数:
SSPx*:* SSP チャネルを指定します。
NewState: スレーブモード SPxDO 出力の許可/禁止を選択します。

- **ENABLE:** 許可。
- **DISABLE:** 禁止。

機能:
スレーブモード SPxDO 出力の許可/禁止を選択します。

戻り値:
なし

6.2.3.10 SSP_SetMSMode

マスタ/ スレーブモードの選択

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

引数:
SSPx*:* SSP チャネルを指定します。
Mode: マスタ/ スレーブモードを選択します。
- **SSP_MASTER:** デバイスがマスタ。
- **SSP_SLAVE:** デバイスがスレーブ。

機能:
マスタ/ スレーブモードを選択します。

戻り値:
なし

6.2.3.11 SSP_SetLoopBackMode

ループバックモードの制御

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

引数:
SSPx*:* SSP チャネルを指定します。
NewState: ループバックモードの許可/禁止を選択します。
- **ENABLE**: 許可。
- **DISABLE**: 禁止。

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

戻り値:

なし

6.2.3.12 SSP_SetTxData

送信 FIFO のデータ設定

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

引数:
SSPx*:* SSP チャネルを指定します。
Data: 送信データを 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

6.2.3.13 SSP_GetRxData

受信 FIFO からのデータ読み込み

関数のプロトタイプ宣言:
uint16_t
SSP_GetRxData(TSB_SSP_TypeDef * ***SSPx***)

引数:
SSPx*:* SSP チャネルを指定します。

機能:
受信 FIFO から受信データを読み込みます。

戻り値:
受信データ

6.2.3.14 SSP_GetWorkState

ビジーフラグの読み込み

関数のプロトタイプ宣言:
WorkState
SSP_GetWorkState(TSB_SSP_TypeDef * ***SSPx***)

引数:
SSPx*:* SSP チャネルを指定します。

機能:
ビジーフラグを読み込みます。

戻り値:
ビジーフラグ
BUSY: ビジー
DONE: アイドル

6.2.3.15 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 チ次ャネルを指定します。
Direction: 送受信方向を選択します。
- **SSP_RX**: 受信 FIFO
- **SSP_TX**: 送信 FIFO

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

```
SSP_FIFOState    fifoState;

fifoState = SSP_GetFIFOState(TSB_SSP0, SSP_TX);
if ((fifoState == SSP_FIFO_EMPTY) || (fifoState == SSP_FIFO_NORMAL))
      { SSP_SetTxData(SSP0, data_to_be_sent ); }
```

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

6.2.3.16 SSP_SetINTConfig

割り込みの制御

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

引数:
SSPx: SSP チャネルを指定します。
IntSrc: 割り込みの許可/禁止を選択します。
- **SSP_INTCFG_NONE:** すべて禁止。
- **SSP_INTCFG_ALL** :すべて許可。

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

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

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

戻り値:
なし

6.2.3.17 SSP_GetINTConfig

割り込み制御の読み込み

関数のプロトタイプ宣言:
SSP_INTState
SSP_GetINTConfig(TSB_SSP_TypeDef * ***SSPx***)

引数:
SSPx*:* SSP チャネルを指定します。

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

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

6.2.3.18 SSP_GetPreEnableINTState

許可前の割り込み状態の読み込み

関数のプロトタイプ宣言:
SSP_INTState
SSP_GetPreEnableINTState(TSB_SSP_TypeDef * ***SSPx***)

引数:
SSPx*:* SSP チャネルを指定します。

機能:
許可前の割り込み状態を読み込みます。

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

6.2.3.19 SSP_GetPostEnableINTState

許可後の割り込み状態の読み込み

関数のプロトタイプ宣言:
SSP_INTState
SSP_GetPostEnableINTState(TSB_SSP_TypeDef * ***SSPx***)

引数:
SSPx*:* SSP チャネルを指定します。

機能:
禁止前の割り込み状態を読み込みます。

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

6.2.3.20 SSP_ClearINTFlag

割り込みフラグのクリア

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

引数:
SSPx*:* SSP チャネルを指定します。
IntSrc: クリアする割り込みフラグを選択します。
- **SSP_INTCFG_RX_OVERRUN:** 受信オーバーラン割り込みフラグ。
- **SSP_INTCFG_RX_TIMEOUT:** 受信タイムアウト割り込みフラグ
- **SSP_INTCFG_ALL:** すべての割り込みフラグ。

機能:
割り込みフラグをクリアします。

戻り値:
なし

6.2.4 データ構造

6.2.4.1 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

6.2.4.2 SSP_INTState

メンバ:
uint32_t
All: 割り込み要因
Bit
uint32_t
OverRun: 1 オーバー欄割り込み
uint32_t
TimeOut: 1 受信タイムアウト
uint32_t
Rx: 1 受信
uint32_t
Tx: 1 送信
uint32_t
Reserved: 28未定義

7 TEMP

7.1 概要

温度センサにより相対温度を計測できます。
温度センサは、基準電圧回路(BGR)の電圧を受けて温度に対応した電圧を出力します。出力された電圧は、ΔΣ型アナログデジタルコンバータ(DSADC)のユニット D に入力されており、DSAD 変換によりデジタル値として温度に対応する値を得ることができます。

温度変化による温度センサ出力電圧の差には直線性があるため、複数の温度条件でのデータを取得することで相対的な温度を計測することができます。

温度センサ、DSADC を使用しない場合、基準電圧回路に関し下記のとおり端子処理を行ってください。
・DSRVDD3、SRVDD は DVDD3 に接続
・DSRVSS は DGND に接続

TEMP ドライバ API は AMP 動作制御、BGR 動作制御、温度センサ制御などの機能を提供します。

本ドライバ API は、アプリで使用する API 定義を含む以下のファイルで構成されています。
\Libraries\TX03_Periph_Driver\src\tmpm311_ temp.c
\Libraries/TX03_Periph_Driver\inc\tmpm311_temp.h

補足: 基準電圧回路(BGR)はΔΣ 変換方式アナログ/デジタルコンバータ (DSADC)と共用しています。

7.2 API 関数

7.2.1 関数一覧

- void TEMP_SetAMPState(FunctionalState ***NewState***)
- FunctionalState TEMP_GetAMPState(void)
- void TEMP_SetBGRState(FunctionalState ***NewState***)
- FunctionalState TEMP_GetBGRState(void)
- void TEMP_SetSensorState(FunctionalState ***NewState***)
- FunctionalState TEMP_GetSensorState(void)

7.2.2 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) 温度センサ制御:
   TEMP_SetAMPState(), TEMP_SetBGRState(), TEMP_SetSensorState()
2) 温度センサ制御状態の取得:
   TEMP_GetAMPState(), TEMP_GetBGRState(), TEMP_GetSensorState()

7.2.3 関数仕様

7.2.3.1 TEMP_SetAMPState

ΔΣADC 用 AMP の有効/無効設定

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

引数:
NewState: AMP 動作を選択します。
- **ENABLE**: 有効
- **DISABLE**: 無効

機能:
ΔΣADC 用 AMP の有効/無効を設定します。

補足:
AMP 動作を有効にする前に BGR 動作を有効にしてください。

戻り値:
なし

7.2.3.2 TEMP_GetAMPState

ΔΣADC 用 AMP の有効/無効設定状態の取得

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

引数:
なし

機能:
ΔΣADC 用 AMP の有効/無効設定状態を取得します。

戻り値:
ΔΣADC 用 AMP の有効/無効設定状態:
ENABLE: 有効
DISABLE: 無効

7.2.3.3 TEMP_SetBGRState

基準電圧回路の有効/無効設定

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

引数:
NewState: 基準電圧回路の有効/無効を選択します。
- **ENABLE**: 有効
- **DISABLE**: 無効

機能:

基準電圧回路の有効/無効を設定します。

戻り値:
なし

7.2.3.4 TEMP_GetBGRState

基準電圧回路の有効/無効設定状態の取得

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

引数:
なし

機能:
基準電圧回路の有効/無効設定状態を取得します。

戻り値:
基準電圧回路の有効/無効設定状態:
ENABLE: 有効
DISABLE: 無効

7.2.3.5 TEMP_SetSensorState

温度センサの有効/無効設定

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

引数:
NewState: 温度センサの有効/無効を選択します。
- **ENABLE**: 有効
- **DISABLE**: 無効

機能:
温度センサの有効/無効を設定します。

戻り値:
なし

7.2.3.6 TEMP_GetSensorState

温度センサの有効/無効設定状態の取得

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

引数:
なし

機能:
温度センサの有効/無効設定状態を取得します。

戻り値:
温度センサの有効/無効設定状態:
ENABLE: 有効
DISABLE: 無効

7.2.4 データ構造

なし

8 TMR16A

8.1 概要

TMPM311 は TMR16A を 1 チャネル内蔵しています。
TMR16A には以下の機能があります。
・一致割り込み
・矩形波出力
・リードキャプチャ

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

/Libraries/TX03_Periph_Driver/src/tmpm311_tmr16a.c
/Libraries/TX03_Periph_Driver/inc/tmpm311_tmr16a.h

8.2 API 関数

8.2.1 関数一覧

◆ void TMR16A_SetIdleMode(TSB_T16A_TypeDef * ***T16Ax***);
◆ void TMR16A_SetClkInCoreHalt(TSB_T16A_TypeDef * ***T16Ax***, uint8_t ***ClkState***);
◆ void TMR16A_SetRunState(TSB_T16A_TypeDef * ***T16Ax***, uint32_t ***Cmd***);
◆ void TMR16A_SetSrcClk(TSB_T16A_TypeDef * ***T16Ax***, uint32_t ***SrcClk***);
◆ void TMR16A_SetFlipFlop(TSB_T16A_TypeDef * ***T16Ax***,
TMR16A_FFOutputTypeDef * ***FFStruct***);
◆ void TMR16A_ChangeCycle(TSB_T16A_TypeDef * ***T16Ax***, uint32_t ***Cycle***);
◆ uint16_t TMR16A_GetCaptureValue(TSB_T16A_TypeDef* ***T16Ax***);

8.2.2 関数の種類

関数は、主に以下の 3 種類に分かれています。
1) 各タイマの設定:
TMR16A_SetSrcClk(), TMR16A_SetRunState(), TMR16A_ChangeCycle()
2) ステータスの確認:
TMR16A_GetCaptureValue()
3) その他:
TMR16A_SetFlipFlop(),TMR16A_SetClkInCoreHalt (),TMR16A_SetIdleMode()

8.2.3 関数仕様

補足: 引数に記述されている “TSB_T16A_TypeDef* ***T16Ax***” は下記から選択してください。
TSB_T16A0

8.2.3.1 TMR16A_SetIdleMode

IDLE 時の動作設定

関数のプロトタイプ宣言:
void
TMR16A _SetIdleMode(TSB_T16A_TypeDef* ***T16Ax***,
FunctionalState ***NewState***)

引数:
T16Ax: TMR16A チャネルを指定します。
NewState: IDLE 時の動作を指定します。
- **ENABLE:** 動作
- **DISABLE:** 停止

機能:
NewState が **ENABLE** の場合、IDLE 時でも TMR16A チャネルは動作します。
DISABLE の場合、IDLE 時は動作を停止します。

戻り値:
なし

8.2.3.2 TMR16A_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
TMR16A_SetClkInCoreHalt (TSB_T16A_TypeDef* ***T16Ax***,
uint8_t ***ClkState***)

引数:
T16Ax: TMR16A チャネルを指定します。
***ClkState*:** デバッグ HALT 中のクロック動作を選択します。
- **TMR16A_RUNNING_IN_CORE_HALT:** 動作
- **TMR16A_STOP_IN_CORE_HALT:** 停止

機能:
デバッグツール使用時に HALT モードに遷移した場合、TMR16A クロック動作/停止の設定を行ないます。

戻り値:
なし

8.2.3.3 TMR16A_SetRunState

カウンタ動作の設定

関数のプロトタイプ宣言:
void
TMR16A_SetRunState(TSB_T16A_TypeDef* ***T16Ax***,
uint32_t ***Cmd***)

引数:
T16Ax: TMR16A チャネルを指定します。
Cmd: カウンタ動作を選択します。
- **TMR16A_RUN:** カウント
- **TMR16A_STOP:** 停止&クリア

機能:
Cmd が **TMR16A_RUN** の場合、アップカウンタがカウントを開始します。
Cmd が **TMR16A_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

8.2.3.4 TMR16A_SetSrcClk

ソースクロックの選択

関数のプロトタイプ宣言:
void
TMR16A_SetSrcClk(TSB_T16A_TypeDef* ***T16Ax***,
uint32_t ***SrcClk***)

引数:
T16Ax: TMR16A チャネルを指定します。
SrcClk: 以下からソースクロックを選択します。
- **TMR16A_SYSCK:** fsys
- **TMR16A_PRCK:** φT0

機能:
ソースクロックを選択します。

戻り値:
なし

8.2.3.5 TMR16A_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
TMR16A_SetFlipFlop(TSB_T16A_TypeDef* ***T16Ax***,
TMR16A_FFOutputTypeDef* ***FFStruct***)

引数:
T16Ax: TMR16A チャネルを指定します。
***FFStruct*:** TMRB のフリップフロップ機能に関する構造体です。(詳細は"データ構造"を参照)

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

戻り値:
なし

8.2.3.6 TMR16A_ChangeCycle

周期の設定

関数のプロトタイプ宣言:
void
TMR16A_ChangeCycle(TSB_T16A_TypeDef* ***T16Ax***,
uint32_t ***Cycle***)

引数:
T16Ax: TMR16A チャネルを指定します。
Cycle: 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

8.2.3.7 TMR16A_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
TMR16A_GetCaptureValue(TSB_T16A_TypeDef* ***T16Ax***)

引数:
T16Ax: TMR16A チャネルを指定します。

機能:
キャプチャレジスタの値を読み込みます。

戻り値:
キャプチャレジスタの値

8.2.4 データ構造

8.2.4.1 TMR16A_FFOutputTypeDef

メンバ:
uint32_t
TMR16AFlipflopCtrl: フリップフロップのレベルを選択します。
- **TMR16A_FLIPFLOP_INVERT**: 出力を反転(ソフト反転)します。
- **TMR16A_FLIPFLOP_SET**: 出力を"1"にセットします。
- **TMR16A_FLIPFLOP_CLEAR**: 出力を"0"にセットします。

uint32_t
TMR16AFlipflopReverseTrg: フリップフロップの反転トリガを選択します。
- **TMR16A_DISALBE_FLIPFLOP**: 反転トリガを無効にします。

- **TMR16A_FLIPFLOP_MATCH_CYCLE**: 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。

9 TMRB

9.1 概要

TMPM311 は、4 チャンネルの多機能 16 ビットタイマ/ イベントカウンタ (TMRB0 ～ TMRB3)を内蔵しています。各チャネルは下記モードで動作します。

- 16 ビットインタバルタイマモード
- 16 ビットイベントカウンタモード
- 16 ビットプログラマブル矩形波出力 (PPG) モード
- タイマ同期モード(各 4 チャネルの出力設定可能)

また、キャプチャ機能を利用することで、次のような用途に使用することができます。

- 周波数測定
- パルス幅測定

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

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm311_tmrb.c
/Libraries/TX03_Periph_Driver/inc/tmpm311_tmrb.h

9.2 API 関数

9.2.1 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***)
- 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***)

9.2.2 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()

9.2.3 関数仕様

補足: 引数に記述されている “TSB_TB_TypeDef* ***TBx***” は特に記載の無い限り以下から選択してください。
TSB_TB0, TSB_TB1, TSB_TB2, TSB_TB3

9.2.3.1 TMRB_Enable

TMRB 機能の許可

関数のプロトタイプ宣言:
void
TMRB_Enable(TSB_TB_TypeDef* ***TBx***)

引数:
***TBx*:** TMRB チャネルを指定します。

機能:
TMRB 機能を有効にします。

戻り値:
なし

9.2.3.2 TMRB_Disable

TMRB 機能の禁止

関数のプロトタイプ宣言:
void
TMRB_Disable(TSB_TB_TypeDef* ***TBx***)

引数:
***TBx*:** TMRB チャネルを指定します。

機能:

TMRB 機能を無効にします。

戻り値:
なし

9.2.3.3 TMRB_SetRunState

カウンタ動作の設定

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

引数:
***TBx*:** TMRB チャネルを指定します。
Cmd: カウンタ動作を選択します。
- **TMRB_RUN:** カウント
- **TMRB_STOP:** 停止&クリア

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

戻り値:
なし

9.2.3.4 TMRB_Init

TMRB チャネルの初期化

関数のプロトタイプ宣言:
void
TMRB_Init(TSB_TB_TypeDef* ***TBx***,
TMRB_InitTypeDef* ***InitStruct***)

引数:
***TBx*:** TMRB チャネルを指定します。
***InitStruct*:** TMRB に関する構造体です。(詳細は"データ構造"を参照)

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

戻り値:
なし

9.2.3.5 TMRB_SetCaptureTiming

キャプチャタイミングの設定

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

引数:
***TBx*:** TMRB チャネルを指定します。
CaptureTiming: キャプチャタイミングを選択します。

- **TMRB_DISABLE_CAPTURE**: キャプチャ禁止
- **TMRB_CAPTURE_TBIN_RISING_FALLING**: TBxIN 端子入力の立ち上がりでキャプチャレジスタ 0 (TBxCP0)にカウント値を取り込み、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。

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

9.2.3.6 TMRB_SetFlipFlop

フリップフロップ機能の設定

関数のプロトタイプ宣言:
void
TMRB_SetFlipFlop(TSB_TB_TypeDef* ***TBx***,
TMRB_FFOutputTypeDef* ***FFStruct***)

引数:
***TBx*:** TMRB チャネルを指定します。
***FFStruct*:** TMRB のフリップフロップ機能に関する構造体です。(詳細は"データ構造"を参照)

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

戻り値:
なし

9.2.3.7 TMRB_GetINTFactor

割り込み要因の取得

関数のプロトタイプ宣言:
TMRB_INTFactor
TMRB_GetINTFactor(TSB_TB_TypeDef* ***TBx***)

引数:
***TBx*:** TMRB チャネルを指定します。

機能:
割り込み要因を取得します。

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

補足:
異なる割り込み要因を処理する場合は、以下のように記述してください。

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

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

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

9.2.3.8 TMRB_SetINTMask

割り込みマスク要因の設定

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

引数:
***TBx*:** TMRB チャネルを指定します。

***INTMask*:** マスクする割り込みを選択します。

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

9.2.3.9 TMRB_ChangeLeadingTiming

デューティの設定

関数のプロトタイプ宣言:
void
TMRB_ ChangeLeadingTiming(TSB_TB_TypeDef* ***TBx***,
uint32_t ***LeadingTiming***)

引数:
***TBx*:** TMRB チャネルを指定します。
***LeadingTiming*:** デューティ値を設定します。最大値は 0xFFFF です。

機能:
デューティを設定します。実際のデューティのインターバルは、CG の校正と ***ClkDiv***(詳細は"データ構造"を参照) の値によります。

戻り値:
なし。

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

9.2.3.10 TMRB_ChangeTrailingTiming

周期の設定

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

引数:

***TBx*:** TMRB チャネルを指定します。

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

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

戻り値:
なし。

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

9.2.3.11 TMRB_GetUpCntValue

アップカウンタ値の読み込み

関数のプロトタイプ宣言:
uint16_t
TMRB_GetUpCntValue(TSB_TB_TypeDef* ***TBx***)

引数:
***TBx*:** TMRB チャネルを指定します。

機能:
アップカウンタ値の読み込みを行います。

戻り値:
アップカウンタ値

9.2.3.12 TMRB_GetCaptureValue

キャプチャレジスタの読み込み

関数のプロトタイプ宣言:
uint16_t
TMRB_GetCaptureValue(TSB_TB_TypeDef* ***TBx***,
uint8_t ***CapReg***)

引数:
***TBx*:** TMRB チャネルを指定します。
CapReg: キャプチャレジスタを選択します。
- **TMRB_CAPTURE_0:** キャプチャレジスタ 0
- **TMRB_CAPTURE_1:** キャプチャレジスタ 1

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

戻り値:

キャプチャされた値

9.2.3.13 TMRB_ExecuteSWCapture

ソフトウェアキャプチャの実行

関数のプロトタイプ宣言:
void
TMRB_ExecuteSWCapture(TSB_TB_TypeDef* ***TBx***)

引数:
***TBx*:** TMRB チャネルを指定します。

機能:
キャプチャレジスタ 0 (TBxCP0)にカウント値を取り込みます。

戻り値:
なし

9.2.3.14 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***)

引数:
***TBx*:** TMRB チャネルを指定します。

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

戻り値:
なし

補足:
TMRB の低消費電力モード動作機能は意味を持ちません。<I2TB>には"0"をライトします。

9.2.3.15 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*:** 以下から TMRB チャネルをします。
TSB_TB1, TSB_TB2, TSB_TB3

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

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

戻り値:
なし

9.2.3.16 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)
```

引数:
***TBx*:** TMRB チャネルを指定します。
***NewState*:** ダブルバッファの有効/無効を選択します。
- **ENABLE:** 許可。
- **DISABLE:** 禁止。

機能:
ダブルバッファ動作の許可/禁止を設定します。

戻り値:
なし

9.2.3.17 TMRB_SetExtStartTrg

外部トリガの設定

関数のプロトタイプ宣言:
```
void
TMRB_SetExtStartTrg (TSB_TB_TypeDef* TBx,
                     FunctionalState NewState,
                     uint8_t TrgMode)
```

引数:
***TBx*:** TMRB チャネルを指定します。

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

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

機能:
外部トリガによる変換開始の有無とアクティブエッジの設定を行います。

補足:
NewState が **ENABLE** の場合のみ ***TrgMode*** を選択できます。

戻り値:
なし

9.2.3.18 TMRB_SetClkInCoreHalt

デバッグ HALT 中のクロック動作

関数のプロトタイプ宣言:
void
TMRB_SetClkInCoreHalt (TSB_TB_TypeDef* ***TBx***, uint8_t ***ClkState***)

引数:
***TBx*:** TMRB チャネルを指定します。

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

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

戻り値:
なし

9.2.4 データ構造

9.2.4.1 TMRB_InitTypeDef

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

uint32_t
ClkDiv: インターバルタイマのソースクロックの分周を選択します。
- **TMRB_CLK_DIV_2**: fperiph / 2
- **TMRB_CLK_DIV_8**: fperiph / 8
- **TMRB_CLK_DIV_32**: fperiph / 32
- **TMRB_CLK_DIV_64**: fperiph / 64
- **TMRB_CLK_DIV_128**: fperiph / 128

- **TMRB_CLK_DIV_256**: fperiph / 256
- **TMRB_CLK_DIV_512**: fperiph / 512

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

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

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

9.2.4.2 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。

9.2.4.3 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　　未定義

10 SIO/UART

10.1 概要

TMPM311 はシリアル I/O を 1 チャネル内蔵しています(UART0)。各チャネルは I/O インタフェースモード(同期通信モード)と 7, 8, 9 ビット長の UART モード(非同期通信)の 2 つの動作モードを持っています。

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

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm311_uart.c
/Libraries/TX03_Periph_Driver/inc/tmpm311_uart.h

10.2 API 関数

10.2.1 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***)
- void UART_SetIdleMode(TSB_SC_TypeDef* ***UARTx***)
- 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***)

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) 初期化と設定:
   UART_Enable(), UART_Disable(),UART_SetInputClock(),UART_Init(),
   UART_DefaultConfig(), SIO_Enable(), SIO_Disable(), SIO_SetInputClock(),
   SIO_Init()
2) 送受信設定とエラー確認:
   UART_GetBufState(), UART_GetRxData(), UART_SetTxData(),
   UART_GetErrState(), SIO_GetRxData(), SIO_SetTxData()
3) 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()
4) その他の設定
   UART_SWReset(), UART_SetWakeUpFunc(), UART_SetIdleMode()

10.2.3 関数仕様

補足: 下記関数の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

また、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

10.2.3.1 UART_Enable

UART 機能の許可

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

引数:
UARTx: UART チャネルを指定します。

機能:
UART 機能を有効にします。

戻り値:

なし

10.2.3.2 UART_Disable

UART 機能の禁止

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

引数:
UARTx: UART チャネルを指定します。

機能:
UART 機能を無効にします。

戻り値:
なし

10.2.3.3 UART_GetBufState

送受信バッファ状態の読み込み

関数のプロトタイプ宣言:
WorkState
UART_GetBufState(TSB_SC_TypeDef* ***UARTx***,
uint8_t ***Direction***)

引数:
UARTx: UART チャネルを指定します。

Direction: 送信/受信を選択します。
- **UART_RX**: 受信
- **UART_TX**: 送信

機能:
Direction が **UART_RX** の場合、以下の受信バッファの状態を返します。
DONE:受信データはバッファに保存済み
BUSY:データ受信中
Direction が **UART_TX** の場合、以下の送信バッファの状態を返します。
DONE:バッファ中のデータは送信済み
BUSY:データ送信中

戻り値:
DONE: バッファリード/ライト可能状態
BUSY: 送受信中

10.2.3.4 UART_SWReset

ソフトウェアリセットの実行

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

引数:
UARTx: UART チャネルを指定します。

機能:
ソフトウェアリセットを実行します。

戻り値:
なし

10.2.3.5 UART_Init

UART チャネルの初期化

関数のプロトタイプ宣言:
void
UART_Init(TSB_SC_TypeDef* ***UARTx***,
UART_InitTypeDef* ***InitStruct***)

引数:
UARTx: UART チャネルを指定します。

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

機能:
ボーレート、ビット単位の転送、ストップビット、パリティ、転送モード、フローコントロールなどの初期設定を行います。

戻り値:
なし

10.2.3.6 UART_GetRxData

受信データの読み込み

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

引数:
UARTx: UART チャネルを指定します。

機能:
受信データを読み込みます。**UART_GetBufState(*UARTx*, UART_RX)**にて **DONE** を読み出した後、もしくは UART (シリアルチャネル) 割り込み関数の中で実行してください。

戻り値:

受信データです。データ範囲は 0x00～0x1FF です。

10.2.3.7 UART_SetTxData

送信データの設定

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

引数:
UARTx: UART チャネルを指定します。

Data: 送信データ(7 ビット、8 ビット、9 ビット)

機能:
送信データを設定します。**UART_GetBufState(*UARTx,* UART_TX)**にて **DONE** を読み出した後、もしくは UART (シリアルチャネル) 割り込み関数の中で実行してください。

戻り値:
なし

10.2.3.8 UART_DefaultConfig

デフォルト構成での初期化

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

引数:
UARTx: UART チャネルを指定します。

機能:
以下の構成で初期化します:
ボーレート: 115200 bps
データ長: 8 ビット
ストップビット: 1 ビット
パリティ: なし
フローコントロール: なし
送受信有効。ボーレートジェネレータはソースクロックとして使用。

戻り値:
なし

10.2.3.9 UART_GetErrState

転送エラーフラグの読み出し

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

引数:
UARTx: UART チャネルを指定します。

機能:
転送エラーフラグを読み出します。

戻り値:
UART_NO_ERR: エラーなし
UART_OVERRUN: オーバーランエラー
UART_PARITY_ERR: パリティエラー
UART_FRAMING_ERR: フレーミングエラー
UART_ERRS: 上記の 2 つ以上のエラーが発生している

10.2.3.10 UART_SetWakeUpFunc

9 ビットモード時のウェイクアップ機能の設定

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

引数:
UARTx: UART チャネルを指定します。
NewState:ウェイクアップ機能の有効/無効を選択します。
- **ENABLE**: 有効
- **DISABLE**: 無効

機能:
9 ビットモード時のウェイクアップ機能を設定します。***NewState*** が **ENABLE** の場合、ウェイクアップ機能を有効に、***NewState*** が **DISABLE** の場合、ウェイクアップ機能を無効に設定します。 ウェイクアップ機能は、9 ビットモード時のみ機能します。

戻り値:
なし

10.2.3.11 UART_SetInputClock

入力クロックの設定

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

引数:
UARTx: UART チャネルを指定します。

Clock : 以下から、プリスケーラの入力クロックを選択します。
- **0** : ΦT0/2
- **1** :ΦT0

機能:
プリスケーラの入力クロックを選択します。

戻り値:
なし

10.2.3.12 UART_SetIdleMode

IDLE 時の動作

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

引数:
UARTx: UART チャネルを指定します。

NewState: IDLE 時の動作を選択します。
- **ENABLE**: 動作
- **DISABLE**: 停止

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

戻り値:
なし

10.2.3.13 UART_FIFOConfig

FIFO の許可

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

引数:
UARTx: UART チャネルを指定します。

NewState: FIFO の許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
FIFO の許可/禁止を選択します。

NewState が **ENABLE** の場合、FIFO を許可します。**DISABLE** の場合、FIFO を禁止します。

戻り値:
なし

10.2.3.14 UART_SetFIFOTransferMode

転送モードの選択

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

引数:
UARTx: UART チャネルを指定します。

TransferMode: 転送モードを選択します。
- **UART_TRANSFER_PROHIBIT**： 転送禁止
- **UART_TRANSFER_HALFDPX_RX**： 半二重(受信)
- **UART_TRANSFER_HALFDPX_TX**： 半二重(送信)
- **UART_TRANSFER_FULLDPX**： 全二重

機能:
転送モードを選択します。

戻り値:
なし

10.2.3.15 UART_TRxAutoDisable

送信/受信の自動禁止

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

引数:
UARTx: UART チャネルを指定します。

TRxAutoDisable: 送信/受信の自動禁止機能を制御します。
- **UART_RXTXCNT_NONE**: なし
- **UART_RXTXCNT_AUTODISABLE**: 自動禁止

機能:
送信/受信の自動禁止機能を制御します。

戻り値:
なし

10.2.3.16 UART_RxFIFOINTCtrl

受信 FIFO 使用時の受信割り込み許可

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

引数:
UARTx: UART チャネルを指定します。

NewState: 受信 FIFO 使用時の受信割り込みの許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

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

戻り値:
なし

10.2.3.17 UART_TxFIFOINTCtrl

送信 FIFO 使用時の送信割り込み許可

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

引数:
UARTx: UART チャネルを指定します。

NewState: 送信 FIFO 使用時の送信割り込みの許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

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

戻り値:
なし

10.2.3.18 UART_RxFIFOByteSel

受信 FIFO 使用バイト数

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

引数:
UARTx: UART チャネルを指定します。

BytesUsed: 受信 FIFO 使用バイト数を設定します。
- **UART_RXFIFO_MAX**: 最大
- **UART_RXFIFO_RXFLEVEL**: 受信 FIFO の FILL レベルに同じ

機能:
受信 FIFO 使用バイト数を設定します。

戻り値:
なし

10.2.3.19 UART_RxFIFOFillLevel

受信割り込みが発生する受信 FIFO の fill レベルの設定

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

引数:
UARTx: UART チャネルを指定します。

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

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

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

戻り値:
なし

10.2.3.20 UART_RxFIFOINTSel

受信割り込み発生条件の選択

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

引数:
UARTx: UART チャネルを指定します。

RxINTCondition: 受信 割り込み発生条件を選択します。

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

機能:
受信割り込み発生条件を選択します。

戻り値:
なし

10.2.3.21 UART_RxFIFOClear

受信 FIFO クリア

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

引数:
UARTx: UART チャネルを指定します。

機能:
受信 FIFO をクリアします。

戻り値:
なし

10.2.3.22 UART_TxFIFOFillLevel

送信割り込みが発生する送信 FIFO の fill レベルの設定

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

引数:
UARTx: UART チャネルを指定します。

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

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

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

戻り値:

なし

10.2.3.23 UART_TxFIFOINTSel

送信割り込み発生条件の選択

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

引数:
UARTx: UART チャネルを指定します。

TxINTCondition: 受信 割り込み発生条件を選択します。

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

機能:
送信割り込み発生条件を選択します。

戻り値:
なし

10.2.3.24 UART_TxFIFOClear

送信 FIFO クリア

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

引数:
UARTx: UART チャネルを指定します。

機能:
送信 FIFO をクリアします。

戻り値:
なし

10.2.3.25 UART_TxBufferClear

送信バッファのクリア

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

引数:

UARTx: UART チャネルを指定します。

機能:
送信バッファをクリアします。

戻り値:
なし

10.2.3.26 UART_GetRxFIFOFillLevelStatus

受信 FIFO の fill レベルの取得

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

引数:
UARTx: UART チャネルを指定します。

機能:
受信 FIFO の fill レベルを取得します。

戻り値:
- **UART_TRXFIFO_EMPTY**: Empty
- **UART_TRXFIFO_1B**: 1 バイト
- **UART_TRXFIFO_2B**: 2 バイト
- **UART_TRXFIFO_3B**: 3 バイト
- **UART_TRXFIFO_4B**: 4 バイト

10.2.3.27 UART_GetRxFIFOOverRunStatus

受信 FIFO オーバーラン状態の取得

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

引数:
UARTx: UART チャネルを指定します。

機能:
受信 FIFO オーバーラン状態を取得します。

戻り値:
UART_RXFIFO_OVERRUN: オーバーラン発生

10.2.3.28 UART_GetTxFIFOFillLevelStatus

送信 FIFO の fill レベルの取得

関数のプロトタイプ宣言:

uint32_t
UART_GetTxFIFOFillLevelStatus (TSB_SC_TypeDef* ***UARTx***);

引数:
UARTx: UART チャネルを指定します。

機能:
送信 FIFO の fill レベルの取得

戻り値:
- **UART_TRXFIFO_EMPTY**: Empty
- **UART_TRXFIFO_1B**: 1 バイト
- **UART_TRXFIFO_2B**: 2 バイト
- **UART_TRXFIFO_3B**: 3 バイト
- **UART_TRXFIFO_4B**: 4 バイト

10.2.3.29 UART_GetTxFIFOUnderRunStatus

送信 FIFO アンダーラン状態の取得

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

引数:
UARTx: UART チャネルを指定します。

機能:
送信 FIFO アンダーラン状態を取得します。

戻り値:
UART_TXFIFO_UNDERRUN: アンダーラン発生

10.2.3.30 SIO_SetInputClock

入力クロックの設定

関数のプロトタイプ宣言:
void
SIO_SetInputClock (TSB_SC_TypeDef * SIOx,
uint32_t Clock)
引数:
SIOx: SIO チャネルを指定します。
Clock : 以下から、プリスケーラの入力クロックを選択します。
- **SIO_CLOCK_T0_HALF** : ΦT0/2
- **SIO_CLOCK_T0** :ΦT0

機能:
プリスケーラの入力クロックを選択します。

戻り値:

なし

10.2.3.31 SIO_Enable

SIO 動作の許可

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

引数:
SIOx: SIO チャネルを指定します。

機能:
SIO 動作を許可します。

戻り値:
なし

10.2.3.32 SIO_Disable

SIO 動作の禁止

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

引数:
SIOx: SIO チャネルを指定します。

機能:
SIO 動作を禁止します。

戻り値:
なし

10.2.3.33 SIO_GetRxData

受信用バッファの取得

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

引数:
SIOx: SIO チャネルを指定します。

機能:
受信用バッファを取得します。

戻り値:

受信用バッファ(値の範囲は 0x00 ~ 0xFF です)

10.2.3.34 SIO_SetTxData

送信用バッファの設定

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

引数:
SIOx: SIO チャネルを指定します。

Data: 送信用バッファ

機能:
送信用バッファを指定します。

戻り値:
なし

10.2.3.35 SIO_Init

SIO モードの初期設定

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

引数:
SIOx: SIO チャネルを指定します。

IOClkSel: クロックを選択します。
- **SIO_CLK_BAUDRATE**: ボーレートジェネレータ
- **SIO_CLK_SCLKINPUT**: SCLKx 端子入力

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

機能:
SIO モードの初期設定を行います。

戻り値:
なし

10.2.4 データ構造

10.2.4.1 UART_InitTypeDef

メンバ

uint32_t
BaudRate :UART 通信ボーレートを 2400(bps) から 115200(bps) に設定。(*)

uint32_t
DataBits： 転送ビット数を選択します。
- **UART_DATA_BITS_7** : 7 ビットモード
- **UART_DATA_BITS_8** : 8 ビットモード
- **UART_DATA_BITS_9** :9 ビットモード

uint32_t
StopBits :ストップビット長を選択します。
- **UART_STOP_BITS_1:** 1 ビット
- **UART_STOP_BITS_2 :** 2 ビット

uint32_t
Parity : パリティを選択します。
- **UART_NO_PARITY :** パリティなし
- **UART_EVEN_PARITY**: 偶数(Even) パリティ
- **UART_ODD_PARITY**: 偶数(Even) パリティ

uint32_t
Mode: 転送モードを選択します。送受信の場合は、送信と受信をOR 演算子によって組み合わせてください。
- **UART_ENABLE_TX:** 送信許可
- **UART_ENABLE_RX:** 受信許可

uint32_t
FlowCtrl: フロー制御モードを選択します。(**)
- **UART_NONE_FLOW_CTRL**: フロー制御 無効

(*)補足:
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。
(**)補足:
UART_NONE_FLOW_CTRLのみ選択可能です。

10.2.4.2 SIO_InitTypeDef

メンバ:
uint32_t
InputClkEdge: 入力クロックエッジを選択します。
- **SIO_SCLKS_TXDF_RXDR**: SCLKx 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 TXDx 端子へ出力します。
- **SIO_SCLKS_TXDR_RXDF**: SCLKx 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 TXDx 端子へ出力します。

uint32_t
TIDLE: 最終ビット出力後の TXDx 端子の状態を選択します。
- **SIO_TIDLE_LOW**: "Low"出力保持
- **SIO_TIDLE_HIGH**: "High"出力保持
- **SIO_TIDLE_LAST**: 最終ビット保持

uint32_t
TXDEMP: アンダーランエラーが発生したときの TXDx 端子の状態を選択します。
- **SIO_TXDEMP_LOW**: "Low"出力
- **SIO_TXDEMP_HIGH**: "High"出力保持

uint32_t
EHOLDTime: クロック入力モードの TXDx 端子の最終ビットホールド時間を設定します。
- **SIO_EHOLD_FC_2**: 2/fc.
- **SIO_EHOLD_FC_4**: 4/fc.
- **SIO_EHOLD_FC_8**: 8/fc.
- **SIO_EHOLD_FC_16**: 16/fc.
- **SIO_EHOLD_FC_32**: 32/fc.
- **SIO_EHOLD_FC_64**: 64/fc.
- **SIO_EHOLD_FC_128**: 128/fc.

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

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

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

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

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

uint32_t
BaudRateClock: ボーレートジェネレータ入力クロックを選択します。

- **SIO_BR_CLOCK_TS0**: ΦTS0
- **SIO_BR_CLOCK_TS2**: ΦTS2
- **SIO_BR_CLOCK_TS8**: ΦTS8
- **SIO_BR_CLOCK_TS32**: ΦTS32

uint32_t
Divider: 分周値"N"を選択します。

- **SIO_BR_DIVIDER_16**: 16 分周
- **SIO_BR_DIVIDER_1**: 1 分周
- **SIO_BR_DIVIDER_2**: 2 分周
- **SIO_BR_DIVIDER_3**: 3 分周
- **SIO_BR_DIVIDER_4**: 4 分周
- **SIO_BR_DIVIDER_5**: 5 分周
- **SIO_BR_DIVIDER_6**: 6 分周
- **SIO_BR_DIVIDER_7**: 7 分周
- **SIO_BR_DIVIDER_8**: 8 分周
- **SIO_BR_DIVIDER_9**: 9 分周
- **SIO_BR_DIVIDER_10**: 10 分周
- **SIO_BR_DIVIDER_11**: 11 分周
- **SIO_BR_DIVIDER_12**: 12 分周
- **SIO_BR_DIVIDER_13**: 13 分周
- **SIO_BR_DIVIDER_14**: 14 分周
- **SIO_BR_DIVIDER_15**: 15 分周

11 uDMAC

11.1 概要

TMPM311 は µDMAC を 1 ユニット内蔵しています。
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		-
The number of transfer	1 to 1024 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		-
Interrupt function	Transfer end interrupt		Output for each channel
	Error interrupt		
Operation mode	Basic mode Automatic request mode Ping-pong mode Memory scatter / gather mode Peripheral scatter / gather mode		-

µDMAC API は、MCU の µDMAC モジュールを使用するための機能セットを提供します。本 API には、µDMAC 転送タイプセット、チャネルセット、マスクセット、初期/代替データエリアセット、チャネル優先、初期化データ設定などが含まれます。
全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver\src\tmpm311_udmac.c
/Libraries/TX03_Periph_Driver\inc\tmpm311_udmac.h

補足: 本ドキュメントでは uDMAC を単に DMAC と表現します。

11.2 API 関数

11.2.1 関数一覧

- FunctionalState DMAC_GetDMACState(TSB_DMA_TypeDef * ***DMACx***)
- void DMAC_Enable(TSB_DMA_TypeDef * ***DMACx***)
- void DMAC_Disable(TSB_DMA_TypeDef * ***DMACx***)
- void DMAC_SetPrimaryBaseAddr(TSB_DMA_TypeDef * ***DMACx***, uint32_t ***Addr***)
- uint32_t DMAC_GetBaseAddr(TSB_DMA_TypeDef * ***DMACx***,
  DMAC_PrimaryAlt ***PriAlt***)
- void DMAC_SetSWReq(TSB_DMA_TypeDef * ***DMACx*** ,

uint8_t ***Channel***)

- void DMACA_SetTransferType(DMACA_Channel ***Channel***, DMAC_TransferType ***Type***)
- DMAC_TransferType DMACA_GetTransferType(DMACA_Channel ***Channel***)
- void DMAC_SetMask(TSB_DMA_TypeDef * ***DMACx*** , uint8_t ***Channel*** , FunctionalState ***NewState***)
- FunctionalState DMAC_GetMask(TSB_DMA_TypeDef * ***DMACx*** , uint8_t ***Channel***)
- void DMAC_SetChannel(TSB_DMA_TypeDef * ***DMACx*** , uint8_t ***Channel*** , FunctionalState ***NewState***)
- FunctionalState DMAC_GetChannelState(TSB_DMA_TypeDef * ***DMACx*** , uint8_t ***Channel***)
- void DMAC_SetPrimaryAlt(TSB_DMA_TypeDef * ***DMACx*** , uint8_t ***Channel*** DMAC_PrimaryAlt ***PriAlt***)
- DMAC_PrimaryAlt DMAC_GetPrimaryAlt(TSB_DMA_TypeDef * ***DMACx*** , uint8_t ***Channel***)
- void DMAC_SetChannelPriority(TSB_DMA_TypeDef * ***DMACx*** , uint8_t ***Channel*** , DMAC_Priority ***Priority***)
- DMAC_Priority DMAC_GetChannelPriority(TSB_DMA_TypeDef * ***DMACx*** , uint8_t ***Channel***)
- void DMAC_ClearBusErr(TSB_DMA_TypeDef * ***DMACx***)
- Result DMAC_GetBusErrState(TSB_DMA_TypeDef * ***DMACx***)
- void DMAC_FillInitData(TSB_DMA_TypeDef * ***DMACx*** , uint8_t ***Channel*** , DMAC_InitTypeDef * ***InitStruct***)

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) uDMAC configuration by DMACA_SetTransferType(), DMACA_GetTransferType(), DMAC_SetMask(),DMAC_GetMask(),DMAC_SetChannel(),DMAC_GetChannelState(),DMAC_SetPrimaryAlt(),DMAC_GetPrimaryAlt(),DMAC_SetChannelPriority(),DMAC_GetChannelPriority().
2) uDMAC enable/disable by DMAC_GetDMACState(), DMAC_Enable(), DMAC_Disable().
3) uDMAC software trigger by DMAC_SetSWReq().
4) uDMAC bus error by DMAC_ClearBusErr(), DMAC_GetBusErrState().
5) uDMAC control data area filled by: DMAC_FillInitData(), DMAC_SetPrimaryBaseAddr(), DMAC_GetBaseAddr().

11.2.3 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

Channel: チャネル選択です。

- **DMACA_SSP0_RX :** SSP0 受信
- **DMACA_SSP0_TX :** SSP0 送信

- **DMACA_UART0_RX :** UART0 受信
- **DMACA_UART0_TX :** UART0 送信

11.2.3.1 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

11.2.3.2 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

11.2.3.3 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

11.2.3.4 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

11.2.3.5 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

11.2.3.6 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***,
uint8_t ***Channel***)

引数:
DMACx: DMAC ユニットを選択します。
Channel: チャネルを選択します。

機能:
ソフトウェア転送要求を設定します。

戻り値:
なし

11.2.3.7 DMACA_SetTransferType

DMAC 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:チャネルを選択します。
Type が **DMAC_BURST** の場合:
- **DMACA_SSP0_RX :** SSP0 受信
- **DMACA_SSP0_TX :** SSP0 送信
- **DMACA_UART0_RX :** UART0 受信
- **DMACA_UART0_TX :** UART0 送信

Type が **DMAC_SINGLE** の場合**:**
- **DMACA_SSP0_RX :** SSP0 受信
- **DMACA_SSP0_TX :** SSP0 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

11.2.3.8 DMACA_GetTransferType

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: チャネルを選択します。
- **DMACA_SSP0_RX :** SSP0 受信
- **DMACA_SSP0_TX :** SSP0 送信
- **DMACA_UART0_RX :** UART0 受信
- **DMACA_UART0_TX :** UART0 送信

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

11.2.3.9 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,
             uint8_t Channel,
             FunctionalState NewState)
```

引数:
DMACx: DMAC ユニットを選択します。
Channel: チャネルを選択します。
NewState: DMA 要求のマスク設定/クリアを選択します。
- **ENABLE:** DMA 要求マスクのクリア。
- **DISABLE:** DMA 要求のマスク設定

機能:
DMA 要求のマスク設定/クリアを選択します。

戻り値:
なし

11.2.3.10 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,
             uint8_t Channel)
```

引数:
DMACx: DMAC ユニットを選択します。
Channel: チャネルを選択します。

機能:
DMA 要求のマスク状態を読み出します。

戻り値:
DMA 要求のマスク状態:
- **ENABLE:** DMA 要求のマスク設定なし
- **DISABLE:** DMA 要求のマスク設定あり

11.2.3.11 DMAC_SetChannel

DMA チャネルの有効/無効設定

関数のプロトタイプ宣言:
void
DMAC_SetChannel(TSB_DMA_TypeDef * ***DMACx***,
uint8_t ***Channel***,
FunctionalState ***NewState***)

引数:
DMACx: DMAC ユニットを選択します。
Channel: チャネルを選択します。
NewState: DMA チャネルの有効/無効を選択します。
- **ENABLE:** 有効
- **DISABLE:** 無効

機能:
DMA チャネルの有効/無効を設定します。

戻り値:
なし

11.2.3.12 DMAC_GetChannelState

DMA チャネルの有効/無効状態の取得

関数のプロトタイプ宣言:
FunctionalState
DMAC_GetChannelState(TSB_DMA_TypeDef * ***DMACx***,
uint8_t ***Channel***)

引数:
DMACx: DMAC ユニットを選択します。
Channel: チャネルを選択します。

機能:
DMA チャネルの有効/無効状態を取得します。

戻り値:
DMA チャネルの有効/無効状態:
- **ENABLE:** 有効
- **DISABLE:** 無効

11.2.3.13 DMAC_SetPrimaryAlt

一次データあるは代替データの選択

関数のプロトタイプ宣言:
void
DMAC_SetPrimaryAlt(TSB_DMA_TypeDef * ***DMACx***,
uint8_t ***Channel***,
DMAC_PrimaryAlt ***PriAlt***)

引数:
DMACx: DMAC ユニットを選択します。
Channel: チャネルを選択します。
PriAlt: 一次データあるいは代替データを選択します。

- **DMAC_PRIMARY:** 一次データ使用
- **DMAC_ALTERNATE:** 代替データ使用

機能:
一次データあるは代替データの使用有無を設定します。

戻り値:
なし

11.2.3.14 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***,
uint8_t ***Channel***)

引数:
DMACx: DMAC ユニットを選択します。
Channel: チャネルを選択します。

機能:
一次データあるは代替データの選択状態を読み出します。

戻り値:
一次データあるは代替データの選択状態:

- **DMAC_PRIMARY:** 一次データ
- **DMAC_ALTERNATE:** 代替データ

11.2.3.15 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***,
uint8_t ***Channel***,

DMAC_Priority ***Priority***)

引数:
DMACx: DMAC ユニットを選択します。
Channel: チャネルを選択します。
Priority: 優先順位を選択します。
- **DMAC_PRIOTIRY_NORMAL:** 通常
- **DMAC_PRIOTIRY_HIGH:** 最優先

機能:
優先度を設定します。

戻り値:
なし

11.2.3.16 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***,
uint8_t ***Channel***)

引数:
DMACx: DMAC ユニットを選択します。
Channel: チャネルを選択します。

機能:
優先度の設定状態を読み出します。

戻り値:
優先度の設定状態:
- **DMAC_PRIOTIRY_NORMAL:** 通常
- **DMAC_PRIOTIRY_HIGH:** 最優先

11.2.3.17 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

11.2.3.18 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

11.2.3.19 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***,
uint8_t ***Channel***,
DMAC_InitTypeDef * ***InitStruct***)

引数:
DMACx: DMAC ユニットを選択します。
Channel: チャネルを選択します。
InitStruct: DMA 設定に関する構造体です。

機能:
DMA 設定状態を読み出します。

戻り値:
なし

11.2.4 データ構造

11.2.4.1 DMAC_InitTypeDef

メンバ:
uint32_t
SrcEndPointer: データ送信元最終アドレス

uint32_t

DstEndPointer: データ送信先最終アドレス

DMAC_CycleCtrl
Mode: 動作モード

- **DMAC_INVALID:** 無効。DMA は動作を停止します。
- **DMAC_BASIC:** 基本モード
- **DMAC_AUTOMATIC:** 自動要求モード
- **DMAC_PINGPONG:** ピンポンモード
- **DMAC_MEM_SCATTER_GATHER_PRI:** メモリスキャッターギャザーモード(一次データ)
- **DMAC_MEM_SCATTER_GATHER_ALT:** メモリスキャッターギャザーモード(代替データ)
- **DMAC_PERI_SCATTER_GATHER_PRI:** 周辺スキャッターギャザーモード(一次データ)
- **DMAC_PERI_SCATTER_GATHER_ALT:** 周辺スキャッターギャザーモード(代替データ)

DMAC_Next_UseBurst
NextUseBurst: 周辺スキャッターギャザーモードで代替データを用いた DMA 転送終了時に<chnl_useburst_set>ビットに"1"を設定するかどうかを指定します。

- **DMAC_NEXT_NOT_USE_BURST:** <chnl_useburst_set>の値を変更しない。
- **DMAC_NEXT_USE_BURST:** <chnl_useburst_set> に"1"を設定する。

uint32_t
TxNum: 転送数(最大値は 1024 回)

DMAC_Arbitration
ArbitrationMoment: アービトレーションを選択します。設定した回数の転送後に転送要求を確認し、優先度の高い要求があれば制御が最優先のチャネルに切り替わります。

DMAC_BitWidth
SrcWidth: 転送元データサイズ

- **DMAC_BYTE:** 1 バイト
- **DMAC_HALF_WORD:** 2 バイト
- **DMAC_WORD:** 4 バイト

DMAC_IncWidth
SrcInc: 転送元アドレスのインクリメント

- **DMAC_INC_1B:** 1 バイト
- **DMAC_INC_2B:** 2 バイト
- **DMAC_INC_4B:** 4 バイト
- **DMAC_INC_0B:** インクリメントなし

DMAC_BitWidth
DstWidth: 転送先データサイズ

- **DMAC_BYTE:** 1 バイト
- **DMAC_HALF_WORD:** 2 バイト
- **DMAC_WORD:** 4 バイト

DMAC_IncWidth

DstInc: 転送先アドレスのインクリメント

- **DMAC_INC_1B:** 1 バイト
- **DMAC_INC_2B:** 2 バイト
- **DMAC_INC_4B:** 4 バイト
- **DMAC_INC_0B:** インクリメントなし

12 WDT

12.1 概要

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

検出時間、カウンタのオーバーフロー時の出力、アイドルモードでの動作設定などの引数等、ウォッチドッグタイマの設定を行う関数を提供します。

本ドライバは、以下のファイルで構成されています。
\Libraries\TX03_Periph_Driver\src\tmpm311_wdt.c
\Libraries/TX03_Periph_Driver\inc\tmpm311_wdt.h

12.2 API 関数

12.2.1 関数一覧

- Result WDT_SetDetectTime(uint32_t ***DetectTime***)
- Result WDT_SetIdleMode(void)
- Result WDT_SetOverflowOutput(uint32_t ***OverflowOutput***)
- Result WDT_Init(WDT_InitTypeDef * ***InitStruct***)
- Result WDT_Enable(void)
- Result WDT_Disable(void)
- Result WDT_WriteClearCode(void)
- FunctionalState WDT_GetWritingFlg(void)

12.2.2 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) ウォッチドッグタイマ設定:
   WDT_SetDetectTime(), WDT_SetOverflowOutput(), WDT_Init(), WDT_Enable(), WDT_Disable(), WDT_WriteClearCode()
2) IDLE モード時の開始・停止:
   WDT_SetIdleMode().
3) WDMOD または WDCR レジスタ書き込みステータスの取得:
   WDT_GetWritingFlg()

12.2.3 関数仕様

12.2.3.1 WDT_SetDetectTime

検出時間の設定

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

引数:
***DetectTime*:** 検出時間を選択します。

- **WDT_DETECT_TIME_EXP_15: *DetectTime*** is 2^15/fsys
- **WDT_DETECT_TIME_EXP_17: *DetectTime*** is 2^17/fsys

- **WDT_DETECT_TIME_EXP_19:** ***DetectTime*** is 2^19/fsys
- **WDT_DETECT_TIME_EXP_21:** ***DetectTime*** is 2^21/fsys
- **WDT_DETECT_TIME_EXP_23:** ***DetectTime*** is 2^23/fsys
- **WDT_DETECT_TIME_EXP_25:** ***DetectTime*** is 2^25/fsys

機能:
WDT の検出時間を設定します。

戻り値:
SUCCESS: 設定成功
ERROR: 設定失敗

12.2.3.2 WDT_SetIdleMode

IDLE モード時の動作

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

引数:
なし

機能:
IDLE モード時の WDT カウンタの動作を停止します。

補足:
WDT の低消費電力モードの設定は意味がありません。<I2WDT>に"0"をライトします。

戻り値:
SUCCESS: 設定成功
ERROR: 設定失敗

12.2.3.3 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:
result
WDT_SetOverflowOutput(uint32_t ***OverflowOutput***)

引数:
OverflowOutput: カウンタオーバーフロー時の設定を選択します。
- **WDT_NMIINT**: NMI 割り込み発生
- **WDT_WDOUT**: リセット

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

戻り値:

SUCCESS: 設定成功
ERROR: 設定失敗

12.2.3.4 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:
SUCCESS: 設定成功
ERROR: 設定失敗

12.2.3.5 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:
SUCCESS: 設定成功
ERROR: 設定失敗

12.2.3.6 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:
SUCCESS: 設定成功
ERROR: 設定失敗

12.2.3.7 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:
SUCCESS: 設定成功
ERROR: 設定失敗

12.2.3.8 WDT_GetWritingFlg

レジスタ書き込みステータスの取得

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

引数:
なし

機能:
レジスタ書き込みステータスを取得します。

補足:
WDMOD および WDCR に書き込む際は、このフラグが **ENABLE** であることを確認してください。

戻り値:
レジスタ書き込みステータス:
ENABLE: レジスタ書き込み可能
DISABLE: レジスタ書き込み禁止

12.2.4 データ構造

12.2.4.1 WDT_InitTypeDef

メンバ:
uint32_t
DetectTime 検出時間を選択します。

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

uint32_t
***OverflowOutput*:** カウンタオーバーフロー時の設定を選択します。

- **WDT_WDOUT:** リセット
- **WDT_NMIINT:** NMI 割り込み