TOSHIBA

TX03 ペリフェラルドライバ
ユーザガイド
(TMPM366)

第一版
2017 年 9 月

東芝デバイス＆ストレージ株式会社

CMDR-M366UG-01J

本製品取り扱い上のお願い

- ソフトウエア使用権許諾契約書の同意無しに使用しないで下さい。

目次

1. はじめに

本ソフトウェアは、東芝製TX03シリーズマイコンTMPM366用ペリフェラルドライバセットです。

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。

TMPM366 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

/Libraries
TX03 CMSIS ファイルと TMPM366 ペリフェラルドライバが格納されています。

/Libraries/ TX03_CMSIS
このフォルダには TMPM366 CMSIS ファイルのデバイス・ペリフェラル・アクセス・レイヤーが格納されています。

/Libraries/TX03_Periph_Driver
TMPM366 ペリフェラルドライバの全てのソースコードが格納されています。

/Libraries/TX03_Periph_Driver/inc
TMPM366 ペリフェラルドライバのヘッダファイルが格納されています。

/Libraries/TX03_Periph_Driver/src
TMPM366 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
TMPM366 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
TMPM366 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
TMPM366 ペリフェラルドライバの使用例が格納されています。

/Utilities/TMPM366-SK
M366-SK ボードのハードウェアリソース用の設定ファイル、およびドライバファイル (例： led, key) が格納されています。

3. ADC

3.1 概要

本デバイスは、12 ビット逐次変換方式アナログ/デジタルコンバータ(AD コンバータ)を内蔵しており、12チャネルのアナログ入力を持っています。

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、最優先 AD 変換の起動
　ソフトウェアによる起動
　外部トリガ入力(ADTRG)によるハードウェア起動
　16 ビットタイマによる起動
(2) 通常 AD 変換機能の動作モード
　チャネル固定シングル変換モード
　チャネルスキャンシングル変換モード
　チャネル固定リピート変換モード
　チャネルスキャンリピート変換モード
(3) 最優先 AD 変換機能の動作モード
　チャネル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、割り込み発生機能
(5) 通常 AD 変換機能、最優先 AD 変換機能は以下のステータスフラグを持っています。
　AD 変換結果格納フラグ、オーバーランフラグ、AD 変換終了フラグ、AD 変換ビジーフラグ
(6) AD 監視機能
　AD 変換結果とあらかじめ設定した値とを比較し、特定の条件で割り込みを発生
(7) AD 変換クロックを 1/fc～1/16fc まで制御可能
(8) AD 変換終了時、2 種類の DMA リクエストをサポート
(9) スタンバイモードをサポート
(10)出力スイッチングモニタ機能

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

全ドライバ API は、アプリで使用する API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver\src\tmpm366_adc.c
/Libraries/TX03_Periph_Driver\inc\tmpm366_adc.h

3.2 API 関数

3.2.1 関数一覧

- void ADC_SWReset(void)
- void ADC_SetClk(uint32_t ***Sample_HoldTime***, uint32_t ***Prescaler_Output***)
- void ADC_Start(void)
- void ADC_SetScanMode(FunctionalState ***NewState***)
- void ADC_SetRepeatMode(FunctionalState ***NewState***)
- void ADC_SetINTMode(uint8_t ***INTMode***)
- void ADC_SetInputChannel(uint8_t ***InputChannel***)
- void ADC_SetScanChannel(uint8_t ***StartChannel***, uint8_t ***Range***)
- void ADC_SetVrefCut(uint8_t ***VrefCtrl***)
- void ADC_SetIdleMode(FunctionalState ***NewState***)
- void ADC_SetVref(FunctionalState ***NewState***)

- void ADC_SetInputChannelTop(uint8_t ***TopInputChannel***)
- void ADC_StartTopConvert(void)
- void ADC_SetMonitor(ADC_CMPCRx ***ADCMPx***, FunctionalState ***NewState***)
- void ADC_ConfigMonitor(ADC_CMPCRx ***ADCMPx***, ADC_MonitorTypeDef* ***Monitor***)
- void ADC_SetHWTrg(uint8_t ***HwSource***, FunctionalState ***NewState***)
- void ADC_SetHWTrgTop(uint8_t ***HwSource***, FunctionalState ***NewState***)
- ADC_State ADC_GetConvertState(void)
- ADC_Result ADC_GetConvertResult (uint8_t ***ADREGx***)
- void ADC_SetClkSupply(FunctionalState ***NewState***)
- void ADC_SetDMAReq(uint8_t ***DMAReq***, FunctionalState ***NewState***)

3.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。

1) AD 変換設定:
   ADC_SetClk(), ADC_SetScanMode(), ADC_SetRepeatMode(), ADC_SetINTMode(), ADC_SetInputChannel(), ADC_SetScanChannel(), ADC_SetVref(), ADC_SetInputChannelTop(), ADC_SetMonitor(), ADC_ConfigMonitor(), ADC_SetHWTrg(), ADC_SetHWTrgTop()
2) AD 変換の許可/禁止と開始:
   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(), ADC_SetClkSupply(), ADC_SetDMAReq()

3.2.3 関数仕様

3.2.3.1 ADC_SWReset

ADC 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(void)

引数:
なし

機能:
ADC をソフトウェアリセットします。

補足:
ADxCLK<ADCLK>を除くレジスタは、すべて初期化されます。
ソフトウェアリセットを行う場合、初期化に 3μs の時間が必要となります。

戻り値:
なし

3.2.3.2 ADC_SetClk

AD 変換サンプルホールド時間とプリスケーラ出力(SCLK)の設定

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

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

Prescaler_Output: 以下から ADC プリスケーラ出力(ADCLK)を選択します。
- **ADC_FC_DIVIDE_LEVEL_1:** fc
- **ADC_FC_DIVIDE_LEVEL_2:** fc / 2
- **ADC_FC_DIVIDE_LEVEL_4:** fc / 4
- **ADC_FC_DIVIDE_LEVEL_8:** fc / 8

機能:
Sample_HoldTime で ADC サンプルホールド時間を設定し、***Prescaler_Output*** でプリスケーラ出力を設定します。

補足:
AD変換中は、この関数を使わないでください。またAD変換状態を確認するための**ADC_GetConvertState()**がBUSYでない場合、この関数をコールすることができます。

サンプルホールド時間と変換時間例は下記のようになります。

Prescaler_Output	***Sample_HoldTime***	**Conversion time**		
		fc=32MHz	fc=40MHz	fc=54MHz
ADC_FC_DIVIDE_LEVEL_1 (fc)	ADC_CONVERSION_CLK_10	1.25 µs	1.00 µs	-
	ADC_CONVERSION_CLK_20	1.56 µs	1.25 µs	-
	ADC_CONVERSION_CLK_30	1.88 µs	1.50 µs	-
	ADC_CONVERSION_CLK_40	2.19 µs	1.75 µs	-
	ADC_CONVERSION_CLK_80	3.44 µs	2.75 µs	-
ADC_FC_DIVIDE_LEVEL_2 (fc / 2)	ADC_CONVERSION_CLK_10	2.50 µs	2.00 µs	1.48 µs
	ADC_CONVERSION_CLK_20	3.13 µs	2.50 µs	1.85 µs
	ADC_CONVERSION_CLK_30	3.75 µs	3.00 µs	2.22 µs
	ADC_CONVERSION_CLK_40	4.38 µs	3.50 µs	2.59 µs
	ADC_CONVERSION_CLK_80	6.88 µs	5.50 µs	4.07 µs
ADC_FC_DIVIDE_LEVEL_4 (fc / 4)	ADC_CONVERSION_CLK_10	5.00 µs	4.00 µs	2.96 µs
	ADC_CONVERSION_CLK_20	6.25 µs	5.00 µs	3.70 µs
	ADC_CONVERSION_CLK_30	7.50 µs	6.00 µs	4.44 µs
	ADC_CONVERSION_CLK_40	8.75 µs	7.00 µs	5.19 µs
	ADC_CONVERSION_CLK_80	-	-	8.15 µs
ADC_FC_DIVIDE_LEVEL_8 (fc / 8)	ADC_CONVERSION_CLK_10	10.0 µs	8.00 µs	5.93 µs
	ADC_CONVERSION_CLK_20	-	10.0 µs	7.41 µs
	ADC_CONVERSION_CLK_30	-	-	8.89 µs
	ADC_CONVERSION_CLK_40	-	-	-
	ADC_CONVERSION_CLK_80	-	-	-

上記一覧で"-" で示される部分の設定は禁止されています。ADCLK は1µs から 10µs 間の値で設定してください。

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(void)

引数:
なし

機能:
AD 変換を開始します。

補足:
この関数をコールする前に、以下のいずれかのモードを選択してください:
チャネル固定シングル変換モード
チャネルスキャンシングル変換モード
チャネル固定リピート変換モード
チャネルスキャンリピート変換モード
詳細は、**ADC_SetScanMode(), ADC_SetRepeatMode(), ADC_SetInputChannel(), ADC_SetScanChannel()** を参照してください。

AD 変換をスタートさせる場合、**ADC_SetVref (ENABLE)**をコールして Vref を有効にしてください。なお、Vref 有効後、3 µs の安定時間が必要です。その後、**ADC_Start()**をコールしてください。

戻り値:
なし

3.2.3.4 ADC_SetScanMode

スキャンモードの設定

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

引数:
NewState: 以下から、スキャンモードを設定します。
- **ENABLE:** チャネルスキャン
- **DISABLE:** チャネル固定

機能:
AD 変換スキャンモードを設定します。

戻り値:

なし

3.2.3.5 ADC_SetRepeatMode

リピートモードの設定

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

引数:
NewState: 以下から、リピートモードを設定します。
- **ENABLE:** リピート変換
- **DISABLE:** シングル変換

機能:
リピートモードを設定します。

戻り値:
なし

3.2.3.6 ADC_SetINTMode

チャネル固定リピート変換モード時の割り込みタイミングの設定

関数のプロトタイプ宣言:
void
ADC_SetINTMode(uint8_t ***INTMode***)

引数:
INTMode: 以下から、割り込みタイミングを選択します。
- **ADC_INT_SINGLE:** 1 回毎、割り込み発生
- **ADC_INT_CONVERSION_2:** 2 回毎、割り込み発生
- **ADC_INT_CONVERSION_3:** 3 回毎、割り込み発生
- **ADC_INT_CONVERSION_4:** 4 回毎、割り込み発生
- **ADC_INT_CONVERSION_5:** 5 回毎、割り込み発生
- **ADC_INT_CONVERSION_6:** 6 回毎、割り込み発生
- **ADC_INT_CONVERSION_7:** 7 回毎、割り込み発生
- **ADC_INT_CONVERSION_8:** 8 回毎、割り込み発生

機能:
チャネル固定リピート変換モード時の割り込みタイミングを設定します。

補足:
この関数は、チャネル固定リピート変換モード時のみ有効です。
以下は、チャネル固定リピート変換モードの例です:
1. **ADC_SetScanMode(DISABLE)**.
2. **ADC_SetRepeatMode(ENABLE)**.

戻り値:
なし

3.2.3.7 ADC_SetInputChannel

アナログ入力チャネルの選択

関数のプロトタイプ宣言:
void
ADC_SetInputChannel(uint8_t ***InputChannel***)

引数:
InputChannel: 以下から、いずれか 1 つのアナログ入力チャネルを使用します。
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

機能:
アナログ入力チャネルを選択します。

補足:
通常変換入力の場合 1 チャネルのみ選択できます。

戻り値:
なし

3.2.3.8 ADC_SetScanChannel

スキャンチャネルの設定

関数のプロトタイプ宣言:
void
ADC_SetScanChannel (uint8_t ***StartChannel***, uint8_t ***Range***)

引数:
StartChannel: 以下から、チャネルスキャンの先頭チャネルを設定します。
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

Range: チャネルスキャンの範囲を 1～12 のいずれか選択できます。

機能:
StartChannel にてチャネルスキャンの先頭チャネルの設定を、***Range*** にてチャネルスキャンの範囲を設定します。

補足:
設定可能なチャネルスキャンの範囲を下表に示します。

StartChannel	***Range***
ADC_AN_00	1 ~ 12
ADC_AN_01	1 ~ 11
ADC_AN_02	1 ~ 10
ADC_AN_03	1 ~ 9
ADC_AN_04	1 ~ 8
ADC_AN_05	1 ~ 7
ADC_AN_06	1 ~ 6

ADC_AN_07	1 ~ 5
ADC_AN_08	1 ~ 4
ADC_AN_09	1 ~ 3
ADC_AN_10	1 ~ 2
ADC_AN_11	1

上記以外の場合、**ADC_Start()**をコールしても 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(uint8_t ***VrefCtrl***)

引数:
VrefCtrl: AVREFH-AVREFL 間のリファレンス電流を制御します。
- **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:
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

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

引数:
NewState: 以下から、IDLE モード時の ADC 動作を選択します。
- **ENABLE**: 動作
- **DISABLE**: 停止

機能:
IDLE モード時の ADC 動作を制御します。
IDLE モードに移行する前にこの関数をコールする必要があります。

戻り値:
なし

3.2.3.11 ADC_SetVref

Vref 回路の on/off 制御

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

引数:
NewState: 以下から、Vref 回路の状態を選択します。
- **ENABLE**: ON
- **DISABLE**: OFF

機能:
Vref 回路の on/off を制御します。

補足:
低消費電力モードに移行する前に、**ADC_SetVref(DISABLE**) をコールしてください。

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(uint8_t ***TopInputChannel***)

引数:
TopInputChannel: 以下から、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

機能:
最優先 AD 変換入力チャネルを設定します。

補足:
最優先 AD 変換入力を 1 チャネルのみ選択できます。

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(void

引数:
なし

機能:
最優先 AD 変換を開始します。

補足:
この関数をコールする前 **ADC_SetInputChannelTop()**をコールしてください。

戻り値:
なし

3.2.3.14 ADC_SetMonitor

AD 監視機能の許可/禁止

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

引数:
ADCMPx: 以下から、監視機能設定レジスタを選択します。
- **ADC_CMPCR_0:** ADCMPCR0
- **ADC_CMPCR_1:** ADCMPCR1

NewState: 以下から、監視機能を設定します。
- **ENABLE**: 許可(条件成立で AD 監視割り込みを発生します)
- **DISABLE**: 禁止(大小判定カウント数はクリア)

機能:
本デバイスは、2 つの AD 監視機能を持ち、それぞれ設定レジスタで制御します。
ADCMPx 設定で AD 監視レジスタを選択し、NewState で許可/禁止を設定します。

戻り値:
なし

3.2.3.15 ADC_ConfigMonitor

AD 監視機能の設定

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

引数:
ADCMPx: 以下から、AD 変換レジスタを選択します。
- **ADC_CMPCR_0:** ADCMPCR0
- **ADC_CMPCR_1:** ADCMPCR1

Monitor: AD 監視機能関する構造体で、大小判定カウント数、判定カウント条件、判定条件、比較対象のアナログ入力チャネルが含まれます。詳細は"データ構成"のADC_MonitorTypeDef を参照してください。

機能:
本デバイスは、2 つの AD 監視機能を持ち、それぞれ設定レジスタで制御します。
ADCMPx 設定で AD 監視レジスタを選択し、***Monitor*** で監視機能を設定します。

補足: この関数をコールする前に ADC 監視機能を禁止してください。

戻り値:
なし

3.2.3.16 ADC_SetHWTrg

通常 AD 変換を開始するためのハードウェア起動要因の選択

関数のプロトタイプ宣言:
void
ADC_SetHWTrg(uint8_t ***HwSource***,
FunctionalState ***NewState***)

引数:
HwSource: 以下から、通常 AD 変換のハードウェア起動要因を選択します。
- **ADC_EXT_TRG:** ADTRG 端子で起動することが可能です。
- **ADC_MATCH_TB5RG0:** 16 ビットタイマ/イベントカウンタのコンペアレジスタ 0 一致割り込みで起動することが可能です。(TB5RG0)

NewState: 以下から、ハードウェア起因による通常 AD 変換開始の許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
HwSource の設定により通常 AD 変換のハードウェア起動要因を設定し、***NewState*** により通常 AD 変換のハードウェア起動の許可/禁止を選択します。
この関数は TB5 の設定にも関連しています。

補足:
最優先 AD 変換のハードウェア起動要因に使用する場合、外部トリガを通常 AD 変換のハードウェア要因起動に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 変換を開始するためのハードウェア起動要因の選択

関数のプロトタイプ宣言:
void

ADC_SetHWTrgTop(uint8_t ***HwSource***,
FunctionalState ***NewState***)

引数:
HwSource: 以下から、最優先 AD 変換のハードウェア起動要因を選択します。
- **ADC_EXT_TRG:** ADTRG 端子で起動することが可能です。
- **ADC_MATCH_TB4RG0:** 16 ビットタイマ/イベントカウンタのコンペアレジスタ 0 一致割り込みで起動することが可能です。(TB4RG0)

NewState: 以下から、ハードウェア起因による通常 AD 変換開始の許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
HwSource の設定により最優先 AD 変換のハードウェア起動要因を設定し、***NewState*** により最優先 AD 変換のハードウェア起動の許可/禁止を選択します。
この関数は TB4 の設定にも関連しています。

補足:
最優先 AD 変換のハードウェア起動要因に使用する場合、外部トリガを通常 AD 変換のハードウェア要因起動に使用することはできません。

戻り値:
なし

3.2.3.18 ADC_GetConvertState

AD 変換終了フラグの取得(通常と最優先)

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

引数:
なし

機能:
AD 変換終了フラグ (通常と最優先の両方)を取得します。この関数は、AD 変換が終了したかどうかを確認するために使います。

戻り値:
AD 変換状態:
NormalComplete (Bit 1) : 通常 AD 変換終了
TopComplete (Bit 3) : 最優先 AD 変換終了

3.2.3.19 ADC_GetConvertResult

AD 変換結果の取得

関数のプロトタイプ宣言:

ADC_Result
ADC_GetConvertResult(uint8_t ***ADREGx***)

引数:
ADREGx: 以下から、ADC 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_SP

機能:
AD 変換結果格納フラグ、オーバーランフラグ、変換結果を取得します。

補足:
変換結果が格納されると AD 変換格納フラグ ***ADREGx*** が **DONE** になります。本関数によって変換結果が読み出されると、AD 変換結果格納フラグ ***ADREGx*** がクリアされます。

変換結果格納レジスタ(ADREGx)の値が読み出される前に変換結果が上書きされた場合、AD 変換結果格納フラグ ***ADREGx*** に **ADC_OVERRUN** がセットされます。本関数によってオーバーランフラグが読み出されるとオーバーランフラグがクリアされます。

アナログチャネル入力と 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変換モード	
チャネル	格納レジスタ
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

チャネル固定リピート変換モード	
割り込み発生タイミング	格納レジスタ
Interrupt by each time AD/C	ADC_REG_00
Interrupt by each time 2 AD/C	ADC_REG_00 to ADC_REG_01
Interrupt by each time 3 AD/C	ADC_REG_00 to ADC_REG_02
Interrupt by each time 4 AD/C	ADC_REG_00 to ADC_REG_03
Interrupt by each time 5 AD/C	ADC_REG_00 to ADC_REG_04
Interrupt by each time 6 AD/C	ADC_REG_00 to ADC_REG_05
Interrupt by each time 7 AD/C	ADC_REG_00 to ADC_REG_06
Interrupt by each time 8 AD/C	ADC_REG_00 to ADC_REG_07

チャネルスキャンシングル変換モード / リピート変換モード		
スタートチャネル	スキャンチャネル幅	格納レジスタ
ADC_AN_00	12 channels	ADC_REG_00 to ADC_REG_11

ADC_AN_01	11 channels	ADC_REG_01 to ADC_REG_11
ADC_AN_02	10 channels	ADC_REG_02 to ADC_REG_11
ADC_AN_03	9 channels	ADC_REG_03 to ADC_REG_11
ADC_AN_04	8 channels	ADC_REG_04 to ADC_REG_11
ADC_AN_05	7 channels	ADC_REG_05 to ADC_REG_11
ADC_AN_06	6 channels	ADC_REG_06 to ADC_REG_11
ADC_AN_07	5 channels	ADC_REG_07 to ADC_REG_11
ADC_AN_08	4 channels	ADC_REG_08 to ADC_REG_11
ADC_AN_09	3 channels	ADC_REG_09 to ADC_REG_11
ADC_AN_10	2 channels	ADC_REG_10 to ADC_REG_11
ADC_AN_11	1 channels	ADC_REG_11 to ADC_REG_11

The AD 変換モードの詳細は、関連 API を参照ください。
最優先 AD 変換結果は、ADC_REG_SP に格納されます。

戻り値:
AD 変換結果:
ADResult (Bit 0 ～ Bit 11) : AD 変換結果が格納されます
Stored (Bit 12) : AD 変換結果格納フラグ
OverRun (Bit 13) : オーバーランフラグ
OutputSwitching (Bit 14) : AIN 兼用ポートの出力スイッチングフラグ

3.2.3.20 ADC_SetClkSupply

ADC クロック選択

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

引数:
NewState: 以下から、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

3.2.3.21 ADC_SetDMAReq

通常 AD 変換、最優先 AD 変換の各 DMA 起動要因の設定

関数のプロトタイプ宣言:
void
ADC_SetDMAReq(uint8_t ***DMAReq***,
FunctionalState ***NewState***)

引数:
DMAReq: 以下から AD 変換の種類を選択します。

- **ADC_DMA_REQ_NORMAL**: 通常 AD 変換
- **ADC_DMA_REQ_TOP**: 最優先 AD 変換

NewState: 以下から DMA 起動の許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
通常 AD 変換、最優先 AD 変換の各 DMA 起動要因を選択します。

戻り値:
なし

3.2.4 データ構造

3.2.4.1 ADC_MonitorTypeDef

メンバ:

uint8_t
CmpChannel 以下から、比較対象のアナログ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.

uint32_t
CmpCnt : 大小判定カウント数を選択します。(**1** ~ **16**)

ADC_CmpCondition
Condition : 以下から、判定条件を選択します。
- **ADC_LARGER_THAN_CMP_REG:** 比較レジスタ(ADCMPn (n=0/1))より AD 変換結果が大
- **ADC_SMALLER_THAN_CMP_REG:** 比較レジスタ(ADCMPn (n=0/1))より AD 変換結果が小

ADC_CmpCntMode
CntMode : 以下から、判定カウント条件を選択します。
- **ADC_SEQUENCE_CMP_MODE:** 連続方式
- **ADC_CUMULATION_CMP_MODE:** 蓄積方式

uint32_t
CmpValue : AD 変換結果比較値を設定します。(**0** ~ **4095**)

3.2.4.2 ADC_State

メンバ:
uint32_t
All : すべての AD 変換状態

ビットフィールド:
uint32_t

Reserved0 (Bit 0) : 未使用
uint32_t
NormalComplete (Bit 1) : 通常 AD 変換終了フラグ
uint32_t
Reserved1 (Bit 2) : 未使用
uint32_t
TopComplete (Bit 3) : 最優先 AD 変換終了フラグ
uint32_t
Reserved2 (Bit 4 ~ Bit 31) : 未使用

3.2.4.3 ADC_Result

メンバ:
uint32_t
All : すべての AD 変換結果

ビットフィールド:
uint32_t
ADResult (Bit 0 ~ Bit 11) : AD 変換結果の値
uint32_t
Stored (Bit 12) : AD 結果終了フラグ
uint32_t
OverRun (Bit 13) : オーバーランフラグ
uint32_t
OutputSwitching (Bit 14) : AIN 兼用ポートの出力スイッチングフラグ
uint32_t
Reserved (Bit 15 ~ Bit 31) : 未使用

4. CG

4.1 概要

本 CG API は以下の機能を提供します。
- 高速発振器、PLL(逓倍回路)の設定
- クロックギア、プリスケーラクロック、PLL、発振器の設定
- ウォームアップタイマの設定と結果の読み出し
- 低消費電力モードの設定
- 動作モードの変更 (ノーマルモード、低消費電力モード)
- 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/TX03_Periph_Driver\src\tmpm366_cg.c
/Libraries/TX03_Periph_Driver\inc\tmpm366_cg.h

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

fosc : 内部発振回路で生成されるクロック、X1、X2 端子より入力されるクロック
fPLL : PLL により逓倍されたクロック
fc : CGPLLSEL<PLL0SEL>で選択されたクロック(高速クロック)
fgear : CGSYSCR<GEAR[2:0]>で選択されたクロック
fsys : fgear と同等のクロック
fperiph : CGSYSCR<FPSEL>で選択されたクロック
ΦT0 : CGSYSCR<PRCK[2:0]>で選択されたクロック (プリスケーラクロック)

4.2 API 関数

4.2.1 関数一覧

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

- FunctionalState CG_GetPLLState(void)
- Result CG_SetFosc(CG_FoscSrc Source, FunctionalState ***NewState***)
- void CG_SetFoscSrc(CG_FoscSrc ***Source***)
- CG_FoscSrc CG_GetFoscSrc(void)
- FunctionalState CG_GetFoscState(CG_FoscSrc ***Source***)
- void CG_SetSTBYMode(CG_STBYMode ***Mode***)
- CG_STBYMode CG_GetSTBYMode(void)
- void CG_SetPinStateInStop1Mode(FunctionalState ***NewState***)
- void CG_SetPortKeepInStop2Mode(FunctionalState ***NewState***)
- FunctionalState CG_GetPortKeepInStop2Mode(void)
- FunctionalState CG_GetPinStateInStop1Mode(void)
- Result CG_SetFcSrc(CG_FcSrc ***Source***)
- CG_FcSrc CG_GetFcSrc(void)
- void CG_SetUSBSrcClk(CG_USBSrc ***Source***)
- CG_USBSrc CG_GetUSBSrcClk(void)
- void CG_SetUSBClkState(FunctionalState ***NewState***)
- FunctionalState CG_GetUSBClkState(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)
- CG_ResetFlag CG_GetResetFlag(void)

4.2.2 関数の種類

上記関数は以下の 3 種類に分けられます。

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_SetUSBSrcClk(), CG_GetUSBSrcClk(), CG_SetUSBClkState(), CG_GetUSBClkState(), CG_SetProtectCtrl()
2) スタンバイモードの設定:
   CG_SetSTBYMode(), CG_GetSTBYMode( ), CG_SetPinStateInStop1Mode(), CG_GetPinStateInStop1Mode(), CG_SetPortKeepInStop2Mode(), CG_GetPortKeepInStop2Mode()
3) 割り込みの設定:
   CG_SetSTBYReleaseINTSrc(), CG_GetSTBYReleaseINTState(), CG_ClearINTReq(), CG_GetNMIFlag(), CG_GetResetFlag()

4.2.3 関数仕様

4.2.3.1 CG_SetFgearLevel

fgear,fc 間の分周レベル設定

関数のプロトタイプ宣言:

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

引数:
DivideFgearFromFc: 以下から、fgear,fc 間の分周レベルを選択します。
- **CG_DIVIDE_1**: fgear = fc
- **CG_DIVIDE_2**: fgear = fc/2
- **CG_DIVIDE_4**: fgear = fc/4
- **CG_DIVIDE_8**: fgear = fc/8
- **CG_DIVIDE_16**: fgear = fc/16

機能：
fgear,fc 間の分周レベルを設定します。

戻り値:
なし

4.2.3.2 CG_GetFgearLevel

fgear,fc 間の分周レベルの取得

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

引数:
なし。

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

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

4.2.3.3 CG_SetPhiT0Src

PhiT0(fperiph)ソースの設定

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

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

機能:

PhiT0 (ΦT0) ソースを選択します。

戻り値:
なし

4.2.3.4 CG_GetPhiT0Src

PhiT0 (ΦT0) ソースの取得

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

引数:
なし。

機能:
PhiT0 (ΦT0) ソースを取得します。

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

4.2.3.5 CG_SetPhiT0Level

PhiT0 (ΦT0) と fc 間の分周レベルの設定

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

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

4.2.3.6 CG_GetPhiT0Level

ΦT0 ,fc 間の分周レベルの取得

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

引数:
なし。

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

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

4.2.3.7 CG_SetSCOUTSrc

SCOUT 出力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 出力のソースクロックを選択します。
- **CG_SCOUT_SRC_HALF_FSYS**: fsys/2 に設定
- **CG_SCOUT_SRC_FSYS**: fsys に設定
- **CG_SCOUT_SRC_PHIT0**: ΦT0 に設定

機能:
SCOUT 出力のソースクロックを設定します。

戻り値:
なし

4.2.3.8 CG_GetSCOUTSrc

SCOUT 出力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 出力ソースクロック設定を取得します。

戻り値:
SCOUT 出力のソースクロック:
- **CG_SCOUT_SRC_HALF_FSYS**: fsys/2 に設定
- **CG_SCOUT_SRC_FSYS**: fsys に設定
- **CG_SCOUT_SRC_PHIT0**: ΦT0 に設定
- **CG_SCOUT_SRC_UNKNOWN**: 無効データ

4.2.3.9 CG_SetWarmUpTime

ウォーミングアップ時間の設定

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、ウォーミング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**: 内部高速発振器を選択
- **CG_WARM_UP_SRC_OSC_EXT**: 外部高速発振器を選択

Time: ウォーミングアップタイマーのカウント数を選択します。最大値は 0xFFFF です。

機能:
ウォーミングアップ時間とウォーミングアップカウンタを設定します。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) / (ウォームアップクロック周期)

高速発振子 8MHz 使用時、ウォーミングアップ時間 5ms を設定する場合のウォーミングアップサイクル数は以下になります:

(ウォーミングアップ時間) / (ウォームアップクロック周期) = 5ms / (1/8MHz) = 4000cycle = 0x9C40
従って、***Time*** = 0x9C40 となります。

戻り値:
なし

4.2.3.10 CG_StartWarmUp

ウォーミングアップ開始

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

引数:
なし。

機能:
ウォーミングアップを開始します。

戻り値:
なし

4.2.3.11 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, 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

4.2.3.12 CG_SetFPLLValue

PLL (fsys 用)の逓倍数を設定。

関数のプロトタイプ宣言:
Result
CG_SetFPLLValue(CG_FpllValue ***NewValue***)

引数:
NewValue:
- **CG_FPLL_MULTIPLY_8**: 8 逓倍
- **CG_FPLL_MULTIPLY_6**: 6 逓倍

機能:
PLL (fsys 用)の逓倍数を設定します。

戻り値:
SUCCESS: 成功
ERROR: 失敗

4.2.3.13 CG_GetFPLLValue

PLL 逓倍値の取得

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

引数:
なし

機能:
PLL 逓倍値を取得します。
レジスタ値が “Reserved” の場合、本 API の戻り値は **CG_FPLL_MULTIPLY_UNKNOWN** です。

戻り値:
PLL 逓倍値:
CG_FPLL_MULTIPLY_8: 8 逓倍値
CG_FPLL_MULTIPLY_6: 6 逓倍値
CG_FPLL_MULTIPLY_UNKNOWN: 無効値

4.2.3.14 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 回路を使用する
- **DISABLE**: PLL 回路を使用しない

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

4.2.3.15 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 回路の状態
ENABLE: PLL 有効
DISABLE: PLL 無効

4.2.3.16 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: 以下から、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: 以下から、高速発振器の有効/無効を設定します。
- **ENABLE**: 有効
- **DISABLE**: 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

4.2.3.17 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

4.2.3.18 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

4.2.3.19 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: 以下から、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: fosc が有効

DISABLE: fosc が無効

4.2.3.20 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: 以下から、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

4.2.3.21 CG_GetSTBYMode

スタンバイモードの取得

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

引数:
なし。

機能:
スタンバイモードの設定状態を取得します。
“Reserved”の場合、“**CG_STBY_MODE_UNKNOWN**” を返却します。

戻り値:
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

4.2.3.22 CG_SetPinStateInStop1Mode

STOP1 モード中の端子状態の設定

関数のプロトタイプ宣言:
void

CG_SetPinStateInStop1Mode(FunctionalState ***NewState***)

引数:
NewState:
- **DISABLE:** STOP1 モード中端子をドライブしません
- **ENABLE:** STOP1 モード中端子をドライブします

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

機能:
STOP1 モード時の端子状態を設定します。

戻り値:
なし

4.2.3.23 CG_GetPinStateInStop1Mode

STOP1 モード中の端子状態の取得。

関数のプロトタイプ宣言:
FunctionalState
CG_GetPinStateInStop1Mode(void)

引数:
なし。

機能:
STOP1 モード中の端子状態を取得します。

戻り値:
DISABLE: STOP1 モード中端子をドライブしません
ENABLE: STOP1 モード中端子をドライブします

4.2.3.24 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:** DISABLE->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”を参照してくだいさい。

機能:
STOP2 モード中の I/O 制御信号保持の有効/無効を切り替えます。

戻り値:
なし

4.2.3.25 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 モード時の端子状態:
DISABLE: ポートによる制御
ENABLE: DISABLE->ENABLE 設定時の状態を保持

4.2.3.26 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 を使用
- **CG_FC_SRC_ QUARTER_FPLL**: fpll/4 を使用

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

4.2.3.27 CG_GetFcSrc

fc ソースの取得

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

引数:

なし

機能:
fc ソースを取得します。

戻り値:
fc のソース:
CG_FC_SRC_FOSC : fosc
CG_FC_SRC_ QUARTER_FPLL: fpll/4

4.2.3.28 CG_SetUSBSrcClk

USB ソースクロック選択

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

引数:
Source: 以下から USB クロックソースを選択します。
- **CG_USB_SRC_CLK_PLL**: PLL クロック(fpllI)
- **CG_USB_SRC_CLK_EXT**: 外部入力クロック(EHCLKIN)

機能:
USB ソースクロックを選択します。

戻り値:
なし

4.2.3.29 CG_GetUSBSrcClk

USB ソースクロックの設定状態取得

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

引数:
なし

機能:
USB ソースクロックの設定状態を取得します。

戻り値:
USB ソースクロックの設定状態:
CG_USB_SRC_CLK_PLL: PLL クロック(fpllI)
CG_USB_SRC_CLK_EXT: 外部入力クロック(EHCLKIN)

4.2.3.30 CG_SetUSBClkState

USB ソースクロックの設定

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

引数:
NewState
- **DISABLE:** 停止(OFF)
- **ENABLE:** 設定(ON)

機能:
USB ソースクロックを設定します。

戻り値:
なし

4.2.3.31 CG_GetUSBClkState

USB ソースクロック設定状態の取得

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

引数:
なし。

機能:
USB ソースクロック設定状態を取得します。

戻り値:
USB ソースクロックの設定状態:
DISABLE: 停止(OFF)
ENABLE: 設定(ON)

4.2.3.32 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

4.2.3.33 CG_SetSTBYReleaseINTSrc

スタンバイモードの解除割り込みソースの設定

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

引数:
INTSource: 以下から、スタンバイモードの解除割り込みソースを選択します。
- **CG_INT_SRC_0** : INT0
- **CG_INT_SRC_1** : INT1
- **CG_INT_SRC_2** : INT2
- **CG_INT_SRC_3** : INT3
- **CG_INT_SRC_4** : INT4
- **CG_INT_SRC_5** : INT5
- **CG_INT_SRC_6** : INT6
- **CG_INT_SRC_7** : INT7
- **CG_INT_SRC_8** : INT8
- **CG_INT_SRC_9** : INT9
- **CG_INT_SRC_USB_PON** : USB Poewe ON(V-Bus Connect)検出割り込み
- **CG_INT_SRC_USB_WKUP** : USB Wake-up 割り込み

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

NewState: 以下から、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

4.2.3.34 CG_GetSTBYReleaseINTState

スタンバイモードの解除割り込みソースのアクティブ状態の取得

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

引数:
INTSource: 以下から、解除割り込みソースを選択します。
CG_INT_SRC_0, CG_INT_SRC_1, CG_INT_SRC_2, CG_INT_SRC_3,
CG_INT_SRC_4, CG_INT_SRC_5, CG_INT_SRC_6, CG_INT_SRC_7,
CG_INT_SRC_8, CG_INT_SRC_9,
CG_INT_SRC_USB_PON, CG_INT_SRC_USB_WKUP.

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_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
CG_INT_ACTIVE_STATE_INVALID: 無効な値

4.2.3.35 CG_ClearINTReq

スタンバイ解除割り込み要求のクリア

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

引数:
INTSource: 以下から、解除割り込みソースを選択します。
CG_INT_SRC_0, CG_INT_SRC_1, CG_INT_SRC_2, CG_INT_SRC_3,
CG_INT_SRC_4, CG_INT_SRC_5, CG_INT_SRC_6, CG_INT_SRC_7,
CG_INT_SRC_8, CG_INT_SRC_9,
CG_INT_SRC_USB_PON, CG_INT_SRC_USB_WKUP.

機能:
スタンバイ解除割り込み要求をクリアします。

戻り値:
なし

4.2.3.36 CG_GetNMIFlag

NMI 起動要因フラグの取得

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

引数:
なし

機能:

NMI 起動要因フラグを取得します。

戻り値:
NMI 起動要因:
WDT (Bit 0) :WDT による NMI 発生
NMIPin(Bit 1):NMI 端子 による NMI 発生

4.2.3.37 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 (Bit 0) リセット端子によるリセット
Reserved(Bit1) 未使用
WDTReset (Bit 2) WDT リセット
STOP2Reset(Bit3) STOP2 モード解除
DebugReset (Bit 4) SYSRESETREQ リセット

4.2.4 データ構造

4.2.4.1 CG_NMIFactor

メンバ:
uint32_t
All すべての NMI 要因

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

4.2.4.2 CG_ResetFlag

メンバ:
uint32_t
All すべてのリセット要因

ビットフィールド:
uint32_t

ResetPin(Bit 0)　　RESET 端子によるリセット
uint32_t
Reserved(Bit 1)　　未使用
uint32_t
WDTReset(Bit 2)　　WDT によるリセット
uint32_t
STOP2Reset(Bit 3)　STOP2 モード解除
uint32_t
DebugReset(Bit 4)　<SYSResetREQ>によるリセット

5. DMAC

5.1 概要

本デバイスは、DMA 要求選択レジスタにより制御される 2 ユニットの DMA コントローラ (UNITA) を内蔵しています。ユニット A、B は、 4 つの転送タイプのどれかで動作します。4 つの転送タイプは、メモリ-メモリ、メモリ-周辺回路、周辺回路–メモリ、周辺回路-周辺回路です。ユニット A、B は 2 チャネルの DMAC を内蔵し、DMA チャネル 0 は DMA チャネル 1 より優先度が高くなります。

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

全ドライバ API は、アプリ使用の API 定義を格納する以下のファイルで構成されています。
\Libraries\TX03_Periph_Driver\src\tmpm366_dmac.c
\Libraries/TX03_Periph_Driver\inc\tmpm366_dmac.h

5.2 API 関数

5.2.1 関数一覧

- void DMAC_Enable(TSB_DMAC_TypeDef * ***DMACx***);
- void DMAC_Disable(TSB_DMAC_TypeDef * ***DMACx***);
- DMAC_INTReq DMAC_GetINTReq(TSB_DMAC_TypeDef * ***DMACx***);
- DMAC_TxINTReq DMAC_GetTxINTReq(TSB_DMAC_TypeDef * ***DMACx***, DMAC_Channel ***Chx***);
- void DMAC_ClearTxINTReq(TSB_DMAC_TypeDef * ***DMACx***, DMAC_Channel ***Chx***, DMAC_INTSrc ***INTSource***);
- DMAC_TxINTReq DMAC_GetRawTxINTReq(TSB_DMAC_TypeDef * ***DMACx***, DMAC_Channel ***Chx***);
- WorkState DMAC_GetChannelTxState(TSB_DMAC_TypeDef * ***DMACx***, DMAC_Channel ***Chx***);
- void DMACA_SetSWBurstReq(DMACA_ReqNum ***BurstReq***);
- void DMACB_SetSWBurstReq(DMACB_ReqNum ***BurstReq***);
- DMAC_BurstReqState DMAC_GetSWBurstReqState(TSB_DMAC_TypeDef * ***DMACx***);
- void DMACA_SetSWSingleReq(DMACA_ReqNum ***SingleReq***);
- void DMACB_SetSWSingleReq(DMACB_ReqNum ***SingleReq***);
- DMAC_SingleReqState DMAC_GetSWSingleReqState(TSB_DMAC_TypeDef * ***DMACx***);
- void DMAC_SetLinkedList(TSB_DMAC_TypeDef * ***DMACx,*** DMAC_Channel ***Chx***, uint32_t ***LinkedAddr***);
- WorkState DMAC_GetFIFOState(TSB_DMAC_TypeDef * ***DMACx,*** DMAC_Channel ***Chx***);
- void DMAC_SetDMAHalt(TSB_DMAC_TypeDef * ***DMACx,*** DMAC_Channel ***Chx***, FunctionalState ***NewState***);
- void DMAC_SetLockedTx(TSB_DMAC_TypeDef * ***DMACx,*** DMAC_Channel ***Chx***, FunctionalState ***NewState***)

- void DMAC_SetTxINTConfig(TSB_DMAC_TypeDef * ***DMACx,*** DMAC_Channel ***Chx***, DMAC_INTSrc ***INTSource***, FunctionalState ***NewState***);
- void DMAC_SetDMAChannel(TSB_DMAC_TypeDef * ***DMACx,*** DMAC_Channel ***Chx***, FunctionalState ***NewState***);
- void DMAC_Init(TSB_DMAC_TypeDef * ***DMACx*** , DMAC_Channel ***Chx***, DMAC_InitTypeDef * ***InitStruct***);

5.2.2 関数の種類

関数は、主に以下の 5 種類に分かれています。

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

5.2.3 関数仕様

補足: 下記の全 API において、パラメータ"TSB_DMAC_TypeDef * ***DMACx***" は以下をのいずれかを選択してください。
DMAC_UNIT_A, DMAC_UNIT_B

5.2.3.1 DMAC_Enable

DMA 回路動作の許可

関数のプロトタイプ宣言:
void
DMAC_Enable(TSB_DMAC_TypeDef * ***DMACx***);

引数:
DMACx: ユニットを選択します。

機能:
DMA 回路動作を許可します。

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

戻り値:
なし

5.2.3.2 DMAC_Disable

DMA 回路動作の禁止

関数のプロトタイプ宣言:
void
DMAC_Disable(TSB_DMAC_TypeDef * ***DMACx***);

引数:
DMACx: ユニットを選択します。

機能:
DMA 回路動作を禁止します。

戻り値:
なし

5.2.3.3 DMAC_GetINTReq

DMA チャネル割り込みステータスの取得

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

引数:
DMACx: ユニットを選択します。

機能:
DMA チャネル割り込み要求状態を取得します。

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

5.2.3.4 DMAC_GetTxINTReq

DMA チャネル転送割り込み要求状態の取得

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

引数:
DMACx: ユニットを選択します。

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

機能:

DMA チャネル転送割り込み要求状態を取得します。

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

5.2.3.5 DMAC_ClearTxINTReq

転送割り込み要求のクリア

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

引数:
DMACx: ユニットを選択します。

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

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

機能:
転送割り込み要求をクリアします。

戻り値:
なし

5.2.3.6 DMAC_GetRawTxINTReq

DMA チャネルの許可前転送終了割り込み発生状態の取得

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

引数:
DMACx: ユニットを選択します。

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

機能:
DMA チャネルの許可前転送終了割り込み発生状態を取得します。

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

5.2.3.7 DMAC_GetChannelTxState

DMA チャネル転送状態の取得

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

引数:
DMACx: ユニットを選択します。

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

機能:
本関数は、***Chx*** が **DMAC_CHANNEL_0** の時、DMA チャネル 0 転送状態を取得します。***Chx*** が **DMAC_CHANNEL_1** の時、DMA チャネル 1 転送状態を取得します。戻り値が **BUSY** の時は、DMA チャネルは有効で、データ送信中であることを示します。戻り値が **DONE** の時は、DMA チャネルは無効で、データ送信は終了していることを示します。

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

5.2.3.8 DMACA_SetSWBurstReq

ソフトウェアによるユニット A の DMA バースト転送要求の設定

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

引数:
BurstReq: 以下のいずれかのバースト要求番号を選択します。
- **DMACA_SIO0_UART0_RX**: SIO0/UART0 受信
- **DMACA_SIO0_UART0_TX**: SIO0/UART0 送信
- **DMACA_SIO1_UART1_RX**: SIO1/UART1 受信
- **DMACA_SIO1_UART1_TX**: SIO1/UART1 送信
- **DMACA_TMRB8_CMP_MATCH**: TMRB8 コンペアー致

- **DMACA_TMRB9_CMP_MATCH**: TMRB9 コンペアー致
- **DMACA_TMRB0_CAPTURE0** : TMRB0 キャプチャ 0 割り込み
- **DMACA_TMRB4_CAPTURE0**: TMRB4 キャプチャ 0 割り込み
- **DMACA_TMRB4_CAPTURE1**: TMRB4 キャプチャ 1 割り込み
- **DMACA_TMRB5_CAPTURE0**: TMRB5 キャプチャ 0 割り込み
- **DMACA_TMRB5_CAPTURE1**: TMRB5 キャプチャ 1 割り込み
- **DMACA_TOP_PRIORITY_ADC**: 最優先 A/D 変換終了
- **DMACA_FULL_UART_RX**: フル UART 受信(PL011)
- **DMACA_FULL_UART_TX**: フル UART 送信(PL011)

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

戻り値:
なし

5.2.3.9 DMACB_SetSWBurstReq

ソフトウェアによるユニット B の DMA バースト転送要求の設定

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

引数:
BurstReq: 以下のいずれかのバースト要求番号を選択します。
- **DMACB_TMRB6_CMP_MATCH**: TMRB6 コンペアー致
- **DMACB_TMRB7_CMP_MATCH**: TMRB7 コンペアー致
- **DMACB_TMRB0_CAPTURE1**: TMRB0 キャプチャ 1 割り込み
- **DMACB_TMRB2_CAPTURE0**: TMRB2 キャプチャ 0 割り込み
- **DMACB_TMRB2_CAPTURE1**: TMRB2 キャプチャ 1 割り込み
- **DMACB_TMRB3_CAPTURE0**: TMRB3 キャプチャ 0 割り込み
- **DMACB_TMRB3_CAPTURE1**: TMRB3 キャプチャ 1 割り込み
- **DMACB_TMRB6_CAPTURE0**: TMRB6 キャプチャ 0 割り込み
- **DMACB_TMRB6_CAPTURE1**: TMRB6 キャプチャ 1 割り込み
- **DMACB_NORMAL_ADC**: 通常 AD 変換終了
- **DMACB_SSP0_TX**: SSP0 送信
- **DMACB_SSP0_RX**: SSP0 受信
- **DMACB_SSP1_TX**: SSP1 送信
- **DMACB_SSP1_RX**: SSP1 受信
- **DMACB_SSP2_TX**: SSP2 送信
- **DMACB_SSP2_RX**: SSP2 受信

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

戻り値:
なし

5.2.3.10 DMAC_GetSWBurstReqState

ソフトウェアによる DMA バースト要求状態の取得

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

引数:
DMACx: 以下からユニットを選択します。

機能:
ソフトウェアによる DMA バースト要求状態を取得します。

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

5.2.3.11 DMACA_SetSWSingleReq

ソフトウェアによるユニット A の DMA シングル転送要求の設定

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

引数:
SingleReq: 以下から、シングル要求番号を選択します。
- **DMACA_FULL_UART_RX**: フル UART 受信(PL011)
- **DMACA_FULL_UART_TX**: フル UART 送信(PL011)

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

戻り値:
なし

5.2.3.12 DMACB_SetSWSingleReq

ソフトウェアによるユニット B の DMA シングル転送要求の設定

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

引数:
SingleReq: 以下から、シングル要求番号を選択します。
- **DMACB_SSP0_TX**: SSP0 送信
- **DMACB_SSP0_RX**: SSP0 受信
- **DMACB_SSP1_TX**: SSP1 送信
- **DMACB_SSP1_RX**: SSP1 受信

- **DMACB_SSP2_TX**: SSP2 送信
- **DMACB_SSP2_RX**: SSP2 受信

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

戻り値:
なし

5.2.3.13 DMAC_GetSWSingleReqState

ソフトウェアによる DMA シングル要求状態の取得

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

引数:
DMACx: 以下からユニットを選択します。

機能:
ソフトウェアによる DMA シングル要求状態を取得します。

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

5.2.3.14 DMAC_SetLinkedList

DMA チャネル・コレクションアイテムレジスタの設定

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

引数:
DMACx: 以下からユニットを選択します。

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

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

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

補足:

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

戻り値:
なし

5.2.3.15 DMAC_GetFIFOState

FIFO 状態の取得

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

引数:
DMACx: 以下からユニットを選択します。

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

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

戻り値:
FIFO 状態:
BUSY、または **DONE**

5.2.3.16 DMAC_SetDMAHalt

DMA 要求の設定

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

引数:
DMACx: 以下からユニットを選択します。

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

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

機能:
DMA 要求受付制御を設定します。

戻り値:
なし

5.2.3.17 DMAC_SetLockedTx

ロック転送の設定

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

引数:
DMACx: 以下からユニットを選択します。

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

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

機能:
ロック転送を設定します。

戻り値:
なし

5.2.3.18 DMAC_SetTxINTConfig

転送割り込みの設定

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

引数:
DMACx: 以下からユニットを選択します。

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

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

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

機能:
転送割り込みを設定します。

戻り値:
なし

5.2.3.19 DMAC_SetDMAChannel

DMA チャネルの許可/禁止設定

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

引数:
DMACx: 以下からユニットを選択します。

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

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

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

戻り値:

なし

5.2.3.20 DMAC_Init

DMA チャネルの初期設定

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

引数:
DMACx: 以下からユニットを選択します。

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

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

機能:
DMA チャネルの初期設定を行います。

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

戻り値:
なし

5.2.4 データ構造

5.2.4.1 DMAC_InitTypeDef

メンバ:

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

uint32_t
SrcAddr: 転送元アドレスを設定します。

uint32_t
DstAddr: 転送先アドレスを設定します。

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

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

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

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

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

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

DMACA_ReqNum
A_TxDstPeriph: 以下のいずれかのバースト要求番号を選択します。
- **DMACA_SIO0_UART0_RX:** SIO0/UART0 受信
- **DMACA_SIO0_UART0_TX**: SIO0/UART0 送信
- **DMACA_SIO1_UART1_RX**: SIO1/UART1 受信
- **DMACA_SIO1_UART1_TX**: SIO1/UART1 送信
- **DMACA_TMRB8_CMP_MATCH**: TMRB8 コンペアー致
- **DMACA_TMRB9_CMP_MATCH**: TMRB9 コンペアー致

- **DMACA_TMRB0_CAPTURE0** : TMRB0 キャプチャ 0 割り込み
- **DMACA_TMRB4_CAPTURE0**: TMRB4 キャプチャ 0 割り込み
- **DMACA_TMRB4_CAPTURE1**: TMRB4 キャプチャ 1 割り込み
- **DMACA_TMRB5_CAPTURE0:** TMRB5 キャプチャ 0 割り込み
- **DMACA_TMRB5_CAPTURE1**: TMRB5 キャプチャ 1 割り込み
- **DMACA_TOP_PRIORITY_ADC**: 最優先 A/D 変換終了
- **DMACA_FULL_UART_RX**: フル UART 受信(PL011)
- **DMACA_FULL_UART_TX**: フル UART 送信(PL011)

DMACA_ReqNum

A_TxSrcPeriph: 以下のいずれかのバースト要求番号を選択します。

- **DMACA_SIO0_UART0_RX:** SIO0/UART0 受信
- **DMACA_SIO0_UART0_TX**: SIO0/UART0 送信
- **DMACA_SIO1_UART1_RX**: SIO1/UART1 受信
- **DMACA_SIO1_UART1_TX**: SIO1/UART1 送信
- **DMACA_TMRB8_CMP_MATCH**: TMRB8 コンペアー致
- **DMACA_TMRB9_CMP_MATCH**: TMRB9 コンペアー致
- **DMACA_TMRB0_CAPTURE0** : TMRB0 キャプチャ 0 割り込み
- **DMACA_TMRB4_CAPTURE0**: TMRB4 キャプチャ 0 割り込み
- **DMACA_TMRB4_CAPTURE1**: TMRB4 キャプチャ 1 割り込み
- **DMACA_TMRB5_CAPTURE0:** TMRB5 キャプチャ 0 割り込み
- **DMACA_TMRB5_CAPTURE1**: TMRB5 キャプチャ 1 割り込み
- **DMACA_TOP_PRIORITY_ADC**: 最優先 A/D 変換終了
- **DMACA_FULL_UART_RX**: フル UART 受信(PL011)
- **DMACA_FULL_UART_TX**: フル UART 送信(PL011)

DMACB_ReqNum

B_TxDstPeriph: 以下のいずれかのバースト要求番号を選択します。

- **DMACB_TMRB6_CMP_MATCH**: TMRB6 コンペアー致
- **DMACB_TMRB7_CMP_MATCH**: TMRB7 コンペアー致
- **DMACB_TMRB0_CAPTURE1**: TMRB0 キャプチャ 1 割り込み
- **DMACB_TMRB2_CAPTURE0**: TMRB2 キャプチャ 0 割り込み
- **DMACB_TMRB2_CAPTURE1**: TMRB2 キャプチャ 1 割り込み
- **DMACB_TMRB3_CAPTURE0**: TMRB3 キャプチャ 0 割り込み
- **DMACB_TMRB3_CAPTURE1**: TMRB3 キャプチャ 1 割り込み
- **DMACB_TMRB6_CAPTURE0**: TMRB6 キャプチャ 0 割り込み
- **DMACB_TMRB6_CAPTURE1**: TMRB6 キャプチャ 1 割り込み
- **DMACB_NORMAL_ADC**: 通常 AD 変換終了
- **DMACB_SSP0_TX**: SSP0 送信
- **DMACB_SSP0_RX**: SSP0 受信
- **DMACB_SSP1_TX**: SSP1 送信
- **DMACB_SSP1_RX**: SSP1 受信
- **DMACB_SSP2_TX**: SSP2 送信
- **DMACB_SSP2_RX**: SSP2 受信

DMACB_ReqNum

B_TxSrcPeriph: 以下のいずれかのバースト要求番号を選択します。

- **DMACB_TMRB6_CMP_MATCH**: TMRB6 コンペアー致
- **DMACB_TMRB7_CMP_MATCH**: TMRB7 コンペアー致
- **DMACB_TMRB0_CAPTURE1**: TMRB0 キャプチャ 1 割り込み

- **DMACB_TMRB2_CAPTURE0**: TMRB2 キャプチャ 0 割り込み
- **DMACB_TMRB2_CAPTURE1**: TMRB2 キャプチャ 1 割り込み
- **DMACB_TMRB3_CAPTURE0**: TMRB3 キャプチャ 0 割り込み
- **DMACB_TMRB3_CAPTURE1**: TMRB3 キャプチャ 1 割り込み
- **DMACB_TMRB6_CAPTURE0**: TMRB6 キャプチャ 0 割り込み
- **DMACB_TMRB6_CAPTURE1**: TMRB6 キャプチャ 1 割り込み
- **DMACB_NORMAL_ADC**: 通常 AD 変換終了
- **DMACB_SSP0_TX**: SSP0 送信
- **DMACB_SSP0_RX**: SSP0 受信
- **DMACB_SSP1_TX**: SSP1 送信
- **DMACB_SSP1_RX**: SSP1 受信
- **DMACB_SSP2_TX**: SSP2 送信
- **DMACB_SSP2_RX**: SSP2 受信

FunctionalState
TxINT: 以下から、転送割り込みステートを選択します。

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

6. EXB

6.1 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ーフェース機能を内蔵しています。外部バスインターフェース回路 (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/TX03_Periph_Driver/src/tmpm366_exb.c
/Libraries/TX03_Periph_Driver/inc/tmpm366_exb.h

6.2 API 関数

6.2.1 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***);

6.2.2 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) EXB バスモード、バスサイクルウェイト拡張、データバス幅、チップセレクタを元にした外部バスサイクルの設定:
   EXB_SetBusMode(),EXB_SetBusCycleExtension(), EXB_Init()
2) 許可/禁止制御:
   EXB_Enable(), EXB_Disable()

6.2.3 関数仕様

6.2.3.1 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_SEPARATE**: セパレートバスモード
- **EXB_BUS_MULTIPLEX**: マルチプレクスバスモード

機能:
外部バスモードを設定します。***BusMode*** に **EXB_BUS_SEPARATE** を設定した場合、バスモードはセパレートバスモードになります。***BusMode*** に **EXB_BUS_MULTIPLEX** を設定した場合、バスモードはマルチプレクスモードになります。

戻り値:
なし

6.2.3.2 EXB_SetBusCycleExtension

バスサイクルウェイト拡張の設定

関数のプロトタイプ宣言:
void
EXB_SetBusCycleExtension(uint8_t ***Cycle***)

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

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

戻り値:
なし

6.2.3.3 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

機能:
チップセレクトを許可します。

戻り値:

なし

6.2.3.4 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

機能:
チップセレクトを禁止します。

戻り値:
なし

6.2.3.5 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

InitStruct : 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、外部バスサイクルを設定する構造体です。(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

6.2.4 データ構造

6.2.4.1 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 0x1FF 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)

6.2.4.2 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

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。TMPM366FDFG のフラッシュメモリのサイズは、512Kbyte です。

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

ブロック構成は、デバイスのデータシートを参照してください。

全ドライバ API は、アプリで使用する API 定義、マクロ、データタイプ、構造を格納する以下のファイルで構成されています。
\Libraries\TX03_Periph_Driver\src\tmpm366_fc.c
\Libraries\TX03_Periph_Driver \inc\tmpm366_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_ProgramBlockProtectState(uint8_t ***BlockNum***)
- FC_Result FC_EraseBlockProtectState(uint8_t ***BlockGroup***)
- FC_Result FC_WritePage(uint32_t ***PageAddr***, uint32_t * ***Data***)
- FC_Result FC_EraseBlock(uint32_t ***BlockAddr***)
- FC_Result FC_EraseChip(void)

7.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。

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

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:
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:
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。
- **FC_BLOCK_0 ~ FC_BLOCK_5**

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_ProgramBlockProtectState

ブロックのプロテクト設定

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

引数:
BlockNum: ブロック番号を選択します。
- **FC_BLOCK_0 ~ FC_BLOCK_5**

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

戻り値:
FC_SUCCESS: プロテクト設定の成功
FC_ERROR_PROTECTED: プロテクト設定の失敗(すでにプロテクト済の場合は再度プロテクト設定を行いません)

FC_ERROR_OVER_TIME: プロテクト設定の失敗(自動動作のタイムアウト)

7.2.3.6 FC_EraseBlockProtectState

プロテクトの解除

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

引数:
BlockGroup: ブロックグループを指定してください。
- **FC_BLOCK_GROUP_1**: ブロック 4, 5
- **FC_BLOCK_GROUP_0**: ブロック 0~3

機能:
プロテクトビットを"0"にすることでプロテクトを解除します。

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

7.2.3.7 FC_WritePage

ページ単位の書き込み

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

引数:
PageAddr: ページの開始アドレスを指定します。
Data: 書き込むデータバッファへのポインタを指定します。サイズは 512Byte 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: あらかじめデータを消去せずに書き込みを行うと、デバイスに損傷を与える恐れがあります。

戻り値:
FC_SUCCESS: 書き込み成功
FC_ERROR_PROTECTED: 書き込み失敗(ブロックにプロテクトが設定されている)
FC_ERROR_OVER_TIME: 書き込みの失敗(自動動作のタイムアウト)

7.2.3.8 FC_EraseBlock

ブロック単位の消去

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

引数:
BlockAddr: ブロック開始アドレスを指定してください。

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:
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)

7.2.3.9 FC_EraseChip

チップ消去

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

引数:
なし。

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

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

7.2.4 データ構造

なし

8. GPIO

8.1 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/TX03_Periph_Driver/src/tmpm366_gpio.c
/Libraries/TX03_Periph_Driver/inc/tmpm366_gpio.h

8.2 API 関数

8.2.1 関数一覧

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

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) 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()

8.2.3 関数仕様

8.2.3.1 GPIO_ReadData

DATA データレジスタの読み込み

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

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

機能:
DATA レジスタを読み込みます。

戻り値:
DATA レジスタの値

8.2.3.2 GPIO_ReadDataBit

ビット単位での DATA レジスタの読み込み

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

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

Bit_x: GPIO 端子を選択します。

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

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 端子の値:
- **GPIO_BIT_VALUE_0**: 0
- **GPIO_BIT_VALUE_1**: 1

8.2.3.3 GPIO_WriteData

DATA レジスタへの書き込み

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

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

Data: DATA レジスタに書き込む値を設定します。

機能:
DATA レジスタへ指定された値を書き込みます。

戻り値:
なし

8.2.3.4 GPIO_WriteDataBit

ビット単位での DATA レジスタの書き込み

関数のプロトタイプ宣言:

void
GPIO_WriteDataBit(GPIO_Port ***GPIO_x***,
uint8_t ***Bit_x***,
uint8_t ***BitValue***);

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

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

BitValue: GPIO 端子値を選択します。
- **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

8.2.3.5 GPIO_Init

GPIO ポートの初期設定

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

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

- **GPIO_PD:** GPIO port D
- **GPIO_PE:** GPIO port E
- **GPIO_PF:** GPIO port F
- **GPIO_PG:** GPIO port G
- **GPIO_PH:** GPIO port H
- **GPIO_PI:** GPIO port I
- **GPIO_PJ:** GPIO port J
- **GPIO_PK:** GPIO port K

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

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

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

戻り値:
なし

8.2.3.6 GPIO_SetOutput

出力ポートの設定

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

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

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

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

機能:
出力ポートに設定します。

戻り値:
なし

8.2.3.7 GPIO_SetInput

入力ポートの設定

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

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

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

機能:
入力ポートに設定します。

戻り値:

なし

8.2.3.8 GPIO_SetOutputEnableReg

出力ポートの許可/禁止設定

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

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

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

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

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

戻り値:
なし

8.2.3.9 GPIO_SetInputEnableReg

入力ポートの許可/禁止設定

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

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

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

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

機能:
GPIO 端子入力の許可/禁止を設定します。***NewState*** が **ENABLE** の時、入力を許可します。***NewState*** が **DISABLE** の時、入力を禁止します。

戻り値:
なし

8.2.3.10 GPIO_SetPullUp

プルアップポートの設定

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

引数:
GPIO_x: GPIO ポートを選択します。

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

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

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

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

戻り値:
なし

8.2.3.11 GPIO_SetPullDown

プルダウンポートの設定

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

引数:
GPIO_x: GPIO ポートを選択します。
- **GPIO_PI :** GPIO port I.

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

- **GPIO_BIT_4:** GPIO pin 4
- **GPIO_BIT_5:** GPIO pin 5
- **GPIO_BIT_6:** GPIO pin 6
- **GPIO_BIT_7:** GPIO pin 7
- **GPIO_BIT_ALL:** すべての GPIO 端子

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

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

戻り値:
なし

8.2.3.12 GPIO_SetOpenDrain

CMOS/オープンドレインポートの設定

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

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

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

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

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

戻り値:
なし

8.2.3.13 GPIO_EnableFuncReg

機能ポートの有効設定

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

引数:

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

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

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

機能:
GPIO 端子の機能を有効に設定します。

戻り値:

なし

8.2.3.14 GPIO_DisableFuncReg

機能ポートの無効設定

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

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

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

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

機能:
GPIO 端子の機能を無効に設定します。

戻り値:
なし

8.2.4 データ構造

8.2.4.1 GPIO_InitTypeDef

メンバ:

uint8_t
IOMode　ポートの入出力設定
- **GPIO_INPUT:** 入力ポートに設定
- **GPIO_OUTPUT:** 出力ポートに設定
- **GPIO_IO_MODE_NONE:** 入出力モードを変更しない

uint8_t
PullUp　プルアップポートの許可/禁止設定
- **GPIO_PULLUP_ENABLE:** プルアップ許可
- **GPIO_PULLUP_DISABLE:** プルアップ禁止
- **GPIO_PULLUP_NONE:** プルアップ機能が無い、または設定変更しない

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

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

9. SBI

9.1 概要

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

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

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm366_sbi.c
/Libraries/TX03_Periph_Driver/inc/tmpm366_sbi.h

9.2 API 関数

9.2.1 関数一覧

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

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) 共通機能の設定:
   SBI_Enable(), SBI_Disable(), SBI_SetI2CACK(), SBI_SetI2CBitNum(), SBI_InitI2C()
2) 転送制御:
   SBI_ClearI2CINTReq(), SBI_GenerateI2Cstart(), SBI_GenerateI2Cstop(),
   SBI_IsI2ClastRxBitSet(), SBI_GetReceiveData()
3) ステータス確認:
   SBI_GetI2CState()

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

9.2.3 関数仕様

補足: 下記の全 API において、パラメータ"TSB_SBI_TypeDef* ***SBIx***" は 以下のいずれかを選択してください。
TSB_SBI0, TSB_SBI1

9.2.3.1 SBI_Enable

SBI 動作の許可

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

引数:
SBIx: SBI チャネルを指定します。

機能:
SBI 動作を有効にします。

戻り値:
なし

9.2.3.2 SBI_Disable

SBI 動作の禁止

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

引数:
SBIx: SBI チャネルを指定します。

機能:
SBI 動作を無効にします。

戻り値:
なし

9.2.3.3 SBI_SetI2CACK

I2C バスモードにおける ACK 選択

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

引数:

SBIx: SBI チャネルを指定します。
NewState: ACK の発生有無を選択します。
- **ENABLE**: 発生する。
- **DISABLE**: 発生しない。

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

戻り値:
なし

9.2.3.4 SBI_InitI2C

I2C バスモードにおける通信の初期化

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

引数:
SBIx: SBI チャネルを指定します。
InitI2CStruct: SBI に関する構造体です。(詳細は"データ構造"を参照)

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

戻り値:
なし

9.2.3.5 SBI_SetI2CBitNum

I2C バスモードにおける転送ビット数の選択

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

引数:
SBIx: SBI チャネルを指定します。
I2CBitNum: 転送ビット数(1～8)を選択します。
- **SBI_I2C_DATA_LEN_8:** データ長 8
- **SBI_I2C_DATA_LEN_1:** データ長 1
- **SBI_I2C_DATA_LEN_2:** データ長 2
- **SBI_I2C_DATA_LEN_3:** データ長 3
- **SBI_I2C_DATA_LEN_4:** データ長 4
- **SBI_I2C_DATA_LEN_5:** データ長 5

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

機能:
転送ビット数を選択します。

戻り値:
なし

9.2.3.6 SBI_SWReset

ソフトウェアリセットの発生

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

引数:
SBIx: SBI チャネルを指定します。

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

戻り値:
なし

9.2.3.7 SBI_ClearI2CINTReq

I2C バスモードにおける INTSBIx 割り込み要求解除

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

引数:
SBIx: SBI チャネルを指定します。

機能:
SBI 割り込み要求を解除します。

戻り値:
なし

9.2.3.8 SBI_GenerateI2CStart

I2C バスモードにおけるスタート状態の発生

関数のプロトタイプ宣言:
void
SBI_GenerateI2CStart(TSB_SBI_TypeDef* ***SBIx***)

引数:
SBIx: SBI チャネルを指定します。

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

戻り値:
なし

9.2.3.9 SBI_GenerateI2CStop

I2C バスモードにおけるストップ状態の発生

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

引数:
SBIx: SBI チャネルを指定します。

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

戻り値:
なし

9.2.3.10 SBI_GetI2CState

I2C バスモードにおける SBI チャネルの状態の読み込み

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

引数:
SBIx: SBI チャネルを指定します。

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

戻り値:
I2C モードでの SBI チャネルの状態

9.2.3.11 SBI_SetIdleMode

IDLE モード時の動作の許可/禁止

関数のプロトタイプ宣言:
void

SBI_SetIdleMode(TSB_SBI_TypeDef* ***SBIx***,
FunctionalState ***NewState***)

引数:
SBIx: SBI チャネルを指定します。
NewState: システムが idle モードの時の動作を指定します。
- **ENABLE:** 許可。
- **DISABLE:** 禁止。

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

戻り値:
なし

9.2.3.12 SBI_SetSendData

データ送信

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

引数:
SBIx: SBI チャネルを指定します。
Data: 送信データ。(最大値は 0xFF です)

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

戻り値:
なし

9.2.3.13 SBI_GetReceiveData

データ受信

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

引数:
SBIx: SBI チャネルを指定します。

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

戻り値:
受信データ

9.2.3.14 SBI_SetI2CFreeDataMode

Set SBI channel working in I2C free data mode.
アドレス認識モードの指定

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

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

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

戻り値:
なし

9.2.4 データ構造

9.2.4.1 SBI_InitI2CTypeDef

メンバ:

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

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

uint32_t
***I2CClkDiv*:** I2C 転送のソースクロックを選択します。
- **SBI_I2C_CLK_DIV_104:** fsys/104
- **SBI_I2C_CLK_DIV_136:** fsys/136

- **SBI_I2C_CLK_DIV_200:** fsys/200
- **SBI_I2C_CLK_DIV_328:** fsys/328
- **SBI_I2C_CLK_DIV_584:** fsys/584
- **SBI_I2C_CLK_DIV_1096:** fsys/1096
- **SBI_I2C_CLK_DIV_2120:** fsys/2120

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

- **ENABLE:** 有効。
- **DISABLE:** 無効。

9.2.4.2 SBI_I2CState

メンバ:

uint32_t
***All*:** I2C モードの全ての状態

ビットフィールド:

uint32_t
***LastRxBit*:** 最終受信ビットモニタ

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

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

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

uint32_t
***INTReq*:** 割り込み要求状態モニタ

uint32_t
***BusState*:** バス状態モニタ

uint32_t
***TRx*:** 送信/受信選択状態モニタ

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

10. SSP

10.1 概要

本デバイスは、同期式シリアルインタフェースを (SSP：Synchronous Serial Port) を 3 チャネル内蔵しています。(SSP0, SSP1, SSP2)

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

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

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm365_ssp.c
/Libraries/TX03_Periph_Driver/inc/tmpm365_ssp.h

10.2 API 関数

10.2.1 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***);

10.2.2 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) 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()

10.2.3 関数仕様

補足: 引数に記述されている “TSB_SSP_TypeDef* ***SSPx***” は特に記載の無い限り以下から選択してください。
SSP0, SSP1, SSP2

10.2.3.1 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

10.2.3.2 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

10.2.3.3 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

10.2.3.4 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 はシステム周波数です。

戻り値:
なし

10.2.3.5 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

10.2.3.6 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

10.2.3.7 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

10.2.3.8 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

10.2.3.9 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

10.2.3.10 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

10.2.3.11 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

10.2.3.12 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

10.2.3.13 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

10.2.3.14 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

10.2.3.15 SSP_GetFIFOState

送受信 FIFO の読み込み

関数のプロトタイプ宣言:
SSP_FIFOState
SSP_GetFIFOState(TSB_SSP_TypeDef*_**SSPx**_, SSP_Direction _**Direction**_)

引数:
SSPx: SSP チャネルを指定します。

Direction: 送受信方向を選択します。
- **SSP_RX**: 受信 FIFO
- **SSP_TX**: 送信 FIFO

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

```
SSP_FIFOState  fifoState;

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

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

10.2.3.16 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

10.2.3.17 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。

10.2.3.18 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。

10.2.3.19 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。

10.2.3.20 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

10.2.3.21 SSP_SetDMACtrl

送受信 FIFO の DMA 制御

関数のプロトタイプ宣言:

void
SSP_SetDMACtrl(TSB_SSP_TypeDef*__SSPx__, SSP_Direction __Direction__,
FunctionalState __NewState__)

引数:
__SSPx:__ SSP チャネルを指定します。

__Direction__: 送受信方向を選択します。
- **SSP_RX**: 受信。
- **SSP_TX**: 送信。

__NewState__: DMA FIFO の状態。
- **ENABLE**: 許可。
- **DISABLE**: 禁止。

機能:
送受信 FIFO の DMA 許可/禁止を選択します。

戻り値:
なし

10.2.4 データ構造

10.2.4.1 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

10.2.4.2 SSP_INTState

メンバ:

uint32_t
All: 割り込み要因
Bit

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

11. TMRB

11.1 概要

本デバイスは、10 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 ～ TMRB9)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
- 時間差測定

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/tmpm366_tmrb.c
/Libraries/TX03_Periph_Driver/inc/tmpm366_tmrb.h

11.2 API 関数

11.2.1 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***);
- void TMRB_SetExtStartTrg(TSB_TB_TypeDef * ***TBx***, FunctionalState ***NewState***, uint8_t ***TrgMode***);
- void TMRB_SetClkInCoreHalt(TSB_TB_TypeDef * ***TBx***, uint8_t ***ClkState***);
- void TMRB_SetExtInput(TSB_TB_TypeDef * ***TBx***);
- void TMRB_SetDMAReq(TSB_TB_TypeDef * TBx, FunctionalState ***NewState***, uint8_t ***DMAReq***);

11.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。

1) 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 (), TMRB_SetExtInput(), TMRB_SetDMAReq()

11.2.3 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, TSB_TB4, TSB_TB5, TSB_TB6, TSB_TB7, TSB_TB8, TSB_TB9

11.2.3.1 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

11.2.3.2 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

11.2.3.3 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

11.2.3.4 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

11.2.3.5 TMRB_SetCaptureTiming

キャプチャタイミングの設定

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

引数:
***TBx*:** TMRB チャネルを指定します。

CaptureTiming: キャプチャタイミングを選択します。
- **TMRB_DISABLE_CAPTURE**: キャプチャ機能を無効にします。
- **TMRB_CAPTURE_IN_RISING**: TBxIN0↑ TBxIn1↑
- **TMRB_CAPTURE_IN_RISING_FALLING**: TBxIN0↑ TBxIN0
- **TMRB_CAPTURE_OUTPUT_EDGE**: TBxFF0↑ TBxFF0↓

機能:
CaptureTiming が **TMRB_CAPTURE_IN_RISING** の場合、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。
CaptureTiming が **TMRB_CAPTURE_IN_RISING_FALLING** の場合、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。
CaptureTiming が **TMRB_CAPTURE_OUTPUT_EDGE** の場合、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 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。

TMRB7, TMRB8, TMRB9 のフリップフロップ出力を他のチャネルのキャプチャトリガとして使用できます。
TMRB0~1: TB7OUT
TMRB2~3: TB8OUT
TMRB4~6: TB9OUT

戻り値:
なし

11.2.3.6 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

11.2.3.7 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
}
```

11.2.3.8 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。

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、オーバフロー発生時の割り込みは発生しません。
TMRB_NO_INT_MASK 選択時、割り込みマスクはすべてクリアされます。

戻り値:
なし

11.2.3.9 TMRB_ChangeLeadingTiming

デューティーの設定

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*:** デューティー値を設定します。最大値は 0xFFFF です。

機能:
デューティーを設定します。実際のデューティーのインターバルは、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。

11.2.3.10 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。

11.2.3.11 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

11.2.3.12 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

11.2.3.13 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

11.2.3.14 TMRB_SetIdleMode

IDLE 時の動作設定

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

引数:
***TBx*:** TMRB チャネルを指定します。

NewState: IDLE 時の動作を指定します。
- **ENABLE:** 動作
- **DISABLE:** 停止

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

戻り値:
なし

11.2.3.15 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 チャネルを以下から選択します。
TSB_TB0, TSB_TB1, TSB_TB2, TSB_TB3, TSB_TB4, TSB_TB5, TSB_TB6, TSB_TB7.

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

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

戻り値:
なし

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

11.2.3.16 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

11.2.3.17 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

11.2.3.18 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

11.2.3.19 TMRB_SetExtInput

外部入力の設定。

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

引数:
***TBx*:** TMRB チャネルを選択します。

***ExtInput*:** 以下のいずれかの外部入力を選択してください。
- **TMRB_EXT_INPUT_TBxIN:** TBxIN0/1
- **TMRB_EXT_INPUT_PHCxIN:** PHCxIN0/1

機能:
外部入力として TBxIN0/1 または PHCxIN0/1 を設定します。

戻り値:
なし

11.2.3.20 TMRB_SetDMAReq

DMA 要求の制御

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

引数:
TBx: TMRB チャネルを選択します。
NewState: 以下から DMA 要求の許可/禁止を選択します。
- **ENABLE:** 許可
- **DISABLE:** 禁止

DMAReq: 以下から DMA 要求の種類を選択します。
- **TMRB_DMA_REQ_CMP_MATCH:**コンペアー致
- **TMRB_DMA_REQ_CAPTURE_1:** インプットキャプチャ 1
- **TMRB_DMA_REQ_CAPTURE_0:** インプットキャプチャ 0

機能:
DMA 要求の制御を行います。

戻り値:
なし

補足:

TBxIM レジスタで割り込みをマスク設定している場合、DMA 要求を許可しても要求は発生しません。

11.2.4 データ構造

11.2.4.1 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。

11.2.4.2 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。

11.2.4.3 TMRB_INTFactor

メンバ:

uint32_t
All: TMRB 割り込み要因
Bit
uint32_t
MatchLeadingTiming: 1 デューティとの一致検出
uint32_t
MatchTrailingTiming : 1 周期との一致検出
uint32_t
OverFlow : 1 オーバーフロー
uint32_t
Reserverd : 29 未使用

12. SIO/UART

12.1 概要

本デバイスのシリアル I/O チャネルは、I/O インタフェースモード(同期通信モード)と 7, 8, 9 ビット長の UART モード(非同期通信)を実装しています。9 ビット UART モードでは、シリアルリンク(マルチコントローラ・システム) でマスタコントローラがスレーブコントローラを起動するときにウェイクアップ機能が使用されます。

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

全ドライバ API は、マクロ、データタイプ、構造、API 定義を格納する以下のファイルで構成されています。
/Libraries/TX03_Periph_Driver/src/tmpm366_uart.c
/Libraries/TX03_Periph_Driver/inc/tmpm366_uart.h

12.2 API 関数

12.2.1 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_SetRxDMAReq (TSB_SC_TypeDef* ***UARTx***, FunctionalState ***NewState***)
- void UART_SetTxDMAReq (TSB_SC_TypeDef* ***UARTx***, FunctionalState ***NewState***)
- void UART_FIFOConfig(TSB_SC_TypeDef * ***UARTx***, FunctionalState ***NewState***)
- void UART_SetFIFOTransferMode(TSB_SC_TypeDef * ***UARTx***, uint32_t ***TransferMode***)
- void UART_TRxAutoDisable(TSB_SC_TypeDef * ***UARTx***, UART_TRxDisable ***TrxAutoDisable***
- void UART_RxFIFOINTCtrl(TSB_SC_TypeDef * ***UARTx***, FunctionalState ***NewState***)
- void UART_TxFIFOINTCtrl(TSB_SC_TypeDef * ***UARTx***, FunctionalState ***NewState***)
- void UART_RxFIFOByteSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***BytesUsed***)
- void UART_RxFIFOFillLevel(TSB_SC_TypeDef * ***UARTx***,uint32_t ***RxFIFOLevel***)
- void UART_RxFIFOINTSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***RxINTCondition***)
- void UART_RxFIFOClear(TSB_SC_TypeDef * ***UARTx***)
- void UART_TxFIFOFillLevel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***TxFIFOLevel***)
- void UART_TxFIFOINTSel(TSB_SC_TypeDef * ***UARTx***, uint32_t ***TxINTCondition***)
- void UART_TxFIFOClear(TSB_SC_TypeDef * ***UARTx***)
- uint32_t UART_GetRxFIFOFillLevelStatus(TSB_SC_TypeDef * ***UARTx***)

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

12.2.2 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_,UART_Init(), UART_DefaultConfig(), SIO_Enable(), SIO_Disable(), SIO_,SIO_Init()
2. 送受信設定とエラー確認:
   UART_GetBufState(), UART_GetRxData(), UART_SetTxData(), UART_GetErrState(), SIO_GetRxData(), SIO_SetTxData()
3. その他:
   UART_SetRxDMAReq(), UART_SetTxDMAReq(), 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_GetRxFIFOFillLevelStatus(),UART_GetRxFIFOOverRunStatus(), UART_GetTxFIFOFillLevelStatus(),UART_GetTxFIFOUnderRunStatus()

12.2.3 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**
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**

12.2.3.1 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

12.2.3.2 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

12.2.3.3 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

12.2.3.4 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

12.2.3.5 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

12.2.3.6 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。

12.2.3.7 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

12.2.3.8 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

12.2.3.9 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

12.2.3.10 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

12.2.3.11 UART_SetIdleMode

IDLE 時の動作

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

引数:
UARTx: UART チャネルを指定します。

NewState: IDLE 時の動作を選択します。

- **ENABLE**: 動作

- **DISABLE**: 停止

機能:
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

12.2.3.12 UART_SetRxDMAReq

受信割り込みによる DMA 要求の設定

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

引数:
UARTx: UART チャネルを指定します。

NewState: 以下から受信割り込みによる DMA 要求の許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
受信割り込みによる DMA 要求 (受信割り込み INTRX 発生により DMA リクエストを発行) を設定します。

戻り値:
なし

12.2.3.13 UART_SetTxDMAReq

送信割り込みによる DMA 要求の設定

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

引数:
UARTx: UART チャネルを指定します。

NewState: 以下から送信割り込みによる DMA 要求の許可/禁止を選択します。
- **ENABLE**: 許可
- **DISABLE**: 禁止

機能:
送信割り込みによる DMA 要求 (送信割り込み INTTX 発生により DMA リクエストを発行) を設定します。

戻り値:
なし

12.2.3.14 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

12.2.3.15 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

12.2.3.16 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

12.2.3.17 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

12.2.3.18 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

12.2.3.19 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

12.2.3.20 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

12.2.3.21 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

12.2.3.22 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

12.2.3.23 UART_TxFIFOFillLevel

送信割り込みが発生する送信 FIFO の fill レベルの設定

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

uint32_t ***TxFIFOLevel***)

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

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

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

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

戻り値:
なし

12.2.3.24 UART_TxFIFOINTSel

送信割り込み発生条件の選択

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

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

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

機能:
送信割り込み発生条件を選択します。

機能:
送信割り込み発生条件を選択します。

戻り値:
なし

12.2.3.25 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

2.1.1.1. 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

2.1.1.2. 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

2.1.1.3. 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

2.1.1.4. 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

12.2.3.26 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

12.2.3.27 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

12.2.3.28 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)

12.2.3.29 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

12.2.3.30 SIO_Init

SIO チャネルの初期化

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

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

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

機能:
ボーレート、転送方向、転送モードなどの初期設定を行います。

戻り値:
なし

12.2.4 データ構造

12.2.4.1 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。

12.2.4.2 SIO_InitTypeDef

メンバ:

uint32_t
InputClkEdge: 入力クロックエッジを選択します。"0"(SIO_SCLKS_TXDF_RXDR)のみ指定可能です。

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

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

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

uint32_t

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

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

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

uint32_t
Divider: 分周値"N"を選択します。
- **SIO_BR_DIVIDER_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
- **SIO_BR_DIVIDER_16**: 16 分周

13. USBD

13.1 概要

本デバイスは USB デバイスコントローラを内蔵し、その機能は下記の様になります。

1. Universal Serial Bus Specification Rev.2.0 に準拠
2. Full-Speed をサポート (Low-Speed は非対応)
3. USB プロトコル処理
4. SOF/USB_RESET/SUSPEND/RESUME の検出
5. パケット ID の生成とチェック
6. CRC5 チェック、CRC16 の生成とチェック
7. 4 種類の(Control/ Interrupt/ Bulk/ Isochronous)転送モードをサポート
8. 8 エンドポイントのサポート

Endpoint0	コントロール	64byte x 1 FIFO
Endpoint1	コントロール/インターラプト /バルク / アイソクロナス(IN)	64byte x 2 FIFO
Endpoint2	コントロール/インターラプト /バルク / アイソクロナス(OUT)	64byte x 2 FIFO
Endpoint3	コントロール/インターラプト /バルク / アイソクロナス(IN)	64byte x 2 FIFO
Endpoint4	コントロール/インターラプト /バルク / アイソクロナス(OUT)	64byte x 2 FIFO
Endpoint5	コントロール/インターラプト /バルク / アイソクロナス(IN)	64byte x 2 FIFO
Endpoint6	コントロール/インターラプト /バルク / アイソクロナス(OUT)	64byte x 2 FIFO
Endpoint7	コントロール/インターラプト /バルク / アイソクロナス(IN)	64byte x 2 FIFO

9. デュアルパケットモード対応 (エンドポイント 0 は除く)
10. 割り込みコントローラへの割り込み要因信号: INTUSB , INTUSBWKUP

全ての基本ドライバ 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_USBD_Driver\src\usbd_hw.c
\Libraries\TX03_USBD_Driver\inc\usbd_hw.h

13.2 API 関数

13.2.1 関数一覧

- USBD_INTStatus USBD_GetINTStatus(void)
- void USBD_SetINTMask(USBD_INTSrc ***IntSrc***, FunctionalState ***NewState***)
- void USBD_ClearINT(USBD_INTSrc ***IntSrc***)
- USBD_DMACConfig USBD_GetDMACConfig(void)
- void USBD_ConfigDMAC(USBD_DMACConfig ***Config***)
- USBD_DMACStatus USBD_GetDMACStatus(void)
- void USBD_ReadUDC2Reg(uint32_t ***Addr***, uint32_t * ***Data***)
- void USBD_WriteUDC2Reg(uint32_t ***Addr***, const uint32_t ***Data***)
- USBD_DMACAddr USBD_GetDMACMasterAddr(USBD_MasterMode ***MasterMode***)
- void USBD_SetDMACMasterAddr(USBD_MasterMode ***MasterMode***, USBD_DMACAddr ***Addr***)
- USBD_PowerCtrl USBD_GetPowerCtrlStatus(void)

- void USBD_SetPowerCtrl(USBD_PowerCtrl ***PowerCtrl***)
- void USBD_SetEPCMD(USBD_EPx ***EPx***, USBD_EPCMD ***Cmd***)
- USBD_EP0Status USBD_GetEP0Status(void)
- USBD_EPxStatus USBD_GetEPxStatus(USBD_EPx ***EPx***)
- void USBD_ConfigEPx(USBD_EPx ***EPx***, USBD_EPxConfig ***Config***)

13.2.2 関数の種類

関数は、主に以下の 4 種類に分かれています。

1) 初期化と共通関数:
   USBD_SetINTMask(), USBD_ClearINT(), USBD_SetEPCMD(),
   USBD_ConfigEPx(), USBD_SetPowerCtrl()
2) ステータス関係の関数:
   USBD_GetINTStatus(), USBD_GetEP0Status(), USBD_GetEPxStatus(),
   USBD_GetPowerCtrlStatus()
3) DMA 動作関係の関数:
   USBD_GetDMACConfig(), USBD_ConfigDMAC(), USBD_GetDMACStatus(),
   USBD_GetDMACMasterAddr(), USBD_SetDMACMasterAddr()
4) UDC2 モジュールのレジスタへアクセスするための特殊関数:
   USBD_ReadUDC2Reg(), USBD_WriteUDC2Reg()

13.2.3 関数仕様

13.2.3.1 USBD_GetINTStatus

USBD 割り込み状態の取得

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

引数:
なし。

機能:
USBD の割り込み状態を取得します。
例えば、bit0 に UDC2 割り込み **USBD_INT_SETUP**、bit8 に UDC2AB 割り込み **USBD_INT_SUSPEND_RESUME** など。

戻り値:
USBD 割り込み要因。詳細は"データ説明"の USBD_INTStatus を参照してください。

13.2.3.2 USBD_SetINTMask

USBD 割り込みマスクの設定

関数のプロトタイプ宣言:
void
USBD_SetINTMask(USBD_INTSrc ***IntSrc***,
FunctionalState ***NewState***)

引数:
IntSrc: 割り込み要因を選択します。

- **USBD_INT_SETUP:** UDC2 の int_setup 信号
- **USBD_INT_STATUS_NAK:** UDC2 の int_status_nak 信号
- **USBD_INT_STATUS:** UDC2 の int_status 信号
- **USBD_INT_RX_ZERO:** UDC2 の int_rx_zero 信号
- **USBD_INT_SOF:** UDC2 の int_sof signal 信号
- **USBD_INT_EP0:** UDC2 の int_ep0 signal 信号
- **USBD_INT_EP:** UDC2 の int_ep signal 信号
- **USBD_INT_NAK:** UDC2 の int_nak signal 信号
- **USBD_INT_SUSPEND_RESUME:**
  UDC2 の suspend_x 信号が変更されるたびに割り込み発生
- **USBD_INT_USB_RESET:**
  UDC2 の usb_reset アサート時に、割り込み発生
- **USBD_INT_USB_RESET_END:**
  UDC2 の usb_reset デアサート時に、割り込み発生
- **USBD_INT_MW_SET_ADD:**
  マスタライト転送アドレス要求時に割り込み発生
- **USBD_INT_MW_END_ADD:**
  マスタライト転送完了時に割り込み発生
- **USBD_INT_MW_TIMEOUT:**
  マスタライト転送タイムアウト時に割り込み発生
- **USBD_INT_MW_AHBERR:**
  マスタライト転送中の AHB エラー発生時に割り込み発生
- **USBD_INT_MR_END_ADD:**
  マスタリード転送終了時に割り込み発生
- **USBD_INT_MR_EP_DSET:**
  マスタリード転送に使用される UDC2 Tx 用エンドポイントの FIFO が書き込み可になると割り込み発生
- **USBD_INT_MR_AHBERR:**
  マスタリード転送中に AHB エラーが発生すると割り込み発生
- **USBD_INT_UDC2_REG_READ:**
  UDC2 レジスタのリード/ライトが完了すると割り込み発生
- **USBD_INT_DMAC_REG_READ:**
  DMAC 専用レジスタへのリード/ライトが完了すると割り込み発生
- **USBD_INT_MW_READERROR:**
  マスタライトでエンドポイントリードエラーが発生すると割り込み発生

NewState: 割り込み要因マスク状態を下記から選択します。

- **ENABLE : IntSrc** で割り込みを許可
- **DISABLE: IntSrc** で割り込みを禁止

機能:

USBD の割り込みのマスクを設定します。
本関数により使用していない割り込みを禁止することができます。
パワーオンリセット後の UDC2 割込み(**USBD_INT_SETUP** から **USBD_INT_NAK**)の基本設定は、すべて “**ENABLE**”です。
パワーオンリセット後の UDC2AB 割込み (**USBD_INT_SUSPEND_RESUME** から **USBD_INT_MW_READERROR**) の基本設定は、すべて“**DISABLE**”です。

戻り値:

なし

13.2.3.3 USBD_ClearINT

USBD の割り込みフラグのクリア

関数のプロトタイプ宣言:
void
USBD_ClearINT(USBD_INTSrc ***IntSrc***)

引数:
IntSrc: 下記から割り込み要因を設定

- **USBD_INT_SETUP:** UDC2 の int_setup 信号
- **USBD_INT_STATUS_NAK:** UDC2 の int_status_nak 信号
- **USBD_INT_STATUS:** UDC2 の int_status 信号
- **USBD_INT_RX_ZERO:** UDC2 の int_rx_zero 信号
- **USBD_INT_SOF:** UDC2 の int_sof 信号
- **USBD_INT_EP0:** UDC2 の int_ep0 信号
- **USBD_INT_EP:** UDC2 の int_ep 信号
- **USBD_INT_NAK:** UDC2 の int_nak 信号
- **USBD_INT_SUSPEND_RESUME:**
  UDC2 の suspend_x 信号が変かするたびに割り込み発生
- **USBD_INT_USB_RESET:**
  UDC2 が usb_reset 信号をアサートするたびに割り込み発生
- **USBD_INT_USB_RESET_END:**
  UDC2 が usb_reset 信号をデアサートするたびに割り込み発生
- **USBD_INT_MW_SET_ADD:**
  マスタライト転送アドレス要求のたびに割り込み発生
- **USBD_INT_MW_END_ADD:**
  マスタライト転送完了時に割り込み発生
- **USBD_INT_MW_TIMEOUT:**
  マスタライト転送タイムアウト時に割り込み発生
- **USBD_INT_MW_AHBERR:**
  マスタライト転送中に AHB エラーが発生すると割り込み発生
- **USBD_INT_MR_END_ADD:**
  マスタライト転送中完了時に割り込み発生
- **USBD_INT_MR_EP_DSET:**
  マスタリード転送に使用される UDC2 Tx 用エンドポイントの FIFO が書き込み可になると割り込み発生
- **USBD_INT_MR_AHBERR:**
  マスタリード転送中に AHB エラーが発生すると割り込み発生
- **USBD_INT_UDC2_REG_READ:**
  UDC2 レジスタへのリード/ライトが完了すると割り込み発生
- **USBD_INT_DMAC_REG_READ:**
  DMAC 専用レジスタへのリード/ライトが完了すると割り込み発生
- **USBD_INT_MW_READERROR:**
  マスタライト中にエンドポイントエラーが発生すると割り込み発生

機能:
USBD の割り込みフラグをクリアします。
本関数によって割り込みフラグをクリアしてください。

戻り値:
なし

13.2.3.4 USBD_GetDMACConfig

USBD の DMA コントローラ設定状態の読み込み

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

引数:
なし。

機能:
“マスタライト許可”、”マスタリード許可”、”マスタライトリセット”、”マスタリードリセット”など、USBD の DMA コントローラ設定を読み込みます。

戻り値:
USBD の DMA コントローラ設定。詳細は"データ構造"の USBD_DMACConfig を参照してください。

13.2.3.5 USBD_ConfigDMAC

USBD の DMA コントローラの設定

関数のプロトタイプ宣言:
void
USBD_ConfigDMAC(USBD_DMACConfig ***Config***)

引数:
Config: USBD の DMA コントローラ設定。詳細は"データ構造"の USBD_DMACConfig を参照してください。

機能:
“マスタライト許可”、”マスタリード許可”、”マスタライトリセット”、”マスタリードリセット”など、USBD の DMA コントローラの設定をします。

戻り値:
なし

13.2.3.6 USBD_GetDMACStatus

DMA コントローラマスタ動作状態の読み出し

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

引数:
なし。

機能:
DMA コントローラマスタ動作状態を読み出します。

たとえば、“マスタライトバッファは空”は “**MW_Buf_Empty”** ビット、“マスタリードエンドポイント用データがセットされました”は “**MR_EP_DSet”**ビットになります。

戻り値:
DMAC マスタ動作の状態。詳細は"データ構造"の USBD_DMACStatus を参照してください。

13.2.3.7 USBD_ReadUDC2Reg

UDC2AB の“UDC2 リード要求レジスタ”経由での UDC2 レジスタのデータ読み出し

関数のプロトタイプ宣言:
void
USBD_ReadUDC2Reg(uint32_t ***Addr***,
uint32_t * ***Data***)

引数:
Addr: UDC2 モジュールのレジスタアドレスを選択します。

- **UDC2_ADDR:** アドレス状態、デバイスアドレスレジスタ
- **UDC2_FRAME:** フレームレジスタ
- **UDC2_COMMAND:** エンドポイント へのコマンドレジスタ
- **UDC2_BREQ_BMREQTYPE:**
  設定パッケージの bRequeset-bmRequest タイプレジスタ
- **UDC2_WVALUE:** 設定パッケージの wValue レジスタ
- **UDC2_WINDEX:** 設定パッケージの wIndex レジスタ
- **UDC2_WLENGTH:** 設定パッケージの wLength レジスタ
- **UDC2_INT:**
  UDC2 割り込み信号とマスクビットの INT レジスタ
- **UDC2_INTEP:**
  エンドポイント (EP0 を除く)の送受信状態用のフラグ
- **UDC2_INTEP_MASK: UDC2_INTEP** のマスク設定
- **UDC2_INTRX0:**
  エンドポイントで受信される Zero-Length データを示すフラグ.
- **UDC2_EPxMAXPACKETSIZE(x = 0 to 7):** EPx_MaxPacketSize レジスタ
- **UDC2_EPxSTATUS(x = 0 to 7):** EPx status レジスタ
- **UDC2_EPxDATASIZE(x = 0 to 7):** EPx datasize レジスタ
- **UDC2_EPxFIFO(x = 0 to 7):** EPx FIFO レジスタ
- **UDC2_INTNAK:**
  エンドポイント (EP0 を除外)の NAK 送信状態を示すフラグ
- **UDC2_INTNAK_MASK: UDC2_INTNAK** のマスク設定制御

Data**:** レジスタ値が格納されるポインターポイント。UDC2 レジスタは uint32_t 型の low 16 ビットのみを使用します。

機能:
UDC2 レジスタからのデータを読み出します。
例えば、USB バスからの全セットアップパッケージは下記パラメータを順に呼び出すことで取得できます。

UDC2_BREQ_BMREQTYPE,
UDC2_WVALUE,
UDC2_WINDEX,

UDC2_WLENGTH

戻り値:
なし

13.2.3.8 USBD_WriteUDC2Reg

UDC2 レジスタへのデータ書き込み

関数のプロトタイプ宣言:
void
USBD_WriteUDC2Reg(uint32_t ***Addr***,
const uint32_t ***Data***)

引数:
Addr: UDC2 モジュールのレジスタアドレスを選択します。

- **UDC2_ADDR:** アドレスステートレジスタ
- **UDC2_FRAME:** フレームレジスタ
- **UDC2_COMMAND:** エンドポイント レジスタへのコマンド
- **UDC2_INT:**
  UDC2 割り込み信号とそのマスクビットの INT レジスタ
- **UDC2_INTEP:**
  エンドポイント (EP0 を除く)送受信状態のフラグ.
- **UDC2_INTEP_MASK: DC2_INTEP** の制御マスク
- **UDC2_INTRX0:**
  エンドポイントで受信された Zero-Length データを表示するフラグ
- **UDC2_EP0MAXPACKETSIZE:** EP0_MaxPacketSize レジスタ
- **UDC2_EP0FIFO:** EP0_FIFO レジスタ
- **UDC2_EPxMAXPACKETSIZE (x = 1 to 7):** EPx Max Packet Size レジスタ
- **UDC2_EPxSTATUS (x = 0 to 7):** EPx ステータスレジスタ
- **UDC2_EPxFIFO (x = 1 to 7):** EPx FIFO レジスタ
- **UDC2_INTNAK:**
  エンドポイント (EP0 を除く) の NAK 送信状態を表すフラグ
- **UDC2_INTNAK_MASK: UDC2_INTNAK** の制御マスク設定

Data: UDC2 レジスタに書き込まれるデータ

機能:
データを上記 "***Addr*** " によって指定される UDC2 レジスタに書き込みます。

戻り値:
なし

13.2.3.9 USBD_GetDMACMasterAddr

DMAC マスター動作のアドレスを取得

関数のプロトタイプ宣言:
USBD_DMACAddr
USBD_GetDMACMasterAddr(USBD_MasterMode ***MasterMode***)

引数:
MasterMode: DMAC マスタアドレスタイプを選択します。
- **USBD_MASTER_WRITE:** マスタライト
- **USBD_MASTER_READ:** マスタリード

機能:
“Master Write Start Address”, “Master Write End Address”, “Master Read Start Address”, “Master Read End Address” を含む DMAC マスター動作に関連したアドレスレジスタを取得します。

戻り値:
DMAC マスター動作のアドレス。詳細は、"データ構造"の USBD_DMACAddr を参照してください。

13.2.3.10 USBD_SetDMACMasterAddr

DMAC マスター動作のアドレス設定

関数のプロトタイプ宣言:
void
USBD_SetDMACMasterAddr(USBD_MasterMode ***MasterMode***,
USBD_DMACAddr ***Addr***)

引数:
MasterMode: DMAC マスターアドレスタイプを下記から設定
- **USBD_MASTER_WRITE:** マスタライト
- **USBD_MASTER_READ:** マスタリード

Addr: DMAC マスター動作のアドレス
詳細は、"データ構造"の USBD_DMACAddr を参照してください。

機能:
“Master Write Start Address”, “Master Write End Address”, “Master Read Start Address”, “Master Read End Address”を含む DMAC マスター動作に関連したアドレスレジスタの設定をします。

戻り値:
なし

13.2.3.11 USBD_GetPowerCtrlStatus

USBD 電源検出制御値の取得

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

引数:
なし。

機能:

“USB_Reset”, “PHY_Suspend”, “PHY_Resetb” ビットなどの USBD 電源検出制御 レジスタ値を取得します。

戻り値:
USBD 電源検出制御のステータス。詳細は、"データ構造"の USBD_PowerCtrl を参照してください。

13.2.3.12 USBD_SetPowerCtrl

USBD 電源検出制御

関数のプロトタイプ宣言:
void
USBD_SetPowerCtrl(USBD_PowerCtrl ***PowerCtrl***)

引数:
PowerCtrl: USBD 電源検出制御の設定。詳細は、"データ構造"の USBD_PowerCtrl を参照してください。

機能:
“USB_Reset”, “PHY_Suspend”, “PHY_Resetb” ビットなど USBD 電源検出制御レジスタを設定します。

戻り値:
なし

13.2.3.13 USBD_SetEPCMD

エンドポイントへのコマンド送信

関数のプロトタイプ宣言:
void
USBD_SetEPCMD(USBD_EPx ***EPx***,
USBD_EPCMD ***Cmd***)

引数:
EPx: ターゲットのエンドポイントを選択します。
- **USBD_EP0:** USBD エンドポイント 0
- **USBD_EP1:** USBD エンドポイント 1
- **USBD_EP2:** USBD エンドポイント 2
- **USBD_EP3:** USBD エンドポイント 3
- **USBD_EP4:** USBD エンドポイント 4
- **USBD_EP5:** USBD エンドポイント 5
- **USBD_EP6:** USBD エンドポイント 6
- **USBD_EP7:** USBD エンドポイント 7

***Cmd*:** エンドポイントに送られるコマンドを選択します。
EPx が **USBD_EP0** の場合**:**
- **USBD_CMD_SETUP_FIN :**
  DATA ステージ終了または INT_STATUS_NAK 受信時にコマンドを発行してください。

- **USBD_CMD_EP_RESET:**
  エンドポイントのデータおよびステータスをクリアするコマンドです。
- **USBD_CMD_EP_STALL:**
  エンドポイント のステータスを"Stall"にセットするコマンドです。
- **USBD_CMD_ALL_EP_INVALID:**
  EP0 以外の全エンドポイントのステータスを"Invalid"にセットするコマンドです。
- **USBD_CMD_USB_READY:**
  USB ケーブルへの接続をするためのコマンドです。ケーブルに接続されたことを確認後、ホストとの通信が可能になった時点でこのコマンドを発行して下さい。
- **USBD_CMD_SETUP_RECEIVED:**
  Control 転送の SETUP-Stage を認識したことを UDC2 へ知らせるためのコマンドです。INT_SETUP 割込みを受け付けて、リクエストコードを認識した後にこのコマンドを発行して下さい。
- **USBD_CMD_EP_EOP:**
  送信データ書込み終了を UDC2 へ知らせるためのコマンドです。最大転送バイト数よりも少ないバイト数を送信したい場合、このコマンドを発行して下さい。
- **USBD_CMD_EP_FIFO_CLEAR:**
  エンドポイントのデータを削除します。
- **USBD_CMD_EP_TX_0DATA:**
  エンドポイントに Zero-Length データをセットするコマンドです。Zero-Length データを送信したい場合、このコマンドを発行して下さい。

***EPx* が USBD_EPy (y = 1 ～ 7) の場合:**

- **USBD_CMD_SET_DATA0:**
  エンドポイントのトグルをクリアするコマンドです。通常の転送時のトグル更新は UDC2 により自動的に行われますが、ソフトからクリアする必要がある場合はこのコマンドを発行して下さい。
- **USBD_CMD_EP_RESET:**
  エンドポイントのデータおよびステータスをクリアするコマンドです。
- **USBD_CMD_EP_STALL:**
  エンドポイントのステータスを"Stall"にセットするコマンドです。
- **USBD_CMD_EP_INVALID:**
  エンドポイントのステータスを"Invalid"にセットするコマンドです。
- **USBD_CMD_EP_DISABLE:**
  エンドポイントを Disable にするコマンドです。
- **USBD_CMD_EP_ENABLE:**
  エンドポイントを Enable にするコマンドです。
- **USBD_CMD_ALL_EP_INVALID:**
  EP0 以外の全 EP のステータスを"Invalid"にセットするコマンドです。
- **USBD_CMD_EP_EOP:**
  送信データ書込み終了を UDC2 へ知らせるためのコマンドです。最大転送バイト数よりも少ないバイト数を送信したい場合、このコマンドを発行して下さい。
- **USBD_CMD_EP_FIFO_CLEAR:**
  エンドポイントのデータをクリアするコマンドです。
- **USBD_CMD_EP_TX_0DATA:**
  エンドポイントに Zero-Length データをセットするコマンドです。。Zero-Length データを送信したい場合、このコマンドを発行して下さい。

機能:
エンドポイントにコマンドを送信します。

戻り値:
なし

13.2.3.14 USBD_GetEP0Status

EP0 ステータスの読み出し

関数のプロトタイプ宣言:
USBD_EP0Status
USBD_GetEP0Status(void)

引数:
なし。

機能:
Endpoint0 の現在のステータスを読み出します。
“Ready”, “Busy”, “Error”, “Stall” などの EP0 ステータスです。「EP0_FIFO 書き込み可」という情報も読み出します。

戻り値:
Endpoint0 ステータス。詳細は、USBD_EP0Status の"データ構造"を参照ください。

13.2.3.15 USBD_GetEPxStatus

EPx ステータスの読み出し (x = 1～ 7)

関数のプロトタイプ宣言:
USBD_EPxStatus
USBD_GetEPxStatus(USBD_EPx ***EPx***)

引数:
EPx: ターゲットのエンドポイントを選択します。

- **USBD_EP1:** USBD エンドポイント 1
- **USBD_EP2:** USBD エンドポイント 2
- **USBD_EP3:** USBD エンドポイント 3
- **USBD_EP4:** USBD エンドポイント 4
- **USBD_EP5:** USBD エンドポイント 5
- **USBD_EP6:** USBD エンドポイント 6
- **USBD_EP7:** USBD エンドポイント 7

機能:
エンドポイント x (x = 1～7)のステータスを読み出します。
“Ready”, “Busy”, “Error”, “Stall” などの EPx ステータスです。EPx の転送方向、転送モードも取得可能です。

戻り値:
エンドポイント x (x = 1 から 7)のステータス。詳細は、"データ構造"の USBD_EPxStatus を参照してください。

13.2.3.16 USBD_ConfigEPx

EPx (x = 1 ～7) 設定

関数のプロトタイプ宣言:
void
USBD_ConfigEPx(USBD_EPx ***EPx***,
USBD_EPxConfig ***Config***)

引数:
EPx: ターゲットのエンドポイントを選択します。
- **USBD_EP1:** USBD エンドポイント 1
- **USBD_EP2:** USBD エンドポイント 2
- **USBD_EP3:** USBD エンドポイント 3
- **USBD_EP4:** USBD エンドポイント 4
- **USBD_EP5:** USBD エンドポイント 5
- **USBD_EP6:** USBD エンドポイント 6
- **USBD_EP7:** USBD エンドポイント 7

Config: EPx の設定情報。詳細は、USBD_EPxConfig の"データ構造"を参照してください。

機能:
エンドポイント x (x = 1 ～7)の設定をします。
EPx の転送方向、転送モードを設定することができます。
スタンダードリクエスト設定時に呼び出され、Set_Configuration と Set_Interface が受信されます。

戻り値:
なし

13.2.4 データ構造

13.2.4.1 USBD_DMACAddr

メンバ:

uint32_t
StartAddr : DMAC マスター動作の開始アドレスを選択します。
- **USBD_MW_START_ADDR** : マスタライト転送のスタートアドレス
- **USBD_MR_START_ADDR** : マスタリード転送のスタートアドレス

uint32_t
EndAddr: DMAC マスター動作のエンドアドレスを選択します。
- **USBD_MW_END_ADDR** : マスタライト転送のエンドアドレス
- **USBD_MR_END_ADDR** : マスタリード転送のエンドアドレス

13.2.4.2 USBD_INTStatus

メンバ:

uint32_t
All : 下記すべてのビットの OR 値

Bit

uint32_t
Setup : ***1***
UDC2 の int_setup 信号

uint32_t
Status_NAK : ***1***
UDC2 の int_status_nak 信号

uint32_t
Status : ***1***
UDC2 の int_status 信号

uint32_t
Rx_Zero : ***1***
UDC2 の int_rx_zero 信号

uint32_t
SOF : ***1***
UDC2 の int_sof 信号 signal

uint32_t
EP0 : ***1***
UDC2 の int_ep0 信号

uint32_t
EP : ***1***
UDC2 の int_ep 信号

uint32_t
NAK : ***1***
UDC2 の int_nak 信号

uint32_t
Suspend_Resume : 1
UDC2 の suspend_x 信号が変わるたびに 1 をアサートする
0: ステータス変更なし
1: ステータス変更あり

uint32_t
USB_Reset : ***1***
UDC2 が usb_reset 信号をアサートしたかどうかを表示
0: ビットのクリア後、UDC2 による usb_reset 信号のアサートなし
1: UDC2 による usb_reset 信号のアサートあり

uint32_t
USB_Reset_End : ***1***
UDC2 が usb_reset 信号をデアサートしたかどうかを表示
0: ビットのクリア後、UDC2 による usb_reset 信号のデアサートなし
1: : UDC2 による usb_reset 信号のデアサートあり

uint32_t
Reserved1 : ***6***

6ビットのギャップを未使用エリアに配置します。未検出の場合、ゼロが書かれます。

uint32_t
MW_Set_Add : 1
マスタライト転送がディセーブル状態で、該当する Rx 用 EP にマスタライト転送されるべきデータがセットされると 1 にセットされます.
0: 未検出
1: マスタライト転送アドレス要求

uint32_t
MW_End_Add : 1
マスタライト転送が終了した際に、1にセットされます。
0: 未検出
1: マスタライト転送終了

uint32_t
MW_TimeOut : 1
マスターライト転送動作中に、タイムアウトした場合、本ステータスが1にセットされます。
0:未検出
1: マスタライト転送タイムアウト

uint32_t
MW_AHBErr : 1
マスタライト転送動作中に、AHB エラーが発生した場合、本ステータスが1 にセットされます. この割り込み発生後は、DMAC 設定レジスタのmw_resetによりマスタライト転送ブロックをリセットする必要があります。
0: 未検出
1: AHB エラー発生

uint32_t
MR_End_Add : 1
マスタリード転送が終了した際に、1 にセットされます。
0: 未検出
1: マスターリード転送終了

uint32_t
MR_EP_DSet : 1
マスタリード時で使用する、UDC2 Tx 用エンドポイントのFIFO がライト可能(Full ではない状態)となった時に、1 にセットされます。
0: FIFO ライト不可
1: FIFO ライト可

uint32_t
MR_AHBErr : 1
マスタライト転送動作中に、AHB エラーが発生した場合、本ステータスが1 にセットされます。この割り込み発生後は、DMAC 設定レジスタのmr_resetビットによりマスタライト転送ブロックをリセットする必要があります。
0: 未検出
1: AHB エラー発生

uint32_t
UDC2_Reg_Read : 1
UDC2 リード要求レジスタの設定により実行された UDC2 アクセスが完了して、UDC2 リード値レジスタに読み出した値が設定された時、1 がセットされます。また、DC2 内部レジスタへのライトアクセスが完了した時に、1 にセットされます。
0: 未検出
1: レジスタリード/ライト完了

uint32_t
DMAC_Reg_Read : 1
DMAC 設定レジスタの設定により実行されたレジスタアクセスが完了して、DMAC リードレジスタに読み出した値がセットされたときに、1 にセットされます。
0: 未検出
1: レジスタリード完了

uint32_t
Reserved2 : 3
3 ビットのギャップを未使用エリアに配置します。未検出の場合、ゼロが書かれます。

uint32_t
MW_ReadError : 1
共通バスアクセスの設定中にエンドポイントへのアクセスによりマスタライト転送が開始された場合、1 が設定されます。(EPx_Status レジスタの bus_sel ビットは 0)
0: 未検出
1: マスタライトにエンドポイントリードエラー発生

uint32_t
Reserved3 : 2
2 ビットのギャップを未使用エリアに配置します。未検出の場合、ゼロが書かれます。

13.2.4.3 USBD_DMACConfig

メンバ:

uint32_t
All : 下記すべてのビットの OR 値

Bit
uint32_t
MW_Enable : 1
マスタライト転送を制御します。転送アドレスのセット完了時にイネーブル　にして下さい。マスタ転送の終了とともに、自動的にディセーブルされます。本レジスタではマスタライト動作のディセーブルを行うことはできませんのでマスタライト転送を停止させる際は、<mw_abort>を使用して下さい。
0: 禁止
1: 許可

uint32_t
MW_Abort : 1

マスタライト転送を制御します。本ビットに 1 をセットすることによりマスタライト動作を停止させることができます。転送途中にアボートした場合、UDC2 からマスタライト用バッファへの転送を中断して<mw_enable>がクリアされ、マスタライト転送は停止されます。

0: ノーオペレーション
1: アボート

uint32_t
MW_Reset : 1
マスタライト転送ブロックを初期化します。ただしエンドポイントの FIFO は初期化されませんので、本リセットとは別に UDC2 のコマンドレジスタへアクセスして、対応するエンドポイントの初期化を行う必要があります。
本リセットはマスタ動作を停止させてから使用して下さい。
本ビットを 1 へセット後、自動的に 0 にクリアされます。クリアされるまで次のマスタライト転送を行わないで下さい。

0: ノーオペレーション
1: リセット

uint32_t
Reserved1 : 1
1 ビットのギャップを未使用エリアに配置します。未検出の場合、ゼロが書かれます。

uint32_t
MR_Enable : 1
マスタリード転送を制御します。転送アドレスのセット完了時にイネーブルにして下さい。マスタ転送の終了とともに、自動的にディセーブルされます。本レジスタではマスタリード動作のディセーブルを行うことはできませんので、マスタリード転送を停止させる際は<mr_abort>を使用して下さい。

0: 禁止
1: 許可

uint32_t
MR_Abort : 1
マスタリード転送を制御します。本ビットに 1 をセットすることによりマスタリード動作を停止させることができます。
転送途中にアボートした場合、マスタリード用バッファの UDC2 への転送を中断し<mr_enable>がクリアされ、マスタリード転送は停止されます。

0: ノーオペレーション
1: アボート

uint32_t
MR_Reset : 1
マスタライト転送ブロックを初期化します。ただしエンドポイントの FIFO は初期化されませんので、本リセットとは別に UDC2 の UDFS2CMD へアクセスして、対応するエンドポイントの初期化を行う必要があります。
本リセットはマスタ動作を停止させてから使用して下さい。
本ビットを 1 へセット後、自動的に 0 にクリアされます。クリアされるまで次のマスタライト転送を行わないで下さい。

0: ノーオペレーション
1: リセット

uint32_t
Reserved2 : 1
1ビットのギャップを未使用エリアに配置します。未検出の場合、ゼロが書かれます。

uint32_t
M_Burst_Type : 1
マスタライト/リード転送時のバースト転送実行時の HBURST[2:0]のタイプを選択します。UDC2AB が行うバースト転送のタイプは INCR8(8 ビート インクリメント式バースト)となります。従って、通常は初期値で
ある 0 を設定して下さい。但し、システムの AHB 仕様によりバースト転送のタイプとして INCR しか使用できない場合には、このビットに 1 を設定して下さい。この場合、UDC2AB は 8 ビートの INCR 転送を実行し
ます。なお、バースト転送のビート数を変更することはできません。
このビットの設定は UDC2AB への初期設定にて行って下さい。マスタライト/リード転送を開始してからは変更しないで下さい。
注) UDC2AB はマスタライト/リード転送でバースト転送のみを行うわけではなく、バースト転送とシングル転送を組み合わせて転送します。このビットはあくまでバースト転送実行時にのみ影響します。
0: INCR8
1: INCR

uint32_t
Reserved3 : 23
23 ビットのギャップを未使用エリアに配置します。未検出の場合、ゼロが書かれます。

13.2.4.4 USBD_DMACStatus

メンバ:

uint32_t
All : 下記すべてのビットの OR 値

Bit
uint32_t
MW_EP_DSet*: *1
UDC2 の Rx 用エンドポイント へ受信データがセットされると 1 にセットされます。全データがマスタライト用 DMA により読み出されると 0 になります。
0: Endpoint 内にデータはありません
1: Endpoint 内に読み出すべきデータがあります

uint32_t
MR_EP_DSet*: *1
マスタリード DMA 転送により、UDC2 の Tx 用エンドポイントへ送信データがセットされ、Endpoint に書き込むスペースがなくなると 1 にセットされます。ホストからの IN-Token により UDC2 からデータが転送されると 0 になります。このビットが 0 であるときは Endpoint への DMA 転送が可能です。(本ビットは eptx_dataset 入力信号を CLK_H 同期したものです。)
0: Endpoint 内にデータを転送可能です。
1: Endpoint 内にデータを転送するスペースがありません。

uint32_t
MW_Buf_Empty*: *1
UDC2AB のマスタライト DMA 用バッファの空きを示します。
0: バッファにデータがあります。
1: バッファにデータはありません。

uint32_t
MR_Buf_Empty*: *1
UDC2AB のマスタリード DMA 用バッファの空きを示します。
0: バッファにデータがあります。
1: バッファにデータはありません。

uint32_t
MR_EP_Empty*: *1
UDC2Rx のエンドポイントに空きがあるかどうかを示します。UDC2 設定レジスタの tx0 ビットを使用しての NULL パケット送信時に 1 にセットされます。(本ビットは eptx_dataset 入力信号を CLK_H 同期したものです。)
0: Endpoint 内にデータがあります
1: Endpoint 内にデータがありません

uint32_t
Reserved*: *27
27 ビットのギャップを未使用エリアに配置します。未検出の場合、ゼロが書かれます。

13.2.4.5 USBD_PowerCtrl

メンバ:

uint32_t
All : 下記すべてのビットの OR 値

Bit
uint32_t
USB_Reset* : *1
UDC2 からの usb_reset 信号を同期した値 (リードのみ)
0: usb_reset = 0
1: usb_reset = 1

uint32_t
PW_Resetb* : *1
UDC2AB 用のソフトウエアリセットです。本ビットを 0 にセットすることで、PW_RESETB 出力端子が 0 にアサートされます。
マスタ動作が停止した状態でリセットを行って下さい。
このビットは自動解除されませんので、必ずクリアして下さい。
0: リセットアサート
1: リセットデアサート

uint32_t
PW_Detect* : *1
本デバイスではサポートされていないため、常に 0 (リードのみ)

uint32_t

PHY_Suspend*: *1
本ビットを 1 にセットすることで、PHYSUSPEND 出力信号が 0 へアサート(CLK_H 同期)されます。PHY をサスペンドする時の端子として使用可能です。
本ビットを 1 にセットすると、UDC2 レジスタと DMAC Read Request レジスタへのアクセスが禁止となります。
レジューム時(UDC2 の suspend_x デアサート時)に自動的に 0 にクリアされます。
0: 非サスペンド状態
1: サスペンド状態

uint32_t
Suspend_x*: *1
サスペンド信号を検出します(UDC2 からの suspend_x 信号を同期化した値です)。(リードのみ)
0: サスペンド状態 (suspend_x = 0)
1: 非サスペンド状態 (suspend_x = 1)

uint32_t
PHY_Resetb*: *1
本ビットを 0 にセットすることで、PW_RESETB 出力端子が 0 にアサートされます。PHYRESET 信号は PHY のリセットに使用されます。自動的に解除されませんので、PHY のリセット時間経過後に必ずクリアしてください。
0: リセットアサート
1: リセットデアサート

uint32_t
PHY_Remote_Wakeup*: *1
USB のリモートウェイクアップ機能を実行するために使用します。本ビットに 1 をセットすることで、udc2_wakeup 出力信号(UDC2 の wakeup 入力端子)を 1 にアサートすることができます。但し、UDC2 がサスペンドを検出していない時(<suspend_x>＝ 1 の時)に本ビットを 1 にセットした場合は無視されます(1 にセットされません)ので、サスペンド検出時にのみセットして下さい。USB レジューム完了時(<suspend_x>デアサート時)に自動的に 0 にクリアされます。
0: ノーオペレーション
1: ウェイクアップ

uint32_t
Wakeup_En*: *1
本デバイスではサポートされていません。

uint32_t
Reserved*: *24
24 ビットのギャップを未使用エリアに配置します。未検出の場合、ゼロが書かれます。

13.2.4.6 USBD_EP0Status

***メンバ*:**

uint32_t
All : 下記すべてのビットの OR 値

Bit

uint32_t
Reserved1*:** ***9
9 ビットのギャップを未使用エリアに配置します。未検出の場合、ゼロが書かれます。

uint32_t
Status*:** ***3
EP0 の現在のステイタスを示します。Setup-Token 受信後、”Ready”にクリアされます。
000: Ready (通常の状態)
001: Busy (STATUS-Stage で"NAK"を受信した際にセットされます)
010: Error (受信データが CRC エラーの場合、およびデータ送信後タイムアウトした際にセットされます)
011: Control-RD 転送において Length 以上のデータを要求された場合に"STALL"を返信し、status がセットされます。また、コマンドレジスタ により"EP0-STALL"を発行した場合もセットされます
100 ～111: Reserved

uint32_t
Toggle*:** ***2
現在の EP0 のトグル値を示します。
00: DATA0
01: DATA1
10: Reserved
11: Reserved

uint32_t
Reserved2*:** ***1
1 ビットのギャップを未使用エリアに配置します。未検出の場合、ゼロが書かれます。

uint32_t
EP0_Mask*:** ***1
Setup-Token 受信後、1 にセットされます。"Setup_Received"コマンドを発行することにより 0 にクリアされます。この bit が 1 の間は、EP0_FIFO への書込みが行われません。
0: EP0_FIFO への書き込み可
1: EP0_FIFO への書き込み不可

uint32_t
Reserved3*:** ***16
16 ビットのギャップを未使用エリアに配置します。未検出の場合、ゼロが書かれます。

13.2.4.7 USBD_EPxConfig, USBD_EPxStatus

メンバ:

uint32_t
All : 下記すべてのビットの OR 値

Bit
uint32_t
Num_MF* :** ***2

Isochronous 転送を選択した場合、μフレーム中に何回転送をするかを設定します。
00: 1-transaction
01: 2-transaction
10: 3-transaction
11: Reserved

uint32_t
T_Type **:** ***2***
このエンドポイント の転送モードを設定します。
00: Control
01: Isochronous
10: Bulk
11: Interrupt

uint32_t
Reserved1 **:** ***3***
3 ビットのギャップを未使用エリアに配置します。未検出の場合、ゼロが書かれます。

uint32_t
Dir **:** ***1***
このエンドポイント に対する転送方向を設定します
0: OUT (Host-to-device)
1: IN (Device-to-host)
注:
エンドポイント 1,3,5,7 は必ず ‘1’に設定 (IN のみ)
エンドポイント 2,4,6 は必ず‘0’に設定 (OUT のみ)

uint32_t
Disable **:** ***1***
EPx の転送許可状態を示します。"禁止"状態にある場合、このエンドポイント に対する Token に対しては"NAK"を返信し続けます。
0: 許可
1: 禁止

uint32_t
Status **:** ***3***
現在の EPx の状態を示します。UDFS2CMD より EP_Reset を発行することにより status は"Ready"となります。
000: Ready (通常の状態)
001: Reserved
010: エラー(データパケットに受信エラーが発生した時、または送信後タイムアウトが発生した時にセットされます。但し、"Stall"、"Invalid"がセットされている場合にはセットされません)
011: Stall (Command register により"EP_Stall"を発行した場合にセットされます)
100 ～110: Reserved

uint32_t
Toggle **:** ***2***
現在の EPx のトグル値を示します。
00: DATA0
01: DATA1
10: DATA2

11: MDATA

uint32_t
Bus_Sel **:** ***1***
EP1 の FIFO へのアクセスをするバスを選択します。
0: 共通バスアクセス
1: 直接アクセス

uint32_t
Pkt_Mode **:** ***1***
EPx のパケットモードを選択します。Dual モードを選択することにより、EPx に対する 2 つのパケットデータを保持することが可能となります。
0: Single モード
1: Dual モード

uint32_t
Reserved2**:** ***16***
16 ビットのギャップを未使用エリアに配置します。未検出の場合、ゼロが書かれます。

14. WDT

14.1 概要

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

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

本ドライバは、以下のファイルで構成されています。
\Libraries\TX03_Periph_Driver\src\tmpm366_wdt.c
\Libraries/TX03_Periph_Driver\inc\tmpm366_wdt.h

14.2 API 関数

14.2.1 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)

14.2.2 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(), DT_SetOverflowOutput(), WDT_Init(), WDT_Enable(), WDT_Disable(), WDT_WriteClearCode()
2) IDLE モード時の開始・停止:
   WDT_SetIdleMode()

14.2.3 関数仕様

14.2.3.1 WDT_SetDetectTime

WDT 検出時間の設定

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

引数:
***DetectTime*:** 以下から検出時間を選択します。

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

- **WDT_DETECT_TIME_EXP_23:** 2^23/fsys
- **WDT_DETECT_TIME_EXP_25:** 2^25/fsys

機能:
WDT の検出時間を設定します。

戻り値:
なし

14.2.3.2 WDT_SetIdleMode

IDLE 時の動作選択

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

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

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

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

戻り値:
なし

14.2.3.3 WDT_SetOverflowOutput

暴走検出後の動作選択

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

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

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

戻り値:
なし

14.2.3.4 WDT_Init

WDT の初期化

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

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

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

戻り値:
なし

14.2.3.5 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

14.2.3.6 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

14.2.3.7 WDT_WriteClearCode

クリアコードの書き込み

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

引数:
なし

機能:
クリアコードをライトします。

戻り値:
なし

14.2.4 データ構造

14.2.4.1 WDT_InitTypeDef

メンバ:

uint32_t
DetectTime 以下から検出時間を選択します。

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

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

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

15. FUART

15.1 概要

本デバイスは非同期のシリアルチャネル (FullUART)とモデム制御を内蔵します。

FUART ドライバー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/TX03_Periph_Driver/src/tmpm366_fuart.c
/Libraries/TX03_Periph_Driver/inc/tmpm366_fuart.h

15.2 API 関数

15.2.1 関数一覧

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

- FUART_AllModemStatus FUART_GetModemStatus(
  TSB_FUART_TypeDef * ***FUARTx***)
- void FUART_SetRTSStatus(
  TSB_FUART_TypeDef * ***FUARTx***, FUART_ModemStatus ***Status***)
- void FUART_SetDTRStatus(
  TSB_FUART_TypeDef * ***FUARTx***, FUART_ModemStatus ***Status***)

15.2.2 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 の初期化と共通動作
   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) 割り込み設定、割り込みステータスとクリア
   FUART_SetINTMask(), FUART_GetINTMask(), FUART_GetRawINTStatus(), FUART_GetMaskedINTStatus, FUART_ClearINT()
4) モデム制御
   FUART_GetModemStatus(), FUART_SetRTSStatus(), FUART_SetDTRStatus()
5) IrDA 設定
   FUART_EnableIrDA(), FUART_DisableIrDA(), FUART_SetIrDAEncodeMode, FUART_SetIrDADivisor()

15.2.3 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*** を指定してください。

15.2.3.1 FUART_Enable

Full UART チャネルの有効化

関数のプロトタイプ宣言:
void
FUART_Enable(TSB_FUART_TypeDef * ***FUARTx***)

引数:
FUARTx: Full UART チャネルを指定します。

機能:
Full UART チャネルを有効にします。

戻り値:
なし

15.2.3.2 FUART_Disable

Full UART チャネルの無効化

関数のプロトタイプ宣言:

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

引数:
FUARTx: Full UART チャネルを指定します。

機能:
Full UART チャネルを無効にします。

戻り値:
なし

15.2.3.3 FUART_GetRxData

受信データの取得

関数のプロトタイプ宣言:
uint32_t
FUART_GetRxData(TSB_FUART_TypeDef * ***FUARTx***)

引数:
FUARTx: Full UART チャネルを指定します。

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

戻り値:
受信データ

15.2.3.4 FUART_SetTxData

送信データの設定

関数のプロトタイプ宣言:
void
FUART_SetTxData(TSB_FUART_TypeDef * ***FUARTx***,
uint32_t ***Data***)

引数:
FUARTx: Full UART チャネルを指定します。
Data: 送信データポインタです。データサイズは 0x00 - 0xFF です。

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

戻り値:

なし

15.2.3.5 FUART_GetErrStatus

受信エラーステータスの取得

関数のプロトタイプ宣言:
FUART_Err
FUART_GetErrStatus(TSB_FUART_TypeDef * ***FUARTx***)

引数:
FUARTx: Full UART チャネルを指定します。

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

戻り値:
FUART_Err: 受信エラーステータス。(詳細は“データ構成説明”を参照)

15.2.3.6 FUART_ClearErrStatus

受信エラーステータスのクリア

関数のプロトタイプ宣言:
void
FUART_ClearErrStatus(TSB_FUART_TypeDef * ***FUARTx***)

引数:
***FUARTx*:** Full UART チャネルを指定します。

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

戻り値:
なし

15.2.3.7 FUART_GetBusyState

データ送信状態の取得

関数のプロトタイプ宣言:
WorkState
FUART_GetBusyState(TSB_FUART_TypeDef * ***FUARTx***)

引数:
FUARTx: Full UART チャネルを指定します。

機能:

データ送信中であるか停止中であるか状態を取得します。

戻り値:
BUSY: データ送信中
DONE: データ送信が停止中

15.2.3.8 FUART_GetStorageStatus

送受信 FIFO または送受信保持レジスタの取得

関数のプロトタイプ宣言:
FUART_StorageStatus
FUART_GetStorageStatus(TSB_FUART_TypeDef * ***FUARTx***,
FUART_Direction ***Direction***)

引数:
FUARTx: Full UART チャネルを指定します。
Direction: 送信または受信のどちらかを選択します。
- **FUART_RX**: 受信 FIFO または受信保持レジスタ
- **FUART_TX**: 送信 FIFO または送信保持レジスタ

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

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

15.2.3.9 FUART_SetIrDADivisor

IrDA 低電力除数の設定

関数のプロトタイプ宣言:
void
FUART_SetIrDADivisor(TSB_FUART_TypeDef * ***FUARTx***,
uint32_t ***Divisor***)

引数:
FUARTx: Full UART チャネルを指定します。
Divisor: IrDA 低電力除数を設定します。(設定可能範囲: 0x01 - 0xFF)

機能:
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。
IrDA 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

15.2.3.10 FUART_Init

Full UART チャネルの設定

関数のプロトタイプ宣言:
void
FUART_Init(TSB_FUART_TypeDef * ***FUARTx***,
FUART_InitTypeDef * ***InitStruct***)

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

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

戻り値:
なし

15.2.3.11 FUART_EnableFIFO

送受信 FIFO の有効化

関数のプロトタイプ宣言:
void
FUART_EnableFIFO(TSB_FUART_TypeDef * ***FUARTx***)

引数:
FUARTx: Full UART チャネルを指定します。

機能:
送受信 FIFO を許可します。

戻り値:
なし

15.2.3.12 FUART_DisableFIFO

送受信 FIFO の無効化

関数のプロトタイプ宣言:
FUART_DisableFIFO(TSB_FUART_TypeDef * ***FUARTx***)

引数:
FUARTx: Full UART チャネルを指定します。

機能:
送受信 FIFO を禁止します。FIFO は 1 段の保持レジスタとなります。

戻り値:
なし

15.2.3.13 FUART_SetSendBreak

ブレーク付き送信の選択

関数のプロトタイプ宣言:
void
FUART_SetSendBreak(TSB_FUART_TypeDef * ***FUARTx***,
FunctionalState ***NewState***)

引数:
FUARTx: Full UART チャネルを指定します。
NewState: ブレーク付き送信の許可/禁止を選択します。
- **ENABLE**: ブレーク送信する。
- **DISABLE**: ブレーク送信しない。

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

戻り値:
なし

15.2.3.14 FUART_SetIrDAEncodeMode

0 ビット転送用 IrDA エンコードモード選択

関数のプロトタイプ宣言:
void
FUART_SetIrDAEncodeMode(TSB_FUART_TypeDef * ***FUARTx***,
uint32_t ***Mode***)

引数:
FUARTx: Full UART チャネルを指定します。
Mode: 0 ビット転送用 IrDA エンコーディングモードを選択します。
- **FUART_IRDA_3_16_BIT_PERIOD_MODE**: ビット区間の 3/16 の High アクティブパルスとして 0 ビットが転送されます。
- **FUART_IRDA_3_TIMES_IRLPBAUD16_MODE**: IrLPBaud16 入力信号の 3 倍のパルス幅で 0 ビットが転送されます。

機能:
IrDA エンコーディングモードを選択します。IrDA エンコーディングモードを **FUART_IRDA_3_TIMES_IRLPBAUD16_MODE** へ変更すると、消費電力の削減が可能ですが、転送距離が低下する場合があります。

戻り値:
なし

15.2.3.15 FUART_EnableIrDA

IrDA 回路イネーブル

関数のプロトタイプ宣言:
Result
FUART_EnableIrDA(TSB_FUART_TypeDef * ***FUARTx***)

引数:
FUARTx: Full UART チャネルを指定します。

機能:
Full UART がイネーブルの場合、本 API は IrDA 回路をイネーブルします。Full UART がディゼーブルの場合、本 API は何も行わずエラーを返します。

戻り値:
SUCCESS: IrDA イネーブルに成功
ERROR: IrDA イネーブルに失敗 (Full UART チャネルがディゼーブルのため)

15.2.3.16 FUART_DisableIrDA

IrDA 回路ディゼーブル

関数のプロトタイプ宣言:
void
FUART_DisableIrDA(TSB_FUART_TypeDef * ***FUARTx***)

引数:
FUARTx: Full UART チャネルを指定します。

機能:
Full UART がイネーブルの場合、本 API は IrDA 回路をディゼーブルにします。Full UART がディゼーブルの場合、本 API は何も行いません。また IrDA 回路は設定どおりに動作しません。

戻り値:
なし

15.2.3.17 FUART_SetINTFIFOLevel

送受信割り込み FIFO レベルの選択

関数のプロトタイプ宣言:
void
FUART_SetINTFIFOLevel(TSB_FUART_TypeDef * ***FUARTx***,
uint32_t ***RxLevel***,
uint32_t ***TxLevel***)

引数:
FUARTx: Full UART チャネルを指定します。
RxLevel: 受信割り込み FIFO レベルを選択します。(受信 FIFO は 32 段です)

- **FUART_RX_FIFO_LEVEL_4**: 受信 FIFO が 4 バイト以上

- **FUART_RX_FIFO_LEVEL_8**: 受信 FIFO が 8 バイト以上
- **FUART_RX_FIFO_LEVEL_16**: 受信 FIFO が 16 バイト以上
- **FUART_RX_FIFO_LEVEL_24**: 受信 FIFO が 24 バイト以上
- **FUART_RX_FIFO_LEVEL_28**: 受信 FIFO が 28 バイト以上

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

- **FUART_TX_FIFO_LEVEL_4**: 送信 FIFO が 4 バイト以上
- **FUART_TX_FIFO_LEVEL_8**: 送信 FIFO が 8 バイト以上
- **FUART_TX_FIFO_LEVEL_16**: 送信 FIFO が 16 バイト以上
- **FUART_TX_FIFO_LEVEL_24**: 送信 FIFO が 24 バイト以上
- **FUART_TX_FIFO_LEVEL_28**: 送信 FIFO が 28 バイト以上

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

戻り値:
なし

15.2.3.18 FUART_SetINTMask

割り込み発生要因の設定

関数のプロトタイプ宣言:
void
FUART_SetINTMask(TSB_FUART_TypeDef * ***FUARTx***,
uint32_t ***IntMaskSrc***)

引数:
FUARTx: Full UART チャネルを指定します。
IntMaskSrc: 割り込み発生要因を選択します。

- **FUART_NONE_INT_MASK**: すべての割り込みを禁止します。
- **FUART_RIN_MODEM_INT_MASK**: RIN モデム割り込みを許可します。
- **FUART_CTS_MODEM_INT_MASK**: CTS モデム割り込みを許可します。
- **FUART_DCD_MODEM_INT_MASK**: DCD モデム割り込みを許可します。
- **FUART_DSR_MODEM_INT_MASK**: DSR モデム割り込みを許可します。
- **FUART_RX_FIFO_INT_MASK**: 受信割り込みを許可します。
- **FUART_TX_FIFO_INT_MASK**: 送信割り込みを許可します。
- **FUART_RX_TIMEOUT_INT_MASK**: 受信タイムアウト割り込みを許可します。
- **FUART_FRAMING_ERR_INT_MASK**: フレーミングエラー割り込みを許可します。
- **FUART_PARITY_ERR_INT_MASK**: パリティエラー割り込みを許可します。
- **FUART_BREAK_ERR_INT_MASK**: ブレークエラー割り込みを許可します。
- **FUART_OVERRUN_ERR_INT_MASK**: オーバーランエラー割り込みを許可します。
- **FUART_ALL_INT_MASK**: すべての割り込みを許可します。

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

戻り値:

なし

15.2.3.19 FUART_GetINTMask

割り込み発生要因の取得

関数のプロトタイプ宣言:
FUART_INTStatus
FUART_GetINTMask(TSB_FUART_TypeDef * ***FUARTx***)

引数:
FUARTx: Full UART チャネルを指定します。

機能:
割り込み発生の許可/禁止状態を要因毎に取得します。

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

15.2.3.20 FUART_GetRawINTStatus

割り込み許可/禁止設定前の割り込みステータスの取得

関数のプロトタイプ宣言:
FUART_INTStatus
FUART_GetRawINTStatus(TSB_FUART_TypeDef * ***FUARTx***)

引数:
FUARTx: Full UART チャネルを指定します。

機能:
割り込み許可/禁止設定前の割り込みステータスを取得します。

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

15.2.3.21 FUART_GetMaskedINTStatus

割り込み許可/禁止設定後の割り込みステータスの取得

関数のプロトタイプ宣言:
FUART_INTStatus
FUART_GetMaskedINTStatus(TSB_FUART_TypeDef * ***FUARTx***)

引数:
FUARTx: Full UART チャネルを指定します。

機能:
割り込み許可/禁止設定後の割り込みステータスを取得します。

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

15.2.3.22 FUART_ClearINT

割り込み要因のクリア

関数のプロトタイプ宣言:
void
FUART_ClearINT(TSB_FUART_TypeDef * ***FUARTx***,
FUART_INTStatus ***INTStatus***)

引数:
FUARTx: Full UART チャネルを指定します。
INTStatus: クリア対象の割り込み要因を格納してください。(詳細は“データ構成説明”を参照)

機能:
割り込み要因をクリアします。

戻り値:
なし

15.2.3.23 FUART_SetDMAOnErr

DMA オンエラーの許可/禁止選択

関数のプロトタイプ宣言:
void
FUART_SetDMAOnErr(TSB_FUART_TypeDef * ***FUARTx***,
FunctionalState ***NewState***)

引数:
FUARTx: Full UART チャネルを指定します。
NewState: DMA オンエラーの許可/禁止を選択します。
- **ENABLE**: 許可。
- **DISABLE**: 禁止。

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

戻り値:
なし

15.2.3.24 FUART_SetFIFODMA

送受信 DMA の許可/禁止選択

関数のプロトタイプ宣言:
void
FUART_SetFIFODMA(TSB_FUART_TypeDef * ***FUARTx***,
FUART_Direction ***Direction***,
FunctionalState ***NewState***)

引数:
FUARTx: Full UART チャネルを指定します。
Direction: 送信または受信のどちらかを選択します。
- **FUART_RX**: 受信 FIFO または受信保持レジスタ
- **FUART_TX**: 送信 FIFO または送信保持レジスタ

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

機能:
送信 DMA または受信 DMA の許可/禁止を選択します。
DMAC を用いた送信/受信 FIFO のデータ転送の場合、バス幅を 8bit に設定してください。

戻り値:
なし

15.2.3.25 FUART_GetModemStatus

モデム状態の取得

関数のプロトタイプ宣言:
FUART_AllModemStatus
FUART_GetModemStatus(TSB_FUART_TypeDef * ***FUARTx***)

引数:
FUARTx: Full UART チャネルを指定します。

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

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

15.2.3.26 FUART_SetRTSStatus

データ送信要求 (RTS)の設定

関数のプロトタイプ宣言:
void
FUART_SetRTSStatus(TSB_FUART_TypeDef * ***FUARTx***,
FUART_ModemStatus ***Status***)

引数:
FUARTx: Full UART チャネルを指定します。

Status: RTS 出力の補数を選択します。
- **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:
データ送信要求 (RTS)のモデムステータス出力を設定します。

戻り値:
なし

15.2.3.27 FUART_SetDTRStatus

データ送信準備完了(DTR)の設定

関数のプロトタイプ宣言:
void
FUART_SetDTRStatus(TSB_FUART_TypeDef * ***FUARTx***,
FUART_ModemStatus ***Status***)

引数:
FUARTx: Full UART チャネルを指定します。
Status: DTR 出力の補数を選択します。
- **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:
データ送信準備送信準備完了(DTR)のモデムステータス出力を設定します。

戻り値:
なし

15.2.4 データ構造

15.2.4.1 FUART_InitTypeDef

メンバ:

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

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

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

uint32_t

Parity : パリティ状態を設定します。

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

uint32_t
Mode: 受信、送信あるいは両方の許可/禁止を設定します。

- **UART_ENABLE_TX** : 送信許可
- **UART_ENABLE_RX** : 受信許可
- **UART_ENABLE_TX | UART_ENABLE_RX** : 送受信許可

uint32_t
FlowCtrl: ハードウェアフロー制御を設定します。

- **UART_NONE_FLOW_CTRL:** フロー制御なし
- **UART_CTS_FLOW_CTRL:** CTS フロー制御許可
- **UART_RTS_FLOW_CTRL:** RTS フロー制御許可
- **UART_CTS_FLOW_CTRL | UART_RTS_FLOW_CTRL:** CTS/RTS フロー制御許可

15.2.4.2 FUART_Err

メンバ:
uint32_t
All: Full UART エラー

ビットフィールド:

uint32_t		
Reserved1: 8		未使用
uint32_t		
FramingErr: 1		フレーミングエラー
uint32_t		
ParityErr: 1		パリティエラー
uint32_t		
BreakErr: 1		ブレークエラー
uint32_t		
OverrunErr: 1		オーバーランエラー
uint32_t		
Reserved2: 20		未使用

15.2.4.3 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 受信割り込み
uint32_t
TxFIFO: 1 送信割り込み
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

15.2.4.4 FUART_AllModemStatus

メンバ:
uint32_t
All: Full UART の全てのモデムステータス

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
RIN: 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