TRAVEO™ T2G ボディエントリ ファミリのクロック設定

About this document

Scope and purpose

AN220208 は TRAVEO™ T2G ファミリ CYT2B シリーズの MCU におけるクロックソースと PLL/FLL の設定方法を説明し、PLL/FLL の設定例, ILO の校正方法, 補足情報も提供します。

関連製品ファミリ

TRAVEO™ T2G ファミリ CYT2B シリーズ

Intended audience

本書は、TRAVEO™ T2G ボディエントリ ファミリのクロック設定を使用するユーザーを対象とします。

Table of contents

Table of contents

1 はじめに

ボディコントロールユニットなどの車載システム向けの TRAVEO™ T2G ファミリ MCU は高度な 40 nm プロセスで製造され、FPU 付き Arm® Cortex®-M4 プロセッサをベースにした 32 ビットの車載向けマイクロコントローラです。これら製品は安全なコンピュータ プラットフォームを可能にし、インフィニオンの低消費電力フラッシュメモリと複数の高性能アナログおよびデジタル機能を組み込んでいます。

TRAVEO™ T2G クロックシステムは内部クロックソースと外部クロックソースの両方をサポートし、PLL と FLL を用いた高速クロックもサポートします。また TRAVEO™ T2G クロックシステムは、内部クロックと外部クロックで低速クロックもサポートします。クロックソースは外部発振器も使用でき、TRAVEO™ T2G は主に RTC で使用するクロック入力もサポートします。

TRAVEO™ T2G はクロック動作を監視する機能と各クロックのクロック差を測定する機能もサポートします。

このアプリケーションノートで使用されている機能と用語を理解するためには **architecture technical reference manual (TRM)** の Clocking System の章を参照してください。

このドキュメントでは TRAVEO™ T2G ファミリ MCU は、ボディエントリまたは CYT2B シリーズのことを指します。

2 TRAVEO™ T2G ファミリ MCU のクロックシステム

2.1 クロックシステムの概要

この CYT2B シリーズの MCU のクロックシステムは 2 つのブロックに分割できます。1 つのブロックは外部クロックや内部クロックのクロックリソースを選択し、FLL と PLL を使用してクロックを逓倍します。もう 1 つのブロックはクロックを CPU コアおよび周辺機能に分配および分割します。ただし、クロックリソースから周辺回路に直接接続する RTC のようないくつかの例外はあります。

Figure 1 にクロックシステムの構造の概要を示します。

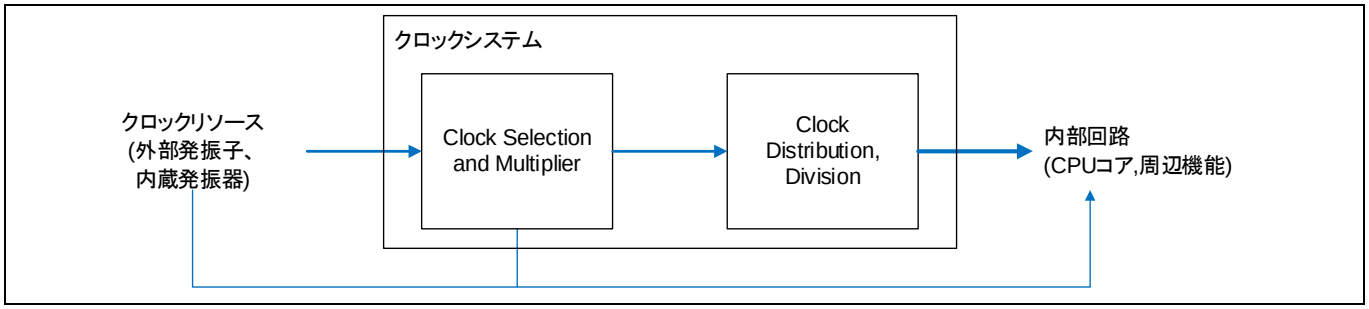

Figure 1 クロックシステム構造の概要

2.2 クロックリソース

内部クロックソースと外部クロックソースの 2 種類のクロックリソースがこの MCU シリーズのクロックシステムに入力されます。内部クロックソースと外部クロックソースにはそれぞれ 3 種類のクロックソースがあります。

- 内部クロックソース
  - IMO: Internal main oscillator (内部主発振器)。IMO は内蔵クロックであり、その周波数は 8 MHz (標準) です。初期設定で IMO は有効です。
  - ILO0: Internal low-speed oscillator 0 (内部低速発振器 0) 。ILO0 は内蔵クロックであり、その周波数は 32.768 kHz (標準) です。初期設定で ILO0 は有効です。
  - ILO1: Internal low-speed oscillator 1 (内部低速発振器 1) 。ILO1 は ILO0 と同様の機能を持っていますが、ILO1 は ILO0 のクロック監視に利用できます。初期設定で ILO1 は無効です。
- 外部クロックソース
  - ECO: External crystal oscillator (外部水晶発振器)。ECO は外部発振子を使用します。入力周波数範囲は 3.988 MHz から 33.34 MHz です。初期設定で ECO は無効です。
  - WCO: Watch crystal oscillator (時計水晶発振器)。WCO は主に RTC で使用されます。32.768 kHz を使用します。初期設定で WCO は無効です。
  - EXT_CLK: 外部クロック。EXT_CLK は 0.25 MHz から 80 MHz の範囲のクロックであり、そのクロックを専用 I/O ピンの信号から供給できます。このクロックは PLL か FLL のソースクロックとして使用することや、高周波クロックとしても直接使用できます。初期設定で EXT_CLK は無効です。

IMO や PLL などの機能や周波数のような数値の詳細については、TRAVEO™ T2G **architecture TRM** および**データシート**を参照してください。

2.3 クロックシステムの機能

クロックシステムの機能について説明します。

TRAVEO™ T2G ファミリ MCU のクロックシステム

Figure 2 に、**Figure 1** に記載されている Clock selection and multiplier ブロックの詳細を示します。このブロックはクロックリソースから CLK_HF0, CLK_HF1 および CLK_HF2 を生成します。CLK_HF0, CLK_HF1 および CLK_HF2 は CYT2B シリーズの MCU を動作するための基本クロックです。また、このブロックはクロックリソースを選択し、高速クロックを生成するために FLL と PLL を選択できます。

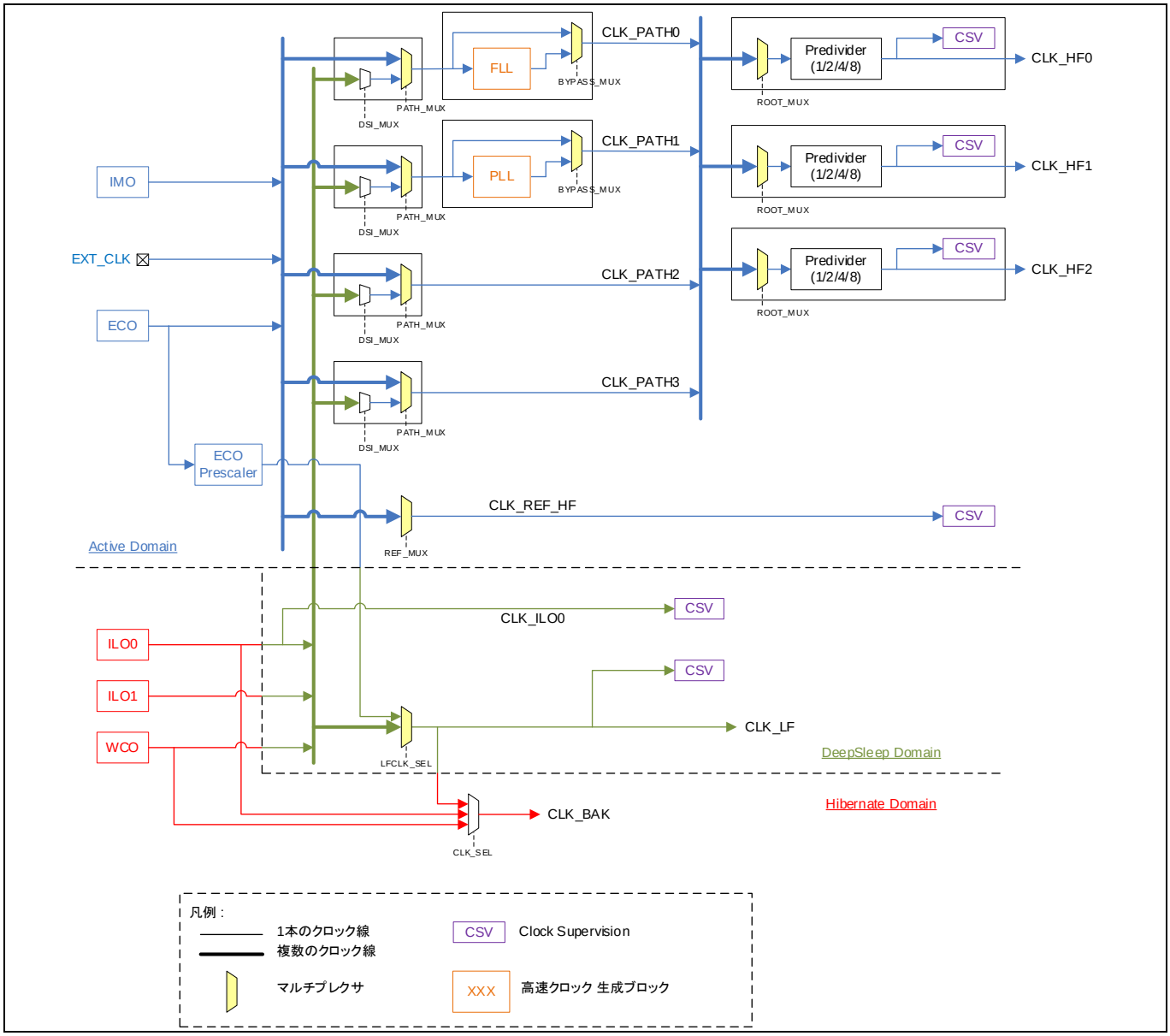

Figure 2　ブロックダイヤグラム

Active domain　Active domain はアクティブパワーモード中のみで動作する領域です。

DeepSleep domain　DeepSleep domain はアクティブモードと DeepSleep モード中のみ動作する領域です。

Hibernate domain　Hibernate domain はすべてのパワーモードで動作する領域です。

ECO prescaler　ECO_Prescaler は ECO を分周し、LFCLK クロックで使用できるクロックを作成します。分周機能には 10 ビット整数分周と 8 ビット分数分周があります。

DSI_MUX　DSI_MUX には ILO0, ILO1 および WCO からクロックを選択する機能があります。

TRAVEO™ T2G ファミリ MCU のクロックシステム

PATH_MUX	PATH_MUX には IMO, ECO, EXT_CLK および DSI_MUX の出力からクロックを選択する機能があります。
CLK_PATH	CLK_PATH0, CLK_PATH1, CLK_PATH2 および CLK_PATH3 は CLK_HF0, CLK_HF1 および CLK_HF2 の入力ソースとして使用されます。
CLK_HF	CLK_HF0, CLK_HF1 および CLK_HF2 は高周波クロックです。
FLL	FLL は高速なクロックを生成できる周波数ロックループです。
PLL	PLL は高速なクロックを生成できる Phase ロックループです。
BYPASS_MUX	BYPASS_MUX には CLK_PATH の出力となるクロックを選択する機能があります。BYPASS_MUX は FLL/PLL の出力を選択するか、それらをバイパスすることができます。
ROOT_MUX	ROOT_MUX には CLK_HFx のクロックソースに対する機能があります。選択できるクロックは CLK_PATH0, CLK_PATH1, CLK_PATH2 および CLK_PATH3 です。
Predivider	Predivider は選択された CLK_PATH を分周するために利用できます。1, 2, 4 および 8 分周が選択できます。
REF_MUX	REF_MUX は CLK_REF_HF のクロックソースを選択します。
CLK_REF_HF	CLK_REF_HF は CLK_HF の CSV を監視します。
LFCLK_SEL	LFCLK_SEL は CLK_LF のクロックソース、または ECO の分周クロックを選択します。
CLK_LF	CLK_LF は MCWDT のソースクロックです。
CLK_SEL	CLK_SEL は RTC に入力するクロックを選択します。
CLK_BAK	CLK_BAK は主に RTC で使用されます。
CSV	CSV は clock supervision であり、クロック動作を監視します。監視できるクロックは CLK_HF, CLK_REF_HF, ILO0 および CLK_LF です。

Figure 3 に CLK_HF0 の分配先を示し、**Figure 1** 中の Clock Distribution, Division ブロックの詳細を示します。

CLK_HF0 は CPU サブシステム (CPUSS) および周辺クロック分周器のルートクロックです。図中の機能については **architecture TRM** を参照してください。

TRAVEO™ T2G ファミリ MCU のクロックシステム

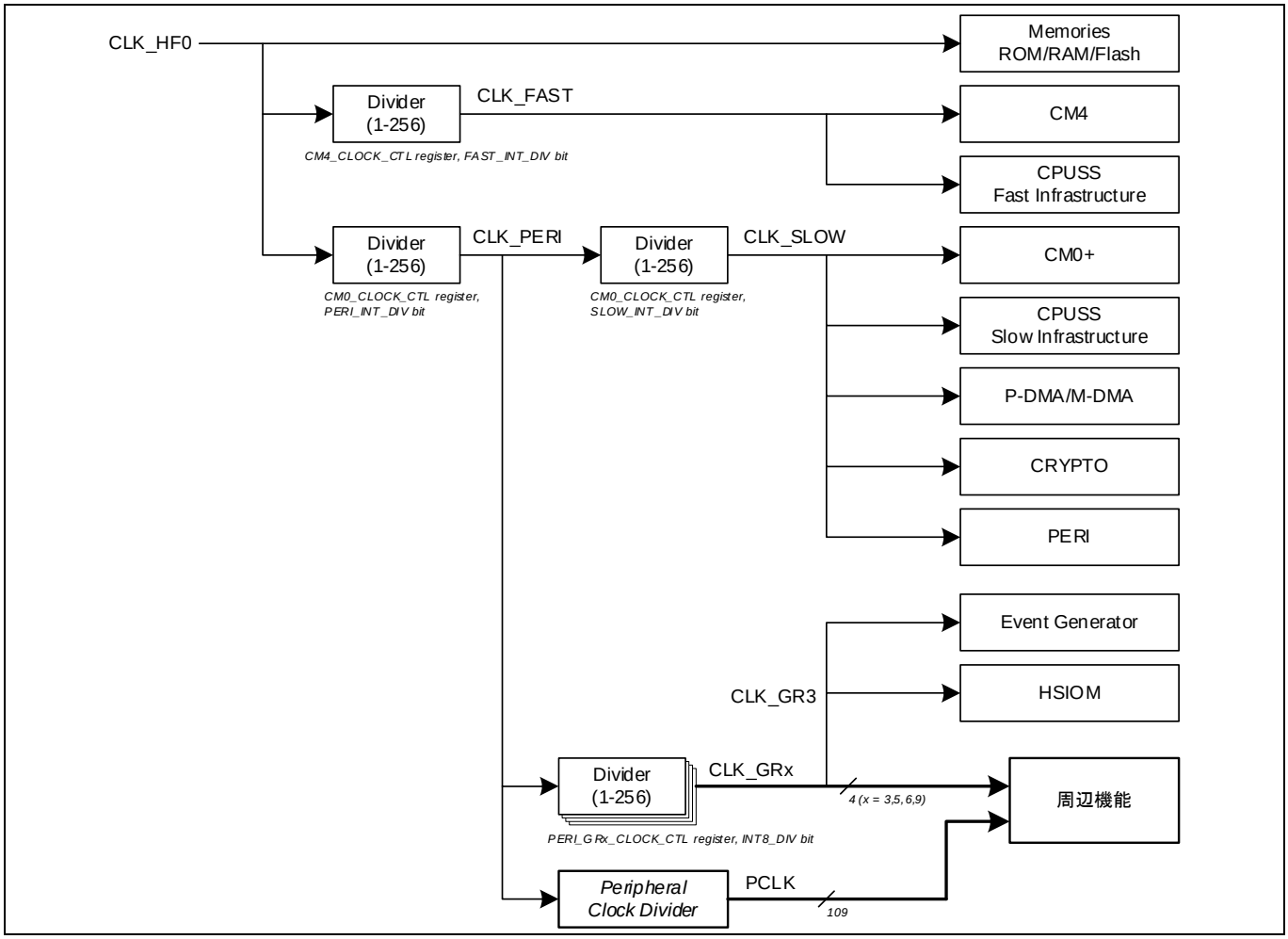

Figure 3　　CLK_HF0 のブロックダイヤグラム

CLK_FAST　　CLK_FAST は CM4 と CPUSS Fast infrastructure の入力クロックです。

CLK_PERI　　CLK_RERI は CLK_SLOW, CLK_GR および周辺クロック分周器のクロックソースです。

CLK_SLOW　　CLK_SLOW は CM0+と CPUSS Slow infrastructure の入力クロックです。

CLK_GR　　CLK_GR は周辺機能への入力クロックです。CLK_GR は Clock gater でグループ分けされます。CLK_GR は 6 つのグループを持っています。

Divider　　Divider は各クロックを分周し、1 から 256 分周まで設定できます。

Figure 4 に CLK_HF1 の分配先を示し、**Figure 1** 中の「Clock Distribution, Division」ブロックの詳細を示します。

CLK_HF1 は割込みとトリガを生成するイベントジェネレータの入力ソースです。割込みとトリガは内部 CPU と GPIO を含む周辺信号に接続します。イベントジェネレータは CLK_GR3 だけでなく CLK_HF1 も使用します。**Figure 4** に CLK_HF1 のクロックの分配先を示します。

イベントジェネレータの詳細は **architecture TRM** を参照してください。

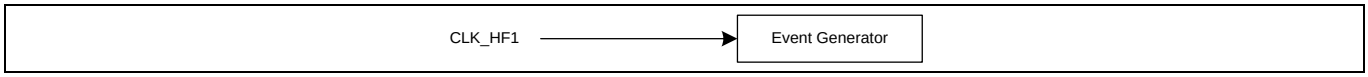

Figure 4　　CLK_HF1 のブロックダイヤグラム

TRAVEO™ T2G ファミリ MCU のクロックシステム

Figure 5 は、**Figure 3** 中の周辺クロック分周器の詳細を示します。

通信機能であるシリアルコミュニケーションブロック (SCB)、波形出力および入力信号測定に使用する Timer, counter, and PWM (TCPWM) のようなこの CYT2B シリーズの MCU の周辺機能には動作クロックが必要となります。これら周辺機能は周辺クロック分周器によりクロック動作します。

この CYT2B シリーズの MCU には、PCLK を生成するための多くの周辺クロック分周器があります。8 ビット分周器が 32 個, 16 ビット分周器が 16 個, 24.5 ビット分周器 (24 個の整数ビット, 5 個の分数ビット) が 8 個あります。これらの分周器の各出力はどの周辺機能にも接続できます。

CLK_PERI のクロック分配を **Figure 5** に示します。**Figure 5** に記載の機能については **architecture TRM** を参照してください。

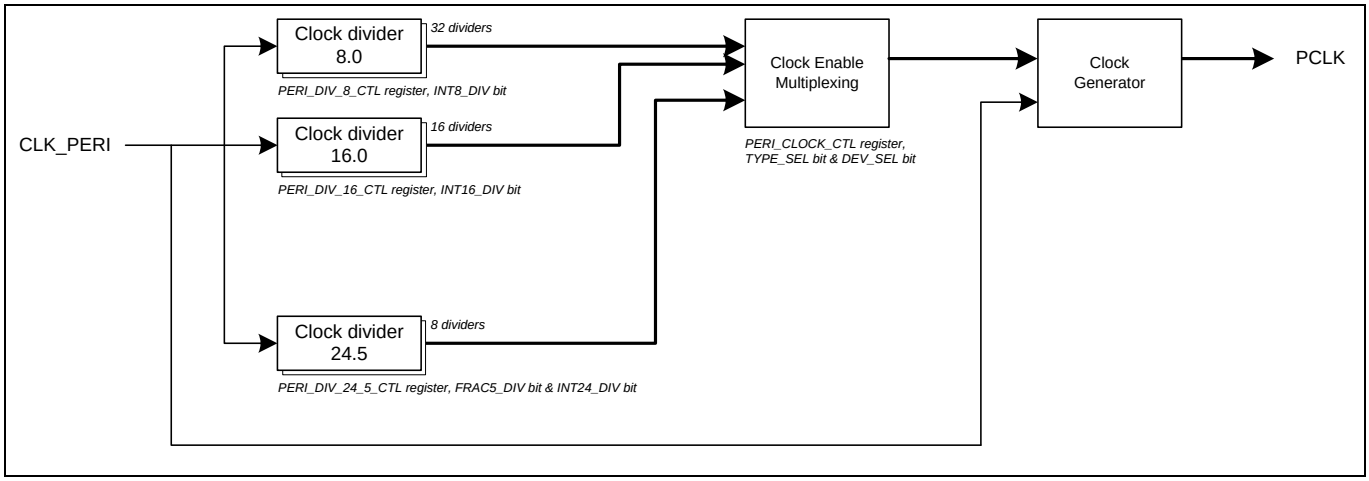

Figure 5　　周辺クロック分周器のブロックダイヤグラム

Clock divider8.0	8 ビットのクロック分周器
Clock divider16.0	16 ビットのクロック分周器
Clock divider24.5	24.5 ビットのクロック分周器
Clock enable multiplexing	Clock enable multiplexing はクロック分周器から出力される信号を有効にします。
Clock generator	Clock generator はクロック分周器を元にして CLK_PERI を分周します。

2.4　　基本的なクロックシステム設定

ここでは、サンプルドライバライブラリ (SDL) を使用して、ユースケースに基づいてクロックシステムを設定する方法について説明します。 このアプリケーションノートのプログラムコードは SDL の一部です。SDL については、**その他の参考資料**を参照してください。

SDL は、設定部とドライバ部があります。 設定部は、主に目的の操作のためのパラメータ値を設定します。

ドライバ部は、設定部のパラメータ値に基づいて各レジスタを設定します。

目的とするシステムに応じて、設定部の設定ができます。

3 クロックリソースの設定

クロックリソースの設定方法について説明します。

3.1 ECO の設定

ECO は初期設定では無効です。ECO は利用に応じて有効にする必要があります。また、ECO を使用するためにはトリミングが必要です。このデバイスは、水晶振動子とセラミック発振子に応じて発振器を制御するトリミングパラメータを設定できます。パラメータの決定方法は、水晶振動子とセラミック発振子で異なります。詳細については、**Setting ECO parameters in TRAVEO™ T2G user guide** を参照してください。

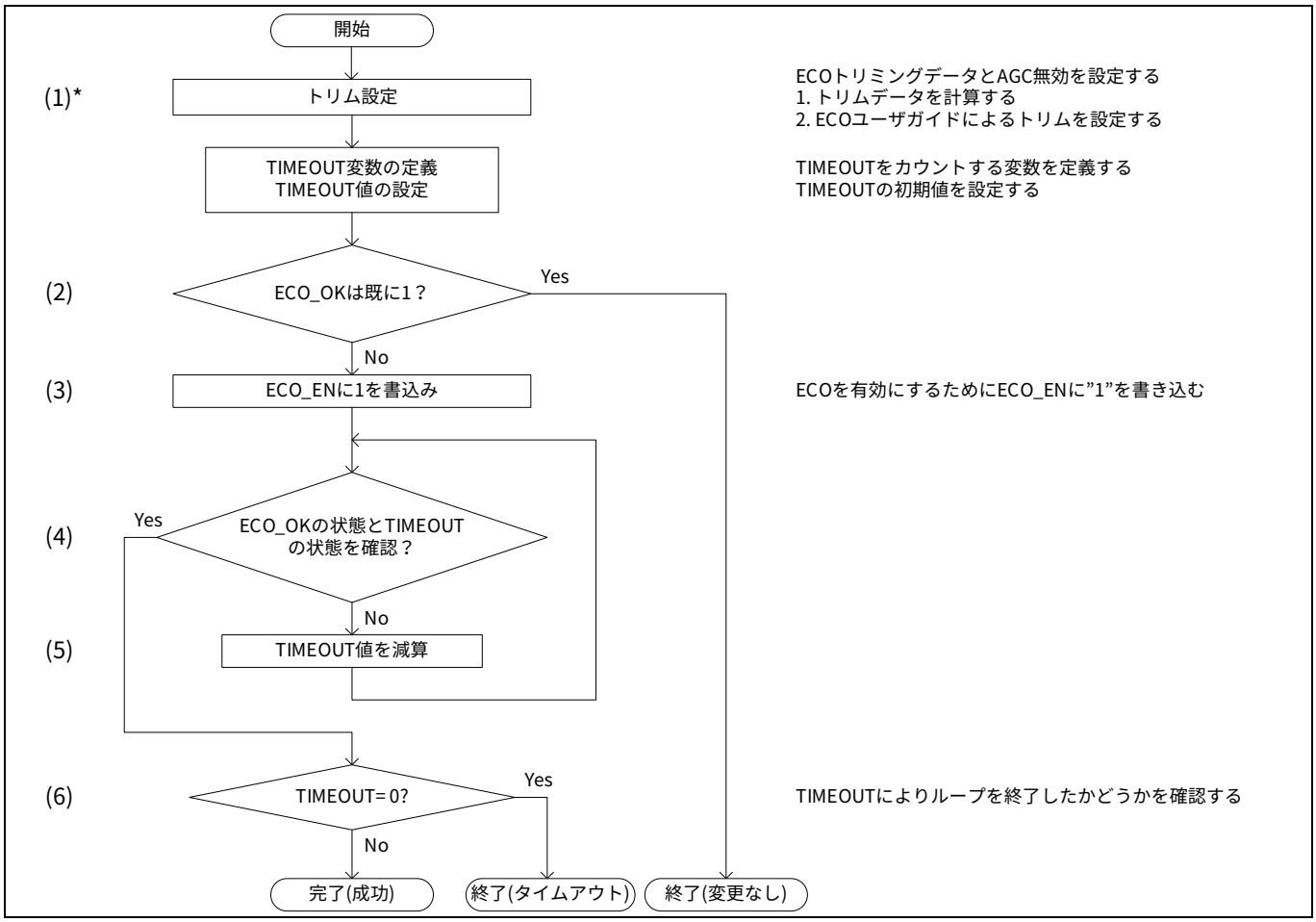

Figure 6 ECO の有効化

* ソフトウェアで計算されたトリミングデータを選択するか、ECO ユーザガイドに従って計算されたデータを使用します。

3.1.1 ユースケース

- 使用する発振器: 水晶振動子
- 基本周波数: 16 MHz
- 最大ドライブレベル: 300.0 uW
- 等価直列抵抗: 150.0 ohm

クロックリソースの設定

- シャント容量: 0.530 pF
- 並列負荷容量: 8.000 pF
- 水晶振動子ベンダーの負性抵抗の推奨値: 1500 ohm
- 自動ゲイン制御: OFF

Note: *これらの値は、水晶振動子ベンダーに確認した上で決めてください。*

3.1.2 コンフィグレーション

ECO の設定における SDL の構成部のパラメータを **Table 1** に、関数を **Table 2** に示します。

Table 1 ECO 設定パラメーター一覧

パラメータ	説明	値
CLK_ECO_CONFIG2.WDTRIM	ウォッチドッグトリム **TRAVEO™ T2G ユーザガイドの「Setting ECO parameters」**から計算	7ul
CLK_ECO_CONFIG2.ATRIM	振幅トリム **TRAVEO™ T2G ユーザガイドの「Setting ECO parameters」**から計算	0ul
CLK_ECO_CONFIG2.FTRIM	3 次高調波発振のフィルタトリム **TRAVEO™ T2G ユーザガイドの「Setting ECO parameters」**から計算	3ul
CLK_ECO_CONFIG2.RTRIM	フィードバック抵抗トリム **TRAVEO™ T2G ユーザガイドの「Setting ECO parameters」**から計算	3ul
CLK_ECO_CONFIG2.GTRIM	ゲイントリムの起動時間 **TRAVEO™ T2G ユーザガイドの「Setting ECO parameters」**から計算	0ul
CLK_ECO_CONFIG.AGC_EN	自動ゲイン制御(AGC)無効 **TRAVEO™ T2G ユーザガイドの「Setting ECO parameters」**から計算	0ul [OFF]
WAIT_FOR_STABILIZATION	発振安定待ち	10000ul
PLL_PATH_NO	PLL 番号	1ul
CLK_FREQ_ECO	ソースクロック周波数	16000000ul
SUM_LOAD_SHUNT_CAP_IN_PF	ロードシャント容量の合計（pF)	17ul
ESR_IN_OHM	等価直列抵抗（ESR) (ohm)	250ul
MAX_DRIVE_LEVEL_IN_UW	最大ドライブレベル（uW)	100ul
MIN_NEG_RESISTANCE	最小負性抵抗	5 * ESR_IN_OHM

Table 2 ECO 設定関数一覧

関数	説明	値
`Cy_WDT_Disable()`	ウォッチドッグタイマ無効	-
`Cy_SysClk_FllDisableSequence(Wait Cycle)`	FLL 無効	Wait cycle = 100ul

クロックリソースの設定

関数	説明	値
`Cy_SysClk_PllDisable(PLL Number)`	PLL 無効	PLL number = PLL_PATH_NO
`AllClockConfiguration()`	クロック設定	-
`Cy_SysClk_EcoEnable(Timeout value)`	ECO の有効化とタイムアウト値の設定	Timeout value = WAIT_FOR_STABILIZATION
`Cy_SysLib_DelayUs(Wait Time)`	指定されたマイクロ秒数による遅延	Wait time = 1u (1us)

3.1.3 サンプルコード

サンプルコードを **Code Listing 1** に示します。

以下の説明は、SDL のドライバ部分のレジスタ表記の理解に役立ちます。

- SRSS->unCLK_ECO_CONFIG.stcField.u1ECO_EN は、**registers TRM** に記載されている SRSS_CLK_ECO_CONFIG.ECO_EN です。他のレジスタも同じように記述されます。
- パフォーマンス改善策
- レジスタ設定のパフォーマンスを向上させるために、SDL は完全な 32 ビットデータをレジスタに書き込みます。各ビットフィールドは、ビット書き込み可能なバッファで事前に生成され、最終的な 32 ビットデータとしてレジスタに書き込まれます。

```
tempTrimEcoCtlReg.u32Register        = SRSS->unCLK_ECO_CONFIG2.u32Register;
tempTrimEcoCtlReg.stcField.u3WDTRIM  = wdtrim;
tempTrimEcoCtlReg.stcField.u4ATRIM   = atrim;
tempTrimEcoCtlReg.stcField.u2FTRIM   = ftrim;
tempTrimEcoCtlReg.stcField.u2RTRIM   = rtrim;
tempTrimEcoCtlReg.stcField.u3GTRIM   = gtrim;
SRSS->unCLK_ECO_CONFIG2.u32Register = tempTrimEcoCtlReg.u32Register;
```

レジスタの共用体と構造体の詳細については、*hdr/rev_x/ip* の下の *cyip_srss_v2.h* を参照してください。

Code Listing 1　ECO の基本設定

```
 :
/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul)
 :
#define CLK_FREQ_ECO       (16000000ul)

#define PLL_PATH_NO      (1u)
:
#define SUM_LOAD_SHUNT_CAP_IN_PF         (17ul)
 :
#define ESR_IN_OHM            (250ul)
 :
#define MIN_NEG_RESISTANCE     (5 * ESR_IN_OHM)
#define MAX_DRIVE_LEVEL_IN_UW  (100ul)
 :
static void AllClockConfiguration(void);
 :
int main(void)
{
    /* disable watchdog timer */
    Cy_WDT_Disable();
 :
    /* Disable Fll */
    Cy_SysClk_FllDisableSequence(100ul);

    /* Disable Pll */
    CY_ASSERT(Cy_SysClk_PllDisable(PLL_PATH_NO) == CY_SYSCLK_SUCCESS);
```

TIMEOUT 変数の宣言

ソフトウェア計算に使用する発振器パラメータを宣言

PLL 番号の宣言

ウォッチドッグタイマ無効。

FLL 無効

PLL 無効

クロックリソースの設定

```
    /* Enable interrupt */
    __enable_irq();

    /* Set Clock Configuring registers */
    AllClockConfiguration();
:
    /* Please check clock output using oscilloscope after CPU reached here. */
    for(;;);
}
```

トリムと ECO の設定。**Code Listing 2** 参照。

Code Listing 2　AllClockConfiguration() 関数

```
static void AllClockConfiguration(void)
{
:
    /***** ECO setting ******/
        cy_en_sysclk_status_t ecoStatus;
        ecoStatus = Cy_SysClk_EcoConfigureWithMinRneg(
                            CLK_FREQ_ECO,
                            SUM_LOAD_SHUNT_CAP_IN_PF,
                            ESR_IN_OHM,
                            MAX_DRIVE_LEVEL_IN_UW,
                            MIN_NEG_RESISTANCE
                            );
        CY_ASSERT(ecoStatus == CY_SYSCLK_SUCCESS);
    {
        SRSS->unCLK_ECO_CONFIG2.stcField.u3WDTRIM = 7ul;
        SRSS->unCLK_ECO_CONFIG2.stcField.u4ATRIM  = 0ul;
        SRSS->unCLK_ECO_CONFIG2.stcField.u2FTRIM  = 3ul;
        SRSS->unCLK_ECO_CONFIG2.stcField.u2RTRIM  = 3ul;
        SRSS->unCLK_ECO_CONFIG2.stcField.u3GTRIM  = 0ul;
        SRSS->unCLK_ECO_CONFIG.stcField.u1AGC_EN  = 0ul;

        ecoStatus = Cy_SysClk_EcoEnable(WAIT_FOR_STABILIZATION);
        CY_ASSERT(ecoStatus == CY_SYSCLK_SUCCESS);
    }
:
   return;
}
```

(1)-1. ソフトウェア計算によるトリム設定。**Code Listing 4** 参照。

(1)-2. ECO ユーザガイドによるトリム設定

ECO 有効。**Code Listing 3** 参照。

(1)-1 または (1)-2 のいずれかを使用できます。

(1)-1 または (1)-2 の使用しないプログラムコード表記をコメントアウトもしくは削除します。

Code Listing 3　Cy_SysClk_EcoEnable() 関数

```
cy_en_sysclk_status_t Cy_SysClk_EcoEnable(uint32_t timeoutus)
{
    cy_en_sysclk_status_t rtnval;

    /* invalid state error if ECO is already enabled */
    if (SRSS->unCLK_ECO_CONFIG.stcField.u1ECO_EN != 0ul) /* 1 = enabled */
    {
        return CY_SYSCLK_INVALID_STATE;
    }

    /* first set ECO enable */
    SRSS->unCLK_ECO_CONFIG.stcField.u1ECO_EN = 1ul; /* 1 = enable */

    /* now do the timeout wait for ECO_STATUS, bit ECO_OK */
    for (;
         (SRSS->unCLK_ECO_STATUS.stcField.u1ECO_OK == 0ul) &&(timeoutus != 0ul);
         timeoutus--)
    {
        Cy_SysLib_DelayUs(1u);
    }

    rtnval = ((timeoutus == 0ul) ? CY_SYSCLK_TIMEOUT : CY_SYSCLK_SUCCESS);
    return rtnval;
}
```

(2) ECO_OK が既に有効か確認。

(3) ECO_EN ビットに”1”を書き込み。ECO が利用可能

(4) ECO OK と TIMEOUT の状態を確認

(5) TIMEOUT 値を減算

1 us 待機。

(6) TIMEOUT によりループが終了したかどうか確認

クロックリソースの設定

Code Listing 4　Cy_SysClk_EcoConfigureWithMinRneg() 関数

```
cy_en_sysclk_status_t Cy_SysClk_EcoConfigureWithMinRneg(uint32_t freq, uint32_t cSum, uint32_t esr, uint32_t
driveLevel, uint32_t minRneg)
{
    /* Check if ECO is disabled */
    if(SRSS->unCLK_ECO_CONFIG.stcField.u1ECO_EN == 1ul)
    {
        return(CY_SYSCLK_INVALID_STATE);
    }

    /* calculate intermediate values */
    float32_t freqMHz      = (float32_t)freq / 1000000.0f;
    float32_t maxAmplitude = (1000.0f * ((float32_t)sqrt((float64_t)((float32_t)driveLevel / (2.0f *
(float32_t)esr))))) /
                             (M_PI * freqMHz * (float32_t)cSum);

    float32_t gm_min       = (157.91367042f /*4 * M_PI * M_PI * 4*/ * minRneg * freqMHz * freqMHz * (float32_t)cSum *
(float32_t)cSum) /
                             1000000000.0f;

    /* Get trim values according to caluculated values */
    uint32_t atrim, agcen, wdtrim, gtrim, rtrim, ftrim;
    atrim  = Cy_SysClk_SelectEcoAtrim(maxAmplitude);
    if(atrim == CY_SYSCLK_INVALID_TRIM_VALUE)
    {
        return(CY_SYSCLK_BAD_PARAM);
    }

    agcen = Cy_SysClk_SelectEcoAGCEN(maxAmplitude);
    if(agcen == CY_SYSCLK_INVALID_TRIM_VALUE)
    {
        return(CY_SYSCLK_BAD_PARAM);
    }

    wdtrim = Cy_SysClk_SelectEcoWDtrim(maxAmplitude);
    if(wdtrim == CY_SYSCLK_INVALID_TRIM_VALUE)
    {
        return(CY_SYSCLK_BAD_PARAM);
    }

    gtrim  = Cy_SysClk_SelectEcoGtrim(gm_min);
    if(gtrim == CY_SYSCLK_INVALID_TRIM_VALUE)
    {
        return(CY_SYSCLK_BAD_PARAM);
    }

    rtrim  = Cy_SysClk_SelectEcoRtrim(freqMHz);
    if(rtrim == CY_SYSCLK_INVALID_TRIM_VALUE)
    {
        return(CY_SYSCLK_BAD_PARAM);
    }

    ftrim  = Cy_SysClk_SelectEcoFtrim(atrim);

    /* update all fields of trim control register with one write, without
       changing the ITRIM field:
    */
    un_CLK_ECO_CONFIG2_t tempTrimEcoCtlReg;
    tempTrimEcoCtlReg.u32Register        = SRSS->unCLK_ECO_CONFIG2.u32Register;
    tempTrimEcoCtlReg.stcField.u3WDTRIM  = wdtrim;
    tempTrimEcoCtlReg.stcField.u4ATRIM   = atrim;
    tempTrimEcoCtlReg.stcField.u2FTRIM   = ftrim;
    tempTrimEcoCtlReg.stcField.u2RTRIM   = rtrim;
    tempTrimEcoCtlReg.stcField.u3GTRIM   = gtrim;
    SRSS->unCLK_ECO_CONFIG2.u32Register = tempTrimEcoCtlReg.u32Register;

    SRSS->unCLK_ECO_CONFIG.stcField.u1AGC_EN = agcen;

    return(CY_SYSCLK_SUCCESS);
}
```

ソフトウェアによるトリム計算

Atrim 値を取得。**Code Listing 5** 参照。

AGC を有効に設定。**Code Listing 6** 参照。

Wdtrim 値を取得。**Code Listing 7** 参照。

Gtrim 値を取得。**Code Listing 8** 参照。

Rtrim 値を取得。**Code Listing 9** 参照。

Ftrim 値を取得。**Code Listing 10** 参照。

トリム値を設定

Code Listing 5　Code 1. Cy_SysClk_SelectEcoAtrim() 関数

```
__STATIC_INLINE uint32_t Cy_SysClk_SelectEcoAtrim(float32_t maxAmplitude)
{
    if((0.50f <= maxAmplitude) && (maxAmplitude < 0.55f))
    {
        return(0x04ul);
    }
    else if(maxAmplitude < 0.60f)
    {
        return(0x05ul);
    }
```

Atrim 値を取得。

クロックリソースの設定

```
    else if(maxAmplitude < 0.65f)
    {
        return(0x06ul);
    }
    else if(maxAmplitude < 0.70f)
    {
        return(0x07ul);
    }
    else if(maxAmplitude < 0.75f)
    {
        return(0x08ul);
    }
    else if(maxAmplitude < 0.80f)
    {
        return(0x09ul);
    }
    else if(maxAmplitude < 0.85f)
    {
        return(0x0Aul);
    }
    else if(maxAmplitude < 0.90f)
    {
        return(0x0Bul);
    }
    else if(maxAmplitude < 0.95f)
    {
        return(0x0Cul);
    }
    else if(maxAmplitude < 1.00f)
    {
        return(0x0Dul);
    }
    else if(maxAmplitude < 1.05f)
    {
        return(0x0Eul);
    }
    else if(maxAmplitude < 1.10f)
    {
        return(0x0Ful);
    }
    else if(1.1f <= maxAmplitude)
    {
        return(0x00ul);
    }
    else
    {
        // invalid input
        return(CY_SYSCLK_INVALID_TRIM_VALUE);
    }
}
```

Code Listing 6　　Code 2. Cy_SysClk_SelectEcoAGCEN() 関数

```
__STATIC_INLINE uint32_t Cy_SysClk_SelectEcoAGCEN(float32_t maxAmplitude)
{
    if((0.50f <= maxAmplitude) && (maxAmplitude < 1.10f))
    {
        return(0x01ul);
    }
    else if(1.10f <= maxAmplitude)
    {
        return(0x00ul);
    }
    else
    {
        return(CY_SYSCLK_INVALID_TRIM_VALUE);
    }
}
```

AGC 有効設定を取得。

クロックリソースの設定

Code Listing 7　Code 3. Cy_SysClk_SelectEcoWDtrim() 関数

```
__STATIC_INLINE uint32_t Cy_SysClk_SelectEcoWDtrim(float32_t amplitude)
{
    if( (0.50f <= amplitude) && (amplitude < 0.60f))
    {
        return(0x02ul);
    }
    else if(amplitude < 0.7f)
    {
        return(0x03ul);
    }
    else if(amplitude < 0.8f)
    {
        return(0x04ul);
    }
    else if(amplitude < 0.9f)
    {
        return(0x05ul);
    }
    else if(amplitude < 1.0f)
    {
        return(0x06ul);
    }
    else if(amplitude < 1.1f)
    {
        return(0x07ul);
    }
    else if(1.1f <= amplitude)
    {
        return(0x07ul);
    }
    else
    {
        // invalid input
        return(CY_SYSCLK_INVALID_TRIM_VALUE);
    }
}
```

Wdtrim 値を取得。

Code Listing 8　Code 4. Cy_SysClk_SelectEcoGtrim() 関数

```
__STATIC_INLINE uint32_t Cy_SysClk_SelectEcoGtrim(float32_t gm_min)
{
    if( (0.0f <= gm_min) && (gm_min < 2.2f))
    {
        return(0x00ul+1ul);
    }
    else if(gm_min < 4.4f)
    {
        return(0x01ul+1ul);
    }
    else if(gm_min < 6.6f)
    {
        return(0x02ul+1ul);
    }
    else if(gm_min < 8.8f)
    {
        return(0x03ul+1ul);
    }
    else if(gm_min < 11.0f)
    {
        return(0x04ul+1ul);
    }
    else if(gm_min < 13.2f)
    {
        return(0x05ul+1ul);
    }
    else if(gm_min < 15.4f)
    {
        return(0x06ul+1ul);
    }
    else if(gm_min < 17.6f)
    {
        // invalid input
        return(CY_SYSCLK_INVALID_TRIM_VALUE);
    }
    else
    {
        // invalid input
        return(CY_SYSCLK_INVALID_TRIM_VALUE);
    }
}
```

Gtrim 値を取得。

Code Listing 9　　Code 5. Cy_SysClk_SelectEcoRtrim() 関数

```
__STATIC_INLINE uint32_t Cy_SysClk_SelectEcoRtrim(float32_t freqMHz)
{
    if(freqMHz > 28.6f)
    {
        return(0x00ul);
    }
    else if(freqMHz > 23.33f)
    {
        return(0x01ul);
    }
    else if(freqMHz > 16.5f)
    {
        return(0x02ul);
    }
    else if(freqMHz > 0.0f)
    {
        return(0x03ul);
    }
    else
    {
        // invalid input
        return(CY_SYSCLK_INVALID_TRIM_VALUE);
    }
}
```

Rtrim 値を取得。

Code Listing 10　　Code 6. Cy_SysClk_SelectEcoFtrim() 関数

```
__STATIC_INLINE uint32_t Cy_SysClk_SelectEcoFtrim(uint32_t atrim)
{
    return(0x03ul);
}
```

Ftrim 値を取得。

3.2　　WCO の設定

3.2.1　　操作概要

WCO は初期設定では無効です。そのため WCO を有効にしない限り使用できません。**Figure 7** は WCO を有効にするためのレジスタの設定方法を示します。

WCO を無効にするためには、BACKUP_CTL レジスタの WCO_EN ビットに’0’を書き込んでください。

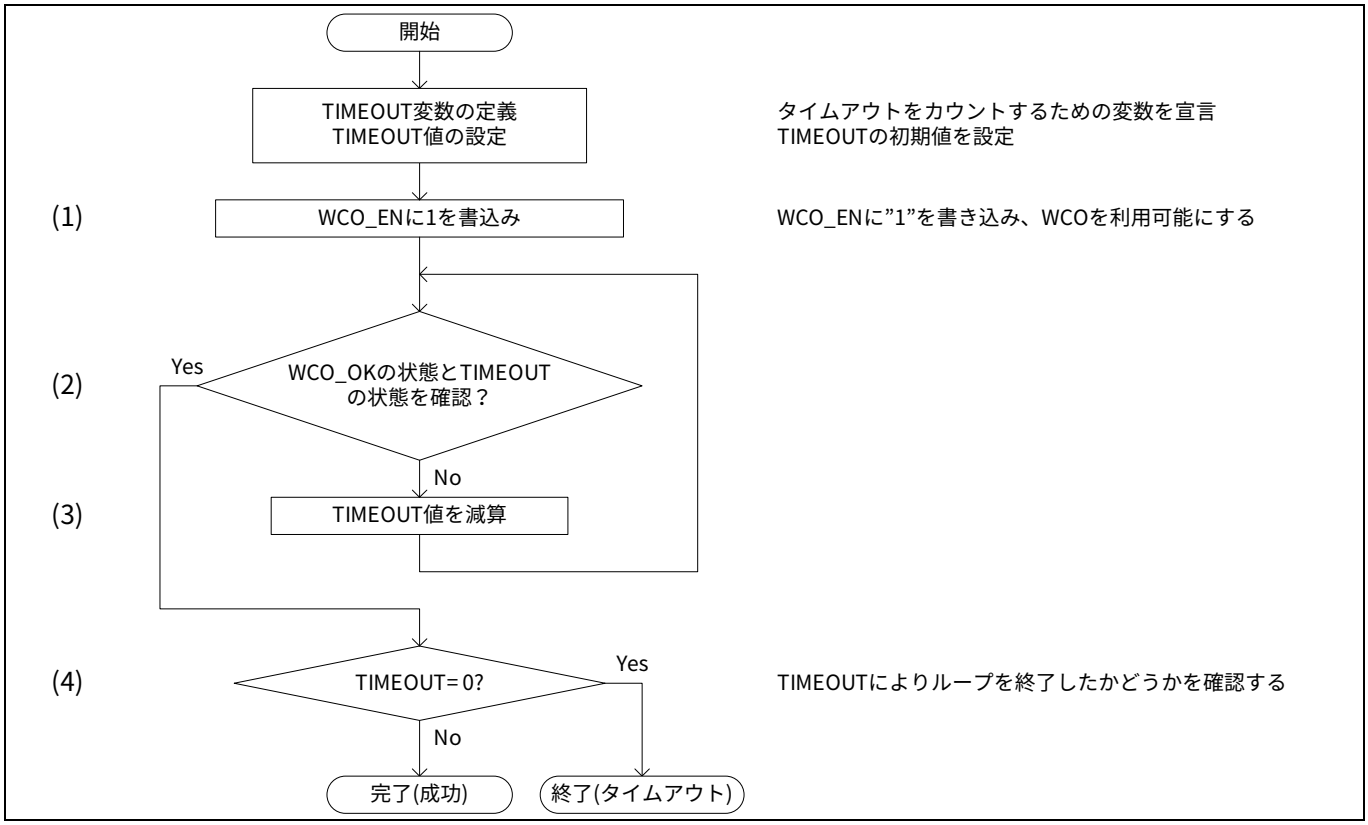

Figure 7 WCO の有効化

3.2.2 コンフィグレーション

WCO の設定における SDL の構成部のパラメータを **Table 3** に、関数を **Table 4** に示します。

Table 3 WCO 設定パラメーター一覧

パラメータ	説明	値
WAIT_FOR_STABILIZATION	発振安定待ち	10000ul
PLL_PATH_NO	PLL PATH 番号	1ul

Table 4 WCO 設定関数一覧

関数	説明	値
`Cy_WDT_Disable()`	ウォッチドッグタイマ無効	-
`Cy_SysClk_FllDisableSequence(Wait Cycle)`	FLL 無効	Wait cycle = 100ul
`Cy_SysClk_PllDisable(PLL Number)`	PLL 無効	PLL number = PLL_PATH_NO
`AllClockConfiguration()`	クロック設定	-
`Cy_SysClk_WcoEnable(Timeout value)`	WCO の有効化とタイムアウト値の設定	Timeout value = WAIT_FOR_STABILIZATION
`Cy_SysLib_DelayUs(Wait Time)`	指定されたマイクロ秒数による遅延	Wait time = 1u (1us)

3.2.3 サンプルコード

サンプルコードを **Code Listing 11** から **Code Listing 13** に示します。

Code Listing 11 WCO の基本設定

```
/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul)
#define PLL_PATH_NO      (1u)

int main(void)
{
    /* disable watchdog timer */
    Cy_WDT_Disable();
 :
    /* Disable Fll */
    Cy_SysClk_FllDisableSequence(100ul);

    /* Disable Pll */
    CY_ASSERT(Cy_SysClk_PllDisable(PLL_PATH_NO) == CY_SYSCLK_SUCCESS);

    /* Enable interrupt */
    __enable_irq();

    /* Set Clock Configuring registers */
    AllClockConfiguration();
 :
    /* Please check clock output using oscilloscope after CPU reached here. */
    for(;;);
}
```

TIMEOUT 変数の宣言

PLL 番号の宣言

ウォッチドッグタイマ無効。

FLL 無効

PLL 無効

WCO の設定。**Code Listing 12** 参照。

Code Listing 12 AllClockConfiguration() 関数

```
static void AllClockConfiguration(void)
{
 :
    /***** WCO setting ******/
    {
        cy_en_sysclk_status_t wcoStatus;
        wcoStatus = Cy_SysClk_WcoEnable(WAIT_FOR_STABILIZATION*10ul);
        CY_ASSERT(wcoStatus == CY_SYSCLK_SUCCESS);
    }

   return;
}
```

WCO が有効。**Code Listing 13** 参照。

Code Listing 13 Cy_Sysclk_WcoEnable() 関数

```
 :
__STATIC_INLINE cy_en_sysclk_status_t Cy_SysClk_WcoEnable(uint32_t timeoutus)
{
    cy_en_sysclk_status_t rtnval = CY_SYSCLK_TIMEOUT;

    BACKUP->unCTL.stcField.u1WCO_EN = 1ul;

    /* now do the timeout wait for STATUS, bit WCO_OK */
    for (; (Cy_SysClk_WcoOkay() == false) && (timeoutus != 0ul); timeoutus--)
    {
        Cy_SysLib_DelayUs(1u);
    }
    if (timeoutus != 0ul)
    {
        rtnval = CY_SYSCLK_SUCCESS;
    }

    return (rtnval);
}
```

(1) WCO_EN ビットに"1"を書込み、WCO を利用可能にする

(2) WCO_OK と TIMEOUT の状態を確認

1 us 待機。

(3) TIMEOUT 値を減算する

(4) TIMEOUT によりループが終了したかどうか確認

3.3 IMO の設定

IMO は初期設定で有効になっているため、すべての機能は適切に動作します。Deep Sleep, Hibernate および XRES 中では IMO は自動的に無効になります。したがって IMO を設定する必要はありません。

3.4 ILO0/ILO1 の設定

ILO0 は初期設定で有効です。

注意として、ILO0 はウォッチドッグタイマ (WDT) の動作クロックとして使用されます。したがって、ILO0 を無効にする場合は WDT を無効にする必要があります。ILO0 を無効にするためには WDT_CTL レジスタの WDT_LOCK ビットに'01b'を書き込み、それから CLK_ILO0_CONFIG レジスタの ENABLE ビットに'0'を書き込みます。ILO1 は初期設定で無効です。ILO1 を有効にするためには、CLK_ILO1_CONFIG レジスタの ENABLE ビットに’1’を書き込んでください。

4 FLL と PLL の設定

クロックシステムの FLL と PLL の設定方法について示します。

4.1 FLL の設定

4.1.1 操作概要

FLL を使用するためには FLL を設定する必要があります。FLL は電流制御オシレータ (CCO) を搭載しており、この CCO の出力周波数は CCO のトリミングを調整することによって制御されます。**Figure 8** に FLL を設定する手順を示します。

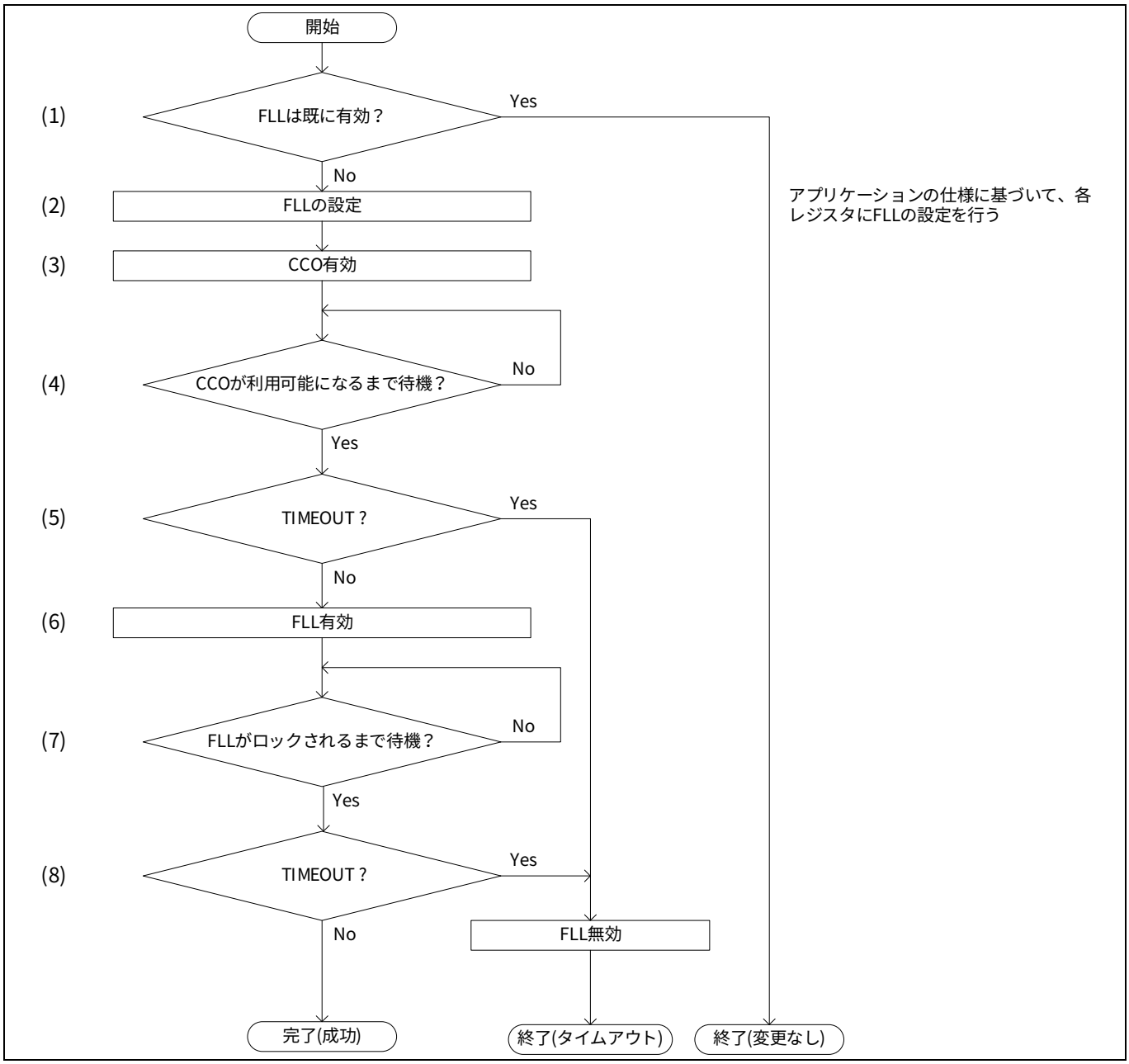

Figure 8 FLL 設定の手順

FLL および FLL 設定レジスタの詳細については **architecture TRM** と **registers TRM** を参照してください。

4.1.2 ユースケース

- 入力クロック周波数: 16 MHz (ECO)
- 出力クロック周波数: 100 MHz

4.1.3 コンフィグレーション

FLL の設定における SDL の構成部のパラメータを **Table 5** に、関数を **Table 6** に示します。

Table 5 FLL 設定パラメーター一覧

パラメータ	説明	値
WAIT_FOR_STABILIZATION	発振安定待ち	10000ul
FLL_PATH_NO	FLL 番号	0ul
FLL_TARGET_FREQ	FLL ターゲット周波数	100000000ul (100 MHz)
CLK_FREQ_ECO	ソースクロック周波数	16000000ul (16 MHz)
PATH_SOURCE_CLOCK_FREQ	FLL 入力周波数	CLK_FREQ_ECO
CY_SYSCLK_FLLPLL_OUTPUT_AUTO	FLL 出力モード CY_SYSCLK_FLLPLL_OUTPUT_AUTO: ロックインジケータを自動使用。 CY_SYSCLK_FLLPLL_OUTPUT_LOCKED_OR_NOTHING: AUTO と同様にロック解除でクロックがゲートオフされることを除外。 CY_SYSCLK_FLLPLL_OUTPUT_INPUT: FLL リファレンス入力を選択 (バイパスモード)。 CY_SYSCLK_FLLPLL_OUTPUT_OUTPUT: FLL 出力を選択。ロックインジケータを無視。 詳細については、**registers TRM** の SRSS_CLK_FLL_CONFIG3 を参照	0ul

Table 6 FLL 設定関数一覧

関数	説明	値
`AllClockConfiguration()`	クロック設定	-
`Cy_SysClk_FllConfigureStandard (inputFreq, outputFreq, outputMode)`	inputFreq: 入力周波数 outputFreq:出力周波数 outputMode: FLL 出力モード	inputFreq = PATH_SOURCE_CLOCK_FREQ, outputFreq = FLL_TARGET_FREQ, outputMode = CY_SYSCLK_FLLPLL_OUTPUT_AUTO
`Cy_SysClk_FllEnable(Timeout value)`	FLL の有効化とタイムアウト値の設定	Timeout value = WAIT_FOR_STABILIZATION
`Cy_SysLib_DelayUs(Wait Time)`	指定されたマイクロ秒数による遅延	Wait time = 1u (1us)

4.1.4 サンプルコード

サンプルコードを **Code Listing 14** から **Code Listing 18** に示します。

Code Listing 14 FLL の基本設定

```
/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul)
 :
#define FLL_TARGET_FREQ  (100000000ul)
#define CLK_FREQ_ECO     (16000000ul)
#define PATH_SOURCE_CLOCK_FREQ CLK_FREQ_ECO
 :
#define FLL_PATH_NO      (0ul)
 :

int main(void)
{
 :
    /* Enable interrupt */
    __enable_irq();
 :
    /* Set Clock Configuring registers */
    AllClockConfiguration();
 :
    /* Please check clock output using oscilloscope after CPU reached here. */
    for(;;);
}
```

TIMEOUT 変数の宣言

FLL ターゲット周波数の宣言。

FLL 入力周波数の宣言。

FLL 番号の宣言

FLL の設定。**Code Listing 15** 参照。

Code Listing 15 AllClockConfiguration() 関数

```
static void AllClockConfiguration(void)
{
 :
    /***** FLL(PATH0) source setting ******/
    {
 :
        fllStatus = Cy_SysClk_FllConfigureStandard(PATH_SOURCE_CLOCK_FREQ, FLL_TARGET_FREQ,
CY_SYSCLK_FLLPLL_OUTPUT_AUTO);
        CY_ASSERT(fllStatus == CY_SYSCLK_SUCCESS);

        fllStatus = Cy_SysClk_FllEnable(WAIT_FOR_STABILIZATION);
        CY_ASSERT((fllStatus == CY_SYSCLK_SUCCESS) || (fllStatus == CY_SYSCLK_TIMEOUT));
 :
    }
   return;
}
```

FLL の設定。**Code Listing 16** 参照。

FLL が有効。**Code Listing 18** 参照。

FLL と PLL の設定

Code Listing 16　Cy_SysClk_FllConfigureStandard() 関数

```
cy_en_sysclk_status_t Cy_SysClk_FllConfigureStandard(uint32_t inputFreq, uint32_t outputFreq,
cy_en_fll_pll_output_mode_t outputMode)
{
    /* check for errors */
    if (SRSS->unCLK_FLL_CONFIG.stcField.u1FLL_ENABLE != 0ul) /* 1 = enabled */
    {
        return(CY_SYSCLK_INVALID_STATE);
    }
    else if ((outputFreq < CY_SYSCLK_MIN_FLL_OUTPUT_FREQ) || (CY_SYSCLK_MAX_FLL_OUTPUT_FREQ < outputFreq)) /* invalid
output frequency */
    {
        return(CY_SYSCLK_INVALID_STATE);
    }
    else if (((float32_t)outputFreq / (float32_t)inputFreq) < 2.2f) /* check output/input frequency ratio */
    {
        return(CY_SYSCLK_INVALID_STATE);
    }

    /* no error */

    /* If output mode is bypass (input routed directly to output), then done.
       The output frequency equals the input frequency regardless of the
       frequency parameters. */
    if (outputMode == CY_SYSCLK_FLLPLL_OUTPUT_INPUT)
    {
        /* bypass mode */
        /* update CLK_FLL_CONFIG3 register with divide by 2 parameter */
        SRSS->unCLK_FLL_CONFIG3.stcField.u2BYPASS_SEL = (uint32_t)outputMode;
        return(CY_SYSCLK_SUCCESS);
    }

    cy_stc_fll_manual_config_t config = { 0ul };

    config.outputMode = outputMode;

    /* 1. Output division is not required for standard accuracy. */
    config.enableOutputDiv = false;

    /* 2. Compute the target CCO frequency from the target output frequency and output division. */
    uint32_t ccoFreq;
    ccoFreq = outputFreq * ((uint32_t)(config.enableOutputDiv) + 1ul);

    /* 3. Compute the CCO range value from the CCO frequency */
    if(ccoFreq >= CY_SYSCLK_FLL_CCO_BOUNDARY4_FREQ)
    {
        config.ccoRange = CY_SYSCLK_FLL_CCO_RANGE4;
    }
    else if(ccoFreq >= CY_SYSCLK_FLL_CCO_BOUNDARY3_FREQ)
    {
        config.ccoRange = CY_SYSCLK_FLL_CCO_RANGE3;
    }
    else if(ccoFreq >= CY_SYSCLK_FLL_CCO_BOUNDARY2_FREQ)
    {
        config.ccoRange = CY_SYSCLK_FLL_CCO_RANGE2;
    }
    else if(ccoFreq >= CY_SYSCLK_FLL_CCO_BOUNDARY1_FREQ)
    {
        config.ccoRange = CY_SYSCLK_FLL_CCO_RANGE1;
    }
    else
    {
        config.ccoRange = CY_SYSCLK_FLL_CCO_RANGE0;
    }

    /* 4. Compute the FLL reference divider value. */
    config.refDiv = CY_SYSCLK_DIV_ROUNDUP(inputFreq * 250ul, outputFreq);

    /* 5. Compute the FLL multiplier value.
          Formula is fllMult = (ccoFreq * refDiv) /  fref */
    config.fllMult = CY_SYSCLK_DIV_ROUND((uint64_t)ccoFreq * (uint64_t)config.refDiv, (uint64_t)inputFreq);

    /* 6. Compute the lock tolerance.
          Recommendation: ROUNDUP((refDiv / fref ) * ccoFreq  * 3 * CCO_Trim_Step) + 2  */
    config.updateTolerance = CY_SYSCLK_DIV_ROUNDUP(config.fllMult, 100ul /* Reciprocal number of Ratio */ );
    config.lockTolerance = config.updateTolerance + 20ul /*Threshould*/;
    // TODO: Need to check the recommend formula to calculate the value.

    /* 7. Compute the CCO igain and pgain. */
    /* intermediate parameters */
    float32_t kcco = trimSteps_RefArray[config.ccoRange] * fMargin_MHz_RefArray[config.ccoRange];
    float32_t ki_p = (0.85f * (float32_t)inputFreq) / (kcco * (float32_t)(config.refDiv)) / 1000.0f;
     /* find the largest IGAIN value that is less than or equal to ki_p */
    for(config.igain = CY_SYSCLK_N_ELMTS(fll_gains_RefArray) - 1ul;config.igain > 0ul; config.igain--)
    {
        if(fll_gains_RefArray[config.igain] < ki_p)
        {
```

(1) FLL が既に有効か確認

FLL の出力範囲を確認。

FLL の出力比率を確認。

FLL パラメータの計算

FLL と PLL の設定

```
            break;
        }
    }

    /* then find the largest PGAIN value that is less than or equal to ki_p - gains[igain] */
    for(config.pgain = CY_SYSCLK_N_ELMTS(fll_gains_RefArray) - 1ul; config.pgain > 0ul; config.pgain--)
    {
        if(fll_gains_RefArray[config.pgain] < (ki_p - fll_gains_RefArray[config.igain]))
        {
            break;
        }
    }

    /* 8. Compute the CCO_FREQ bits will be set by HW */
    config.ccoHwUpdateDisable = 0ul;

    /* 9. Compute the settling count, using a 1-usec settling time. */
    config.settlingCount = (uint16_t)((float32_t)inputFreq / 1000000.0f);

    /* configure FLL based on calculated values */
    cy_en_sysclk_status_t returnStatus;
    returnStatus = Cy_SysClk_FllManualConfigure(&config);

    return (returnStatus);
}
```

FLL のレジスタを設定。**Code Listing 17** 参照。

Code Listing 17　Cy_SysClk_FllManualConfigure() 関数

```
cy_en_sysclk_status_t Cy_SysClk_FllManualConfigure(const cy_stc_fll_manual_config_t *config)
{
    cy_en_sysclk_status_t returnStatus = CY_SYSCLK_SUCCESS;

    /* check for errors */
    if (SRSS->unCLK_FLL_CONFIG.stcField.u1FLL_ENABLE != 0ul) /* 1 = enabled */
    {
        returnStatus = CY_SYSCLK_INVALID_STATE;
    }
    else
    { /* return status is OK */
    }

    /* no error */
    if (returnStatus == CY_SYSCLK_SUCCESS) /* no errors */
    {
        /* update CLK_FLL_CONFIG register with 2 parameters; FLL_ENABLE is already 0 */
        un_CLK_FLL_CONFIG_t tempConfg;
        tempConfg.u32Register               = SRSS->unCLK_FLL_CONFIG.u32Register;
        tempConfg.stcField.u18FLL_MULT      = config->fllMult;
        tempConfg.stcField.u1FLL_OUTPUT_DIV = (uint32_t)(config->enableOutputDiv);
        SRSS->unCLK_FLL_CONFIG.u32Register  = tempConfg.u32Register;

        /* update CLK_FLL_CONFIG2 register with 2 parameters */
        un_CLK_FLL_CONFIG2_t tempConfg2;
        tempConfg2.u32Register              = SRSS->unCLK_FLL_CONFIG2.u32Register;
        tempConfg2.stcField.u13FLL_REF_DIV  = config->refDiv;
        tempConfg2.stcField.u8LOCK_TOL      = config->lockTolerance;
        tempConfg2.stcField.u8UPDATE_TOL    = config->updateTolerance;
        SRSS->unCLK_FLL_CONFIG2.u32Register = tempConfg2.u32Register;

        /* update CLK_FLL_CONFIG3 register with 4 parameters */
        un_CLK_FLL_CONFIG3_t tempConfg3;
        tempConfg3.u32Register               = SRSS->unCLK_FLL_CONFIG3.u32Register;
        tempConfg3.stcField.u4FLL_LF_IGAIN   = config->igain;
        tempConfg3.stcField.u4FLL_LF_PGAIN   = config->pgain;
        tempConfg3.stcField.u13SETTLING_COUNT = config->settlingCount;
        tempConfg3.stcField.u2BYPASS_SEL     = (uint32_t)(config->outputMode);
        SRSS->unCLK_FLL_CONFIG3.u32Register  = tempConfg3.u32Register;

        /* update CLK_FLL_CONFIG4 register with 1 parameter; preserve other bits */
        un_CLK_FLL_CONFIG4_t tempConfg4;
        tempConfg4.u32Register                  = SRSS->unCLK_FLL_CONFIG4.u32Register;
        tempConfg4.stcField.u3CCO_RANGE         = (uint32_t)(config->ccoRange);
        tempConfg4.stcField.u9CCO_FREQ          = (uint32_t)(config->cco_Freq);
        tempConfg4.stcField.u1CCO_HW_UPDATE_DIS = (uint32_t)(config->ccoHwUpdateDisable);
        SRSS->unCLK_FLL_CONFIG4.u32Register     = tempConfg4.u32Register;
    } /* if no error */

    return (returnStatus);
}
```

(1) FLL が既に有効かどうか確認

(2) FLL の設定

CLK_FLL_CONFIG レジスタの設定

CLK_FLL_CONFIG2 レジスタの設定

CLK_FLL_CONFIG3 レジスタの設定

CLK_FLL_CONFIG4 レジスタの設定

FLL と PLL の設定

Code Listing 18　Cy_SysClk_FllEnable() 関数

```
cy_en_sysclk_status_t Cy_SysClk_FllEnable(uint32_t timeoutus)
{
    /* first set the CCO enable bit */
    SRSS->unCLK_FLL_CONFIG4.stcField.u1CCO_ENABLE = 1ul;

    /* Wait until CCO is ready */
    while(SRSS->unCLK_FLL_STATUS.stcField.u1CCO_READY == 0ul)
    {
        if(timeoutus == 0ul)
        {
            /* If cco ready doesn't occur, FLL is stopped. */
            Cy_SysClk_FllDisable();
            return(CY_SYSCLK_TIMEOUT);
        }
        Cy_SysLib_DelayUs(1u);
        timeoutus--;
    }

    /* Set the FLL bypass mode to 2 */
    SRSS->unCLK_FLL_CONFIG3.stcField.u2BYPASS_SEL = (uint32_t)CY_SYSCLK_FLLPLL_OUTPUT_INPUT;

    /* Set the FLL enable bit, if CCO is ready */
    SRSS->unCLK_FLL_CONFIG.stcField.u1FLL_ENABLE = 1ul;

    /* now do the timeout wait for FLL_STATUS, bit LOCKED */
    while(SRSS->unCLK_FLL_STATUS.stcField.u1LOCKED == 0ul)
    {
        if(timeoutus == 0ul)
        {
            /* If lock doesn't occur, FLL is stopped. */
            Cy_SysClk_FllDisable();
            return(CY_SYSCLK_TIMEOUT);
        }
        Cy_SysLib_DelayUs(1u);
        timeoutus--;
    }

    /* Lock occurred; we need to clear the unlock occurred bit.
       Do so by writing a 1 to it. */
    SRSS->unCLK_FLL_STATUS.stcField.u1UNLOCK_OCCURRED = 1ul;
    /* Set the FLL bypass mode to 3 */
    SRSS->unCLK_FLL_CONFIG3.stcField.u2BYPASS_SEL = (uint32_t)CY_SYSCLK_FLLPLL_OUTPUT_OUTPUT;

    return(CY_SYSCLK_SUCCESS);
}
```

(3) CCO を有効にする。

(4) CCO が利用可能になるまで待機。

(5) タイムアウトの確認。

タイムアウトが発生した場合は FLL が無効。

1 us 待機。

(6) FLL を有効にする

(7) FLL がロックされるまで待機。

(8) タイムアウトの確認。

タイムアウトが発生した場合は FLL が無効。

1 us 待機。

4.2 PLL の設定

4.2.1 操作概要

PLL を使用するためには、PLL を設定する必要があります。**Figure 9** に PLL を設定する手順を示します。PLL の詳細については、**architecture TRM** と **registers TRM** を参照してください。

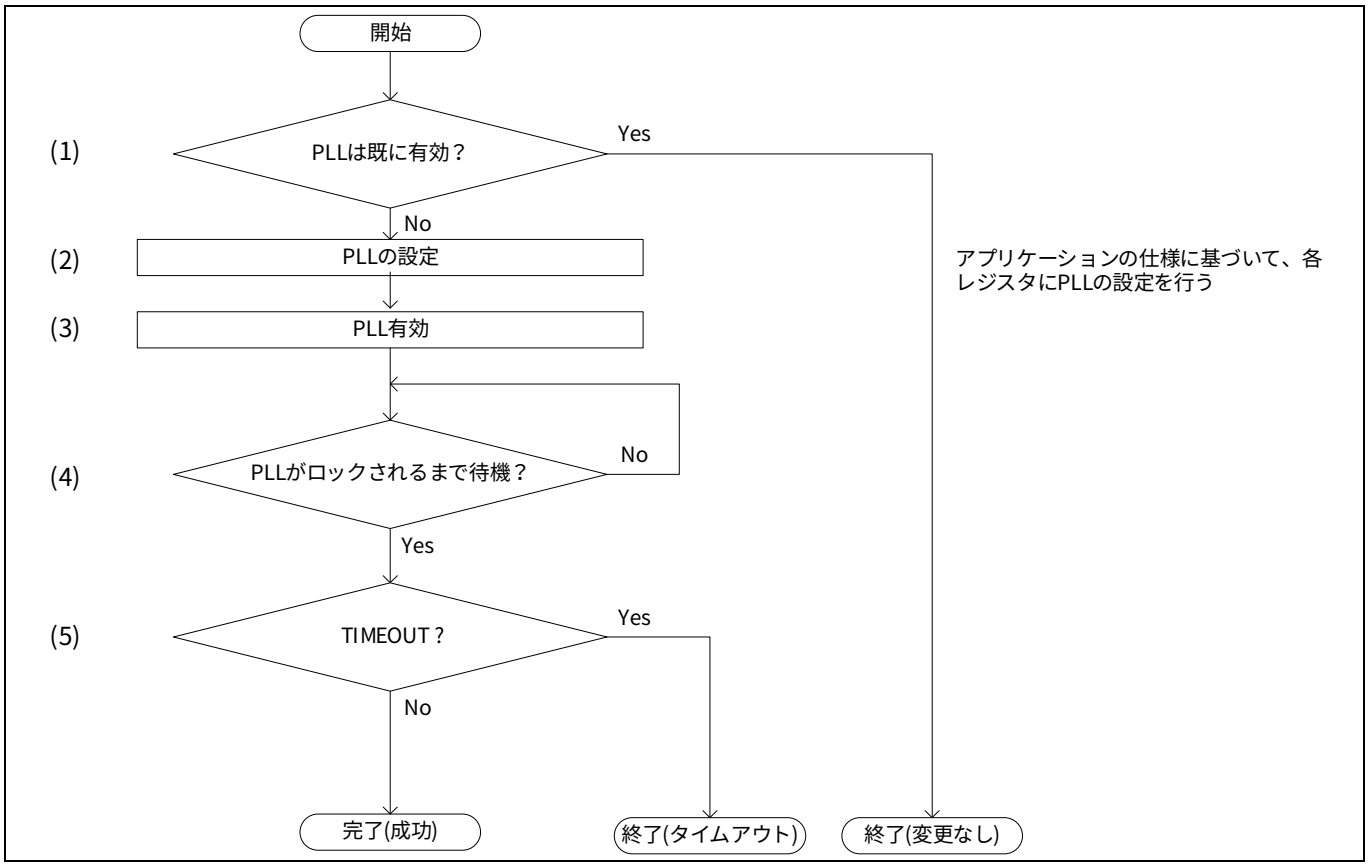

Figure 9 PLL 設定の手順

4.2.2 ユースケース

- 入力クロック周波数: 16 MHz (ECO)
- 出力クロック周波数: 160 MHz
- LF モード: 200 MHz ～ 400 MHz (PLL 出力 320 MHz)

4.2.3 コンフィグレーション

PLL の設定における SDL の構成部のパラメータを **Table 7** に、関数を **Table 8** に示します。

Table 7 PLL 設定パラメーター覧

パラメータ	説明	値
PLL_TARGET_FREQ	PLL ターゲット周波数	160000000ul (160 MHz)
WAIT_FOR_STABILIZATION	発振安定待ち	10000ul
PLL_PATH_NO	PLL 番号	1u
CLK_FREQ_ECO	ECO クロック周波数	16000000ul (16 MHz)

FLL と PLL の設定

パラメータ	説明	値
PATH_SOURCE_CLOCK_FREQ	PLL 入力周波数	CLK_FREQ_ECO
CY_SYSCLK_FLLPLL_OUTPUT_AUTO	FLL 出力モード CY_SYSCLK_FLLPLL_OUTPUT_AUTO: ロックインジケータを自動使用。 CY_SYSCLK_FLLPLL_OUTPUT_LOCKED_OR_NOTHING: AUTO と同様にロック解除でクロックがゲートオフされることを除外。 CY_SYSCLK_FLLPLL_OUTPUT_INPUT: FLL リファレンス入力を選択 (バイパスモード)。 CY_SYSCLK_FLLPLL_OUTPUT_OUTPUT: FLL 出力を選択。ロックインジケータを無視。 詳細については、**registers TRM** の SRSS_CLK_FLL_CONFIG3 を参照	0ul
pllConfig.inputFreq	入力 PLL 周波数	PATH_SOURCE_CLOCK_FREQ
pllConfig.outputFreq	出力 PLL 周波数	PLL_TARGET_FREQ
pllConfig.lfMode	PLL LF モード 0: VCO 周波数 [200MHz, 400MHz] 1: VCO 周波数 [170MHz, 200MHz]	0u (VCO 周波数 320MHz)
pllConfig.outputMode	出力モード 0: CY_SYSCLK_FLLPLL_OUTPUT_AUTO 1: CY_SYSCLK_FLLPLL_OUTPUT_LOCKED_OR_NOTHING 2: CY_SYSCLK_FLLPLL_OUTPUT_INPUT 3: CY_SYSCLK_FLLPLL_OUTPUT_OUTPUT	CY_SYSCLK_FLLPLL_OUTPUT_AUTO

Table 8　　PLL 設定関数一覧

関数	説明	値
`AllClockConfiguration()`	クロック設定	-
`Cy_SysClk_PllConfigure(PLL Number,PLL Configure)`	PLL 番号と PLL の設定	PLL number = PLL_PATH_NO, PLL configure = pllConfig
`Cy_SysClk_PllEnable(PLL Number, Timeout value)`	PLL 番号と PLL モニタの設定	PLL Number = PLL_PATH_NO, Timeout value = WAIT_FOR_STABILIZATION
`Cy_SysLib_DelayUs(Wait Time)`	指定されたマイクロ秒数による遅延	Wait time = 1u (1us)
`Cy_SysClk_PllManualConfigure(PLL Number, PLL Configure)`	PLL の有効化と PLL 値の設定	PLL number = PLL_PATH_NO, PLL manual configure = manualConfig

4.2.4 サンプルコード

サンプルコードを **Code Listing 19** から **Code Listing 23** に示します。

Code Listing 19 PLL の基本設定

```
/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul)


#define CLK_FREQ_ECO           (16000000ul)
#define PATH_SOURCE_CLOCK_FREQ CLK_FREQ_ECO
#define PLL_TARGET_FREQ        (160000000ul)
#define PLL_PATH_NO            (1u)
 :
/*** Parameters for Clock Configuration ***/
cy_stc_pll_config_t pllConfig =
{
    .inputFreq  = PATH_SOURCE_CLOCK_FREQ,       // ECO: 16MHz
    .outputFreq = PLL_TARGET_FREQ,              // target PLL output
    .lfMode     = 0u,                           // VCO frequency is [200MHz, 400MHz]
    .outputMode = CY_SYSCLK_FLLPLL_OUTPUT_AUTO,
};
 :
int main(void)
{
 :
    /* Enable interrupt */
    __enable_irq();

    /* Set Clock Configuring registers */
    AllClockConfiguration();
 :
    /* Please check clock output using oscilloscope after CPU reached here. */
    for(;;);
}
```

TIMEOUT 変数の宣言

ECO 周波数。

PLL 入力周波数。

PLL ターゲット周波数。

PLL 番号の宣言

PLL の設定。

PLL の設定。**Code Listing 20** 参照

Code Listing 20 AllClockConfiguration() 関数

```
static void AllClockConfiguration(void)
{
 :
    /***** PLL (PATH1) source setting ******/
    {
 :
        status = Cy_SysClk_PllConfigure(PLL_PATH_NO , &pllConfig);
        CY_ASSERT(status == CY_SYSCLK_SUCCESS);

        status = Cy_SysClk_PllEnable(PLL_PATH_NO, WAIT_FOR_STABILIZATION);
        CY_ASSERT(status == CY_SYSCLK_SUCCESS);
 :
    }

   return;
}
```

PLL の設定。**Code Listing 21** 参照。

PLL を有効にする。
Code Listing 23 参照。

Code Listing 21 Cy_SysClk_PllConfigure() 関数

```
cy_en_sysclk_status_t Cy_SysClk_PllConfigure(uint32_t clkPath, const cy_stc_pll_config_t *config)
{
    cy_en_sysclk_status_t returnStatus;

    /* check for error */
    if ((clkPath == 0ul) || (clkPath > SRSS_NUM_PLL)) /* invalid clock path number */
    {
        return (CY_SYSCLK_BAD_PARAM);
    }

    if (SRSS->unCLK_PLL_CONFIG[clkPath - 1ul].stcField.u1ENABLE != 0ul) /* 1 = enabled */
    {
        return (CY_SYSCLK_INVALID_STATE);
    }

    /* invalid input frequency */
    if (((config->inputFreq) < MIN_IN_FREQ) || (MAX_IN_FREQ < (config->inputFreq)))
    {
        return (CY_SYSCLK_BAD_PARAM);
```

クロックパスが有効かどうか確認。

(1) PLL が既に有効かどうか確認。

PLL の入力範囲を確認。

FLL と PLL の設定

```
    }

    /* invalid output frequency */
    if (((config->outputFreq) < MIN_OUT_FREQ) || (MAX_OUT_FREQ < (config->outputFreq)))
    {
        return (CY_SYSCLK_BAD_PARAM);
    }

    /* no errors */
    cy_stc_pll_manual_config_t manualConfig = {0ul};

    /* If output mode is bypass (input routed directly to output), then done.
       The output frequency equals the input frequency regardless of the
       frequency parameters. */
    if (config->outputMode != CY_SYSCLK_FLLPLL_OUTPUT_INPUT)
    {
        /* for each possible value of OUTPUT_DIV and REFERENCE_DIV (Q), try
           to find a value for FEEDBACK_DIV (P) that gives an output frequency
           as close as possible to the desired output frequency. */
        uint32_t p, q, out;
        uint32_t error     = 0xFFFFFFFFul;
        uint32_t errorPrev = 0xFFFFFFFFul;

        /* REFERENCE_DIV (Q) selection */
        for (q = MIN_REF_DIV; q <= MAX_REF_DIV; q++)
        {
            /* FEEDBACK_DIV (P) selection */
            for (p = MIN_FB_DIV; p <= MAX_FB_DIV; p++)
            {
                uint64_t inF_MultipliedBy_p  = ((uint64_t)config->inputFreq * (uint64_t)p);
                uint32_t fvco = (uint32_t)(inF_MultipliedBy_p / (uint64_t)q); /* Calculate the intermediate Fvco */
                uint32_t fout;

                /* make sure that fvco in range. */
                if ((fvco < MIN_FVCO) || (MAX_FVCO < fvco))
                {
                    continue;
                }

                /* OUTPUT_DIV selection */
                /* round dividing       */
                out = CY_SYSCLK_DIV_ROUND(inF_MultipliedBy_p, ((uint64_t)config->outputFreq  * (uint64_t)q));

                if(out < MIN_OUTPUT_DIV )
                {
                    out = MIN_OUTPUT_DIV;
                }

                if(MAX_OUTPUT_DIV < out)
                {
                    out = MAX_OUTPUT_DIV;
                }

                /* Calculate what output frequency will actually be produced.
                If it's closer to the target than what we have so far, then save it. */
                fout = (uint32_t)(inF_MultipliedBy_p / (q * out));
                error = abs((int32_t)fout  - (int32_t)config->outputFreq);

                if (error < errorPrev)
                {
                    manualConfig.feedbackDiv  = p;
                    manualConfig.referenceDiv = q;
                    manualConfig.outputDiv    = out;
                    errorPrev = error;
                    if(error == 0ul){break;}
                }
            }
            if(error == 0ul){break;}
        }
        /* exit loops if foutBest equals outputFreq */
    } /* if not bypass output mode */

    /* configure PLL based on calculated values */
    manualConfig.lfMode     = config->lfMode;
    manualConfig.outputMode = config->outputMode;

    returnStatus = Cy_SysClk_PllManualConfigure(clkPath, &manualConfig);
    return (returnStatus);
}
```

PLL の出力範囲を確認。

PLL のパラメータを計算

PLL のレジスタを設定する。
Code Listing 22 参照。

FLL と PLL の設定

Code Listing 22　Cy_SysClk_PllManualConfigure() 関数

```
cy_en_sysclk_status_t Cy_SysClk_PllManualConfigure(uint32_t clkPath, const cy_stc_pll_manual_config_t *config)
{
    /* check for error */
    if ((clkPath == 0ul) || (clkPath > SRSS_NUM_PLL)) /* invalid clock path number */
    {
        return(CY_SYSCLK_BAD_PARAM);
    }

    /* valid divider bitfield values */
    if((config->outputDiv < MIN_OUTPUT_DIV) || (MAX_OUTPUT_DIV < config->outputDiv))
    {
         return(CY_SYSCLK_BAD_PARAM);
    }

    if((config->referenceDiv < MIN_REF_DIV) || (MAX_REF_DIV < config->referenceDiv))
    {
         return(CY_SYSCLK_BAD_PARAM);
    }

    if((config->feedbackDiv  < (config->lfMode ? MIN_FB_DIV_LF : MIN_FB_DIV)) ||
       ((config->lfMode ? MAX_FB_DIV_LF : MAX_FB_DIV) < config->feedbackDiv))
    {
         return(CY_SYSCLK_BAD_PARAM);
    }

    un_CLK_PLL_CONFIG_t tempClkPLLConfigReg;
    tempClkPLLConfigReg.u32Register = SRSS->unCLK_PLL_CONFIG[clkPath - 1ul].u32Register;
    if (tempClkPLLConfigReg.stcField.u1ENABLE != 0ul) /* 1 = enabled */
    {
        return(CY_SYSCLK_INVALID_STATE);
    }

    /* no errors */
    /* If output mode is bypass (input routed directly to output), then done.
       The output frequency equals the input frequency regardless of the frequency parameters. */
    if (config->outputMode != CY_SYSCLK_FLLPLL_OUTPUT_INPUT)
    {
        tempClkPLLConfigReg.stcField.u7FEEDBACK_DIV = (uint32_t)config->feedbackDiv;
        tempClkPLLConfigReg.stcField.u5REFERENCE_DIV = (uint32_t)config->referenceDiv;
        tempClkPLLConfigReg.stcField.u5OUTPUT_DIV = (uint32_t)config->outputDiv;
        tempClkPLLConfigReg.stcField.u1PLL_LF_MODE = (uint32_t)config->lfMode;
    }
    tempClkPLLConfigReg.stcField.u2BYPASS_SEL = (uint32_t)config->outputMode;

    SRSS->unCLK_PLL_CONFIG[clkPath - 1ul].u32Register = tempClkPLLConfigReg.u32Register;

    return (CY_SYSCLK_SUCCESS);
}
```

(2) PLL の設定

CLK_PLL_CONFIG レジスタの設定

Code Listing 23　Cy_SysClk_PllEnable() 関数

```
cy_en_sysclk_status_t Cy_SysClk_PllEnable(uint32_t clkPath, uint32_t timeoutus)
{
    cy_en_sysclk_status_t rtnval = CY_SYSCLK_BAD_PARAM;
    if ((clkPath != 0ul) && (clkPath <= SRSS_NUM_PLL))
    {
        clkPath--; /* to correctly access PLL config and status registers structures */
        /* first set the PLL enable bit */

        SRSS->unCLK_PLL_CONFIG[clkPath].stcField.u1ENABLE = 1ul;

        /* now do the timeout wait for PLL_STATUS, bit LOCKED */
        for (; (SRSS->unCLK_PLL_STATUS[clkPath].stcField.u1LOCKED == 0ul) &&
               (timeoutus != 0ul);
             timeoutus--)
        {
            Cy_SysLib_DelayUs(1u);
        }
        rtnval = ((timeoutus == 0ul) ? CY_SYSCLK_TIMEOUT : CY_SYSCLK_SUCCESS);
    }
    return (rtnval);
}
```

(3) PLL を有効にする

(4) PLL がロックされるまで待機。

(5) タイムアウトの確認。

1 us 待機。

5 内部クロックの設定

クロックシステム中に CLK_HF0 や CLK_FAST などで現れる内部クロックの設定方法について説明します。

5.1 CLK_PATH0, CLK_PATH1, CLK_PATH2 および CLK_PATH3 の設定

CLK_PATH0, CLK_PATH1, CLK_PATH2, および CLK_PATH3 は CLK_HF0, CLK_HF1 および CLK_HF2 の入力ソースとして使用します。CLK_PATH0 と CLK_PATH1 は DSI_MUX と PATH_MUX を使用して FLL と PLL を含むすべてのクロックリソースを選択できます。CLK_PATH2 と CLK_PATH3 は、FLL と PLL を選択できませんが、他のクロックリソースを選択できます。

Figure 10 に CLK_PATH0, CLK_PATH1, CLK_PATH2 および CLK_PATH3 の生成ブロックダイヤグラムを示します。

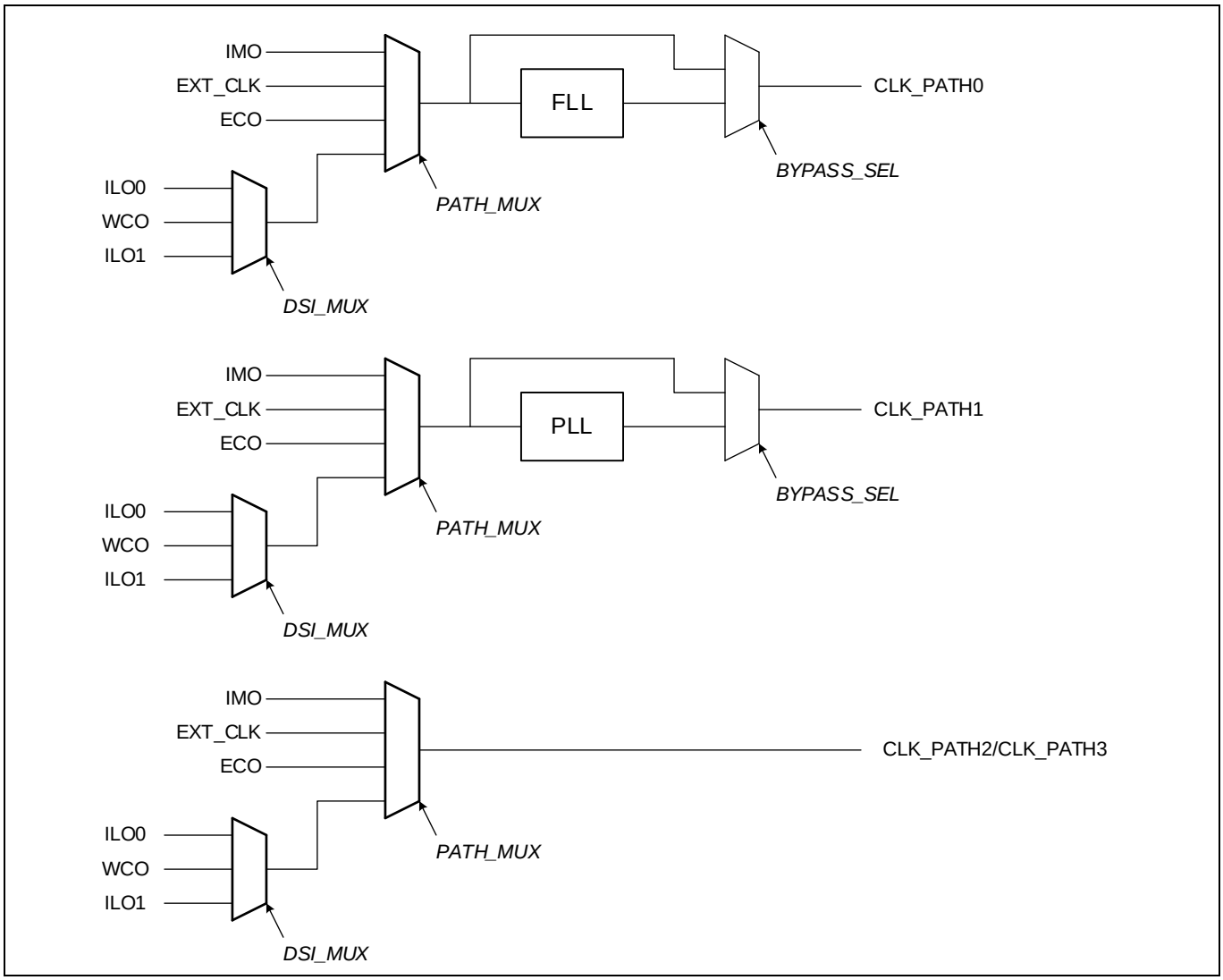

Figure 10 CLK_PATH0, CLK_PATH1, CLKPATH2 および CLKPATH3 の生成ブロックダイヤグラム

CLK_PATH0, CLK_PATH1, CLK_PATH2 および CLK_PATH3 を設定するためには、DSI_MUX と PATH_MUX を設定する必要があります。また BYPASS_SEL は CLK_PATH0 と CLK_PATH1 の設定も必要です。CLK_PATH に必要なレジスタを **Table 9** に示します。詳細については **architecture TRM** と **registers TRM** を参照してください。

内部クロックの設定

Table 9 CLK_PATH0, CLK_PATH1 および CLK_PATH2 の設定

レジスタ名	ビット名	値	選択クロックと項目
CLK_PATH_SELECT	PATH_MUX[2:0]	0 (初期値)	IMO
		1	EXT_CLK
		2	ECO
		4	DSI_MUX
		その他の値	予約済み。使用禁止
CLK_DSI_SELECT	DSI_MUX[4:0]	16	ILO0
		17	WCO
		20	ILO1
		その他の値	予約済み。使用禁止
CLK_FLL_CONFIG3	BYPASS_SEL[29:28]	0 (初期値)	AUTO [1]
		1	LOCKED_OR_NOTHING [2]
		2	FLL_REF (バイパスモード) [3]
		3	FLL_OUT [4]
CLK_PLL_CONFIG	BYPASS_SEL[29:28]	0 (初期値)	AUTO [1]
		1	LOCKED_OR_NOTHING [2]
		2	PLL_REF (バイパスモード) [3]
		3	PLL_OUT [4]

5.2 CLK_HF の設定

CLK_HF0, CLK_HF1 および CLK_HF2 は CLK_PATH0, CLK_PATH1, CLK_PATH2 および CLK_PATH3 から選択できます。Predivider は選択された CLK_PATH0, CLK_PATH1, CLK_PATH2 および CLK_PATH3 を分周するために利用できます。CLK_HF0 は CPU のソースクロックのため、常に有効です。CLK_HF1 と CLK_HF2 は無効にできます。

CLK_HF1 を有効にするためには CLK_ROOT_SELECT レジスタの ENABLE ビットに'1'を書き込みます。CLK_HF1 と CLK_HF2 を無効にするためには、CLK_ROOT_SELECT レジスタの ENABLE ビットに'0'を書き込みます。

CLK_PATH0 は FLL からのクロック出力です。CLK_PATH1 は PLL からのクロック出力です。CLK_PATH2 と CLK_PATH3 は PATH_MUX および DSI_MUX によって選択されたソースクロックになります。CLK_ROOT レジスタの ROOT_DIV ビットは選択肢である分周なし, 2 分周, 4 分周, 8 分周から Predivider の値を設定します。**Figure 11** に ROOT_MUX と Predivider の詳細を示します。

[1] ロック状態に応じて自動的に切り替えます。

[2] ロックが解除されるとクロックはオフになります。

[3] このモードではロック状態は無視されます。

[4] このモードではロック状態は無視されます。

内部クロックの設定

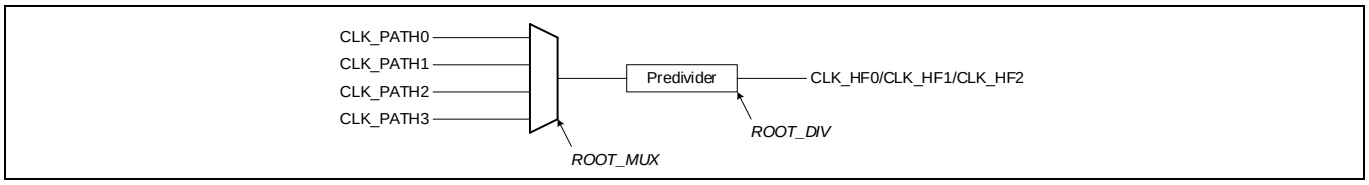

Figure 11　　ROOT_MUX と Predivider

CLK_HF0, CLK_HF1 および CLK_HF2 に必要なレジスタを **Table 10** に示します。**architecture TRM** と **registers TRM** を参照してください。

Table 10　　CLK_HF0 と CLK_HF1 の設定

レジスタ名	ビット名	値	選択項目
CLK_ROOT_SELECT	ROOT_MUX[3:0]	0	CLK_PATH0
		1	CLK_PATH1
		2	CLK_PATH2
		3	CLK_PATH3
		その他の値	予約。使用禁止。
CLK_ROOT_SELECT	ROOT_DIV[5:4]	0	分周なし
		1	2 分周
		2	4 分周
		3	8 分周

5.3　　CLK_LF の設定

CLK_LF は WCO, ILO0, ILO1 および ECO_Prescaler から選択できます。CLK_LF は ILO0 を選択できるため、WDT_CLTL レジスタの WDT_LOCK ビットが無効のときは CLK_LF を設定できません。

Figure 12 に LFCLK_SEL の詳細を示します。

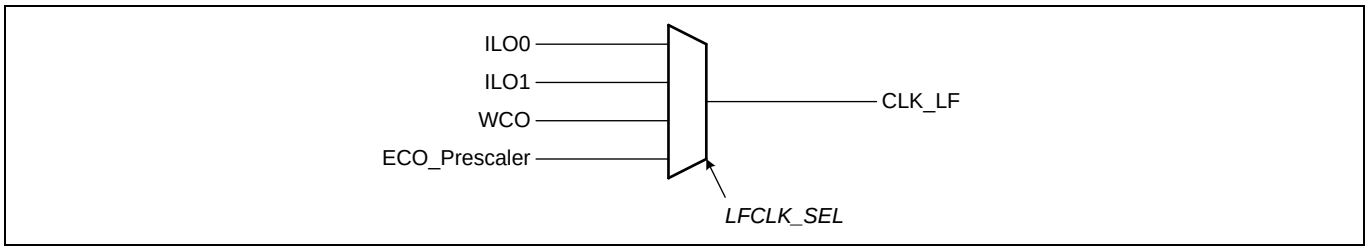

Figure 12　　LFCLK_SEL

CLK_LF に必要なレジスタを **Table 11** に示します。詳細については **architecture TRM** と **registers TRM** を参照してください。

内部クロックの設定

Table 11 CLK_LF の設定

レジスタ名	ビット名	値	選択項目
CLK_SELECT	LFCLK_SEL[2:0]	0	ILO0
		1	WCO
		5	ILO1
		6	ECO_Prescaler
		その他の値	予約。使用禁止

5.4 CLK_FAST の設定

(x+1) による CLK_HF0 の分周によって、CLK_FAST は生成されます。CLK_FAST を設定する場合は、CM4_CLOCK_CTL レジスタの FAST_INT_DIV ビットにより分周する値 (x = 0..255) を設定してください。

5.5 CLK_PERI の設定

CLK_PERI は周辺クロック分周器へのクロック入力です。CLK_HF0 の分周によって CLK_PERI は生成されます。(x+1) での CLK_HF0 分周から得た値から、CLK_PERI の周波数が設定されます。CLK_PERI を設定する場合は、CM0_CLOCK_CTL レジスタの PERI_INT_DIV ビットにより分周する値 (x = 0..255) を設定してください。

5.6 CLK_SLOW の設定

CLK_PERI の分周によって、CLK_SLOW は生成されます。(x+1) で CLK_PERI 分周から得た値で、CLK_SLOW の周波数が設定されます。CLK_PERI を設定した後、CM0_CLOCK_CTL レジスタの SLOW_INT_DIV ビットで分周する値 (x = 0..255) を設定してください。

5.7 CLK_GR の設定

CLK_GR のクロックソースはグループ 1, 2 では CLK_SLOW であり、グループ 3, 5, 6, 9 では CLK_PERI です。グループ 3, 5, 6 および 9 は CLK_PERI から分周されたクロックです。CLK_GR の生成は、分周するための分周値 (1 から 255) を PERI_GR_CLOCK_CTL レジスタの CLOCK_CTL ビットに書き込んでください。

5.8 PCLK の設定

周辺クロック (PCLK) は各周辺機能をアクティブにするクロックです。周辺クロック分周器は CLK_PERI を分周し、各周辺機能に供給するクロックを生成します。周辺クロックの割り当てについては**データシート**の Peripheral Clocks を参照してください。

Figure 13 に周辺クロック分周器を設定する手順を示します。詳細については **architecture TRM** を参照してください。

内部クロックの設定

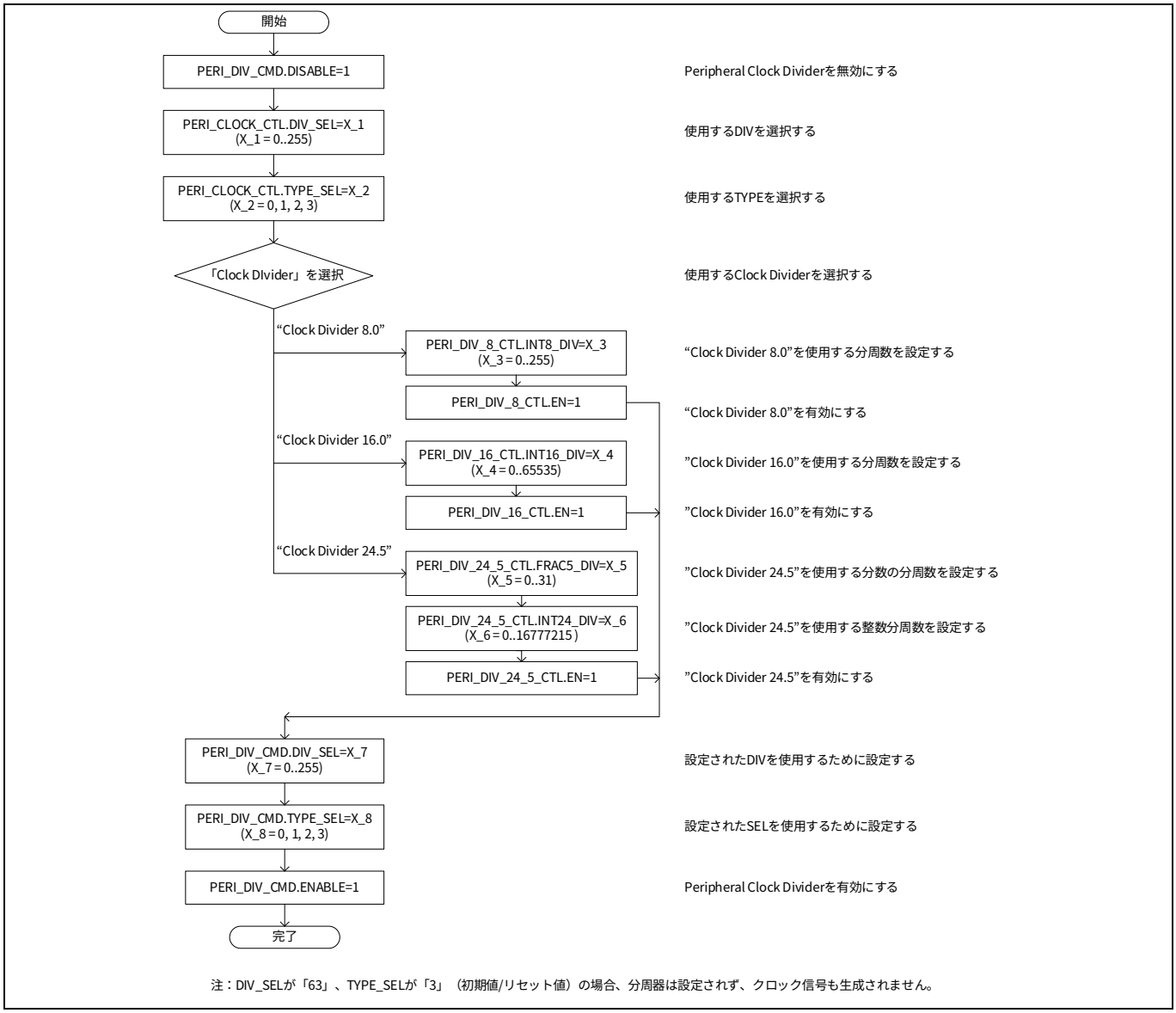

Figure 13　PCLK 生成の設定手順

5.8.1　PCLK の設定例

5.8.1.1　ユースケース

- 入力クロック周波数: 80 MHz
- 出力クロック周波数: 2 MHz
- 分周器のタイプ: Clock divider 16.0
- 使用する分周器: Clock divider 16.0#0
- 周辺機器クロック出力番号: 31 (TCPWM0, Group#0, Counter#0)

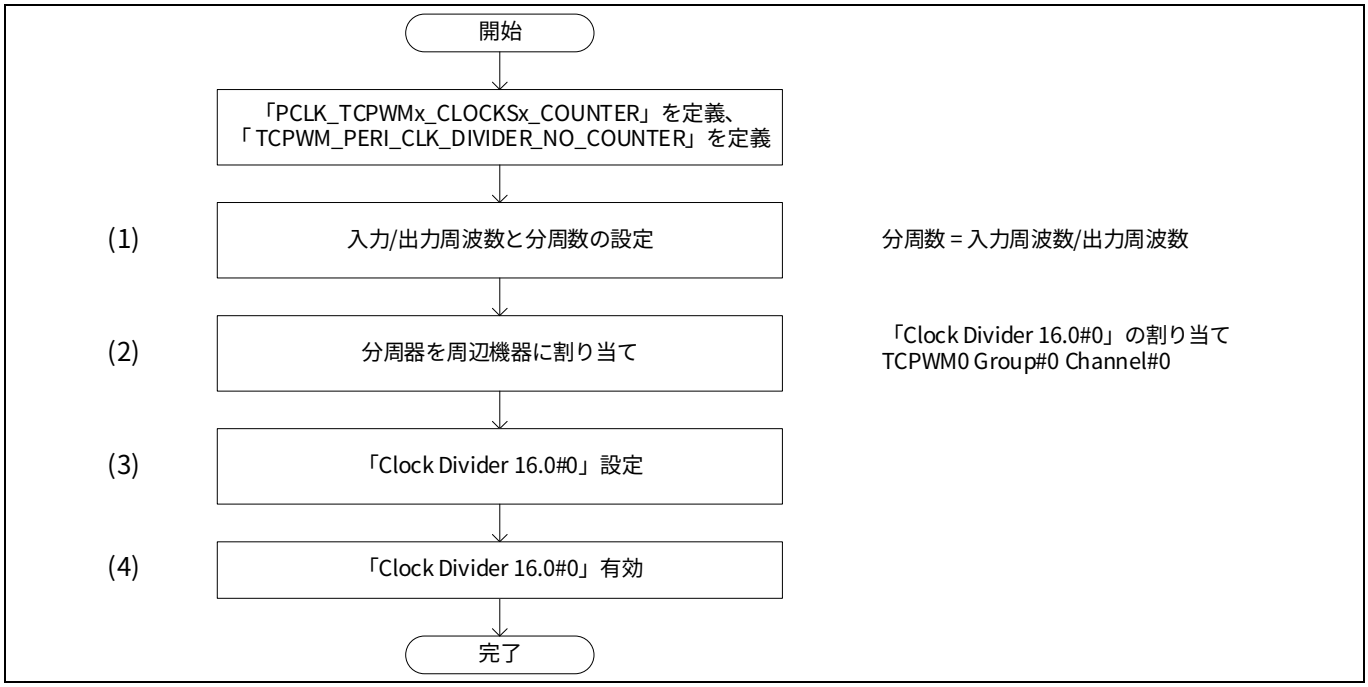

Figure 14　　PCLK の設定手順の例

5.8.2　　コンフィグレーション

PCLK の設定 (TCPWM タイマの例) における SDL の構成部のパラメータを **Table 12** に、関数を Table 13 に示します。

Table 12　　PCLK (TCPWM タイマの例) 設定パラメーター覧

パラメータ	説明	値
PCLK_TCPWMx_CLOCKSx_COUNTER	TCPWM0 の PCLK	PCLK_TCPWM0_CLOCKS0 = 31ul
TCPWM_PERI_CLK_DIVIDER_NO_COUNTER	使用する分周器の番号	0ul
CY_SYSCLK_DIV_16_BIT	分周器のタイプ CY_SYSCLK_DIV_8_BIT = 0u, 8 ビット分周器 CY_SYSCLK_DIV_16_BIT = 1u, 16 ビット分周器 CY_SYSCLK_DIV_16_5_BIT = 2u, 16.5 ビット分数分周器 CY_SYSCLK_DIV_24_5_BIT = 3u, 24.5 ビット分数分周器	1ul
periFreq	**周辺機器のクロック周波数**	80000000ul (80 MHz)
targetFreq	ターゲットクロック周波数	2000000ul (2 MHz)
divNum	分周数	periFreq/targetFreq

内部クロックの設定

Table 13　　PCLK (TCPWM タイマの例) 設定関数一覧

関数	説明	値
`Cy_SysClk_PeriphAssignDivider(IPblock,dividerType, dividerNum)`	選択した IP ブロック (TCPWM など)にプログラム可能な分周器を割り当てる	IPblock = PCLK_TCPWMx_CLOCKSx_COUNTER dividerType = CY_SYSCLK_DIV_16_BIT dividerNum = TCPWM_PERI_CLK_DIVIDER_NO_COUNTER
`Cy_SysClk_PeriphSetDivider(dividerType,dividerNum, dividerValue)`	周辺機器の分周器を設定	dividerType, = CY_SYSCLK_DIV_16_BIT dividerNum = TCPWM_PERI_CLK_DIVIDER_NO_COUNTER dividerValue = divNum-1ul
`Cy_SysClk_PeriphEnableDivider(dividerType,dividerNum)`	周辺機器の分周器を有効にする	dividerType, = CY_SYSCLK_DIV_16_BIT dividerNum = TCPWM_PERI_CLK_DIVIDER_NO_COUNTER

5.8.3　　サンプルコード (TCPWM タイマの例)

サンプルコードを **Code Listing 24** から **Code Listing 27** に示します。

Code Listing 24　　PCLK (TCPWM タイマの例) の基本設定

```
 :
#define PCLK_TCPWMx_CLOCKSx_COUNTER       PCLK_TCPWM0_CLOCKS0
#define TCPWM_PERI_CLK_DIVIDER_NO_COUNTER 0ul
 :
int main(void)
{
    SystemInit();

    __enable_irq(); /* Enable global interrupts. */

    uint32_t periFreq = 80000000ul;
    uint32_t targetFreq = 2000000ul;
    uint32_t divNum = (periFreq / targetFreq);

    CY_ASSERT((periFreq % targetFreq) == 0ul); // inaccurate target clock

    Cy_SysClk_PeriphAssignDivider(PCLK_TCPWMx_CLOCKSx_COUNTER, CY_SYSCLK_DIV_16_BIT,
TCPWM_PERI_CLK_DIVIDER_NO_COUNTER);
    /* Sets the 16-bit divider */
    Cy_SysClk_PeriphSetDivider(CY_SYSCLK_DIV_16_BIT, TCPWM_PERI_CLK_DIVIDER_NO_COUNTER, (divNum-1ul));
    Cy_SysClk_PeriphEnableDivider(CY_SYSCLK_DIV_16_BIT, TCPWM_PERI_CLK_DIVIDER_NO_COUNTER);


    for(;;);
}
```

PCLK_TCPWMx_CLOCKSx_COUNTER の宣言
TCPWM_PERI_CLK_DIVIDER_NO_COUNTER の宣言

(1) 入出力周波数と分周数の設定

分周数の計算

周辺機器の分周器を割り当てる設定。**Code Listing 25** 参照。

周辺機器の分周器を有効にする設定。**Code Listing 27** 参照。

周辺機器の分周器を設定。**Code Listing 26** 参照。

内部クロックの設定

Code Listing 25　Cy_SysClk_PeriphAssignDivider() 関数

```
__STATIC_INLINE cy_en_sysclk_status_t Cy_SysClk_PeriphAssignDivider(en_clk_dst_t ipBlock, cy_en_divider_types_t
dividerType, uint32_t dividerNum)
{
:

    un_PERI_CLOCK_CTL_t tempCLOCK_CTL_RegValue;
    tempCLOCK_CTL_RegValue.u32Register         = PERI->unCLOCK_CTL[ipBlock].u32Register;
    tempCLOCK_CTL_RegValue.stcField.u2TYPE_SEL = dividerType;
    tempCLOCK_CTL_RegValue.stcField.u8DIV_SEL  = dividerNum;
    PERI->unCLOCK_CTL[ipBlock].u32Register     = tempCLOCK_CTL_RegValue.u32Register;

    return CY_SYSCLK_SUCCESS;

}
```

(2) 周辺機器に分周器を割り当て

Code Listing 26　Cy_SysClk_PeriphSetDivider() 関数

```
__STATIC_INLINE cy_en_sysclk_status_t Cy_SysClk_PeriphSetDivider(cy_en_divider_types_t dividerType,
                                              uint32_t dividerNum, uint32_t dividerValue)
{
 :
    if (dividerType == CY_SYSCLK_DIV_8_BIT)
    {
      :
    }
    else if (dividerType == CY_SYSCLK_DIV_16_BIT)
    {
        :
            PERI->unDIV_16_CTL[dividerNum].stcField.u16INT16_DIV =  dividerValue;
        :
    }
    else
    { /* return bad parameter */
        return CY_SYSCLK_BAD_PARAM;
    }

    return CY_SYSCLK_SUCCESS;


}
```

(3) 分周数を“clock divider 16.0#0”に設定

Code Listing 27　Cy_SysClk_PeriphEnableDivider() 関数

```
__STATIC_INLINE cy_en_sysclk_status_t Cy_SysClk_PeriphEnableDivider(cy_en_divider_types_t dividerType, uint32_t
dividerNum)
{
 :
    /* specify the divider, make the reference = clk_peri, and enable the divider */
    un_PERI_DIV_CMD_t tempDIV_CMD_RegValue;
    tempDIV_CMD_RegValue.u32Register            = PERI->unDIV_CMD.u32Register;
    tempDIV_CMD_RegValue.stcField.u1ENABLE      = 1ul;
    tempDIV_CMD_RegValue.stcField.u2PA_TYPE_SEL = 3ul;
    tempDIV_CMD_RegValue.stcField.u8PA_DIV_SEL  = 0xFFul;
    tempDIV_CMD_RegValue.stcField.u2TYPE_SEL    = dividerType;
    tempDIV_CMD_RegValue.stcField.u8DIV_SEL     = dividerNum;
    PERI->unDIV_CMD.u32Register                 = tempDIV_CMD_RegValue.u32Register;

    (void)PERI->unDIV_CMD; /* dummy read to handle buffered writes */

    return CY_SYSCLK_SUCCESS;


}
```

(4) ”clock divider 16#0”を有効にする

分周器のタイプ選択の設定

分周器番号の設定

5.9 ECO プリスケーラの設定

5.9.1 操作概要

ECO プリスケーラは ECO を分周し、LFCLK クロックで使用できるクロックを生成します。分周機能には 10 ビット整数分周器と 8 ビット分数分周器があります。

Figure 15 に ECO プリスケーラを有効にする手順を示します。ECO プリスケーラの詳細については **architecture TRM** と **registers TRM** を参照してください。

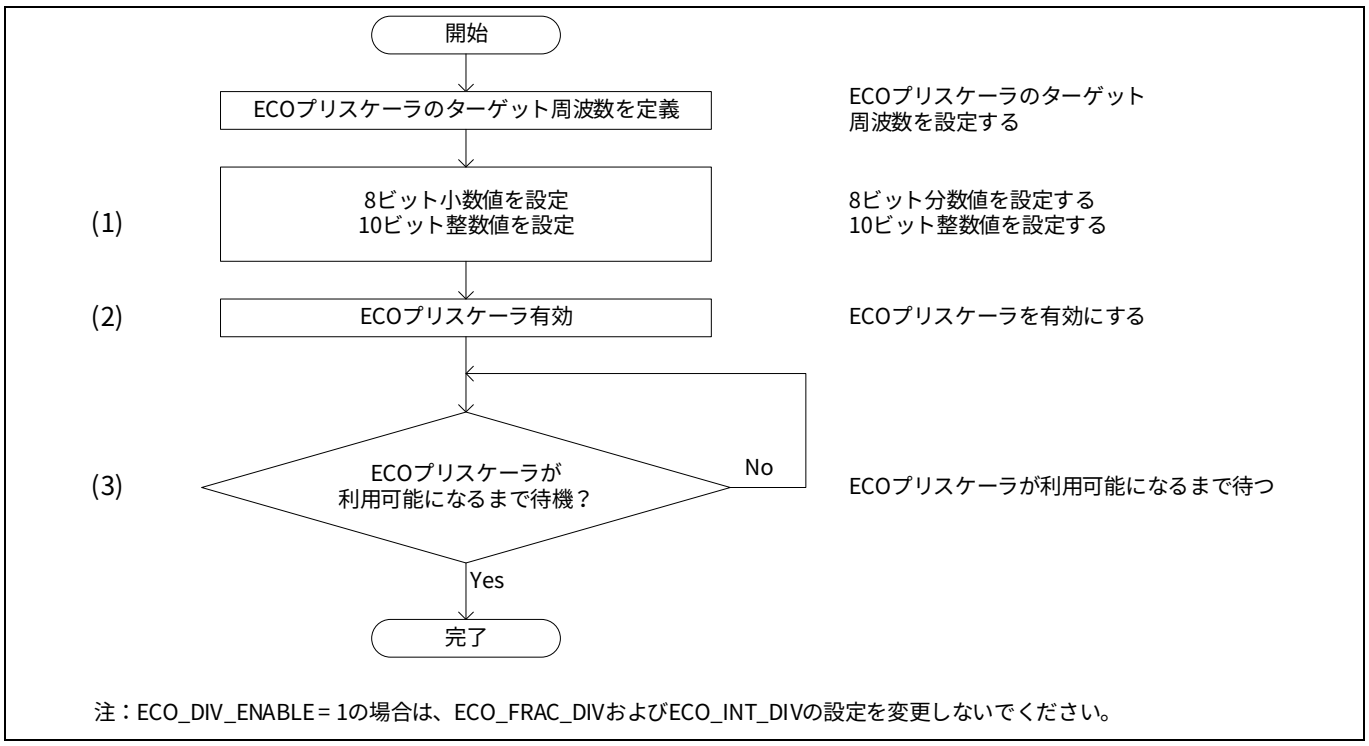

Figure 15 ECO プリスケーラの有効化

Figure 16 に ECO プリスケーラを無効にする手順を示します。ECO プリスケーラの詳細については **architecture TRM** と **registers TRM** を参照してください。

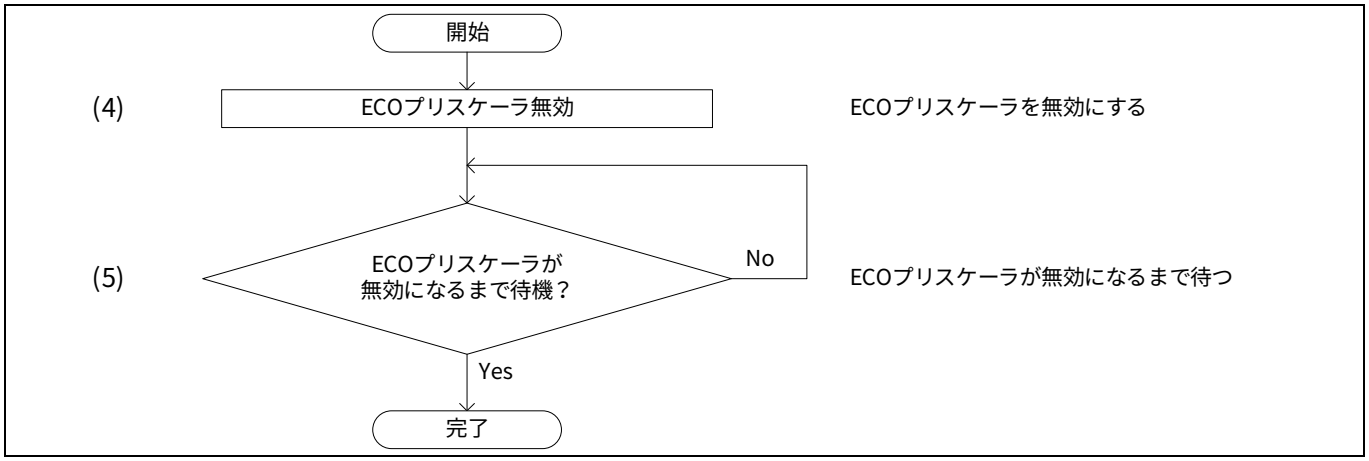

Figure 16 ECO プリスケーラの無効化

5.9.2 ユースケース

- 入力クロック周波数: 16 MHz
- ECO プリスケーラターゲット周波数: 1.234567 MHz

5.9.3 コンフィグレーション

ECO プリスケーラ設定での SDL の設定部のパラメータを **Table 14** に、関数を **Table 15** に示します。

Table 14 ECO プリスケーラ設定パラメーター一覧

パラメータ	説明	値
ECO_PRESCALER_TARGET_FREQ	ECO プリスケーラターゲット周波数	1234567ul
WAIT_FOR_STABILIZATION	発振安定待ち	10000ul
CLK_FREQ_ECO	ECO クロック周波数	16000000ul (16 MHz)
PATH_SOURCE_CLOCK_FREQ	PLL 入力周波数	CLK_FREQ_ECO

Table 15 ECO プリスケーラ設定関数一覧

関数	説明	値
`AllClockConfiguration()`	クロック設定	-
`Cy_SysClk_SetEcoPrescale(Inclk, Targetclk)`	ECO 周波数とターゲット周波数を設定する	Inclk = PATH_SOURCE_CLOCK_FREQ, Targetclk = ECO_PRESCALER_TARGET_FREQ
`Cy_SysClk_EcoPrescaleEnable (Timeout value)`	ECO プリスケーラを有効にしタイムアウト値を設定する	Timeout value = WAIT_FOR_STABILIZATION
`Cy_SysClk_SetEcoPrescaleManual (divInt, divFact)`	divInt: ECO 周波数を考慮に入れた 10 ビット整数値 divFrac: 8 ビット分数値	-
`Cy_SysClk_GetEcoPrescaleStatus`	プリスケーラの状態を確認	-
`Cy_SysLib_DelayUs(WaitTime)`	指定されたマイクロ秒数による遅延	Wait time = 1u (1us)

5.9.4 サンプルコード

サンプルコードを **Code Listing 28** から **Code Listing 34** に示します。

Code Listing 28 ECO プリスケーラの基本設定

```
#define ECO_PRESCALER_TARGET_FREQ (1234567ul)
#define CLK_FREQ_ECO       (16000000ul)
#define PATH_SOURCE_CLOCK_FREQ CLK_FREQ_ECO

/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul)


int main(void)
{
 :
    /* Set Clock Configuring registers */
    AllClockConfiguration();
 :
```

ECO プリスケーラのターゲット周波数を宣言

ECO クロック周波数の宣言

TIMEOUT 変数の宣言

ECO プリスケーラの設定。**Code Listing 29** 参照。

内部クロックの設定

```
    /* Please check clock output using oscilloscope after CPU reached here. */
    for(;;);
}
```

Code Listing 29　AllClockConfiguration() 関数

```
static void AllClockConfiguration(void)
{
    /***** ECO prescaler setting ******/
    {
 :
        cy_en_sysclk_status_t ecoPreStatus;

        ecoPreStatus = Cy_SysClk_SetEcoPrescale(CLK_FREQ_ECO, ECO_PRESCALER_TARGET_FREQ);
        CY_ASSERT(ecoPreStatus == CY_SYSCLK_SUCCESS);

        ecoPreStatus = Cy_SysClk_EcoPrescaleEnable(WAIT_FOR_STABILIZATION);
        CY_ASSERT(ecoPreStatus == CY_SYSCLK_SUCCESS);
    }

   return;
}
```

ECO プリスケーラの設定。**Code Listing 30** 参照。

ECO プリスケーラが有効。
Code Listing 32 参照。

Code Listing 30　Cy_SysClk_SetEcoPrescale() 関数

```
cy_en_sysclk_status_t Cy_SysClk_SetEcoPrescale(uint32_t ecoFreq, uint32_t targetFreq)
{
    // Frequency of ECO (4MHz ~ 33.33MHz) might exceed 32bit value if shifted 8 bit.
    // So, it uses 64 bit data for fixed point operation.
    // Lowest 8 bit are fractional value. Next 10 bit are integer value.
    uint64_t fixedPointEcoFreq = ((uint64_t)ecoFreq << 8ull);
    uint64_t fixedPointDivNum64;
    uint32_t fixedPointDivNum;

    // Culculate divider number
    fixedPointDivNum64 = fixedPointEcoFreq / (uint64_t)targetFreq;

    // Dividing num should be larger 1.0, and smaller than maximum of 10bit number.
    if((fixedPointDivNum64 < 0x100ull) && (fixedPointDivNum64 > 0x40000ull))
    {
        return CY_SYSCLK_BAD_PARAM;
    }

    fixedPointDivNum = (uint32_t)fixedPointDivNum64;

    Cy_SysClk_SetEcoPrescaleManual(
                                   ((((fixedPointDivNum & 0x0003FF00ul) >> 8ul) - 1ul),
                                   (fixedPointDivNum & 0x000000FFul)
                                   );

    return CY_SYSCLK_SUCCESS;
}
```

ECO プリスケーラの設定。**Code Listing 31** 参照。

Code Listing 31　Cy_SysClk_SetEcoPrescaleManual() 関数

```
__STATIC_INLINE void Cy_SysClk_SetEcoPrescaleManual(uint16_t divInt, uint8_t divFract)
{
    un_CLK_ECO_PRESCALE_t tempRegEcoPrescale;
    tempRegEcoPrescale.u32Register             = SRSS->unCLK_ECO_PRESCALE.u32Register;
    tempRegEcoPrescale.stcField.u10ECO_INT_DIV = divInt;
    tempRegEcoPrescale.stcField.u8ECO_FRAC_DIV = divFract;
    SRSS->unCLK_ECO_PRESCALE.u32Register       = tempRegEcoPrescale.u32Register;

    return;
}
```

(1) ECO プリスケーラの設定。

内部クロックの設定

Code Listing 32　Cy_SysClk_EcoPrescaleEnable() 関数

```
cy_en_sysclk_status_t Cy_SysClk_EcoPrescaleEnable(uint32_t timeoutus)
{
    // Send enable command
    SRSS->unCLK_ECO_CONFIG.stcField.u1ECO_DIV_ENABLE = 1ul;

    // Wait eco prescaler get enabled
    while(CY_SYSCLK_ECO_PRESCALE_ENABLE != Cy_SysClk_GetEcoPrescaleStatus())
    {
        if(0ul == timeoutus)
        {
            return CY_SYSCLK_TIMEOUT;
        }

        Cy_SysLib_DelayUs(1u);

        timeoutus--;
    }

    return CY_SYSCLK_SUCCESS;
}
```

(2) ECO プリスケーラを有効にする

(3) ECO プリスケーラが利用可能になるまで待機。**Code Listing 33** 参照。

Code Listing 33　Cy_SysClk_GetEcoPrescaleStatus() 関数

```
__STATIC_INLINE cy_en_eco_prescale_enable_t Cy_SysClk_GetEcoPrescaleStatus(void)
{
    return (cy_en_eco_prescale_enable_t)(SRSS->unCLK_ECO_PRESCALE.stcField.u1ECO_DIV_ENABLED);
}
```

プリスケーラの状態を確認。

ECO プリスケーラを無効にする場合は、上記の関数と同じ方法で待機時間を設定し、次の関数を呼び出します。

Code Listing 34　Cy_SysClk_EcoPrescaleDisable() 関数

```
cy_en_sysclk_status_t Cy_SysClk_EcoPrescaleDisable(uint32_t timeoutus)
{
    // Send disable command
    SRSS->unCLK_ECO_CONFIG.stcField.u1ECO_DIV_DISABLE = 1ul;

    // Wait eco prescaler actually get disabled
    while(CY_SYSCLK_ECO_PRESCALE_DISABLE != Cy_SysClk_GetEcoPrescaleStatus())
    {
        if(0ul == timeoutus)
        {
            return CY_SYSCLK_TIMEOUT;
        }

        Cy_SysLib_DelayUs(1u);

        timeoutus--;
    }

    return CY_SYSCLK_SUCCESS;
}
```

(4) ECO プリスケーラを無効にする。

(5) ECO プリスケーラが無効になるまで待機。**Code Listing 33** 参照。

6 補足情報

6.1 周辺機能へのクロック入力

Table 16 から Table 20 に各周辺機能へのクロック入力を示します。PCLK の詳細値についてはデータシートの Peripheral clocks を参照してください。

Table 16 TCPWM へのクロック入力

周辺機能	動作クロック	チャネルクロック
TCPWM (16 ビット)	CLK_GR3 (グループ 3)	PCLK (PCLK_TCPWM0_CLOCKSx, x = 0〜62)
TCPWM (16 ビット) (Motor Control)		PCLK (PCLK_TCPWM0_CLOCKSy, y = 256〜267)
TCPWM (32 ビット)		PCLK (PCLK_TCPWM0_CLOCKSz, z = 512〜515)

Table 17 CAN FD へのクロック入力

周辺機能	動作クロック (clk_can (cclk))	チャネルクロック (clk_sys (hclk))
CAN FD0	CLK_GR5 (グループ 5)	Ch0: PCLK (PCLK_CANFD0_CLOCK_CANFD0)
		Ch1: PCLK (PCLK_CANFD0_CLOCK_CANFD1)
		Ch2: PCLK (PCLK_CANFD0_CLOCK_CANFD2)
CAN FD1		Ch0: PCLK (PCLK_CANFD1_CLOCK_CANFD0)
		Ch1: PCLK (PCLK_CANFD1_CLOCK_CANFD1)
		Ch2: PCLK (PCLK_CANFD1_CLOCK_CANFD2)

Table 18 LIN へのクロック入力

周辺機能	動作クロック	チャネルクロック (clk_lin_ch)
LIN	CLK_GR5 (グループ 5)	Ch0: PCLK (PCLK_LIN_CLOCK_CH_EN0)
		Ch1: PCLK (PCLK_LIN_CLOCK_CH_EN1)
		Ch2: PCLK (PCLK_LIN_CLOCK_CH_EN2)
		Ch3: PCLK (PCLK_LIN_CLOCK_CH_EN3)
		Ch4: PCLK (PCLK_LIN_CLOCK_CH_EN4)
		Ch5: PCLK (PCLK_LIN_CLOCK_CH_EN5)
		Ch6: PCLK (PCLK_LIN_CLOCK_CH_EN6)
		Ch7: PCLK (PCLK_LIN_CLOCK_CH_EN7)

Table 19 SCB へのクロック入力

周辺機能	動作クロック	チャネルクロック
SCB0	CLK_GR6 (グループ 6)	PCLK (PCLK_SCB0_CLOCK)
SCB1		PCLK (PCLK_SCB1_CLOCK)
SCB2		PCLK (PCLK_SCB2_CLOCK)
SCB3		PCLK (PCLK_SCB3_CLOCK)
SCB4		PCLK (PCLK_SCB4_CLOCK)
SCB5		PCLK (PCLK_SCB5_CLOCK)

補足情報

周辺機能	動作クロック	チャネルクロック
SCB6		PCLK (PCLK_SCB6_CLOCK)
SCB7		PCLK (PCLK_SCB7_CLOCK)

Table 20 SAR ADC へのクロック入力

周辺機能	動作クロック	チャネルクロック
SAR ADC	CLK_GR9 (グループ 9)	Unit0: PCLK (PCLK_PASS_CLOCK_SAR0)
		Unit1: PCLK (PCLK_PASS_CLOCK_SAR1)
		Unit2: PCLK (PCLK_PASS_CLOCK_SAR2)

6.2 クロック調整カウンタ機能のユースケース

6.2.1 クロック調整カウンタの使い方

6.2.1.1 操作概要

クロック調整カウンタには、2 つのクロックソースの周波数を比較するために使用できる 2 つのカウンタがあります。すべてのクロックソースはこれらの 2 つのクロックのクロックソースとして使用できます。

1. Calibration Counter1 は Calibration Clock1 (基準クロックとして使用される高精度クロック) からのクロックパルスをカウントします。Counter1 は降順でカウントします。
2. Calibration Counter2 は Calibration Clock2 (測定クロック) からのクロックパルスをカウントします。このカウンタは昇順でカウントします。
3. Calibration Counter1 が 0 に達すると Calibration Counter2 はカウントを停止し、その値を読み出せます。
4. Calibration Counter2 の周波数はその値と次の式を使用して取得できます。

$$\text{CalibrationClock2} = \frac{\text{Counter2value}}{\text{Counter1value}} \times \text{CalibrationClock1}$$

Figure 17 に ILO0 および ECO を使用した場合のクロック調整カウンタの機能の例を示します。ILO0 および ECO を有効にする必要があります。ECO および ILO0 の設定については、**ILO0/ILO1 の設定**および**ECO の設定**を参照してください。

補足情報

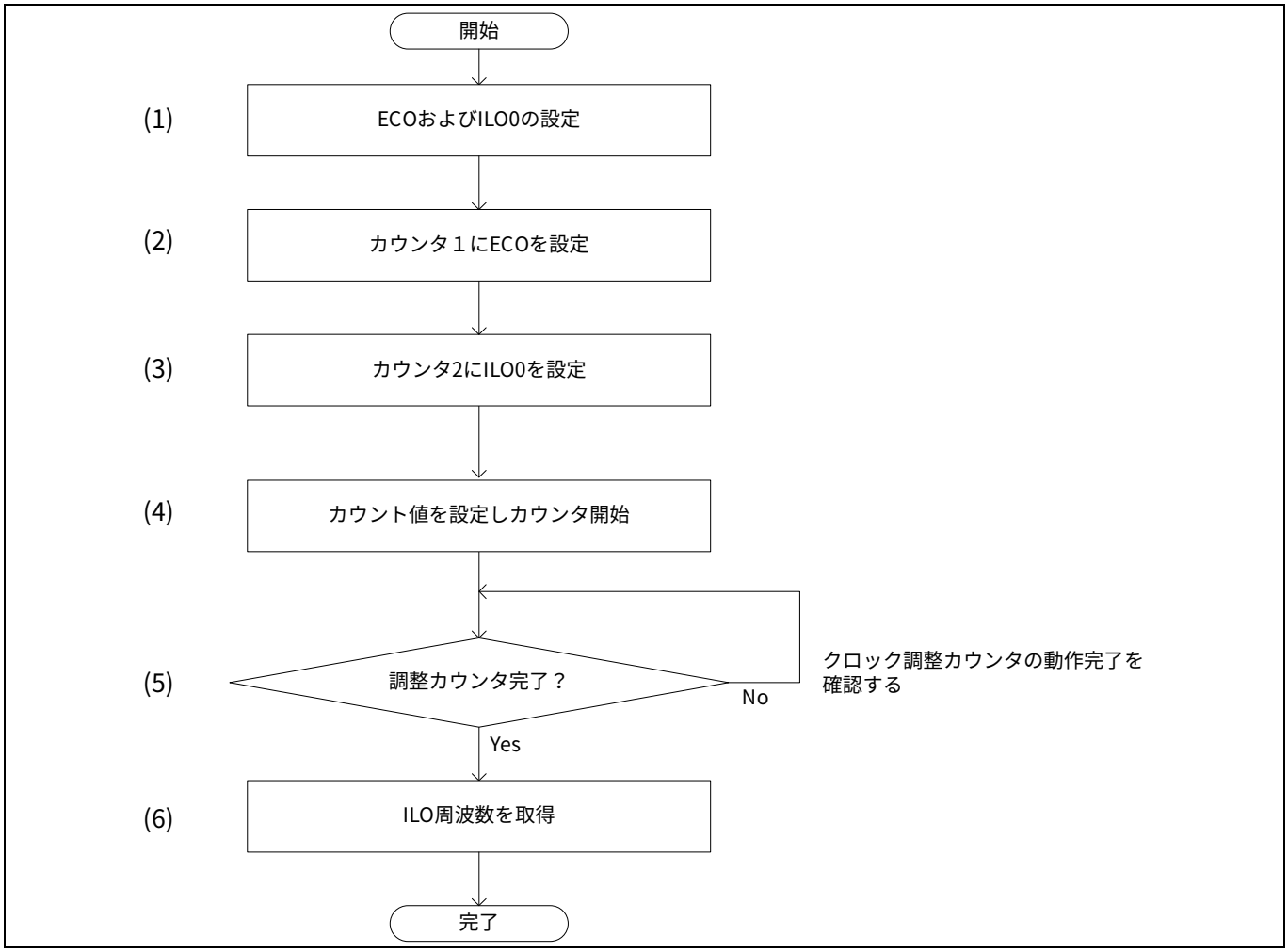

Figure 17　　ILO0 と ECO を用いたクロック調整カウンタの例

6.2.1.2　ユースケース

- 測定クロック: ILO0 クロック周波数 32.768 kHz
- 基準クロック: ECO クロック周波数 16 MHz
- 基準クロックカウント値: 40000ul

6.2.1.3　コンフィグレーション

ILO0 および ECO 設定でのクロック調整カウンタの SDL の設定部のパラメータを **Table 21** に、関数を **Table 22** に示します。

Table 21　　ILO0 および ECO を使用したクロック調整カウンタ設定パラメーター覧

パラメータ	説明	値
ILO_0	ILO_0 設定パラメータを宣言する	0ul
ILO_1	ILO_1 設定パラメータを宣言する	1ul
ILONo	測定クロックを宣言する	ILO_0
clockMeasuredInfo[].name	測定クロック	CY_SYSCLK_MEAS_CLK_ILO0 = 1ul
clockMeasuredInfo[].measuredFreq	測定クロックの分周数を保存する	-

補足情報

パラメータ	説明	値
counter1	基準クロックのカウント値	40000ul
CLK_FREQ_ECO	ECO クロック周波数	16000000ul (16MHz)

Table 22　　ILO0 および ECO を使用したクロック調整カウンタ設定関数一覧

関数	説明	値
`GetILOClockFreq()`	ILO_0 周波数を取得する	-
`Cy_SysClk_StartClkMeasurementCounters(clk1, count1,clk2)`	調整の設定と開始 Clk1: 基準クロック Count1: 測定期間 Clk2: 測定クロック	[カウンタを設定する] clk1 = CY_SYSCLK_MEAS_CLK_ECO = 0x101ul count1 = counter1 clk2 = clockMeasuredInfo[].name
`Cy_SysClk_ClkMeasurementCountersDone()`	カウンタ測定が行われたかどうかを確認する	-
`Cy_SysClk_ClkMeasurementCountersGetFreq (MesauredFreq, refClkFreq)`	測定クロック周波数を取得する MesauredFreq: 保存された測定クロック周波数 refClkFreq: 基準クロック周波数	MesauredFreq = clockMeasuredInfo[].measuredFreq refClkFreq = CLK_FREQ_ECO

6.2.1.4　ILO0 および ECO を使用したクロック調整カウンタの初期設定のサンプルコード

サンプルコードを、**Code Listing 35** に示します。

Code Listing 35　　ILO_0 および ECO を使用したクロック調整カウンタの基本設定

```
#define CY_SYSCLK_DIV_ROUND(a, b) (((a) + ((b) / 2ull)) / (b))

#define ILO_0   0ul
#define ILO_1   1ul
#define ILONo   ILO_0
#define CLK_FREQ_ECO            (16000000ul)

int32_t ILOFreq;

stc_clock_measure clockMeasuredInfo[] =
{
#if(ILONo  == ILO_0)
    {.name = CY_SYSCLK_MEAS_CLK_ILO0,    .measuredFreq= 0ul},
#else
    {.name = CY_SYSCLK_MEAS_CLK_ILO1,    .measuredFreq= 0ul},
#endif
};

int main(void)
{
 :
    /* Enable interrupt */
    __enable_irq();

    /* Set Clock Configuring registers */
    AllClockConfiguration();

    /* return: Frequency of ILO */
    ILOFreq = GetILOClockFreq();

    /* Please check clock output using oscilloscope after CPU reached here. */
    for(;;);
}
```

CY_SYSCLK_DIV_ROUND 関数の宣言

測定クロック(ILO0)の宣言

(1) ECO と ILO0 の設定。**ECO の設定**と **ILO0/ILO1 の設定** 参照

クロック周波数を取得。**Code Listing 36** 参照。

補足情報

Code Listing 36　GetILOClockFreq() 関数

```
uint32_t GetILOClockFreq(void)
{
    uint32_t counter1 = 40000ul;

    if((SRSS->unCLK_ECO_STATUS.stcField.u1ECO_OK == 0ul) || (SRSS->unCLK_ECO_STATUS.stcField.u1ECO_READY == 0ul))
    {
       while(1);
    }

    cy_en_sysclk_status_t status;
    status = Cy_SysClk_StartClkMeasurementCounters(CY_SYSCLK_MEAS_CLK_ECO, counter1, clockMeasuredInfo[0].name);
    CY_ASSERT(status == CY_SYSCLK_SUCCESS);

    while(Cy_SysClk_ClkMeasurementCountersDone() == false);

    status = Cy_SysClk_ClkMeasurementCountersGetFreq(&clockMeasuredInfo[0].measuredFreq, CLK_FREQ_ECO);
    CY_ASSERT(status == CY_SYSCLK_SUCCESS);

:

    uint32_t Frequency  = clockMeasuredInfo[0].measuredFreq;
    return (Frequency);
}
```

ECO の状態を確認

クロック測定カウンタの開始。**Code Listing 37** 参照。

カウンタの測定が終了したかどうか確認。**Code Listing 38** 参照。

ILO 周波数を取得。**Code Listing 39** 参照。

Code Listing 37　Cy_SysClk_StartClkMeasurementCounters() 関数

```
cy_en_sysclk_status_t Cy_SysClk_StartClkMeasurementCounters(cy_en_meas_clks_t clock1, uint32_t count1,
cy_en_meas_clks_t clock2)
{
    cy_en_sysclk_status_t rtnval = CY_SYSCLK_INVALID_STATE;

:
    if (!preventCounting /* don't start a measurement if about to enter DeepSleep mode */  ||
        SRSS->unCLK_CAL_CNT1.stcField.u1CAL_COUNTER_DONE != 0ul/*1 = done*/)
    {
:
    SRSS->unCLK_OUTPUT_FAST.stcField.u4FAST_SEL0 = (uint32_t)clock1;
:
    SRSS->unCLK_OUTPUT_SLOW.stcField.u4SLOW_SEL1 = (uint32_t)clock2;
    SRSS->unCLK_OUTPUT_FAST.stcField.u4FAST_SEL1 =  7ul; /*slow_sel1 output*/;
:
    rtnval = CY_SYSCLK_SUCCESS;

    /* Save this input parameter for use later, in other functions.
    No error checking is done on this parameter.*/
    clk1Count1 = count1;

    /* Counting starts when counter1 is written with a nonzero value. */
    SRSS->unCLK_CAL_CNT1.stcField.u24CAL_COUNTER1 = clk1Count1;
:
    return (rtnval);
}
```

(2) 基準クロック(ECO)を設定

(3) 測定クロック(ILO0)を設定

(4) カウント値とカウンタの開始を設定

Code Listing 38　Cy_SysClk_ClkMeasurementCountersDone() 関数

```
__STATIC_INLINE bool Cy_SysClk_ClkMeasurementCountersDone(void)
{
   return (bool)(SRSS->unCLK_CAL_CNT1.stcField.u1CAL_COUNTER_DONE); /* 1 = done */
}
```

(5) クロック調整カウンタの動作完了を確認

補足情報

Code Listing 39　Cy_SysClk_ClkMeasurementCountersGetFreq() 関数

```
cy_en_sysclk_status_t Cy_SysClk_ClkMeasurementCountersGetFreq(uint32_t *measuredFreq, uint32_t refClkFreq)
{
    if(SRSS->unCLK_CAL_CNT1.stcField.u1CAL_COUNTER_DONE != 1ul)
    {
        return(CY_SYSCLK_INVALID_STATE);
    }

    if(clk1Count1 == 0ul)
    {
        return(CY_SYSCLK_INVALID_STATE);
    }

    volatile uint64_t counter2Value = (uint64_t)SRSS->unCLK_CAL_CNT2.stcField.u24CAL_COUNTER2;

    /* Done counting; allow entry into DeepSleep mode. */
    clkCounting = false;

    *measuredFreq = CY_SYSCLK_DIV_ROUND(counter2Value * (uint64_t)refClkFreq, (uint64_t)clk1Count1 );

    return(CY_SYSCLK_SUCCESS);
}
```

ILO 0 カウント値を取得

(6) ILO 0 周波数を取得

6.2.2　クロック調整カウンタ機能を使用した ILO0 の校正

6.2.2.1　操作概要

ILO 周波数は製造時に決定されます。しかし電圧および温度条件によって、実際の ILO 周波数の値は変化するため、適宜調整 (校正) できます。

ILO 周波数のトリミングは CLK_TRIM_ILOx_CTL レジスタの ILOx_FTRIM ビットを使用して更新できます。ILOx_FTRIM ビットの初期値は 0x2C です。このビットの値を 0x01 増加すると周波数が 1.5% (標準) 増加します。このビット値を 0x01 だけ下げると周波数が 1.5% (標準) 低下します。CLK_TRIM_ILO0_CTL レジスタは WDT_CTL.ENABLE によって保護されています。WDT_CTL レジスタの仕様については TRAVEO™ T2G **architecture TRM** の Watchdog Timer を参照してください。

Figure 18 にクロック調整カウンタと CLK_TRIM_ILOx_CTL レジスタを使用した ILO0 校正のフロー例を示します。

補足情報

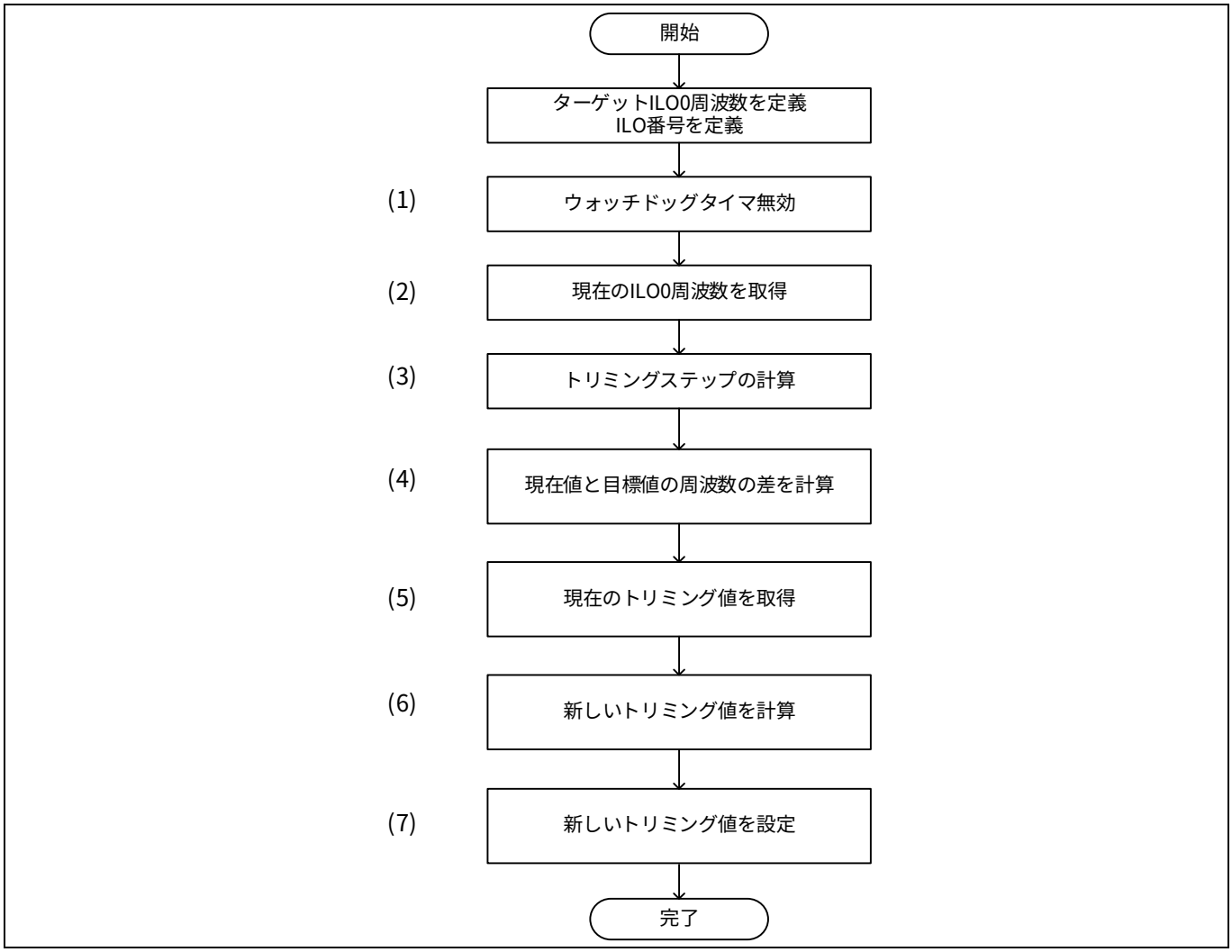

Figure 18　　ILO0 の校正

6.2.2.2　コンフィグレーション

クロック調整カウンタ設定を使用した ILO0 校正での SDL の設定部のパラメータを **Table 23** に、関数を **Table 24** に示します。

Table 23　　クロック調整カウンタ設定を使用した ILO0 校正設定パラメーター一覧

パラメータ	説明	値
CY_SYSCLK_ILO_TARGET_FREQ	ILO ターゲット周波数	32768ul (32.768 KHz)
ILO_0	ILO_0 設定パラメータの宣言	0ul
ILO_1	ILO_1 設定パラメータの宣言	1ul
ILONo	測定クロックの宣言	ILO_0
iloFreq	現在の ILO 0 周波数を保存する	-

補足情報

Table 24　クロック調整カウンタ設定を使用した ILO0 校正設定関数一覧

関数	説明	値
`Cy_WDT_Disable ()`	ウォッチドッグタイマ無効	-
`Cy_WDT_Unlock()`	ウォッチドッグタイマのロックを解除する	-
`GetILOClockFreq()`	現在の ILO 0 周波数を取得する	-
`Cy_SysClk_IloTrim (iloFreq, iloNo)`	トリム設定 iloFreq:現在の ILO 0 周波数 iloNo: ILO 番号のトリミング	iloFreq: iloFreq iloNo: ILONo

6.2.2.3　クロック調整カウンタ設定を使用した ILO0 校正の初期設定のサンプルコード

サンプルコードを **Code Listing 40** に示します。

Code Listing 40　クロック調整カウンタ設定を使用した ILO 0 校正の基本設定

```
#define CY_SYSCLK_DIV_ROUND(a, b) (((a) + ((b) / 2ull)) / (b))

#define CY_SYSCLK_ILO_TARGET_FREQ  32768ul

#define ILO_0   0
#define ILO_1   1
#define ILONo   ILO_0

int32_t iloFreq;

int main(void)
{

    /* Enable global interrupts. */
    __enable_irq();

    Cy_WDT_Disable();
:
    /* return: Frequency of ILO */
    ILOFreq = GetILOClockFreq();
:
    /* Must unlock WDT befor update Trim */
    Cy_WDT_Unlock();
    Trim_diff = Cy_SysClk_IloTrim(ILOFreq,ILONo);
:
    for(;;);

}
```

CY_SYSCLK_DIV_ROUND 関数の宣言

ターゲット ILO 0 周波数を宣言

ILO 0 番号を宣言

(1) ウォッチドッグタイマ無効。

(2) 現在の ILO 0 周波数を取得。**Code Listing 36** 参照。

ウォッチドッグタイマのロック解除

ILO 0 をトリミング。**Code Listing 41** 参照。

補足情報

Code Listing 41　Cy_SysClk_IloTrim() 関数

```
int32_t Cy_SysClk_IloTrim(uint32_t iloFreq, uint8_t iloNo)
{
    /* Nominal trim step size is 1.5% of "the frequency". Using the target frequency. */
    const uint32_t trimStep = CY_SYSCLK_DIV_ROUND((uint32_t)CY_SYSCLK_ILO_TARGET_FREQ * 15ul, 1000ul);

    uint32_t newTrim = 0ul;
    uint32_t curTrim = 0ul;

    /* Do nothing if iloFreq is already within one trim step from the target */
    uint32_t diff = (uint32_t)abs((int32_t)iloFreq - (int32_t)CY_SYSCLK_ILO_TARGET_FREQ);
    if (diff >= trimStep)
    {
        if(iloNo == 0u)
        {
            curTrim = SRSS->unCLK_TRIM_ILO0_CTL.stcField.u6ILO0_FTRIM;
        }
        else
        {
            curTrim = SRSS->unCLK_TRIM_ILO1_CTL.stcField.u6ILO1_FTRIM;
        }

        if (iloFreq > CY_SYSCLK_ILO_TARGET_FREQ)
        { /* iloFreq is too high. Reduce the trim value */
            newTrim = curTrim - CY_SYSCLK_DIV_ROUND(iloFreq - CY_SYSCLK_ILO_TARGET_FREQ, trimStep);
        }
        else
        { /* iloFreq too low. Increase the trim value. */
            newTrim = curTrim + CY_SYSCLK_DIV_ROUND(CY_SYSCLK_ILO_TARGET_FREQ - iloFreq, trimStep);
        }

        /* Update the trim value */
        if(iloNo == 0u)
        {
            if(WDT->unLOCK.stcField.u2WDT_LOCK != 0ul) /* WDT registers are disabled */



            {
                return(CY_SYSCLK_INVALID_STATE);
            }
            SRSS->unCLK_TRIM_ILO0_CTL.stcField.u6ILO0_FTRIM = newTrim;
        }
        else
        {
            SRSS->unCLK_TRIM_ILO1_CTL.stcField.u6ILO1_FTRIM = newTrim;
        }
    }
    return (int32_t)(curTrim - newTrim);
}
```

(3) トリミングステップの計算

(4) 現在の周波数とターゲット周波数の差異を

差分がトリミングステップよりも大きいかどうかを確認。

(5) 現在のトリミング値を読出し

現在の周波数がターゲット周波数よりも小さいかどうか確認。

(6) 新しいトリム値を計算。

ウォッチドッグタイマが無効かどうか確認

(7) 新しいトリミング値を設定

7 用語集

用語	説明
FPU	Floating point unit
RTC	Real time clock
IMO	Internal main oscillator
ILO	Internal low-speed oscillators
ECO	External crystal oscillator
WCO	Watch crystal oscillator
EXT_CLK	External clock
PLL	Phase Locked Loop
FLL	周波数ロックループ
CLK_HF	High frequency clock。CLK_HF は CLK_FAST と CLK_SLOW を動作させます。CLK_HF, CLK_FAST および CLK_SLOW は同期しています。
CLK_FAST	Fast Clock。CLK_FAST は CM4 と CPUSS Fast infrastructure に使用されます。
CLK_SLOW	Slow Clock。CLK_SLOW は CM0+と CPUSS Slow infrastructure に使用されます。
CLK_PERI	Peripheral clock。CLK_PERI は CLK_SLOW, CLK_GR および周辺クロック分周器のクロックソースです。
CLK_GR	Group clock。CLK_GR は周辺機能へのクロック入力です。
Peripheral clock divider	周辺クロック分周器は各周辺機能で使用するためのクロックを駆動します。
MCWDT	Multi-counter watchdog timer。詳細は TRAVEO™ T2G **architecture TRM** の Watchdog timer 章を参照してください。
TCPWM	Timer, counter, and pulse width modulator。詳細は TRAVEO™ T2G **architecture TRM** の Timer, counter, and PWM 章を参照してください。
CAN FD	CAN FD は CAN with Flexible Data rate のことであり、CAN は Controller Area Network です。詳細は TRAVEO™ T2G **architecture TRM** の CAN FD controller 章を参照してください。
LIN	Local Interconnect Network。詳細は TRAVEO™ T2G **architecture TRM** の Local Interconnect Network (LIN) 章を参照してください。
SCB	Serial communications block。詳細は TRAVEO™ T2G **architecture TRM** の Serial communications block (SCB) 章を参照してください。
SAR ADC	Successive approximation register analog-to-digital converter。詳細は TRAVEO™ T2G **architecture TRM** の SAR ADC 章を参照してください。
Clock calibration counter	クロック調整カウンタには、2 つのクロックを使用してクロックを校正する機能があります。

8 関連ドキュメント

- デバイスデータシート
  - **CYT2B7 datasheet 32-bit Arm® Cortex®-M4F microcontroller TRAVEO™ T2G family**
  - **CYT2B9 datasheet 32-bit Arm® Cortex®-M4F microcontroller TRAVEO™ T2G family**
- Body Controller Entry ファミリ
  - **TRAVEO™ T2G automotive body controller entry family architecture technical reference manual (TRM)**
  - **TRAVEO™ T2G automotive body controller entry registers technical reference manual (TRM) for CYT2B7**
  - **TRAVEO™ T2G automotive body controller entry registers technical reference manual (TRM) for CYT2B9**
- ユーザガイド
  - **Setting ECO parameters in TRAVEO™ T2G family user guide**

9 その他の参考資料

さまざまな周辺機器にアクセスするためのサンプルソフトウェアとしてのスタートアップを含むサンプルドライバライブラリ (SDL) が提供されます。SDL は、公式の AUTOSAR 製品でカバーされないドライバの顧客へのリファレンスとしても機能します。SDL は自動車規格に適合していないため、製造目的で使用できません。このアプリケーションノートのプログラムコードは SDL の一部です。SDL の入手については、**テクニカルサポート**に連絡してください。

改訂履歴

Document version	Date of release	Description of changes
**	2019-07-01	このドキュメントは英語版 002-20208 Rev.**を翻訳した日本語版 002-26051 Rev.**です。
*A	2022-01-14	このドキュメントは英語版 002-20208 Rev.*B を翻訳した日本語版 002-26051 Rev.*A です。

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2022-01-14
Published by
Infineon Technologies AG
81726 Munich, Germany

Do you have a question about this document?
Go to www.cypress.com/support

Document reference
002-26051 Rev. *A

重要事項
本文書に記載された情報は、いかなる場合も、条件または特性の保証とみなされるものではありません（「品質の保証」）。本文に記された一切の事例、手引き、もしくは一般的価値、および／または本製品の用途に関する一切の情報に関し、インフィニオンテクノロジーズ（以下、「インフィニオン」）はここに、第三者の知的所有権の不侵害の保証を含むがこれに限らず、あらゆる種類の一切の保証および責任を否定いたします。

さらに、本文書に記載された一切の情報は、お客様の用途におけるお客様の製品およびインフィニオン製品の一切の使用に関し、本文書に記載された義務ならびに一切の関連する法的要件、規範、および基準をお客様が遵守することを条件としています。

本文書に含まれるデータは、技術的訓練を受けた従業員のみを対象としています。本製品の対象用途への適合性、およびこれら用途に関連して本文書に記載された製品情報の完全性についての評価は、お客様の技術部門の責任にて実施してください。

本製品、技術、納品条件、および価格についての詳しい情報は、インフィニオンの最寄りの営業所までお問い合わせください（**www.infineon.com**）。

警告事項
技術的要件に伴い、製品には危険物質が含まれる可能性があります。当該種別の詳細については、インフィニオンの最寄りの営業所までお問い合わせください。

インフィニオンの正式代表者が署名した書面を通じ、インフィニオンによる明示の承認が存在する場合を除き、インフィニオンの製品は、当該製品の障害またはその使用に関する一切の結果が、合理的に人的傷害を招く恐れのある一切の用途に使用することはできないこと予めご了承ください。