TRAVEO™ T2G ファミリ CYT4D シリーズ MCU のクロック設定方法

About this document

Scope and purpose

AN226071 は TRAVEO™ T2G ファミリ CYT4D シリーズの MCU における様々なクロックソースの設定方法を説明し、PLL/FLL の設定例および ILO の校正方法を提供します。

関連製品ファミリ

TRAVEO™ T2G ファミリ CYT4D シリーズ自動車用マイクロコントローラ

Intended audience

本書は、TRAVEO™ T2G ファミリ CYT4D シリーズ MCU 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

インスツルメントクラスターやヘッドアップディスプレイ (HUD) などの車載システム向けの TRAVEO™ T2G ファミリ MCU は、高度な 40nm プロセス技術で製造され、FPU (単精度と倍精度) 付き Arm® Cortex®-M7 プロセッサをベースにした 32 ビット車載向けマイクロコントローラであり、2D グラフィックエンジンやサウンドプロセッサを搭載します。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および外部クロックソースの両方を使用して高速クロックと低速クロックをサポートしています。クロック入力の典型的な使用例の 1 つは内蔵のリアルタイムクロック (RTC) です。TRAVEO™ T2G MCU は高速で内部回路が動作するクロックを生成するための位相ロックループ (PLL) と周波数ロックループ (FLL) に対応します。

TRAVEO™ T2G MCU はクロック動作を監視し、既知のクロックを参照して各クロックのクロック差を測定する機能も対応します。

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より理解するためには **architecture technical reference manual (architecture 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 CYT4D 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されます。1 つのブロックはクロックリソース (外部発振や内部発振など) を選択し、FLL や 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 コアや他の周辺機能に分配および分割をします。ただし、クロックリソースに直接接続している RTC などいくつか例外はあります。

Figure 1 にクロックシステムの構造の概要を示します。

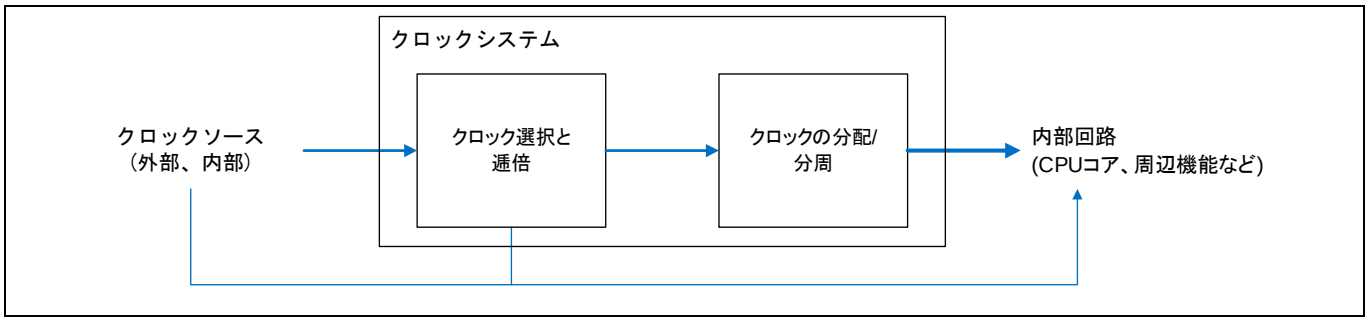

Figure 1 クロックシステム構造の概要

2.2 クロックリソース

MCU は内部と外部の 2 つのタイプのリソース入力に対応します。これらはそれぞれ内部で 3 種類のクロックに対応します。

- 内部クロックソース (これらすべてのクロックは初期設定で有効です。)
  - 内部メイン発振 (IMO): このクロックは 8 MHz (標準) の周波数である内蔵クロックです。
  - 内部低速発振 0 (ILO0): このクロックは 32 kHz (標準) の周波数である内蔵クロックです。
  - 内部低速発振 1 (ILO1): ILO1 は ILO0 と同じ機能を持ちますが、ILO1 は ILO0 のクロックを監視できます。
- 外部クロックソース (これらすべてのクロックは初期値で無効です。)
  - 外部水晶発振 (ECO): このクロックは入力周波数範囲が 3.988 MHz～33.34 MHz の外部発振子を使用します。
  - 時計用水晶発振 (WCO): このクロックも周波数が 32.768kHz で安定している外部発振子を使用し、主に RTC で使用されます。
  - 外部クロック (EXT_CLK): EXT_CLK は 0.25 MHz～100 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、または直接的に高周波クロックとしても使用できます。
  - 低電力外部水晶発振 (LPECO): このクロックは外部発振子を使用します。入力周波数範囲は 4MHz と 8MHz の間です。LPECO は停電モードで動作する ECO と見なせます。

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 **Datasheet** を参照してください。

2.3 クロックシステムの機能説明

ここではクロックシステムの機能を説明します。

TRAVEO™ T2G ファミリ MCU のクロックシステム

クロック選択と逓倍ブロックの詳細を **Figure 2** に示します。このブロックはクロックリソースから CLK_HF0～CLK_HF13 までのルートクロックを生成します。このブロックは対応するクロックリソース, FLL, および PLL から 1 つを選択する機能と要求される高速クロックを生成する機能をもちます。MCU は 2 つのタイプの PLL に対応します。それは SSCG (Spread spectrum clock generator) と Fractional operation を持たない PLL (PLL200#x) と、SSCG と Fractional operation を持つ PLL (PLL400#x) です。

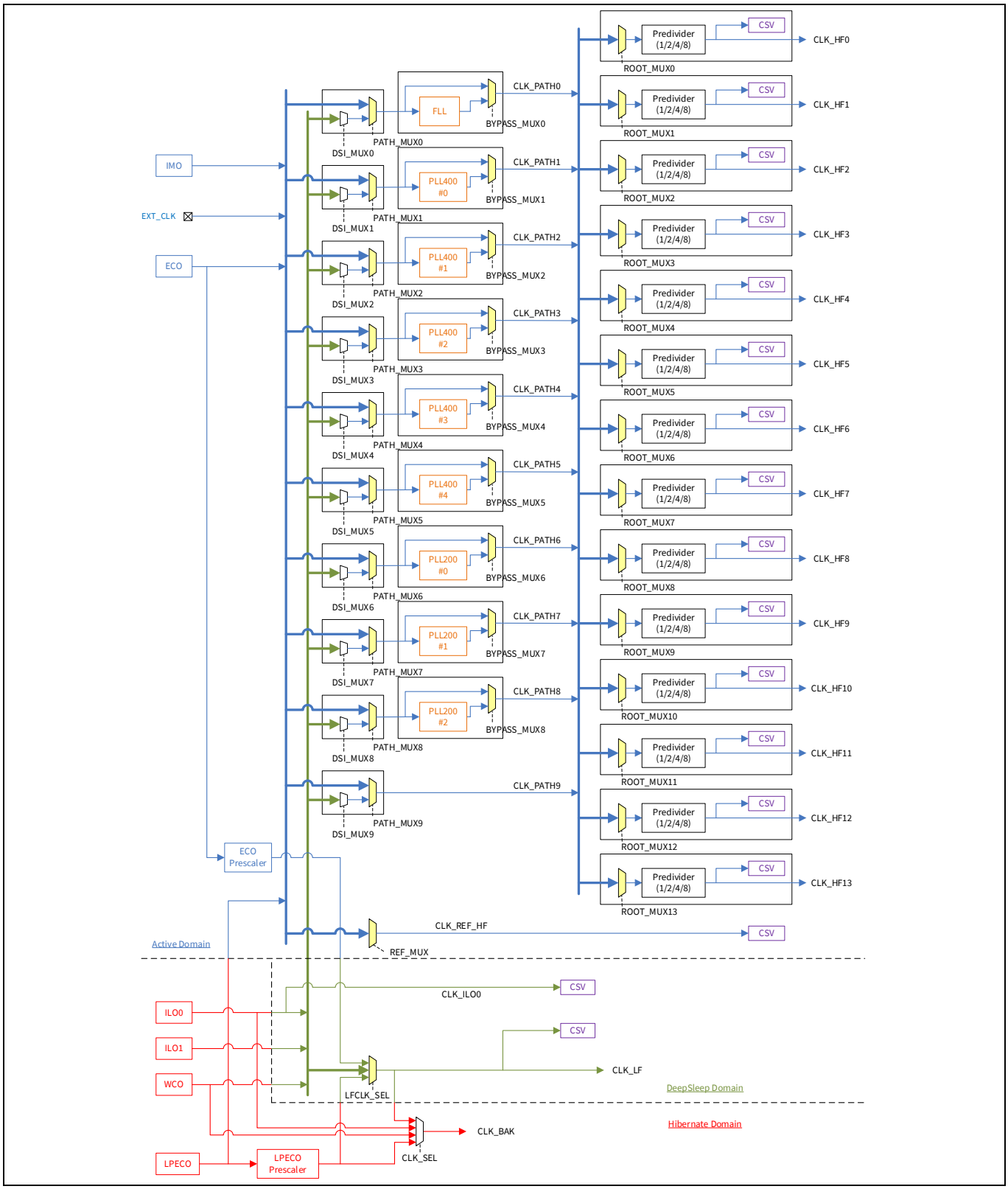

Figure 2　　ブロックダイヤグラム

TRAVEO™ T2G ファミリ MCU のクロックシステム

Active domain	アクティブパワーモードのみで動作する領域。
DeepSleep domain	アクティブと DeepSleep パワーモードで動作する領域。
Hibernate domain	すべてのパワーモードで動作する領域。
ECO prescaler	ECO を分周し、CLK_LF 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。
LPECO prescaler	LPECO を分周し、CLK_BAK 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	ILO0, ILO1 および WCO からクロックを選択します。
PATH_MUX	IMO, ECO, EXT_CLK および DSI_MUX の出力からクロックを選択します。
CLK_PATH	CLK_PATHx(0 から 9)は高周波数クロックの入力ソースとして使用されます。
CLK_HF	CLK_HFs(0 から 13)は高周波数クロックです。
FLL	高周波クロックを生成します。
PLL	高周波クロックを生成します。PLL は PLL200 と PLL400 の 2 種類があります。PLL200 は SSCG と Fractional operation があり、PLL400 は SSCG と Fractional operation があります。
BYPASS_MUX	CLK_PATH に出力させるクロックを選択します。FLL の場合、選択できるクロックは FLL の出か FLL への入力クロックです。
ROOT_MUX	CLK_HFx のクロックソースを選択します。選択できるクロックは CLK_PATHs (0 から 9) です。
Predivider	Predivider (1, 2, 4, または 8 で分周) は選択された CLK_PATH を分周するために利用できます。
REF_MUX	CLK_REF_HF のクロックソースを選択します。
CLK_REF_HF	CLK_HF の CSV を監視するために使用します。
LFCLK_SEL	CLK_LF のクロックソースを選択します。
CLK_LF	MCWDT のソースクロックです。
CLK_SEL	RTC に入力されるクロックを選択します。
CLK_BAK	主に RTC に入力します。
CSV	Clock supervision であり、クロックの動作を監視します。

CLK_HF0 の分配先を **Figure 3** に示します。

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。**Figure 3** の詳細については **architecture TRM** および **Datasheet** を参照してください。

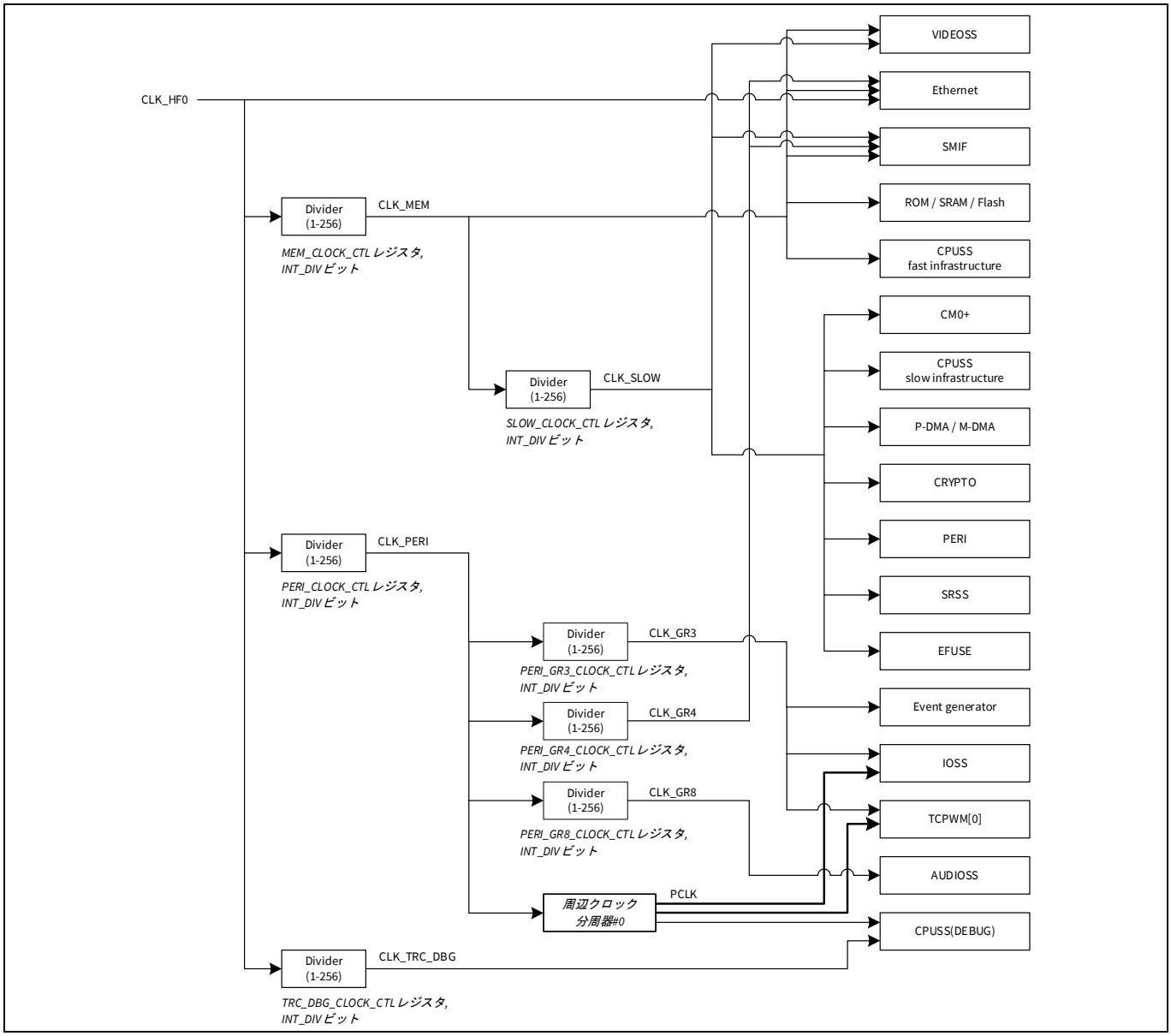

Figure 3 CLK_HF0 のブロックダイヤグラム

CLK_MEM	Fast infrastructure, Ethernet, および Serial memory interface (SMIF) の CPUSS へのクロック入力
CLK_PERI	CLK_GR と周辺クロック分周器のクロックソース
CLK_SLOW	Cortex®-M0+の CPUSS と、 slow infrastructure, SMIF, および VIDEOSS のクロック入力
CLK_GR	周辺機能へのクロック入力です。CLK_GR は Clock gater でグループ分けされます。CLK_GR は 6 つのグループを持ちます。
PCLK	周辺機能で使用される周辺クロックです。PCLK は IP の各チャネルを個別に設定でき、PCLK を生成するため 1 つの分周器を選択します。

CLK_TRC_DBG　　CPUSS (DEBUG) へのクロック入力です。

Divider　　Divider は各クロックを分周する機能があります。1～256 分周まで設定できます。

周辺クロック分周器 #0 の詳細を **Figure 4** に示します。

この MCU は各周辺機能 (例えば、シリアル通信ブロック (SCB), タイマ, カウンタ, PWM (TCPWM) など) およびそれぞれのチャネルに対してクロックが必要です。それらのクロックはそれぞれの分周器によって制御されます。

この周辺クロック分周器#0 は周辺クロック (PCLK) を生成するための多くの周辺クロック分周器を持っています。分周器の数については **Datasheet** を参照してください。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。すでに使用されている分周器は他の周辺機器またはチャネルに使用できないことに注意してください。

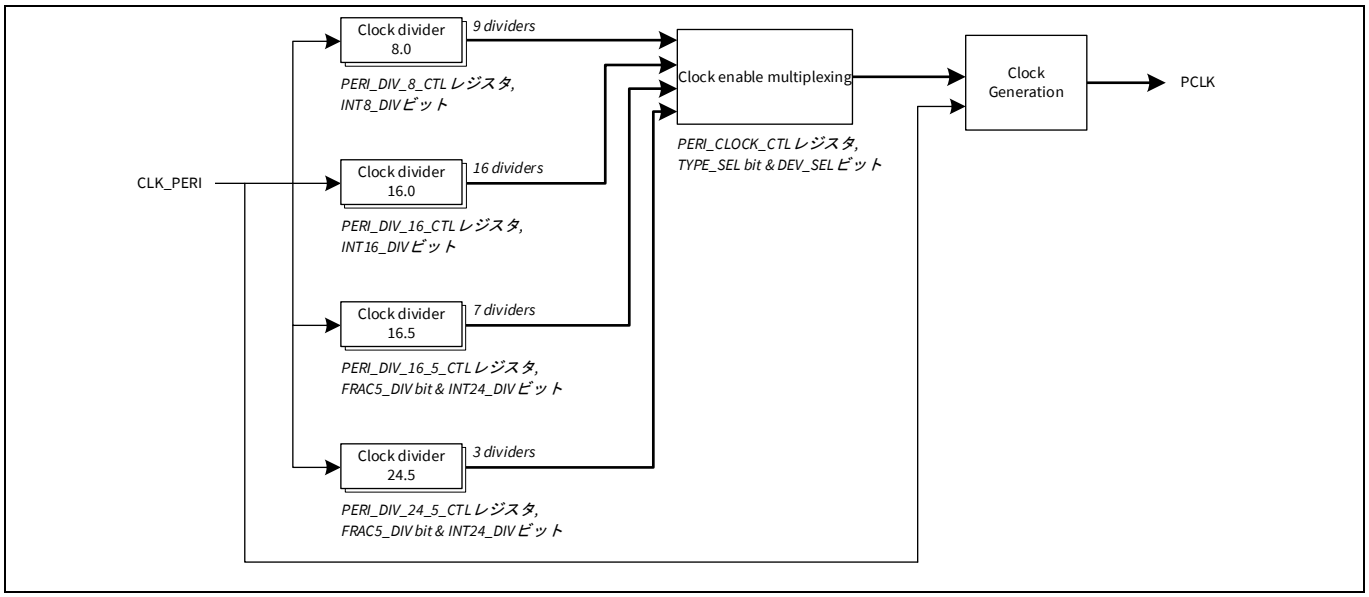

Figure 4　　周辺クロック分周器#0 ブロックダイヤグラム

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 divider16.5　　16.5 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　　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　　クロック分周器を基にして CLK_PERI を分周します

CLK_HF1 の分配先を **Figure 5** に示します。

CLK_HF1 は CLK_FAST_0 と CLK_FAT_1 のクロックソースです。CLK_HF1 のクロック分配を **Figure 5** に示します。CLK_FAST_0 および CLK_FAST_1 は、それぞれ CM7_0 および CM7_1 の入力ソースです。

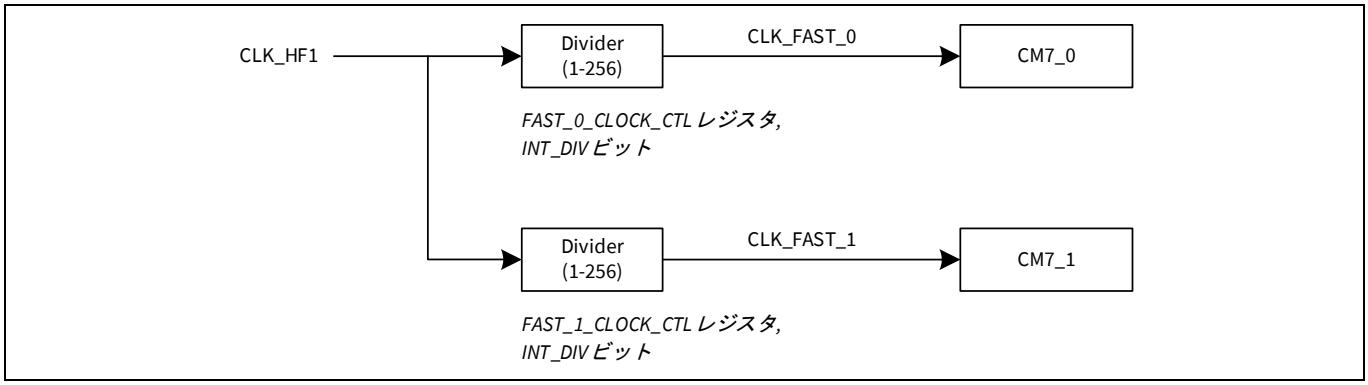

Figure 5　　CLK_HF1 のブロックダイヤグラム

CLK_HF2 の分配先を **Figure 6** に示します。CLK_HF2 は CLK_GR と PCLK のクロックリソースです。

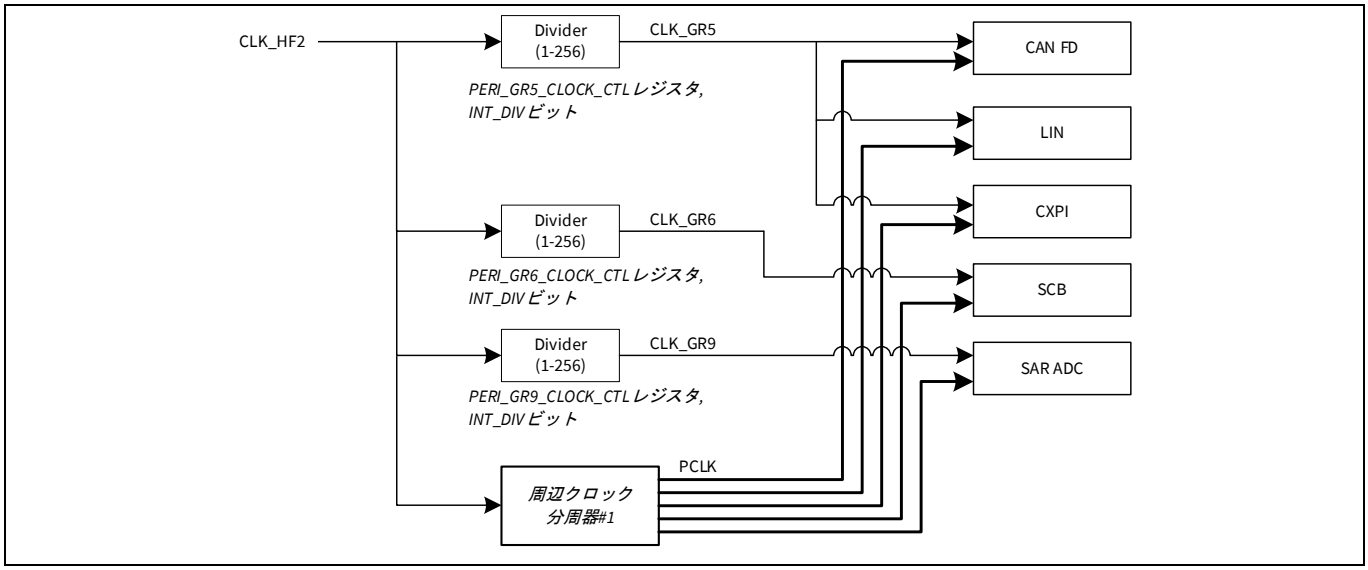

Figure 6　　CLK_HF2 のブロックダイヤグラム

周辺クロック分周器 #1 の詳細を **Figure 7** に示します。

この周辺クロック分周器#1 は PCLK を生成するための多くの周辺クロック分周器を持っています。分周器の数については **Datasheet** を参照してください。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。すでに使用されている分周器は他の周辺機器またはチャネルに使用できないことに注意してください。

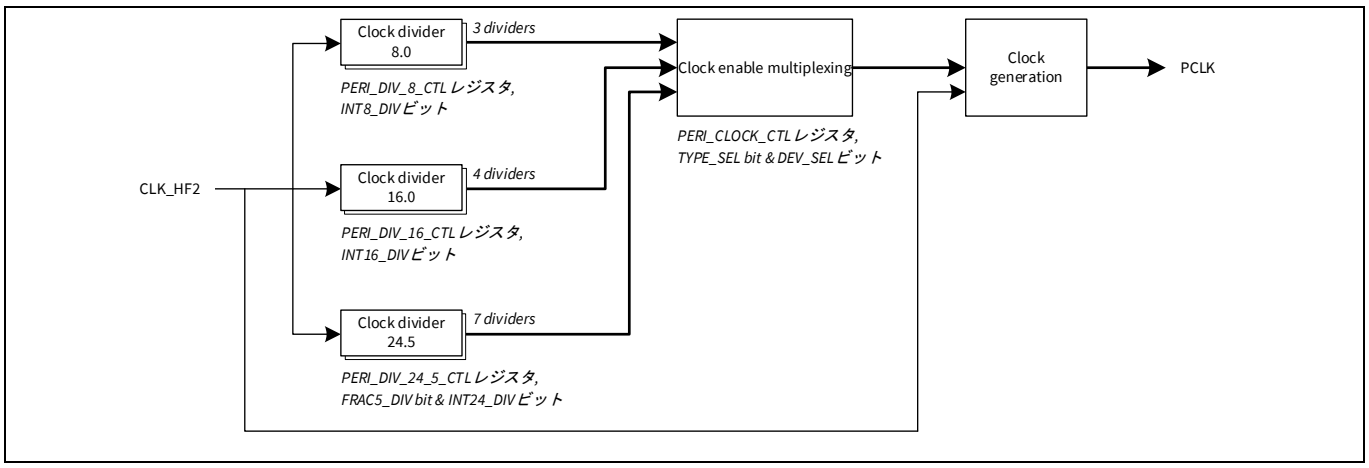

Figure 7　　周辺クロック分周器#1 のブロックダイヤグラム

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	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	クロック分周器を元にして CLK_PERI を分周します

CLK_HF3, CLK_HF4, CLK_HF5 , CLK_HF6, CLK_HF7, CLK_HF8, CLK_HF9, CLK_HF10, CLK_HF11 および CLK_HF12 の分配先を **Figure 8** に示します。**Figure 8** に記載の詳細については **architecture TRM** を参照してください。

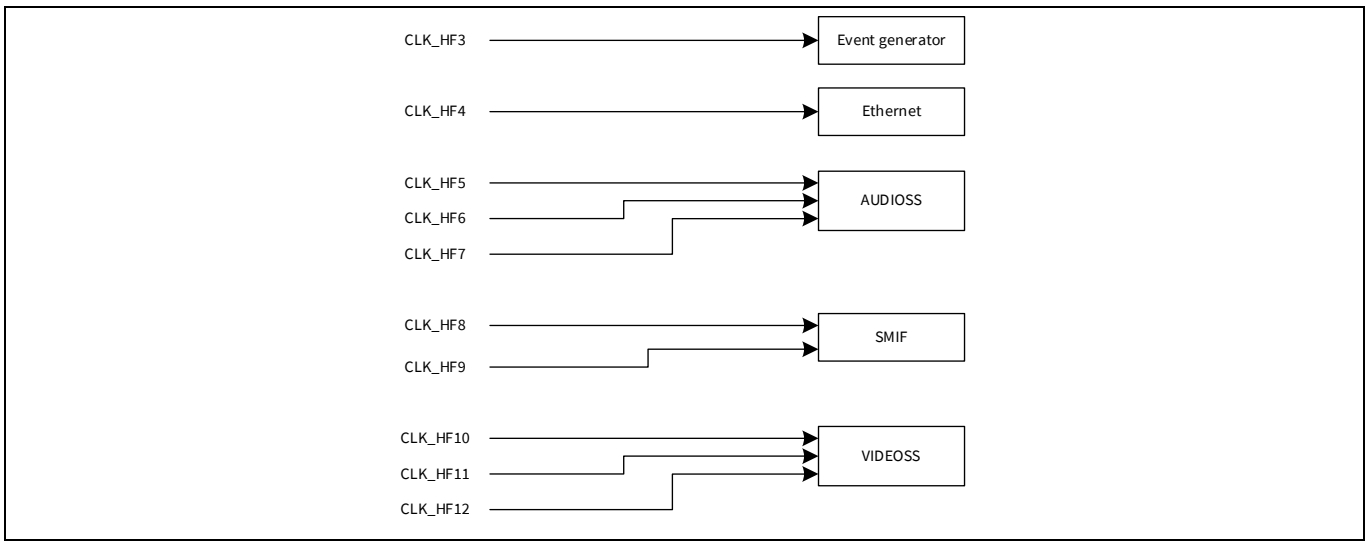

Figure 8　　CLK_HFx (x=3～12)のブロックダイヤグラム

CLK_HF13 は CSV 専用です。 CSV の説明については **architecture TRM** を参照してください。

2.4 基本的なクロックシステム設定

ここでは、インフィニオンの提供する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。目的のシステムに応じて、設定部の設定ができます

3 クロックリソースの設定

ここではクロックの機能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、TRAVEO™ T2G user guide の Setting ECO parameters を参照してください。

Figure 9 に ECO 設定手順を示します。

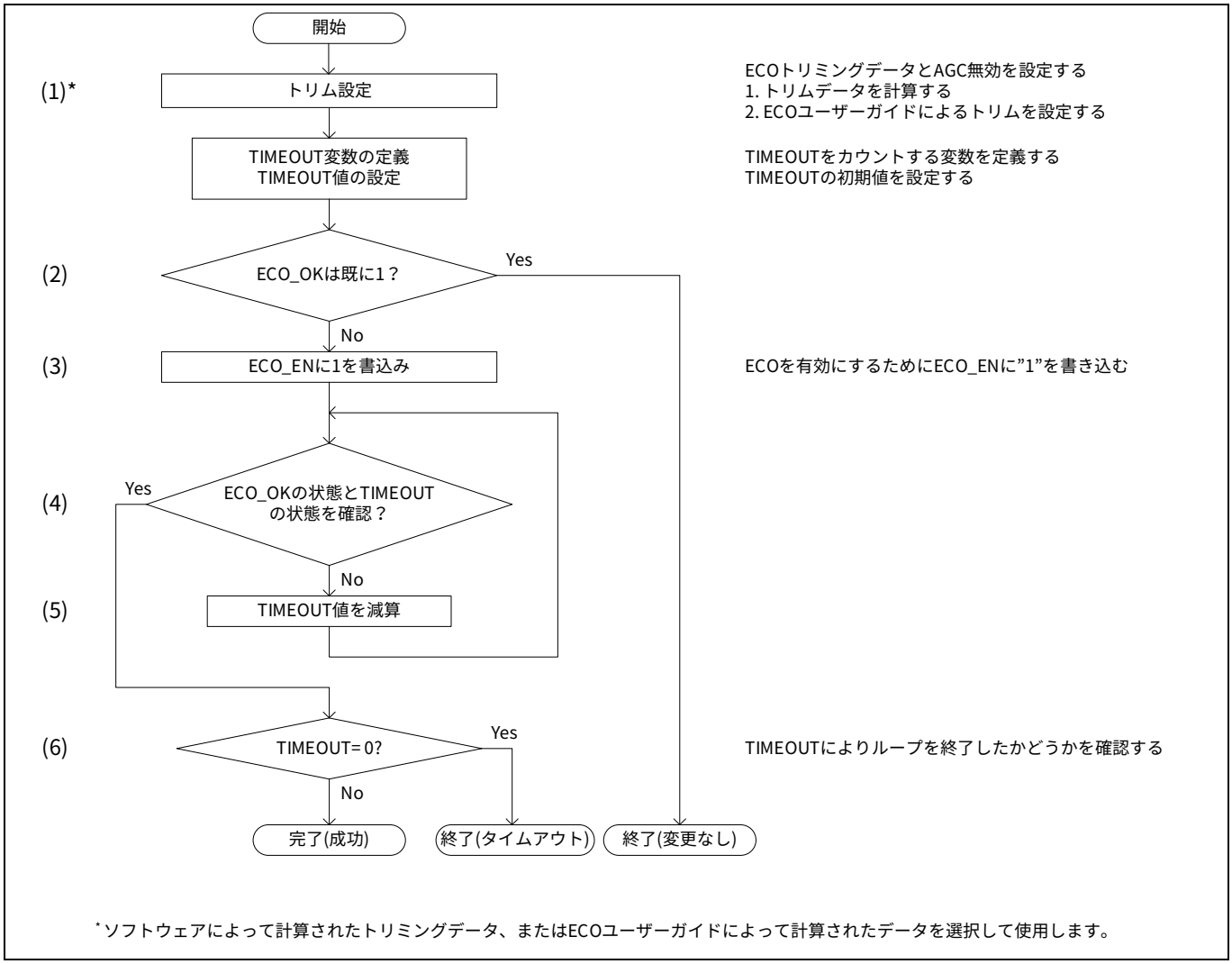

Figure 9 ECO の有効化

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 µW

クロックリソースの設定

- 等価直列抵抗: 150.0 ohm
- 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 の設定部のパラメータを **Table 1** に、関数を **Table 2** に示します。

Table 1 ECO トリム設定パラメーター覧

パラメータ	説明	値
CLK_ECO_CONFIG2.WDTRIM	ウォッチドッグトリム TRAVEO™ T2G user guide の Setting ECO parameters から計算	7ul
CLK_ECO_CONFIG2.ATRIM	振幅トリム TRAVEO™ T2G user guide 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 user guide 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 user guide 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 user guide 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 user guide の Setting ECO parameters から計算	0ul [OFF]
WAIT_FOR_STABILIZATION	発振安定待ち	10000ul
PLL_400M_0_PATH_NO	PLL_400M_0 用の PLL 番号	1ul
PLL_400M_1_PATH_NO	PLL_400M_1 用の PLL 番号	2ul
PLL_200M_0_PATH_NO	PLL_200M_0 用の PLL 番号	3ul
PLL_200M_1_PATH_NO	PLL_200M_1 用の PLL 番号	4ul
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

クロックリソースの設定

Table 2　　ECO トリム設定関数一覧

関数	説明	値
`Cy_WDT_Disable()`	ウォッチドッグタイマ無効	-
`Cy_SysClk_FllDisable Sequence(Wait Cycle)`	FLL 無効	Wait cycle = WAIT_FOR_STABILIZATION
`Cy_SysClk_Pll400M Disable(PLL Number)`	PLL400M_0 無効	PLL number = PLL_400M_0_PATH_NO
	PLL400M_1 無効	PLL number = PLL_400M_1_PATH_NO
`Cy_SysClk_PllDisable (PLL Number)`	PLL200M_0 無効	PLL number = PLL_200M_0_PATH_NO
	PLL200M_1 無効	PLL number = PLL_200M_1_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　　ECO 初期設定のサンプルコード

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)
 :
```

TIMEOUT 変数の宣言。

クロックリソースの設定

Code Listing 1　　ECO の基本設定

```
#define CLK_FREQ_ECO       (16000000ul)
 :
#define PLL_400M_0_PATH_NO   (1ul)
#define PLL_400M_1_PATH_NO   (2ul)
#define PLL_200M_0_PATH_NO   (3ul)
#define PLL_200M_1_PATH_NO   (4ul)
:
#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_ASSERT(Cy_SysClk_FllDisableSequence(WAIT_FOR_STABILIZATION) == CY_SYSCLK_SUCCESS);


    /* Disable Pll */
    CY_ASSERT(Cy_SysClk_Pll400MDisable(PLL_400M_0_PATH_NO) == CY_SYSCLK_SUCCESS);
    CY_ASSERT(Cy_SysClk_Pll400MDisable(PLL_400M_1_PATH_NO) == CY_SYSCLK_SUCCESS);
    CY_ASSERT(Cy_SysClk_PllDisable(PLL_200M_0_PATH_NO) == CY_SYSCLK_SUCCESS);
    CY_ASSERT(Cy_SysClk_PllDisable(PLL_200M_1_PATH_NO) == CY_SYSCLK_SUCCESS);

    /* Enable interrupt */
    __enable_irq();

    /* Set Clock Configuring registers */

    AllClockConfiguration();
 :
    /* Please ensure output clock frequency using oscilloscope */

    for(;;);
}
```

ソフトウェア計算に使用する発振器パラメータの宣言。

PLL 番号の宣言。

ウォッチドッグタイマを無効

FLL 無効。

PLL 無効。

トリムと ECO の設定。**Code Listing 2** 参照。

クロックリソースの設定

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 ユーザー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--)
```

(2) ECO_OK が既に有効か確認する。

(3) ECO_EN ビットに”1”を書き込み、ECO を利用可能にする。

(4) ECO_OK と TIMEOUT の状態を確認する。

(5) TIMEOUT 値を減算する。

クロックリソースの設定

Code Listing 3　Cy_SysClk_EcoEnable() 関数

```
    {
        Cy_SysLib_DelayUs(1u);
    }

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

1 us 待機。

(6) TIMEOUT によりループが終了したかどうか確認する。

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);
    }
```

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。

クロックリソースの設定

Code Listing 4　Cy_SysClk_EcoConfigureWithMinRneg() 関数

```
    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);
}
```

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。

Code Listing 5　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);
    }
    else if(maxAmplitude < 0.65f)
    {
        return(0x06ul);
    }
    else if(maxAmplitude < 0.70f)
```

Atrim 値を取得。

Code Listing 5　　Cy_SysClk_SelectEcoAtrim() 関数

```
    {
        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);
    }
}
```

クロックリソースの設定

Code Listing 6　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。

Code Listing 7　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);
```

Wdtrim 値を取得。

クロックリソースの設定

Code Listing 7　Cy_SysClk_SelectEcoWDtrim() 関数

```
    }
    else
    {
        // invalid input
        return(CY_SYSCLK_INVALID_TRIM_VALUE);
    }
}
```

Code Listing 8　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
    {
```

Gtrim 値を取得。

Code Listing 8　Cy_SysClk_SelectEcoGtrim() 関数

```
        // invalid input
        return(CY_SYSCLK_INVALID_TRIM_VALUE);
    }
}
```

Code Listing 9　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　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

初期設定では WCO は無効になっており、使用するためには有効にする必要があります。WCO を有効にするためのレジスタの設定方法を **Figure 10** に示します。

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

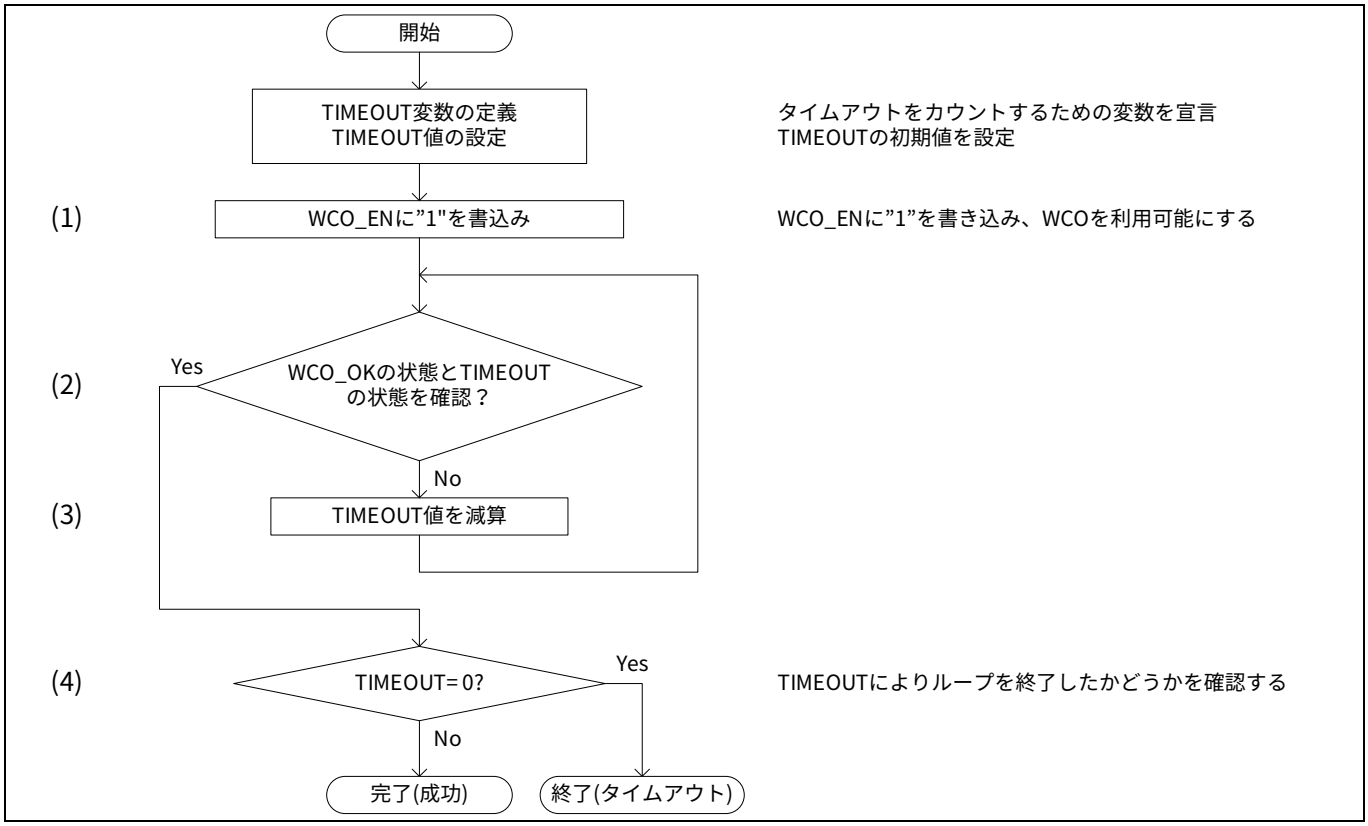

Figure 10 WCO の有効化

3.2.2 コンフィグレーション

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

Table 3 WCO 設定パラメーター覧

パラメータ	説明	値
WAIT_FOR_STABILIZATION	発振安定待ち	10000ul
PLL_400M_0_PATH_NO	PLL_400M_0 の PLL 番号	1ul
PLL_400M_1_PATH_NO	PLL_400M_1 の PLL 番号	2ul
PLL_200M_0_PATH_NO	PLL_200M_0 の PLL 番号	3ul
PLL_200M_1_PATH_NO	PLL_200M_1 の PLL 番号	4ul

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_FllDisable Sequence(Wait Cycle)`	FLL 無効	Wait cycle = WAIT_FOR_STABILIZATION
`Cy_SysClk_Pll400M Disable(PLL Number)`	PLL400M_0 無効	PLL number = PLL_400M_0_PATH_NO
	PLL400M_1 無効	PLL number = PLL_400M_1_PATH_NO
`Cy_SysClk_PllDisable (PLL Number)`	PLL200M_0 無効	PLL number = PLL_200M_0_PATH_NO
	PLL200M_1 無効	PLL number = PLL_200M_1_PATH_NO
`AllClockConfigurationw()`	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　　WCO 設定の初期設定のサンプルコード

サンプル設定を **Code Listing 11**～**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_400M_0_PATH_NO   (1ul)
#define PLL_400M_1_PATH_NO   (2ul)
#define PLL_200M_0_PATH_NO   (3ul)
#define PLL_200M_1_PATH_NO   (4ul)
:
static void AllClockConfiguration(void);
:
int main(void)
{
    /* disable watchdog timer */
    Cy_WDT_Disable();
:
    /* Disable Fll */
    CY_ASSERT(Cy_SysClk_FllDisableSequence(WAIT_FOR_STABILIZATION) == CY_SYSCLK_SUCCESS);

    /* Disable Pll */
    CY_ASSERT(Cy_SysClk_Pll400MDisable(PLL_400M_0_PATH_NO) == CY_SYSCLK_SUCCESS);
    CY_ASSERT(Cy_SysClk_Pll400MDisable(PLL_400M_1_PATH_NO) == CY_SYSCLK_SUCCESS);
    CY_ASSERT(Cy_SysClk_PllDisable(PLL_200M_0_PATH_NO) == CY_SYSCLK_SUCCESS);
    CY_ASSERT(Cy_SysClk_PllDisable(PLL_200M_1_PATH_NO) == CY_SYSCLK_SUCCESS);
```

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

クロックリソースの設定

Code Listing 11　WCO の基本設定

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

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

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 を有効にする。
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”を書き込み、WCO を利用可能にする。

(2) WCO_OK と TIMEOUT の状態を確認する。

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する。

3.3 IMO の設定

初期設定により、IMO はすべての機能が正しく動作するように有効になっています。IMO は DeepSleep, Hibernate, および XRES のモードの間、自動的に無効になります。したがって、IMO を明示的に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。

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ることに注意してください。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 を無効するためには、WDT_CTL レジスタの WDT_LOCK ビットに'0b01'を書き込み、それから CLK_ILO0_CONFIG レジスタの ENABLE ビットに'0b00'を書き込んでください。

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。

3.5 LPECO の設定

LPCEO はデフォルトで無効になっています。LPECO は、有効にしないと使用できません。**Figure 11** に、LPECO を有効にするためのレジスタを設定する方法を示します。LPECO を無効にするためには、BACKUP_LPECO_CTL レジスタの LPECO_EN ビットに'0'を書き込んでください。

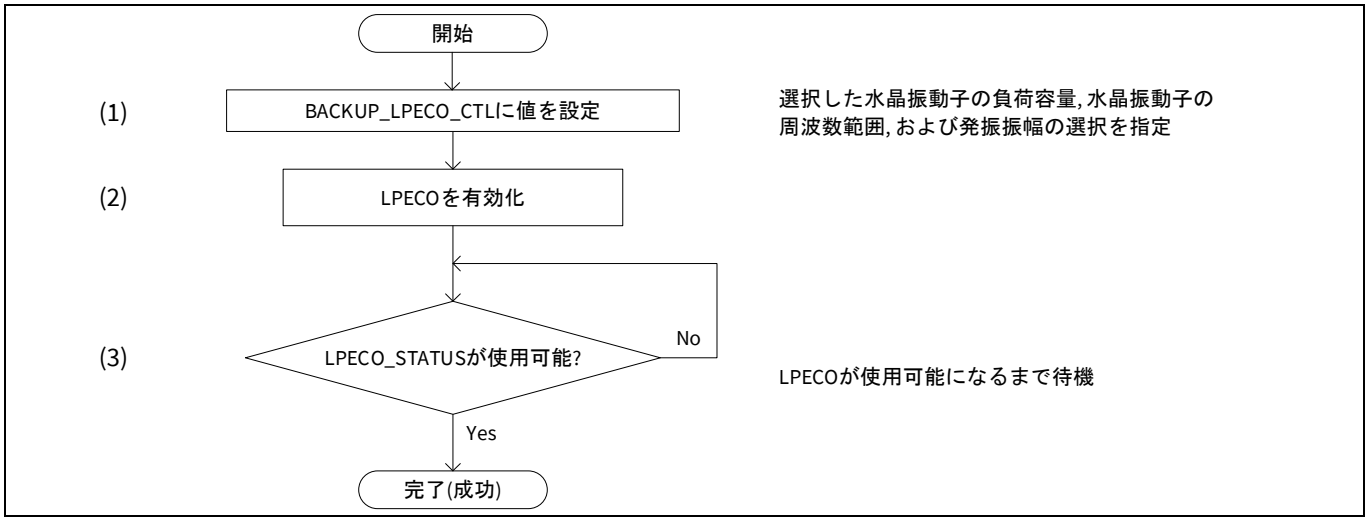

Figure 11 LPECO 設定

3.5.1 ユースケース

- 使用する発振器: 水晶振動子
- 基本周波数: 8 MHz

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。

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

クロックリソースの設定

Table 5　　LPECO 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
CLK_FREQ_LPECO	ソースクロック周波数	8000000ul
CY_SYSCLK_BAK_LPECO_LCAP_5TO10PF	バックアップドメインの LPECO 負荷は、5pF～10pF の範囲です。	0ul
CY_SYSCLK_BAK_LPECO_FREQ_6TO8MHZ	バックアップドメインの LPECO 周波数は、6 MHz～8MHz の範囲です。	1ul
CY_SYSCLK_BAK_LPECO_AMP_MAX_1P35V	バックアップドメインの LPECO の最大発振振幅は 1.35V です。	0ul

Table 6　　LPECO 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_ClkBak_LPECO_SetLoadCap(range)`	LPECO 水晶振動子の負荷容量範囲の設定	CY_SYSCLK_BAK_LPECO_LCAP_5TO10PF
`Cy_SysClk_ClkBak_LPECO_SetFrequency(range)`	LPECO 水晶振動子の周波数範囲の設定	CY_SYSCLK_BAK_LPECO_FREQ_6TO8MHZ
`Cy_SysClk_ClkBak_LPECO_SetAmplitude(value)`	LPECO 水晶振動子の最大発振振幅値の設定	CY_SYSCLK_BAK_LPECO_AMP_MAX_1P35V
`Cy_SysClk_ClkBak_LPECO_Enable()`	LPECO の有効化	–
`Cy_SysClk_ClkBak_LPECO_Ready()`	LPECO 安定化の状態に復帰	–

3.5.2　　LPECO 設定の初期設定のサンプルコード

サンプルコードを **Code Listing 14**～**Code Listing 20** に示します。

Code Listing 14　　LPECO の基本設定

```
:
/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul)
 :
#define CLK_FREQ_LPECO       (8000000ul)
 :
static void AllClockConfiguration(void);
 :
int main(void)
{
    /* disable watchdog timer */
    Cy_WDT_Disable();
:
    /* Enable interrupt */
    __enable_irq();


    /* Set Clock Configuring registers */
```

TIMEOUT 変数の宣言。

ソフトウェア計算を使用した発振器パラメータの宣言。

Code Listing 14　LPECO の基本設定

```
    AllClockConfiguration();

    /* Measure clock frequencies using ECO and check */
    MeasureClockFrequency();
    CompareExpectedAndMeasured();

    /* Read register value and re-calculate the frequency and check */
    RecalucClockFrequencyValues();
    CompareExpectedAndCaluclated();

    /* Start output internal clock */
    Cy_GPIO_Pin_Init(CY_HF3_CLK_OUT_PORT, CY_HF3_CLK_OUT_PIN, &clkOutPortConfig);

    /* Please ensure output clock frequency using oscilloscope */
    for(;;);
}
```

Code Listing 15　AllClockConfiguration () 関数

```
static void AllClockConfiguration(void)
{
 :
#ifdef LPECO_ENABLE
    /***** LPECO setting ******/
    {
        Cy_SysClk_ClkBak_LPECO_SetLoadCap(CY_SYSCLK_BAK_LPECO_LCAP_5TO10PF);
        Cy_SysClk_ClkBak_LPECO_SetFrequency(CY_SYSCLK_BAK_LPECO_FREQ_6TO8MHZ);
        Cy_SysClk_ClkBak_LPECO_SetAmplitude(CY_SYSCLK_BAK_LPECO_AMP_MAX_1P35V);

        Cy_SysClk_ClkBak_LPECO_Enable();
        while(Cy_SysClk_ClkBak_LPECO_Ready() == false);
    }
 :
#endif
 :
   return;
}
```

(1) BACKUP_LPECO_CTL に値を設定。**Code Listing 16**, **Code Listing 17** および **Code Listing 18** を参照。

(2) LPECO が有効。**Code Listing 19** 参照。

(3) LPECO が有効になるまで待機。**Code Listing 20** 参照。

Code Listing 16　Cy_SysClk_ClkBak_LPECO_SetLoadCap () 関数

```
__STATIC_INLINE void Cy_SysClk_ClkBak_LPECO_SetLoadCap(cy_en_clkbak_lpeco_loadcap_range_t capValue)
{
    BACKUP->unLPECO_CTL.stcField.u2LPECO_CRANGE = capValue;
}
```

Code Listing 17　Cy_SysClk_ClkBak_LPECO_SetFrequency () 関数

```
__STATIC_INLINE void Cy_SysClk_ClkBak_LPECO_SetFrequency(cy_en_clkbak_lpeco_frequency_range_t freqValue)
{
    BACKUP->unLPECO_CTL.stcField.u1LPECO_FRANGE = freqValue;
}
```

Code Listing 18　Cy_SysClk_ClkBak_LPECO_SetAmplitude () 関数

```
__STATIC_INLINE void Cy_SysClk_ClkBak_LPECO_SetAmplitude(cy_en_clkbak_lpeco_max_amplitude_t ampValue)
{
    BACKUP->unLPECO_CTL.stcField.u1LPECO_AMP_SEL = ampValue;
}
```

Code Listing 19　Cy_SysClk_ClkBak_LPECO_Enable () 関数

```
__STATIC_INLINE void Cy_SysClk_ClkBak_LPECO_Enable(bool enable)
{
    BACKUP->unLPECO_CTL.stcField.u1LPECO_EN = enable;
}
```

Code Listing 20　Cy_SysClk_ClkBak_LPECO_Ready () 関数

```
__STATIC_INLINE bool Cy_SysClk_ClkBak_LPECO_Ready(void)
{
    return (BACKUP->unLPECO_STATUS.stcField.u1LPECO_READY);
}
 :
```

4 FLL と PLL の設定

ここではクロックシステムの FLL と PLL の設定について説明します。

4.1 FLL の設定

4.1.1 操作概要

FLL は使用する前に設定する必要があります。FLL は電流制御発振器 (CCO) を搭載しており、CCO の出力周波数を CCO の調整によって制御しています。FLL の設定手順を **Figure 12** に示します。

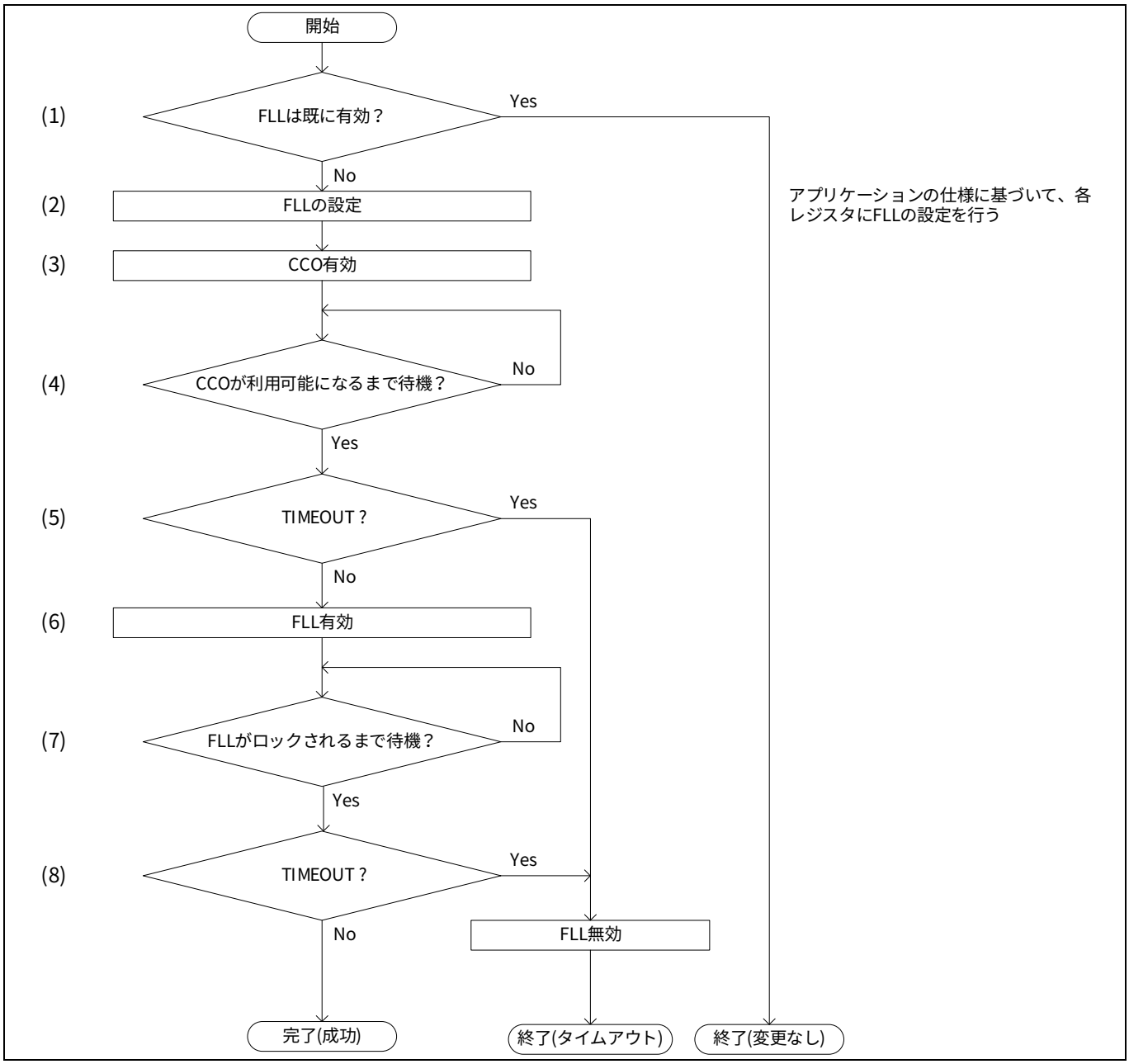

Figure 12 FLL 設定の手順

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
- 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 の設定部のパラメータを **Table 7** に、関数を **Table 8** に示します。

Table 7 FLL 設定パラメーター覧

パラメータ	説明	値
WAIT_FOR_STABILIZATION	発振安定待ち	10000ul
FLL_PATH_NO	FLL 番号	0u
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 8 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 FLL 設定の初期設定のサンプルコード

サンプルコードを **Code Listing 21**～**Code Listing 25** に示します。

Code Listing 21 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
:

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 22** 参照。

Code Listing 22 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 23** 参照。

FLL を有効にする。**Code Listing 25** 参照。

Code Listing 23　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)
    {
```

(1) FLL が既に有効か確認する。

FLL の出力範囲を確認する。

FLL の周波数比率を確認する

FLL パラメータの計算

Code Listing 23　Cy_SysClk_FllConfigureStandard() 関数

```
    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)
    {
        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);
```

FLL と PLL の設定

Code Listing 23　Cy_SysClk_FllConfigureStandard() 関数

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

    return (returnStatus);
}
```

FLL のレジスタを設定する。**Code Listing 24** 参照。

Code Listing 24　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 */
```

(1) FLL が既に有効かどうか確認する。

(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。

FLL と PLL の設定

Code Listing 24　Cy_SysClk_FllManualConfigure() 関数

```
        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);
}
```

CLK_FLL_CONFIG4 レジスタの設定。

Code Listing 25　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--;
    }
```

(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。

Code Listing 25　Cy_SysClk_FllEnable() 関数

```
    /* 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);
}
```

4.2　PLL の設定

PLL は使用する前に PLL を設定する必要があります。PLL400 と PLL200 を設定する手順を **Figure 13** に示します。PLL400 と PLL200 の詳細については **architecture TRM** を参照してください。

FLL と PLL の設定

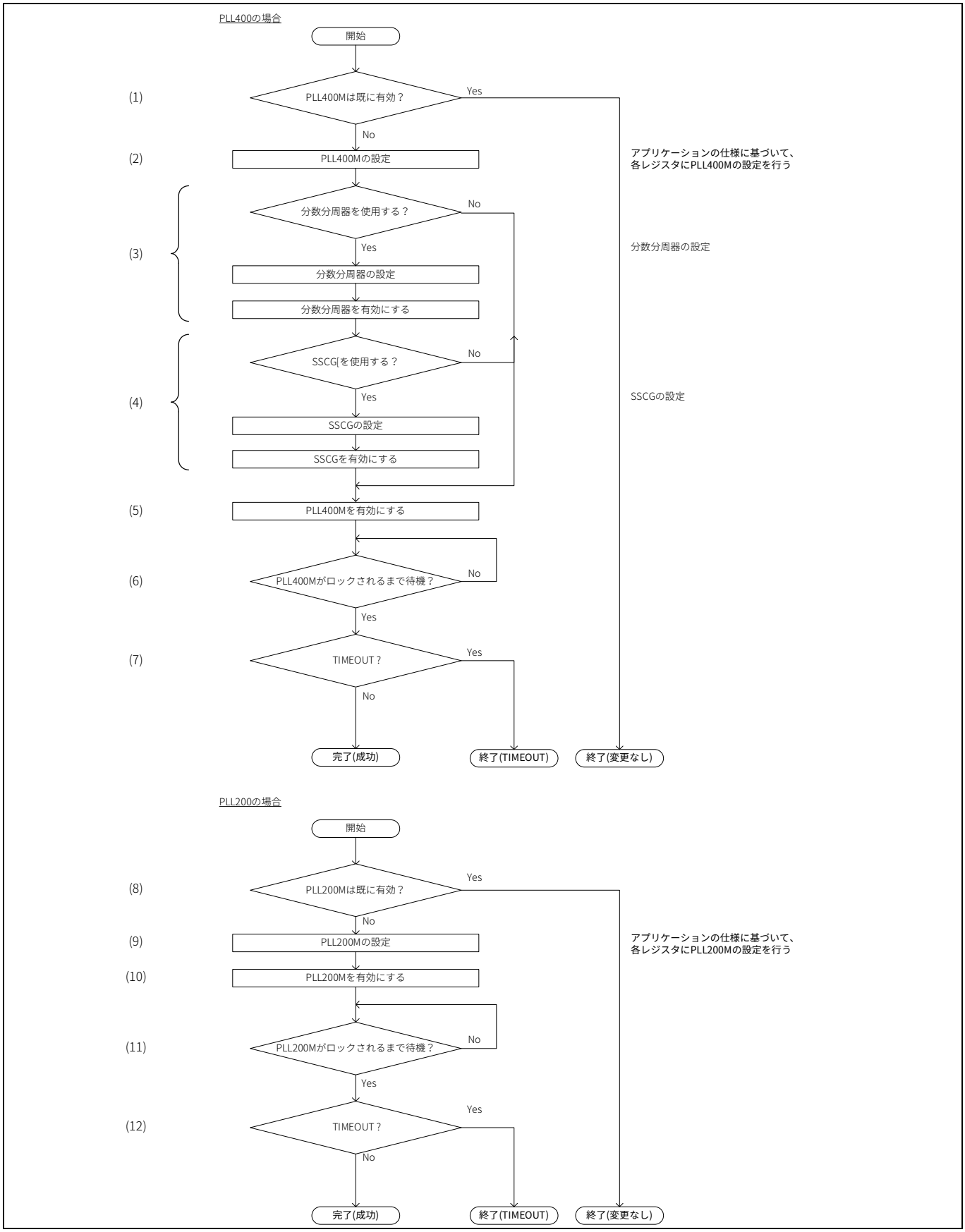

Figure 13　　PLL 設定の手順

4.2.1 ユースケース

- 入力クロック周波数: 16.000 MHz
- 出力クロック周波数:
  250.000 MHz (PLL400 #0)
  196.608 MHz (PLL400 #1)
  160.000 MHz (PLL200 #0)
  80.000 MHz (PLL200 #1)
- 分数分周器:
  無効 (PLL400 #0)
  有効 (PLL400 #1)
- SSCG:
  有効 (PLL400 #0)
  無効 (PLL400 #1)
- SSCG ディザリング:
  有効 (PLL400 #0)
  無効 (PLL400 #1)
- SSCG 変調度: -2.0% (PLL400)
- SSCG 変調速度: 512 分周 (PLL400)
- LF モード: 200 MHz ～ 400 MHz (PLL200)

4.2.2 コンフィグレーション

Table 9 と **Table 11** に、PLL (400/200)の各パラメータを、**Table 10** と **Table 12** に、PLL (400/200)の設定における SDL 設定部の PLL (400/200)の関数を示します。

Table 9 PLL 400 設定パラメーター覧

パラメータ	説明	値
PLL400_0_TARGET_FREQ	PLL400 #0 ターゲット周波数	250 MHz (250000000ul)
PLL400_1_TARGET_FREQ	PLL400 #1 ターゲット周波数	196.608 MHz (196608000ul)
WAIT_FOR_STABILIZATION	発振安定待ち	10000ul
PLL400_0_PATH_NO	PLL400 #0 番号	1u
PLL400_1_PATH_NO	PLL400 #1 番号	2u
CLK_FREQ_ECO	ECO クロック周波数	16000000ul (16 MHz)
PATH_SOURCE_CLOCK_FREQ	PATH ソースクロック周波数	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:	0ul

FLL と PLL の設定

パラメータ	説明	値
	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。	
pllConfig.inputFreq	入力 PLL 周波数	PATH_SOURCE_CLOCK_FREQ
pllConfig.outputFreq	出力 PLL 周波数 (PLL400 #0)	PLL400_0_TARGET_FREQ
	出力 PLL 周波数 (PLL400 #1)	PLL400_1_TARGET_FREQ
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
pllConfig.fracEn	分数分周器有効 (PLL400 #0)	false
	分数分周器有効 (PLL400 #1)	true
pllConfig.fracDitherEn	ディザリング操作有効 (PLL400 #0)	false
	ディザリング操作有効 (PLL400 #1)	true
pllConfig.sscgEn	SSCG 有効 (PLL400 #0)	true
	SSCG 有効 (PLL400 #1)	false
pllConfig.sscgDitherEn	SSCG ディザリング操作有効 (PLL400 #0)	true
	SSCG ディザリング操作有効 (PLL400 #1)	false
pllConfig.sscgDepth	SSCG 変調度設定	CY_SYSCLK_SSCG_DEPTH_MINUS_2_0
pllConfig.sscgRate	SSCG 変調速度設定	CY_SYSCLK_SSCG_RATE_DIV_512
manualConfig.feedbackDiv	フィードバック分周器用制御ビット	p (計算値)
manualConfig.referenceDiv	基準分周器用制御ビット	q (計算値)
manualConfig.outputDiv	出力分周器用制御ビット 0: 不正 (未定義の動作) 1: 不正 (未定義の動作) 2: 2 分周。HFCLK ソースとして直接使用するのに適合。 ... 16: 16 分周。HFCLK ソースとして直接使用するのに適合。 >16: 不正 (未定義の動作)	out (計算値)
manualConfig.lfMode	VCO 周波数レンジ選択 0: VCO 周波数 [200 MHz, 400 MHz] 1: VCO 周波数 [170 MHz, 200 MHz]	config->lfMode (計算値)
manualConfig.outputMode	PLL 出力直後に配置されたマルチプレクサをバイパスする。	config->outputMode (計算値)

FLL と PLL の設定

パラメータ	説明	値
	0: AUTO 1: LOCKED_OR_NOTHING 2: PLL_REF 3: PLL_OUT	

Table 10　　PLL 400 設定関数一覧

関数	説明	値
`AllClockConfiguration()`	クロック設定	–
`Cy_SysClk_Pll400MConfigure(PLL Number,PLL Configure)`	PLL path 番号と PLL の設定 (PLL400 #0)。	PLL number = PLL400_0_PATH_NO, PLL configure = g_pll400_0_Config
	PLL path 番号と PLL の設定 (PLL400 #1)。	PLL number = PLL400_1_PATH_NO, PLL configure = g_pll400_1_Config
`Cy_SysClk_Pll400MEnable(PLL Number, Timeout value)`	PLL path 番号と PLL モニタの設定 (PLL400 #0)。	PLL number = PLL400_0_PATH_NO, Timeout value = WAIT_FOR_STABILIZATION
	PLL path 番号と PLL モニタの設定 (PLL400 #1)。	PLL number = PLL400_1_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 Manual Configure)`	PLL path 番号と PLL の手動設定 (PLL400 #0)。	PLL number = PLL400_0_PATH_NO, PLL manual configure = manualConfig
	PLL path 番号と PLL の手動設定 (PLL400 #1)。	PLL number = PLL400_1_PATH_NO, PLL manual configure = manualConfig
`Cy_SysClk_GetPll400MNo(Clkpath, PllNo)`	入力 PATH 番号に従い PLL 番号をリターン (PLL400 #0)。	Clkpath = 1u
		PllNo = 0u
	入力 PATH 番号に従い PLL 番号をリターン (PLL400 #1)。	Clkpath = 2u
		PllNo = 1u
`Cy_SysClk_PllCalucDividers()`	PLL 入力/出力周波数に従い適切な分周器設定を計算。	
	PLL 番号と PLL モニタの設定 (PLL400 #0)。	PLL number = PLL400_0_PATH_NO,

FLL と PLL の設定

関数	説明	値
`Cy_SysClk_Pll400MEnable(PLL Number,Timeout value)`		Timeout value = WAIT_FOR_STABILIZATION
	PLL 番号と PLL モニタの設定 (PLL400 #1)。	PLL number = PLL400_1_PATH_NO,
		Timeout value = WAIT_FOR_STABILIZATION

Table 11　　PLL 200 設定パラメーター一覧

パラメータ	説明	値
PLL200_0_TARGET_FREQ	PLL200 #0 ターゲット周波数	160 MHz (160000000ul)
PLL200_1_TARGET_FREQ	PLL200 #1 ターゲット周波数	80 MHz (80000000ul)
WAIT_FOR_STABILIZATION	発振安定待ち	10000ul
PLL200_0_PATH_NO	PLL200 #0 番号	3u
PLL200_1_PATH_NO	PLL200 #1 番号	4u
PATH_SOURCE_CLOCK_FREQ	PATH ソースクロック周波数	16000000ul (16 MHz)
pllConfig.inputFreq	PLL 入力周波数	PATH_SOURCE_CLOCK_FREQ
pllConfig.outputFreq	PLL 出力周波数 (PLL200 #0)	PLL200_0_TARGET_FREQ
	PLL 出力周波数 (PLL200 #1)	PLL200_1_TARGET_FREQ
pllConfig.lfMode	PLL LF モード 0: VCO 周波数 [200 MHz, 400 MHz] 1: VCO 周波数 [170 MHz, 200 MHz]	0u (VCO 周波数: 320 MHz)
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
manualConfig.feedbackDiv	フィードバック分周器用制御ビット	p (計算値)
manualConfig.referenceDiv	基準分周器用制御ビット	q (計算値)
manualConfig.outputDiv	出力分周器用制御ビット 0: 不正 (未定義の動作) 1: 不正 (未定義の動作) 2: 2 分周。HFCLK ソースとして直接使用するのに適合。 ... 16: 16 分周。HFCLK ソースとして直接使用するのに適合。 >16: 不正 (未定義の動作)	out (計算値)
manualConfig.lfMode	VCO 周波数レンジ選択 0: VCO 周波数 [200 MHz, 400 MHz] 1: VCO 周波数 [170 MHz, 200 MHz]	config->lfMode (計算値)

FLL と PLL の設定

パラメータ	説明	値
manualConfig.outputMode	PLL 出力直後に配置されたマルチプレクサをバイパス 0: AUTO 1: LOCKED_OR_NOTHING 2: PLL_REF 3: PLL_OUT	config->outputMode (計算値)
manualConfig.fracDiv	分数分周器の値	config->fracDiv (計算値)

Table 12　　PLL 200 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 path 番号と PLL の設定 (PLL200 #0)。	PLL number = PLL200_0_PATH_NO,
		PLL configure = g_pll200_0_Config
	PLL path 番号と PLL の設定 (PLL200 #1)。	PLL number = PLL200_1_PATH_NO,
		PLL configure = g_pll200_1_Config
`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_PllManual Configure(PLL Number, PLL Manual Configure)`	PLL path 番号と PLL の手動設定 (PLL200 #0)。	PLL number = PLL200_0_PATH_NO, PLL manual configure = manualConfig
	PLL path 番号と PLL の手動設定 (PLL200 #1)。	PLL number = PLL200_1_PATH_NO, PLL manual configure = manualConfig
`Cy_SysClk_GetPllNo (Clkpath, PllNo)`	入力 PATH 番号に従い PLL 番号をリターン (PLL200 #0)。	Clkpath = 3u PllNo = 0u
	入力 PATH 番号に従い PLL 番号をリターン (PLL200 #1)。	Clkpath = 4u PllNo = 1u

FLL と PLL の設定

関数	説明	値
`Cy_SysClk_PllCalucDividers(InputFreq, OutputFreq,PLLlimit,FracBitNum,RefDiv,OutputDiv,FeedBackFracDiv)`	PLL 入力/出力周波数に従い適切な分周器設定を計算。	InputFreq = PATH_SOURCE_CLOCK_FREQ OutputFreq = PLL400_0_TARGET_FREQ (PLL 400 #0), PLL400_1_TARGET_FREQ (PLL 400 #1), PLL200_0_TARGET_FREQ (PLL 200 #0), PLL200_1_TARGET_FREQ (PLL 200 #1) PLLlimit = g_limPll400MFrac (PLL 400 #1 only), g_limPll400M (Other) FracBitNum = 24ul (PLL 400 #1 only), 0ul (Other) FeedBackDiv = manualConfig.feedbackDiv RefDiv = manualConfig.referenceDiv OutputDiv = manualConfig.outputDiv FeedBackFracDiv = manualConfig.fracDiv
`Cy_SysClk_PllEnable(PLL Number, Timeout value)`	PLL path 番号と PLL モニタの設定 (PLL200 #0)。	PLL number = PLL200_0_PATH_NO, Timeout value = WAIT_FOR_STABILIZATION
	PLL path 番号と PLL モニタの設定 (PLL200 #1)。	PLL number = PLL200_1_PATH_NO, Timeout value = WAIT_FOR_STABILIZATION

4.2.3 PLL 初期設定のサンプルコード

PLL400 #0 についての例のサンプルコードを **Code Listing 26** ～ **Code Listing 32** に、PLL200 #0 についての例のサンプルコードを **Code Listing 33** ～ **Code Listing 39** に示します。

Code Listing 26 PLL 400 #0 の基本設定

```
:
#define PLL400_0_TARGET_FREQ      (250000000ul)
#define PLL400_1_TARGET_FREQ      (196608000ul)
:
/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul)
:
#define PLL_400M_0_PATH_NO   (1ul)
#define PLL_400M_1_PATH_NO   (2ul)
#define PLL_200M_0_PATH_NO   (3ul)
#define PLL_200M_1_PATH_NO   (4ul)
#define BYPASSED_PATH_NO     (5ul)
 :
/*** Parameters for Clock Configuration ***/
cy_stc_pll_400M_config_t g_pll400_0_Config =
{
    .inputFreq    = PATH_SOURCE_CLOCK_FREQ,
    .outputFreq   = PLL400_0_TARGET_FREQ,
    .outputMode   = CY_SYSCLK_FLLPLL_OUTPUT_AUTO,
    .fracEn       = false,
    .fracDitherEn = false,
    .sscgEn       = true,
    .sscgDitherEn = true,
    .sscgDepth    = CY_SYSCLK_SSCG_DEPTH_MINUS_2_0,
    .sscgRate     = CY_SYSCLK_SSCG_RATE_DIV_512,
};
 :
int main(void)
{
:
    /* Enable interrupt */
    __enable_irq();

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

PLL ターゲット周波数。

TIMEOUT 値の宣言。

PLL 番号の宣言。

PLL400 #0 の設定。

PLL400 #0 の設定。**Code Listing 27** 参照。

FLL と PLL の設定

Code Listing 27　AllClockConfiguration() 関数

```
static void AllClockConfiguration(void)
{
 :
    /***** PLL400M#0(PATH1) source setting ******/
    {
:
        status = Cy_SysClk_Pll400MConfigure(PLL_400M_0_PATH_NO, &g_pll400_0_Config);
        CY_ASSERT(status == CY_SYSCLK_SUCCESS);

        status = Cy_SysClk_Pll400MEnable(PLL_400M_0_PATH_NO, WAIT_FOR_STABILIZATION);
        CY_ASSERT(status == CY_SYSCLK_SUCCESS);
:
   }
    return;
}
```

PLL400 の設定。**Code Listing 28** 参照。

PLL400 有効。**Code Listing 32** 参照。

Code Listing 28　Cy_SysClk_Pll400MConfigure() 関数

```
cy_en_sysclk_status_t Cy_SysClk_Pll400MConfigure(uint32_t clkPath, const cy_stc_pll_400M_config_t *config)
{
    /* check for error */
    uint32_t pllNo;
    cy_en_sysclk_status_t status = Cy_SysClk_GetPll400MNo(clkPath, &pllNo);
    if(status != CY_SYSCLK_SUCCESS)
    {
        return(status);
    }

    if (SRSS->CLK_PLL400M[pllNo].unCONFIG.stcField.u1ENABLE != 0ul) /* 1 = enabled */
    {
        return (CY_SYSCLK_INVALID_STATE);
    }

    cy_stc_pll_400M_manual_config_t manualConfig = {0ul};
    const cy_stc_pll_limitation_t* pllLim;
    uint32_t fracBitNum;
    if(config->fracEn == true)
    {
        pllLim = &g_limPll400MFrac;
        fracBitNum = 24ul;
    }
    else
    {
        pllLim = &g_limPll400M;
        fracBitNum = 0ul;
    }
    status = Cy_SysClk_PllCalucDividers(config->inputFreq,
```

PLL400 番号とクロックパスが有効かどうか確認する。**Code Listing 30** 参照。

(1) PLL400 が既に有効かどうか確認する。

Code Listing 28　Cy_SysClk_Pll400MConfigure() 関数

```
                                        config->outputFreq,
                                        pllLim,
                                        fracBitNum,
                                        &manualConfig.feedbackDiv,
                                        &manualConfig.referenceDiv,
                                        &manualConfig.outputDiv,
                                        &manualConfig.fracDiv
                                        );
    if(status != CY_SYSCLK_SUCCESS)
    {
        return(status);
    }

    manualConfig.outputMode   = config->outputMode;
    manualConfig.fracEn       = config->fracEn;
    manualConfig.fracDitherEn = config->fracDitherEn;
    manualConfig.sscgEn       = config->sscgEn;
    manualConfig.sscgDitherEn = config->sscgDitherEn;



    manualConfig.sscgDepth    = config->sscgDepth;
    manualConfig.sscgRate     = config->sscgRate;

    status = Cy_SysClk_Pll400MManualConfigure(clkPath, &manualConfig);
    return (status);
}
```

PLL400 手動設定。**Code Listing 29** 参照。

Code Listing 29　Cy_SysClk_Pll400MManualConfigure() 関数

```
cy_en_sysclk_status_t Cy_SysClk_Pll400MManualConfigure(uint32_t clkPath, const cy_stc_pll_400M_manual_config_t
*config)
{
    /* check for error */
    uint32_t pllNo;
    cy_en_sysclk_status_t status = Cy_SysClk_GetPll400MNo(clkPath, &pllNo);
    if(status != CY_SYSCLK_SUCCESS)
    {
        return(status);
    }

    /* valid divider bitfield values */
    if((config->outputDiv < PLL_400M_MIN_OUTPUT_DIV) || (PLL_400M_MAX_OUTPUT_DIV < config->outputDiv))
    {
         return(CY_SYSCLK_BAD_PARAM);
    }

    if((config->referenceDiv < PLL_400M_MIN_REF_DIV) || (PLL_400M_MAX_REF_DIV < config->referenceDiv))
    {
         return(CY_SYSCLK_BAD_PARAM);

    }


    if((config->feedbackDiv  < PLL_400M_MIN_FB_DIV) || (PLL_400M_MAX_FB_DIV < config->feedbackDiv))

    {

         return(CY_SYSCLK_BAD_PARAM);

    }
```

PLL400 PATH 番号の取得。 **Code Listing 30** 参照。

FLL と PLL の設定

Code Listing 29　Cy_SysClk_Pll400MManualConfigure() 関数

```
    un_CLK_PLL400M_CONFIG_t tempClkPLL400MConfigReg;
    tempClkPLL400MConfigReg.u32Register = SRSS->CLK_PLL400M[pllNo].unCONFIG.u32Register;
    if (tempClkPLL400MConfigReg.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)
    {
        tempClkPLL400MConfigReg.stcField.u8FEEDBACK_DIV    = (uint32_t)config->feedbackDiv;
        tempClkPLL400MConfigReg.stcField.u5REFERENCE_DIV   = (uint32_t)config->referenceDiv;
        tempClkPLL400MConfigReg.stcField.u5OUTPUT_DIV      = (uint32_t)config->outputDiv;
    }
    tempClkPLL400MConfigReg.stcField.u2BYPASS_SEL          = (uint32_t)config->outputMode;
    SRSS->CLK_PLL400M[pllNo].unCONFIG.u32Register          = tempClkPLL400MConfigReg.u32Register;

    un_CLK_PLL400M_CONFIG2_t tempClkPLL400MConfig2Reg;
    tempClkPLL400MConfig2Reg.u32Register               = SRSS->CLK_PLL400M[pllNo].unCONFIG2.u32Register;
    tempClkPLL400MConfig2Reg.stcField.u24FRAC_DIV      = config->fracDiv;
    tempClkPLL400MConfig2Reg.stcField.u3FRAC_DITHER_EN = config->fracDitherEn;
    tempClkPLL400MConfig2Reg.stcField.u1FRAC_EN        = config->fracEn;
    SRSS->CLK_PLL400M[pllNo].unCONFIG2.u32Register     = tempClkPLL400MConfig2Reg.u32Register;

    un_CLK_PLL400M_CONFIG3_t tempClkPLL400MConfig3Reg;
    tempClkPLL400MConfig3Reg.u32Register               = SRSS->CLK_PLL400M[pllNo].unCONFIG3.u32Register;
    tempClkPLL400MConfig3Reg.stcField.u10SSCG_DEPTH    = (uint32_t)config->sscgDepth;
    tempClkPLL400MConfig3Reg.stcField.u3SSCG_RATE      = (uint32_t)config->sscgRate;
    tempClkPLL400MConfig3Reg.stcField.u1SSCG_DITHER_EN = (uint32_t)config->sscgDitherEn;
    tempClkPLL400MConfig3Reg.stcField.u1SSCG_EN        = (uint32_t)config->sscgEn;
    SRSS->CLK_PLL400M[pllNo].unCONFIG3.u32Register     = tempClkPLL400MConfig3Reg.u32Register;

    return (CY_SYSCLK_SUCCESS);
}
```

(2) PLL400 の設定

(3) 分数分周器設定

(4) SSCG 設定

Code Listing 30　Cy_SysClk_GetPll400MNo() 関数

```
__STATIC_INLINE cy_en_sysclk_status_t Cy_SysClk_GetPll400MNo(uint32_t pathNo, uint32_t* pllNo)
{
    /* check for error */
    if ((pathNo <= 0ul) || (pathNo > SRSS_NUM_PLL400M))
    {
        /* invalid clock path number */
        return(CY_SYSCLK_BAD_PARAM);
    }
```

Code Listing 30　Cy_SysClk_GetPll400MNo() 関数

```
    *pllNo = pathNo - 1ul;
    return(CY_SYSCLK_SUCCESS);
}
```

Code Listing 31　Cy_SysClk_PllCalucDividers() 関数

```
__STATIC_INLINE cy_en_sysclk_status_t Cy_SysClk_PllCalucDividers(uint32_t  inFreq,
                                                                 uint32_t  targetOutFreq,
                                                                 const cy_stc_pll_limitation_t* lim,
                                                                 uint32_t  fracBitNum,
                                                                 uint32_t* feedBackDiv,
                                                                 uint32_t* refDiv,
                                                                 uint32_t* outputDiv,
                                                                 uint32_t* feedBackFracDiv)
{
    uint64_t errorMin = 0xFFFFFFFFFFFFFFFFull;

    if(feedBackDiv == NULL)
    {
        return (CY_SYSCLK_BAD_PARAM);
    }

    if((feedBackFracDiv == NULL) && (fracBitNum != 0ul))
    {
        return (CY_SYSCLK_BAD_PARAM);
    }

    if(refDiv == NULL)
    {
        return (CY_SYSCLK_BAD_PARAM);
    }

    if(outputDiv == NULL)
    {
        return (CY_SYSCLK_BAD_PARAM);
    }

    if ((targetOutFreq < lim->minFoutput) || (lim->maxFoutput < targetOutFreq))
    {
        return (CY_SYSCLK_BAD_PARAM);
    }

    /* REFERENCE_DIV  selection */
    for (uint32_t i_refDiv = lim->minRefDiv; i_refDiv <= lim->maxRefDiv; i_refDiv++)
    {
        uint32_t fpd_roundDown = inFreq / i_refDiv;
```

Code Listing 31　Cy_SysClk_PllCalucDividers() 関数

```
        if (fpd_roundDown < lim->minFpd)
        {
            break;
        }

        uint32_t fpd_roundUp   = CY_SYSCLK_DIV_ROUNDUP(inFreq, i_refDiv);
        if (lim->maxFpd < fpd_roundUp)
        {
            continue;
        }

        /* OUTPUT_DIV selection */
        for (uint32_t i_outDiv = lim->minOutputDiv; i_outDiv <= lim->maxOutputDiv; i_outDiv++)
        {
            uint64_t tempVco = i_outDiv * targetOutFreq;

            if(tempVco < lim->minFvco)
            {
                continue;
            }
            else if(lim->maxFvco < tempVco)
            {
                break;
            }

            // (inFreq / refDiv) * feedBackDiv = Fvco
            // feedBackDiv = Fvco * refDiv / inFreq
            uint64_t tempFeedBackDivLeftShifted = ((tempVco << (uint64_t)fracBitNum) * (uint64_t)i_refDiv) /
(uint64_t)inFreq;
            uint64_t error = abs(((uint64_t)targetOutFreq << (uint64_t)fracBitNum) - ((uint64_t)inFreq *
tempFeedBackDivLeftShifted / ((uint64_t)i_refDiv * (uint64_t)i_outDiv)));

            if (error < errorMin)
            {
                *feedBackDiv      = (uint32_t)(tempFeedBackDivLeftShifted >> (uint64_t)fracBitNum);
                if(feedBackFracDiv != NULL)
                {
                    if(fracBitNum == 0ul)
                    {
                        *feedBackFracDiv  = 0ul;
                    }
                    else
                    {
                        *feedBackFracDiv  = (uint32_t)(tempFeedBackDivLeftShifted & ((1ull << (uint64_t)fracBitNum) -
1ull));
                    }
                }
                *refDiv           = i_refDiv;
```

FLL と PLL の設定

Code Listing 31　Cy_SysClk_PllCalucDividers() 関数

```
                *outputDiv        = i_outDiv;
                errorMin          = error;
                if(errorMin == 0ull){break;}
            }
        }
        if(errorMin == 0ull){break;}
    }

    if(errorMin == 0xFFFFFFFFFFFFFFFFull)
    {
        return (CY_SYSCLK_BAD_PARAM);
    }
    else
    {
        return (CY_SYSCLK_SUCCESS);
    }
}
```

Code Listing 32　Cy_SysClk_Pll400MEnable() 関数

```
cy_en_sysclk_status_t Cy_SysClk_Pll400MEnable(uint32_t clkPath, uint32_t timeoutus)
{
    uint32_t pllNo;
    cy_en_sysclk_status_t status = Cy_SysClk_GetPll400MNo(clkPath, &pllNo);
    if(status != CY_SYSCLK_SUCCESS)
    {
        return(status);
    }

    /* first set the PLL enable bit */
    SRSS->CLK_PLL400M[pllNo].unCONFIG.stcField.u1ENABLE = 1ul;

    /* now do the timeout wait for PLL_STATUS, bit LOCKED */
    for (; (SRSS->CLK_PLL400M[pllNo].unSTATUS.stcField.u1LOCKED == 0ul) &&
           (timeoutus != 0ul);
         timeoutus--)
    {
        Cy_SysLib_DelayUs(1u);
    }

    status = ((timeoutus == 0ul) ? CY_SYSCLK_TIMEOUT : CY_SYSCLK_SUCCESS);

    return (status);
}
```

(5) PLL400 を有効にする

(6) PLL400 がロックされるまで待機。

(7) タイムアウトの確認。

1 us 待機。

FLL と PLL の設定

Code Listing 33　PLL200 #0 の基本設定

```
:
#define PLL200_0_TARGET_FREQ      (160000000ul)
#define PLL200_1_TARGET_FREQ      (80000000ul)
:
/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul)
:
#define PLL_400M_0_PATH_NO   (1ul)
#define PLL_400M_1_PATH_NO   (2ul)
#define PLL_200M_0_PATH_NO   (3ul)
#define PLL_200M_1_PATH_NO   (4ul)
#define BYPASSED_PATH_NO     (5ul)
 :
/*** Parameters for Clock Configuration ***/
cy_stc_pll_config_t g_pll200_0_Config =
{
    .inputFreq  = PATH_SOURCE_CLOCK_FREQ,       // ECO: 16MHz
    .outputFreq = PLL200_0_TARGET_FREQ,         // target PLL output
    .lfMode     = false,                        // 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(;;);
}
```

PLL ターゲット周波数

TIMEOUT 変数の宣言

PLL 番号の宣言。

PLL200 #0 設定

PLL200 #0 設定。**Code Listing 34** 参照。

Code Listing 34　AllClockConfiguration() 関数

```
static void AllClockConfiguration(void)
{
:
    /***** PLL200M#0(PATH3) source setting ******/
    {
:
        status = Cy_SysClk_PllConfigure(PLL_200M_0_PATH_NO , &g_pll200_0_Config);
        CY_ASSERT(status == CY_SYSCLK_SUCCESS);

        status = Cy_SysClk_PllEnable(PLL_200M_0_PATH_NO, WAIT_FOR_STABILIZATION);
        CY_ASSERT(status == CY_SYSCLK_SUCCESS);
```

PLL200 設定。**Code Listing 35** 参照。

PLL200 を有効にする。
Code Listing 39 参照。

Code Listing 34　AllClockConfiguration() 関数

```
:
    }
    return;
}
```

Code Listing 35　Cy_SysClk_PllConfigure() 関数

```
cy_en_sysclk_status_t Cy_SysClk_PllConfigure(uint32_t clkPath, const cy_stc_pll_config_t *config)
{
    /* check for error */
    uint32_t pllNo;
    cy_en_sysclk_status_t status = Cy_SysClk_GetPllNo(clkPath, &pllNo);
    if(status != CY_SYSCLK_SUCCESS)
    {
        return(status);
    }

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

    /* invalid output frequency */
    cy_stc_pll_manual_config_t manualConfig = {0ul};
    const cy_stc_pll_limitation_t* pllLim = (config->lfMode) ? &g_limPllLF : &g_limPllNORM;
    status = Cy_SysClk_PllCalucDividers(config->inputFreq,
                                        config->outputFreq,
                                        pllLim,
                                        0ul, // Frac bit num
                                        &manualConfig.feedbackDiv,
                                        &manualConfig.referenceDiv,
                                        &manualConfig.outputDiv,
                                        NULL
                                        );
    if(status != CY_SYSCLK_SUCCESS)
    {
        return(status);
    }

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

    status = Cy_SysClk_PllManualConfigure(clkPath, &manualConfig);
    return (status);

}
```

(8) PLL200 が既に有効かどうかを確認する。

PLL200 分周器設定の計算。**Code Listing 38** 参照。

PLL200 手動設定。**Code Listing 36** 参照。

Code Listing 36　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 */
    uint32_t pllNo;
    cy_en_sysclk_status_t status = Cy_SysClk_GetPllNo(clkPath, &pllNo);
    if(status != CY_SYSCLK_SUCCESS)
    {
        return(status);
    }

    /* 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_NORM)) ||
       ((config->lfMode ? MAX_FB_DIV_LF : MAX_FB_DIV_NORM) < config->feedbackDiv))
    {
         return(CY_SYSCLK_BAD_PARAM);
    }

    un_CLK_PLL_CONFIG_t tempClkPLLConfigReg;
    tempClkPLLConfigReg.u32Register = SRSS->unCLK_PLL_CONFIG[pllNo].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[pllNo].u32Register = tempClkPLLConfigReg.u32Register;

    return (CY_SYSCLK_SUCCESS);
}
```

(9) PLL200 設定

Code Listing 37 Cy_SysClk_GetPllNo() 関数

```
__STATIC_INLINE cy_en_sysclk_status_t Cy_SysClk_GetPllNo(uint32_t pathNo, uint32_t* pllNo)
{
    /* check for error */
    if ((pathNo <= SRSS_NUM_PLL400M) || (pathNo > (SRSS_NUM_PLL400M + SRSS_NUM_PLL)))
    {
        /* invalid clock path number */
        return(CY_SYSCLK_BAD_PARAM);
    }

    *pllNo = pathNo - (uint32_t)(SRSS_NUM_PLL400M + 1u);
    return(CY_SYSCLK_SUCCESS);
}
```

Code Listing 38 Cy_SysClk_PllCalucDividers() 関数

```
__STATIC_INLINE cy_en_sysclk_status_t Cy_SysClk_PllCalucDividers(uint32_t  inFreq,
                                                                 uint32_t  targetOutFreq,
                                                                 const cy_stc_pll_limitation_t* lim,
                                                                 uint32_t  fracBitNum,
                                                                 uint32_t* feedBackDiv,
                                                                 uint32_t* refDiv,
                                                                 uint32_t* outputDiv,
                                                                 uint32_t* feedBackFracDiv)
{
    uint64_t errorMin = 0xFFFFFFFFFFFFFFFFull;

    if(feedBackDiv == NULL)
    {
        return (CY_SYSCLK_BAD_PARAM);
    }

    if((feedBackFracDiv == NULL) && (fracBitNum != 0ul))
    {
        return (CY_SYSCLK_BAD_PARAM);
    }

    if(refDiv == NULL)
    {
        return (CY_SYSCLK_BAD_PARAM);
    }
    if(outputDiv == NULL)
    {
        return (CY_SYSCLK_BAD_PARAM);
    }

    if ((targetOutFreq < lim->minFoutput) || (lim->maxFoutput < targetOutFreq))
    {
        return (CY_SYSCLK_BAD_PARAM);
```

Code Listing 38 Cy_SysClk_PllCalucDividers() 関数

```
    }

    /* REFERENCE_DIV  selection */
    for (uint32_t i_refDiv = lim->minRefDiv; i_refDiv <= lim->maxRefDiv; i_refDiv++)
    {
        uint32_t fpd_roundDown = inFreq / i_refDiv;
        if (fpd_roundDown < lim->minFpd)
        {
            break;
        }

        uint32_t fpd_roundUp   = CY_SYSCLK_DIV_ROUNDUP(inFreq, i_refDiv);
        if (lim->maxFpd < fpd_roundUp)
        {
            continue;
        }

        /* OUTPUT_DIV selection */
        for (uint32_t i_outDiv = lim->minOutputDiv; i_outDiv <= lim->maxOutputDiv; i_outDiv++)
        {
            uint64_t tempVco = i_outDiv * targetOutFreq;

            if(tempVco < lim->minFvco)
            {
                continue;
            }
            else if(lim->maxFvco < tempVco)
            {
                break;
            }

            // (inFreq / refDiv) * feedBackDiv = Fvco
            // feedBackDiv = Fvco * refDiv / inFreq
            uint64_t tempFeedBackDivLeftShifted = ((tempVco << (uint64_t)fracBitNum) * (uint64_t)i_refDiv) /
(uint64_t)inFreq;
            uint64_t error = abs(((uint64_t)targetOutFreq << (uint64_t)fracBitNum) - ((uint64_t)inFreq *
tempFeedBackDivLeftShifted / ((uint64_t)i_refDiv * (uint64_t)i_outDiv)));

            if (error < errorMin)
            {
                *feedBackDiv      = (uint32_t)(tempFeedBackDivLeftShifted >> (uint64_t)fracBitNum);
                if(feedBackFracDiv != NULL)
                {
                    if(fracBitNum == 0ul)
                    {
                        *feedBackFracDiv  = 0ul;
                    }
                    else
```

Code Listing 38　Cy_SysClk_PllCalucDividers() 関数

```
                    {
                        *feedBackFracDiv  = (uint32_t)(tempFeedBackDivLeftShifted & ((1ull << (uint64_t)fracBitNum) -
1ull));
                    }
                }
                *refDiv           = i_refDiv;
                *outputDiv        = i_outDiv;
                errorMin          = error;
                if(errorMin == 0ull){break;}
            }
        }
        if(errorMin == 0ull){break;}
    }

    if(errorMin == 0xFFFFFFFFFFFFFFFFull)
    {
        return (CY_SYSCLK_BAD_PARAM);
    }
    else
    {
        return (CY_SYSCLK_SUCCESS);
    }
}
```

Code Listing 39　Cy_SysClk_PllEnable() 関数

```
cy_en_sysclk_status_t Cy_SysClk_PllEnable(uint32_t clkPath, uint32_t timeoutus)
{
    uint32_t pllNo;
    cy_en_sysclk_status_t status = Cy_SysClk_GetPllNo(clkPath, &pllNo);
    if(status != CY_SYSCLK_SUCCESS)
    {
        return(status);
    }

    /* first set the PLL enable bit */
    SRSS->unCLK_PLL_CONFIG[pllNo].stcField.u1ENABLE = 1ul;

    /* now do the timeout wait for PLL_STATUS, bit LOCKED */
    for (; (SRSS->unCLK_PLL_STATUS[pllNo].stcField.u1LOCKED == 0ul) &&
           (timeoutus != 0ul);
         timeoutus--)
    {
        Cy_SysLib_DelayUs(1u);
    }

    status = ((timeoutus == 0ul) ? CY_SYSCLK_TIMEOUT : CY_SYSCLK_SUCCESS);
```

(10) PLL200 を有効にする

(11) PLL200 がロックされるまで待機。

(12) タイムアウトの確認。

1 us 待機。

Code Listing 39　Cy_SysClk_PllEnable() 関数

```
    return (status);
}
```

5 内部クロックの設定

ここではクロックシステム中の内部クロックの設定方法について説明します。

5.1 CLK_PATHx の設定

CLK_PATHx は CLK_HFx の入力ソースとして使用されます。CLK_PATHx 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_PATH9 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。

Table 13 に FLL/PLL および CLK_PATHx の関係を示します。

Table 13 FLL/PLL および PATHx の関係

FLL/PLL	CLK_PATHx
FLL	CLK_PATH0
PLL#0	CLK_PATH1
PLL#1	CLK_PATH2
PLL#2	CLK_PATH3
PLL#3	CLK_PATH4
PLL#4	CLK_PATH5
PLL#5	CLK_PATH6
PLL#6	CLK_PATH7
PLL#7	CLK_PATH8
Directly (FLL と PLL は選択できません)	CLK_PATH9

CLK_PATHx の生成ダイヤグラムを **Table 13** に示します。

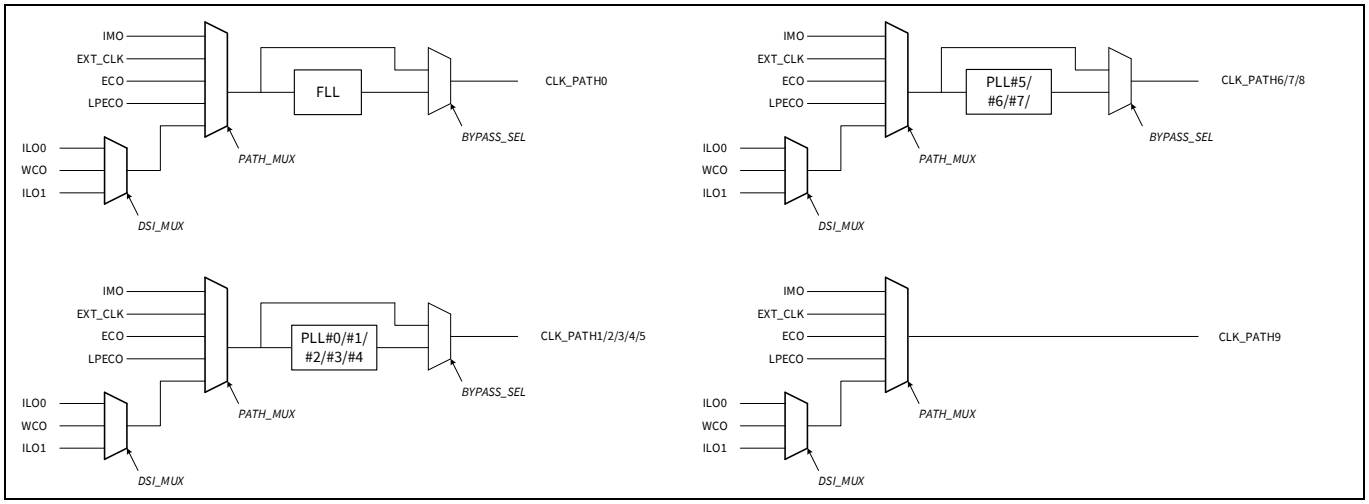

Figure 14 CLK_PATH の生成ダイヤグラム

CLK_PATHx 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 CLK_PATHx には BYPASS_MUX の設定も必要です。CLK_PATHx の設定に必要なレジスタを **Table 14** に示します。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

Table 14 CLK_PATHx 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
		5	LPECO
		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 [3]
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 [3]

[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。

5.2 CLK_HFx の設定

CLK_HFx (x=0〜13) は CLK_PATHy (y=0〜9) から選択できます。Predivider は選択された CLK_PATHx を分周するために利用できます。CLK_HF0 は CPU のソースクロックであるため、常に有効です。CLK_HFx は無効にすることが可能です。

CLK_HFx を有効にするためには、各 CLK_ROOT_SELECT レジスタの ENABLE ビットに'1'を書き込んでください。CLK_HFx を無効にするためには、各 CLK_ROOT_SELECT レジスタの ENABLE ビットに'0'を書き込んでください。

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, および 8 分周から Predivider の値を設定します。

ROOT_MUX と Predivider の詳細を **Figure 15** に示します。

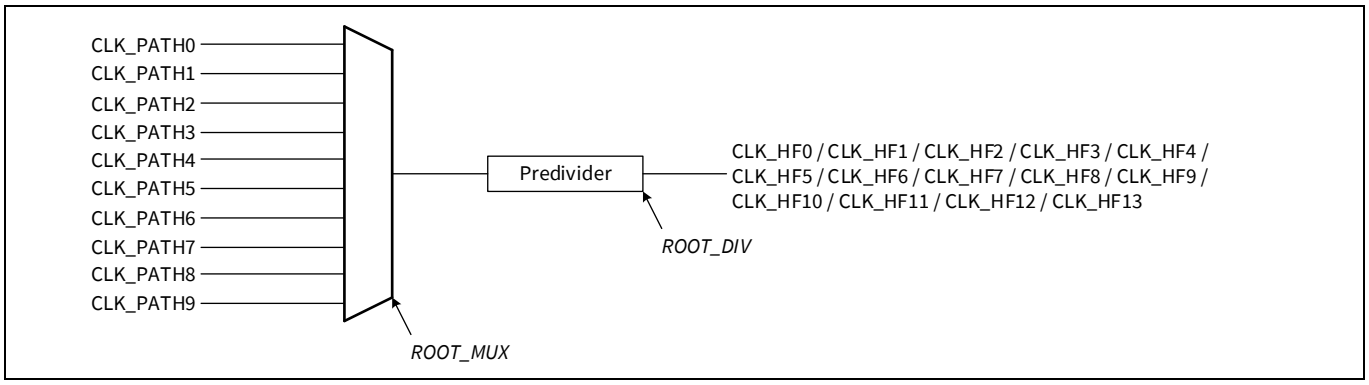

Figure 15 ROOT_MUX と Predivider

CLK_HFx に必要なレジスタを **Table 15** に示します。詳細は **architecture TRM** を参照してください。

Table 15 CLK_HFx 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
		4	CLK_PATH4
		5	CLK_PATH5
		6	CLK_PATH6
		7	CLK_PATH7
		8	CLK_PATH8
		9	CLK_PATH9
		その他の値	予約済み。使用禁止。
CLK_ROOT_SELECT	ROOT_DIV[1:0]	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 は利用可能なソースである WCO, ILO0, ILO1, ECO_Prescaler, および LPECO_Prescaler のいずれかから 1 つ選択できます。CLK_LF は WDT の入力クロックである ILO0 が選択できるため、WDT_CTL レジスタの WDT_LOCK ビットが無効であるときは CLK_LF を設定できません。

CLK_LF を設定している LFCLK_SEL の詳細を **Figure 16** に示します。

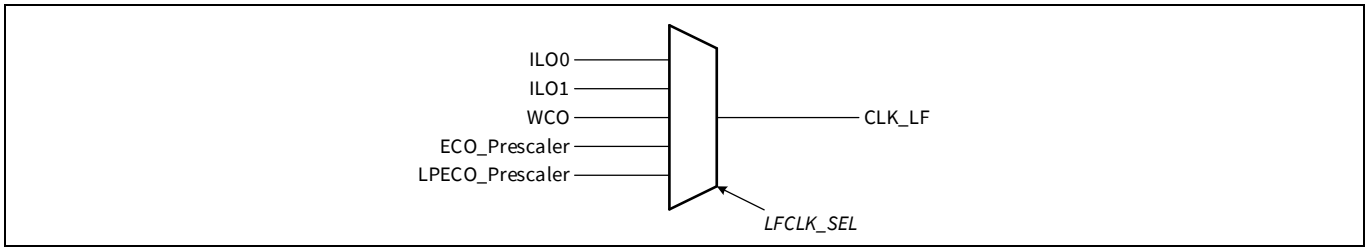

Figure 16 LFCLK_SEL

CLK_LF に必要なレジスタを **Table 16** に示します。詳細については **architecture TRM** を参照してください。

Table 16 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 (初期値)	ILO0
		1	WCO
		4	ILO1
		5	ECO_Prescaler
		6	LPECO_Prescaler
		その他の値	予約済み。使用禁止

5.4 CLK_FAST_0/CLK_FAST_1 の設定

CLK_HF1 を (x+1) で分周して CLK_FAST_0 と CLK_FAST_1 は生成されます。CLK_FAST_0 と CLK_FAST_1 を設定する場合、CPUSS_FAST_0_CLOCK_CTL レジスタおよび CPUSS_FAST_1_CLOCK_CTL レジスタの FRAC_DIV ビットおよび INT_DIV ビットに分周する値 (x=0～255) を設定してください。

5.5 CLK_MEM の設定

CLK_MEM は CLK_HF0 を分周して生成され、その周波数は CLK_HF0 を (x+1) で分周した値で設定されます。CLK_MEM を設定する場合、CPUSS_MEM_CLOCK_CTL レジスタの INT_DIV ビットを分周した値 (x=0～255) を設定してください。

5.6 CLK_PERI の設定

CLK_PERI は周辺クロック分周器と CLK_GR のクロック入力です。CLK_PERI は CLK_HF0 を分周して生成され、その周波数は CLK_HF0 を (x+1) で分周した値で設定されます。CLK_PERI を設定する場合、CPUSS_PERI_CLOCK_CTL レジスタの INT_DIV ビットを分周した値 (x=0～255) を設定してください。

5.7 CLK_SLOW の設定

CLK_SLOW は CLK_MEM を分周して生成され、その周波数は CLK_MEM を (x+1) で分周した値で設定されます。CLK_MEM を設定した後、CPUSS_SLOW_CLOCK_CTL レジスタの INT_DIV ビットを分周した値 (x=0～255) を設定してください。

5.8 CLK_GR の設定

CLK_GR のクロックソースはグループ 3, 4, 8 では CLK_PERI であり、グループ 5, 6, 9 では CLK_HF2 です。グループ 3, 4, 8 は CLK_PERI を分周したクロックです。CLK_GR3, CLK_GR4 および CLK_GR8 を生成するためには、CPUSS_PERI_GRx_CLOCK_CTL レジスタの INT8_DIV ビットに分周する値 (1～255) を書き込んでください。

5.9 PCLK の設定

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。周辺クロックの割当については **Datasheet** の「Peripheral clocks」を参照してください。

周辺クロック分周器を設定する手順を **Figure 17** に示します。詳細については **architecture TRM** を参照してください。

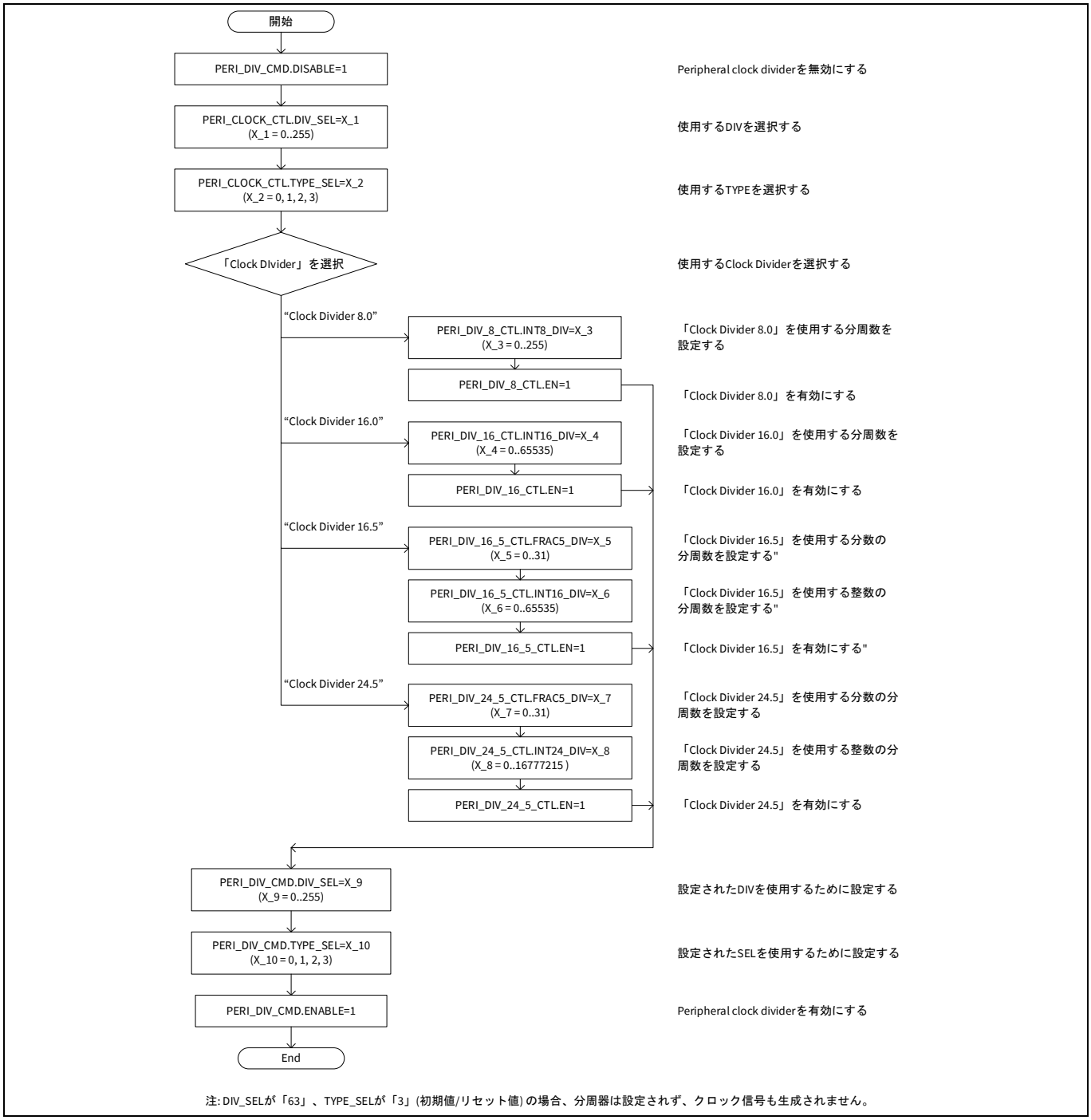

Figure 17 PCLK 生成の設定手順

5.9.1 PCLK の設定例

5.9.1.1 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)

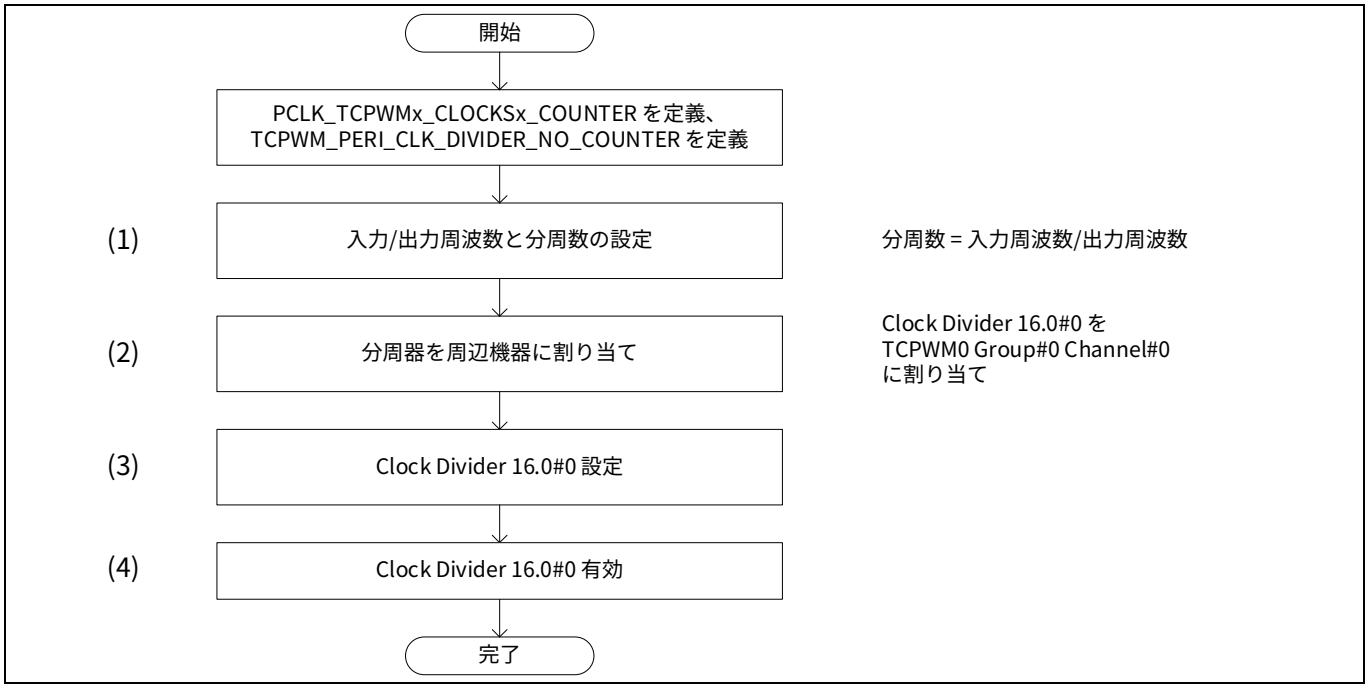

Figure 18 PCLK の設定手順の例

5.9.1.2 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 の設定部のパラメータを **Table 17** に、関数を **Table 18** に示します。

Table 17 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 bit 分周器 CY_SYSCLK_DIV_16_BIT = 1u, 16 bit 分周器 CY_SYSCLK_DIV_16_5_BIT = 2u, 16.5 bit 分数分周器 CY_SYSCLK_DIV_24_5_BIT = 3u, 24.5 bit 分数分周器	1ul
periFreq	周辺機器のクロック周波数	80000000ul (80 MHz)

内部クロックの設定

パラメータ	説明	値
targetFreq	ターゲットクロック周波数	2000000ul (2 MHz)
divNum	分周数	periFreq/targetFreq

Table 18　PCLK(TCPWM タイマの例)設定関数一覧

関数	説明	値
`Cy_SysClk_PeriphAssign Divider(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_PeriphSet Divider(dividerType, dividerNum, dividerValue)`	周辺機器の分周器を設定する	dividerType, = CY_SYSCLK_DIV_16_BIT dividerNum = TCPWM_PERI_CLK_DIVIDER_NO_COUNTER dividerValue = divNum-1ul
`Cy_SysClk_PeriphEnable Divider(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

5.9.2　PCLK 設定の初期設定のサンプルコード (TCPWM タイマの例)

サンプルコードを **Code Listing 40**～**Code Listing 43** に示します。

Code Listing 40　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 0u
:
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 41** 参照。

周辺機器の分周器を有効にする設定。 **Code Listing 43** 参照。

周辺機器の分周器を設定。**Code Listing 42** 参照。

内部クロックの設定

Code Listing 41　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 42　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) Clock Divider 16.0#0 に分周設定

Code Listing 43　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 */
```

(4) ”Clock Divider 16#0”を有効にする。

内部クロックの設定

Code Listing 43　Cy_SysClk_PeriphEnableDivider() 関数

```
    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;
}
```

分周器のタイプ選択の設定。

分周器番号の設定。

5.10 ECO_Prescaler の設定

ECO_Prescaler は ECO を分周し、CLK_LF で使用できるクロックを生成します。分周機能は 10 ビット整数分周と 8 ビット分数分周があります。

ECO_Prescaler を有効する手順を **Figure 19** に示します。ECO_Prescaler の詳細については **architecture TRM** と **registers TRM** を参照してください。

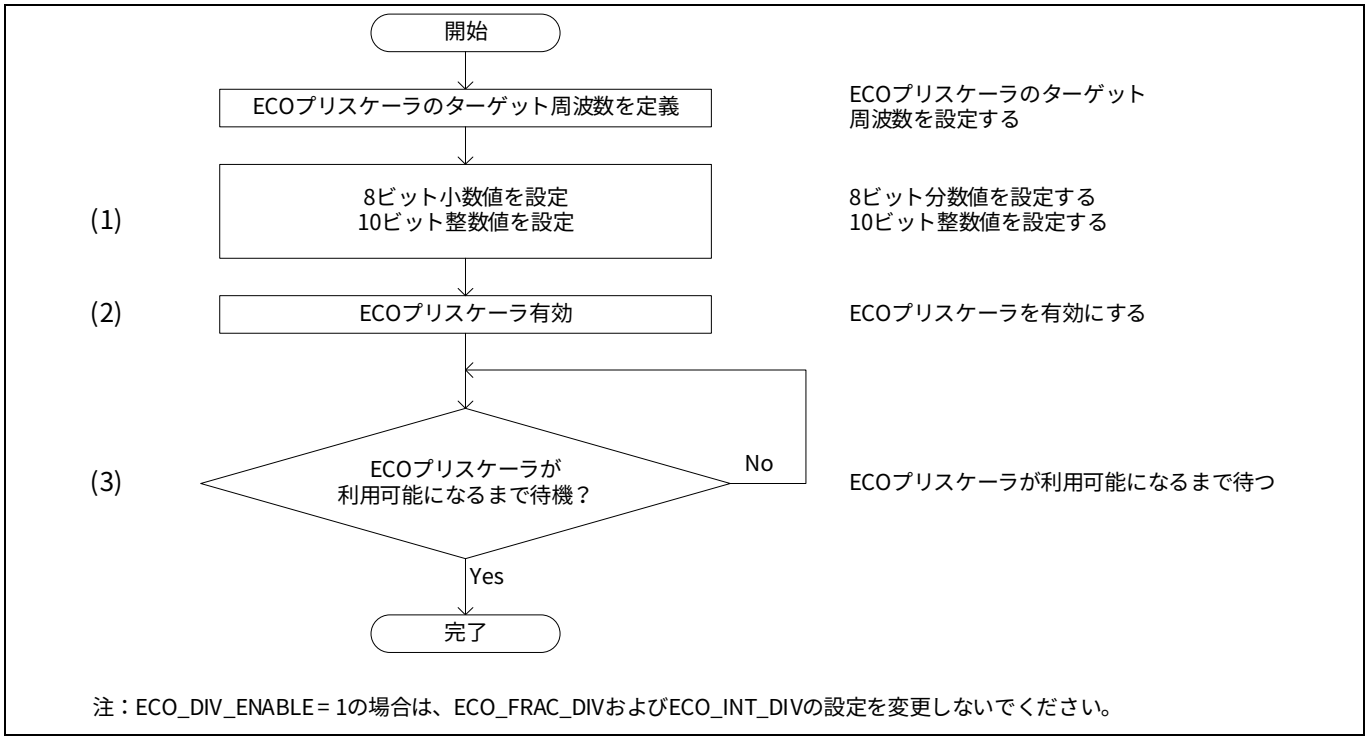

Figure 19 ECO_Prescaler の有効化

ECO_Prescaler を無効にする手順を **Figure 20** に示します。ECO_Prescaler の詳細については **architecture TRM** を参照してください。

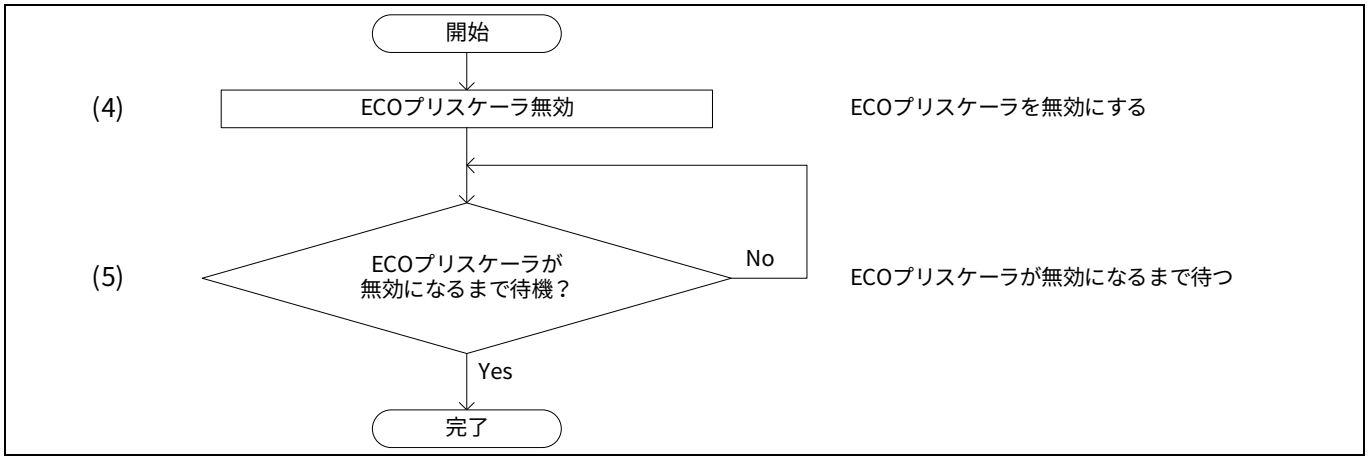

Figure 20 ECO_Prescaler の無効化

5.10.1 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

5.10.2 コンフィグレーション

ECO プリスケーラ設定における SDL の設定部のパラメータを **Table 19** に、関数を **Table 20** に示します。

Table 19 ECO プリスケーラ設定パラメーター覧

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	PATH ソースクロック周波数	CLK_FREQ_ECO

Table 20 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_SetEco Prescale(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_EcoPrescale Enable(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_SetEco PrescaleManual (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_GetEco PrescaleStatus`	プリスケーラの状態を確認	–

5.10.3 ECO プリスケーラ設定の初期設定のサンプルコード

サンプルコードを **Code Listing 44**～**Code Listing 50** に示します。

Code Listing 44 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();
:
    /* Please check clock output using oscilloscope after CPU reached here. */
    for(;;);
}
```

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 45** 参照。

Code Listing 45 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 46** 参照。

ECO プリスケーラを有効にする。**Code Listing 48** 参照。

Code Listing 46 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;
```

内部クロックの設定

Code Listing 46　Cy_SysClk_SetEcoPrescale() 関数

```
    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 47** 参照。

Code Listing 47　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。

Code Listing 48　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--;
```

(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。

内部クロックの設定

Code Listing 48　Cy_SysClk_EcoPrescaleEnable() 関数

```
    }

    return CY_SYSCLK_SUCCESS;
}
```

Code Listing 49　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);
}
```

ECO プリスケーラの状態を確認する。

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 50　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。

5.11 LPECO_Prescaler の設定

LPECO_Prescaler は LPECO を分周します。分周機能は 10 ビット整数分周と 8 ビット分数分周があります。

LPECO_Prescaler を有効にする手順を **Figure 21** に示します。LPECO_Prescaler の詳細については **architecture TRM** を参照してください。

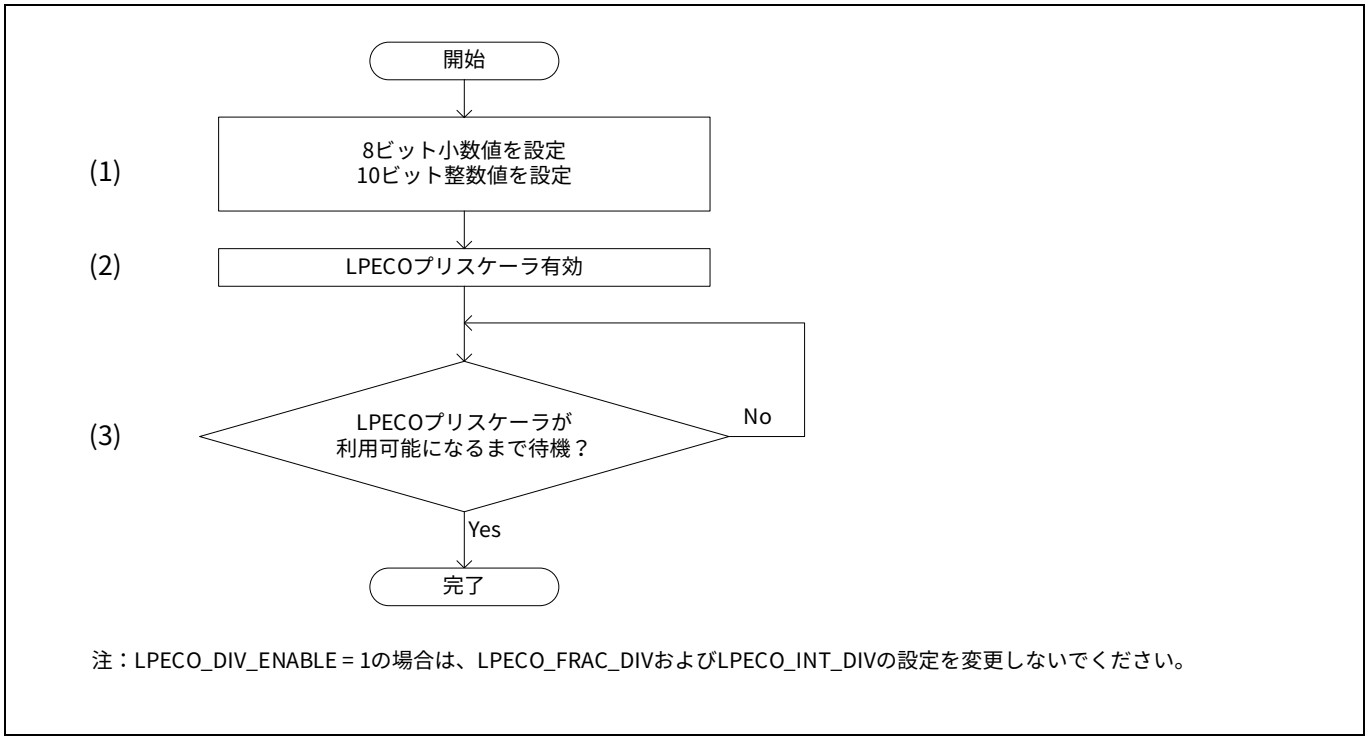

Figure 21 LPECO_Prescaler の有効化

LPECO_Prescaler を無効にする手順を **Figure 22** に示します。LPECO_Prescaler の詳細については **architecture TRM** を参照してください。

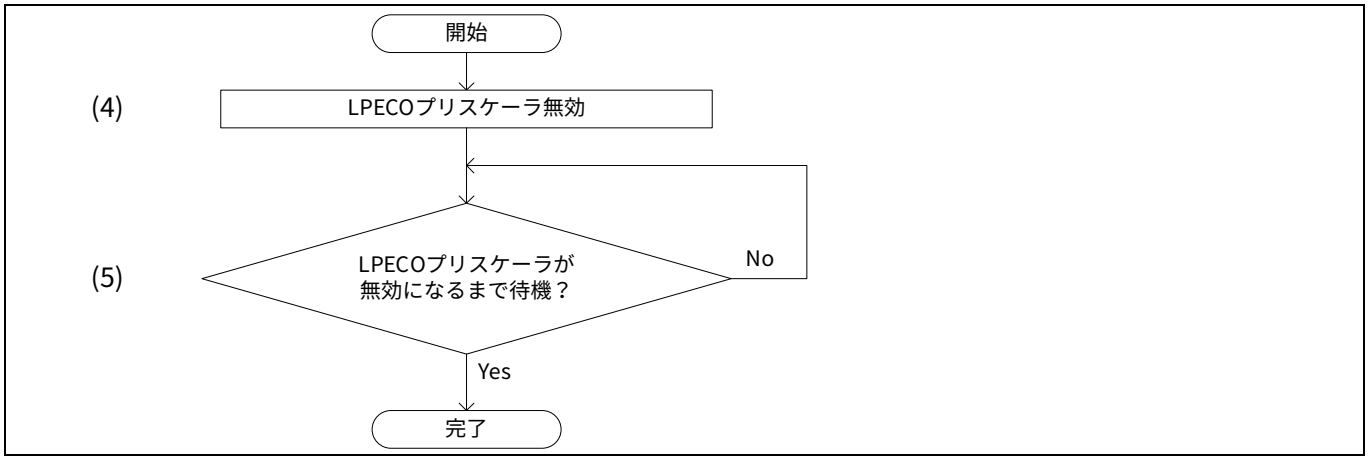

Figure 22 LPECO_Prescaler の無効化

5.11.1 ユースケース

- 入力クロック周波数: 8 MHz
- LPECO プリスケーラターゲット周波数: 1.234567 MHz

5.11.2 コンフィグレーション

LPECO プリスケーラ設定における 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 LPECO プリスケーラ設定パラメーター覧

パラメータ	説明	値
LPECO_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_LPECO	ECO クロック周波数	8000000ul (8 MHz)
PATH_SOURCE_CLOCK_FREQ	PATH ソースクロック周波数	CLK_FREQ_LPECO

Table 22 LPECO 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_ClkBak_LPECO_SetPrescale(frac,int)`	LPECO から 32.768 kHz を生成するプリスケーラ整数分周器および分数分周器	frac = fixedPointDivNum & 0x000000FFul int = (fixedPointDivNum & 0x0003FF00ul) >> 8ul) - 1ul
`Cy_SysClk_ClkBak_LPECO_EnableDivider(divInt,divFract)`	LPECO プリスケーラを有効に設定する	divInt = 0x3FF divFract = 0xFF
`Cy_SysClk_ClkBak_LPECO_PrescalerOkay()`	プリスケーラ分周器を設定した後、LPECO からステータスを返す。	–

5.11.3 LPECO プリスケーラ設定の初期設定のサンプルコード

サンプルコードを **Code Listing 51**～**Code Listing 55** に示します。

Code Listing 51 LPECO プリスケーラの基本設定

```
#define LPECO_PRESCALER_TARGET_FREQ (1234567ul)
#define CLK_FREQ_LPECO       (8000000ul)
#define PATH_SOURCE_CLOCK_FREQ CLK_FREQ_LPECO

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


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

LPECO プリスケーラのターゲット周波数の宣言

LPECO クロック周波数の宣言

TIMEOUT 変数の宣言

LPECO プリスケーラの設定。**Code Listing 52** 参照。

内部クロックの設定

Code Listing 52　AllClockConfiguration() 関数

```
static void AllClockConfiguration(void)
{
:
    /***** LPECO prescaler setting ******/
    {
        cy_en_sysclk_status_t lpecoPreStatus;

        lpecoPreStatus = Cy_SysClk_ClkBak_LPECO_SetPrescale(CLK_FREQ_LPECO, LPECO_PRESCALER_TARGET_FREQ);
        CY_ASSERT(lpecoPreStatus == CY_SYSCLK_SUCCESS);

        lpecoPreStatus = Cy_SysClk_ClkBak_LPECO_PrescaleEnable(WAIT_FOR_STABILIZATION);
        CY_ASSERT(lpecoPreStatus == CY_SYSCLK_SUCCESS);
    }
:
    return;
}
```

LPECO プリスケーラの設定。**Code Listing 53** 参照。

LPECO プリスケーラが有効。**Code Listing 54** 参照。

Code Listing 53　Cy_SysClk_ClkBak_LPECO_SetPrescale () 関数

```
__STATIC_INLINE cy_en_sysclk_status_t Cy_SysClk_ClkBak_LPECO_SetPrescale(uint32_t lpecoFreq, uint32_t targetFreq)
{
    // Frequency of LPECO (4MHz ~ 8MHz) might exceed 32-bit value if shifted 8 bit.
    // So, it uses 64-bit data for fixed-point operation.
    // Lowest 8 bits are fractional value. Next 10 bits are integer value.
    uint64_t fixedPointLPEcoFreq = ((uint64_t)lpecoFreq << 8ull);
    uint64_t fixedPointDivNum64;
    uint32_t fixedPointDivNum;

    // Culculate the divider number
    fixedPointDivNum64 = fixedPointLPEcoFreq / (uint64_t)targetFreq;

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

    fixedPointDivNum = (uint32_t)fixedPointDivNum64;

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

    return CY_SYSCLK_SUCCESS;
}
```

LPECO プリスケーラの設定。**Code Listing 54** 参照。

内部クロックの設定

Code Listing 54　Cy_SysClk_ClkBak_LPECO_SetPrescaleManual() 関数

```
__STATIC_INLINE void Cy_SysClk_ClkBak_LPECO_SetPrescaleManual(uint16_t intDiv, uint8_t fracDiv)
{
    if(BACKUP->unLPECO_PRESCALE.stcField.u1LPECO_DIV_ENABLED == 0)
    {
        BACKUP->unLPECO_PRESCALE.stcField.u8LPECO_FRAC_DIV = fracDiv;
        BACKUP->unLPECO_PRESCALE.stcField.u10LPECO_INT_DIV = intDiv;
    }
}
```

(1) 8 ビット分数値と 10 ビット整数値の設定。

Code Listing 55　Cy_SysClk_ClkBak_LPECO_PrescaleEnable() 関数

```
__STATIC_INLINE cy_en_sysclk_status_t Cy_SysClk_ClkBak_LPECO_PrescaleEnable(uint32_t timeoutus)
{
    // Send enable command
    BACKUP->unLPECO_CTL.stcField.u1LPECO_DIV_ENABLE = 1ul;

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

        Cy_SysLib_DelayUs(1u);

        timeoutus--;
    }

    return CY_SYSCLK_SUCCESS;
}
```

(2) LPECO プリスケーラを有効にする。

(3) LPECO プリスケーラが利用可能になるまで待機。

Code Listing 56　Cy_SysClk_ClkBak_LPECO_PrescalerOkay() 関数

```
__STATIC_INLINE bool Cy_SysClk_ClkBak_LPECO_PrescalerOkay(void)
{
    if(BACKUP->unLPECO_PRESCALE.stcField.u1LPECO_DIV_ENABLED == 1)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

プリスケーラの状態を確認。

LPECO 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。

内部クロックの設定

Code Listing 57　Cy_SysClk_ClkBak_LPECO_PrescaleDisable() 関数

```
__STATIC_INLINE cy_en_sysclk_status_t Cy_SysClk_ClkBak_LPECO_PrescaleDisable(uint32_t timeoutus)
{
    // Send the disable command
    BACKUP->unLPECO_CTL.stcField.u1LPECO_EN = 0ul;

    // Wait for eco prescaler to get enabled
    while(BACKUP->unLPECO_PRESCALE.stcField.u1LPECO_DIV_ENABLED == 1)
    {
        if(0ul == timeoutus)
        {
            return CY_SYSCLK_TIMEOUT;
        }

        Cy_SysLib_DelayUs(1u);

        timeoutus--;
    }

    return CY_SYSCLK_SUCCESS;
}
```

(4) LPECO プリスケーラを無効にする。

(5) LPECO プリスケーラが無効になるまで待機。

6 補足情報

6.1 周辺機能へのクロック入力

Table 23～Table 30 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、Datasheet の「Peripheral clocks」を参照してください。

Table 23 TCPWM[0]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[0]	CLK_GR3 (グループ 3)	PCLK (PCLK_TCPWM0_CLOCKSx, x = 0～37)
		PCLK (PCLK_TCPWM0_CLOCKSy, y = 256～267)
		PCLK (PCLK_TCPWM0_CLOCKSz, z = 512～543)

Table 24 CAN FD へのクロック入力

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

Table 25 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)

Table 26 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)
SCB6		PCLK (PCLK_SCB6_CLOCK)
SCB7		PCLK (PCLK_SCB7_CLOCK)
SCB8		PCLK (PCLK_SCB8_CLOCK)
SCB9		PCLK (PCLK_SCB9_CLOCK)
SCB10		PCLK (PCLK_SCB10_CLOCK)
SCB11		PCLK (PCLK_SCB11_CLOCK)

Table 27 SAR ADC へのクロック入力

周辺機能	動作クロック	ユニットクロック
SAR ADC	CLK_GR9 (グループ 9)	Unit0: PCLK (PCLK_PASS_CLOCK_SAR0)

補足情報

Table 28 CXPI 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
CXPI	CLK_GR5 (グループ 5)	PCLK (PCLK_CXPI0_CLOCK_CH_EN0)
		PCLK (PCLK_CXPI0_CLOCK_CH_EN1)

Table 29 SMIF へのクロック入力

周辺機能	“clk_slow” ドメイン (XIP AHB-Lite Interface0)	“clk_mem”ドメイン (XIP AHB Interface)	“clk_sys”ドメイン (MMIO AHB-Lite Interface)	“clk_if” ドメイン
SMIF	clk_slow	clk_mem	CLK_GR4	CLK_HF6

Table 30 AUDIOSS へのクロック入力

周辺機能	clk_sys_i2s	clk_audio_i2s
AUDIOSS	CLK_HF5	CLK_GR8

*Note: Ethernet クロックの詳細については **architecture TRM** を参照してください。*

6.2 クロック調整カウンタ機能のユースケース

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。詳細については **architecture TRM** を参照してください。

クロック調整カウンタを使用して調整するためには、次の手順を使用してください。

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。このカウンタ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 は昇順のカウントを停止し、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。

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

ILO0 と ECO を使用した場合のクロック調整カウンタ機能の例を **Figure 23** に示します。 ILO0 と ECO を有効にする必要があります。ILO0/ILO1 および ECO については、**3.4 ILO0/ILO1 の設定**および **3.1 ECO の設定**を参照してください。

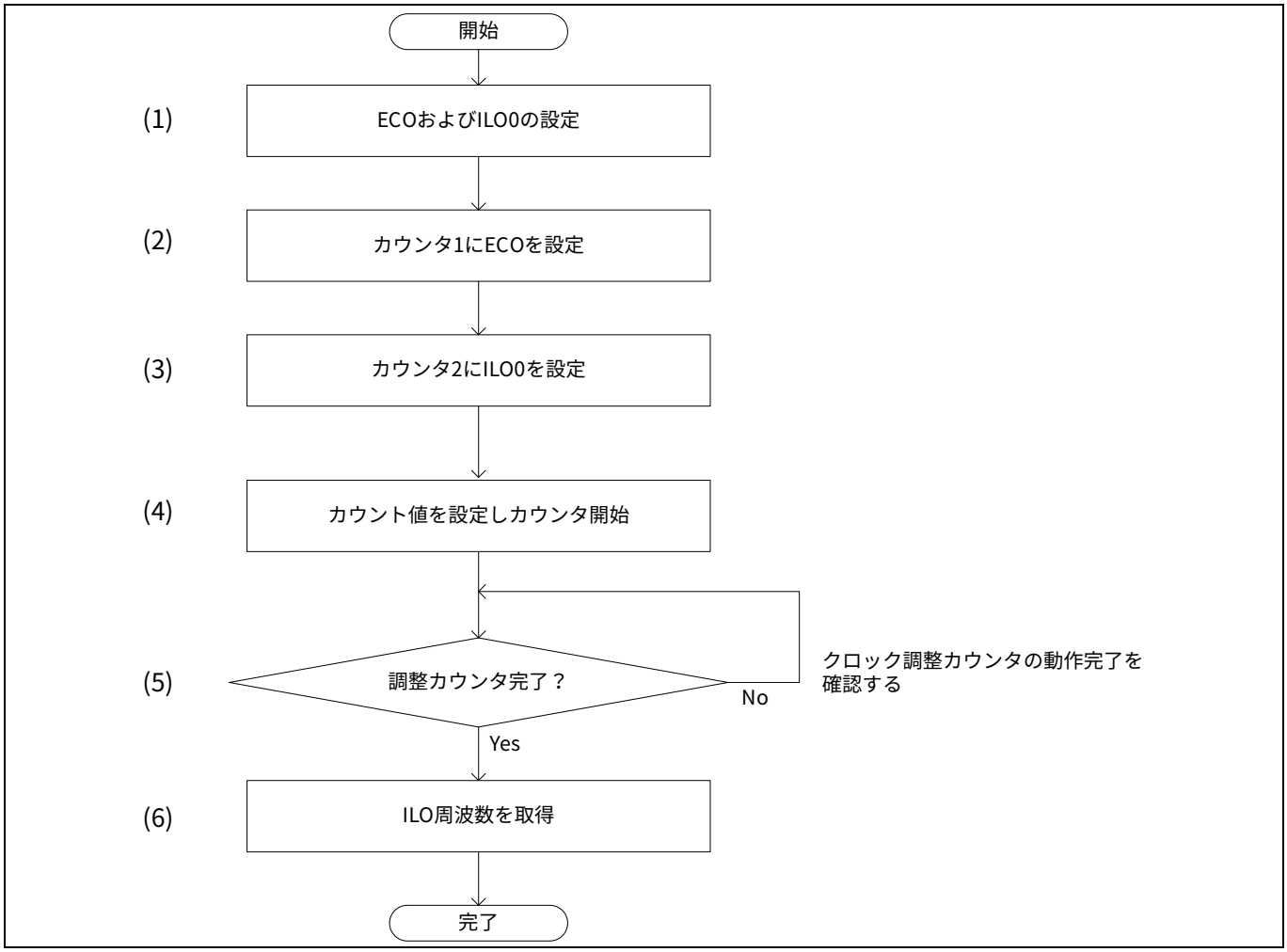

Figure 23　　ILO0 と ECO を用いたクロック調整カウンタの例

6.2.1.1　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

6.2.1.2　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 31** に、関数を **Table 32** に示します。

補足情報

Table 31　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一覧

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	測定クロック周波数を保存する	-
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 32　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

6.2.1.3　ILO0 および ECO 設定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 58**～**Code Listing 62** に示します。

Code Listing 58　ILO_0 および ECO 設定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)
```

CY_SYSCLK_DIV_ROUND 関数の宣言。

測定クロック(ILO0)の宣言。

補足情報

Code Listing 58　ILO_0 および ECO 設定を使用したクロック調整カウンタの基本設定

```
    {.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(;;);
}
```

(1) ECO と ILO0 の設定。
3.1 ECO の設定と **3.4 ILO0/ILO1 の設定**参照。

クロック周波数を取得する。**Code Listing 59** 参照。

Code Listing 59　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 の状態を確認する。

クロック測定カウンタ開始。
Code Listing 60 参照。

カウンタ測定が完了したかどうかを確認する。**Code Listing 61** 参照。

ILO 周波数を取得。
Code Listing 62 参照。

補足情報

Code Listing 60　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)を設定する

(3) 測定クロック(ILO0)を設定する

(4) カウント値を設定しカウンタを開始する。

Code Listing 61　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する。

Code Listing 62　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)
    {
```

Code Listing 62　Cy_SysClk_ClkMeasurementCountersGetFreq() 関数

```
        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

ILO 周波数は製造時に決定されます。ILO 周波数は電圧および温度条件に応じて変化するため、ILO 周波数は適宜更新できます。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.WDT_LOCK によって保護されます。WDT_CTL レジスタの仕様については **architecture TRM** の「Watchdog timer」を参照してください。

補足情報

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 の校正のフロー例を **Figure 24** に示します。

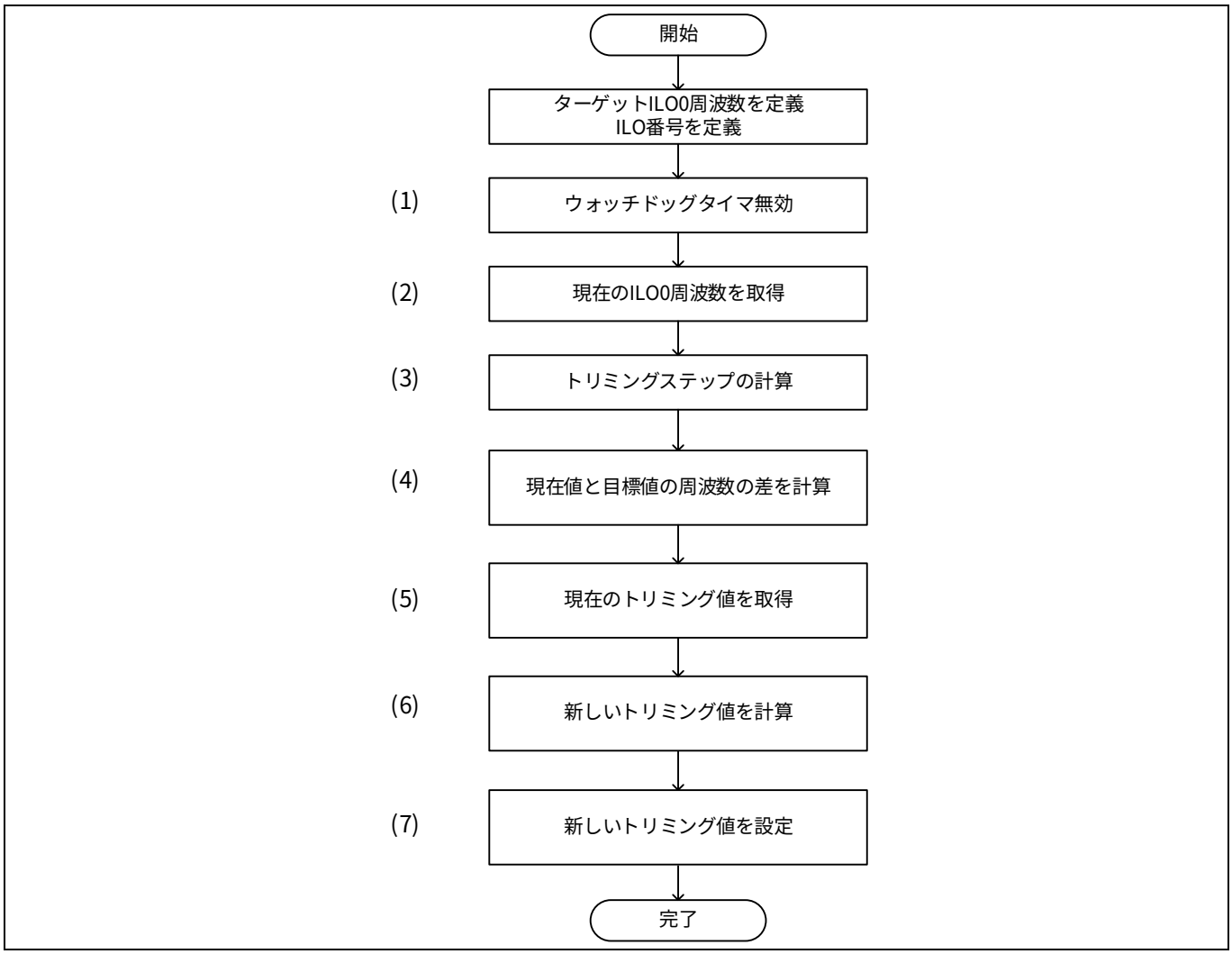

Figure 24　　ILO0 の校正

6.2.2.1　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 33** に、関数を **Table 34** に示します。

Table 33　　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 34　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: 在の ILO 0 周波数 iloNo: ILO 番号のトリミング	 iloFreq: iloFreq iloNo: ILONo

6.2.2.2　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 63**～**Code Listing 64** に示します。

Code Listing 63　クロック調整カウンタ設定を使用した ILO0 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 周波数の宣言。

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 59** 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 64** 参照。

補足情報

Code Listing 64　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計算

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する。

(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する。

(6) 新しいトリム値を計算する。

ウォッチドッグタイマが無効かどうか確認する

(7) 新しいトリミング値を設定する

6.3 CSV ダイヤグラム、およびモニタークロックとリファレンスクロックの関係

Figure 25 に CSV におけるモニタークロックとリファレンスクロックのクロックダイヤグラムを示します。モニタークロックとリファレンスクロックの関係を **Table 35** に示します。

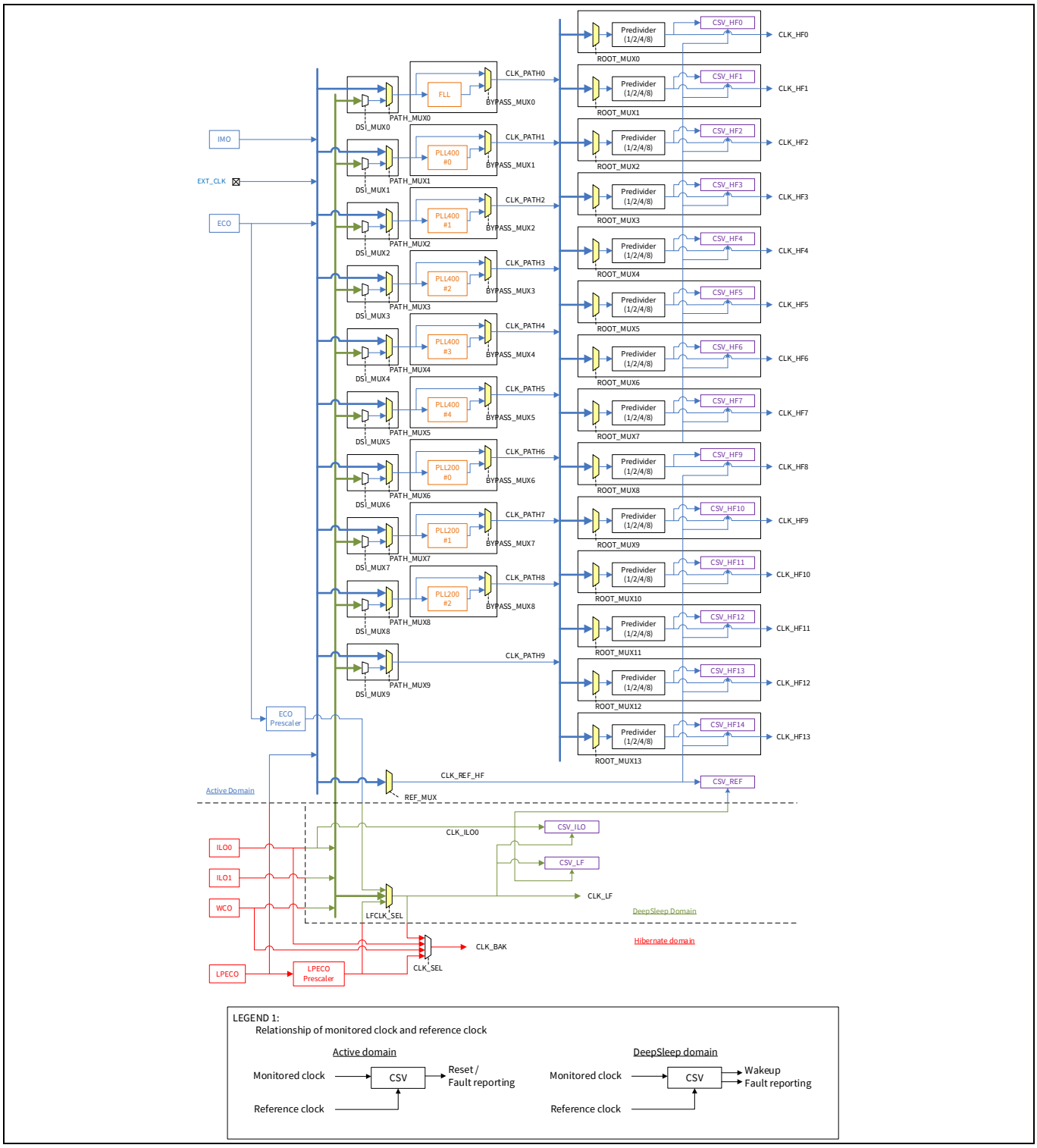

Figure 25 CSV ダイヤグラム

補足情報

Table 35　　モニタークロックとリファレンスクロック

CSV コンポーネント	モニタークロック	リファレンスクロック	注意事項
CSV_HF0	CLK_HF0	CLK_REF_HF	CLK_REF_HF は CLK_IMO, EXT_CLK, CLK_ECO, または CLK_LPECO から選択されます。
CSV_HF1	CLK_HF1	CLK_REF_HF	CLK_REF_HF は CLK_IMO, EXT_CLK, CLK_ECO, または CLK_LPECO から選択されます。
CSV_HF2	CLK_HF2	CLK_REF_HF	CLK_REF_HF は CLK_IMO, EXT_CLK, CLK_ECO, または CLK_LPECO から選択されます。
CSV_HF3	CLK_HF3	CLK_REF_HF	CLK_REF_HF は CLK_IMO, EXT_CLK, CLK_ECO, または CLK_LPECO から選択されます。
CSV_HF4	CLK_HF4	CLK_REF_HF	CLK_REF_HF は CLK_IMO, EXT_CLK, CLK_ECO, または CLK_LPECO から選択されます。
CSV_HF5	CLK_HF5	CLK_REF_HF	CLK_REF_HF は CLK_IMO, EXT_CLK, CLK_ECO, または CLK_LPECO から選択されます。
CSV_HF6	CSV_HF6	CLK_REF_HF	CLK_REF_HF は CLK_IMO, EXT_CLK, CLK_ECO, または CLK_LPECO から選択されます。
CSV_HF7	CLK_HF7	CLK_REF_HF	CLK_REF_HF は CLK_IMO, EXT_CLK, CLK_ECO, または CLK_LPECO から選択されます。
CSV_HF8	CLK_HF7	CLK_REF_HF	CLK_REF_HF は CLK_IMO, EXT_CLK, CLK_ECO, または CLK_LPECO から選択されます。
CSV_HF9	CLK_HF7	CLK_REF_HF	CLK_REF_HF は CLK_IMO, EXT_CLK, CLK_ECO, または CLK_LPECO から選択されます。
CSV_HF10	CLK_HF7	CLK_REF_HF	CLK_REF_HF は CLK_IMO, EXT_CLK, CLK_ECO, または CLK_LPECO から選択されます。
CSV_HF11	CLK_HF7	CLK_REF_HF	CLK_REF_HF は CLK_IMO, EXT_CLK, CLK_ECO, または CLK_LPECO から選択されます。
CSV_HF12	CLK_HF7	CLK_REF_HF	CLK_REF_HF は CLK_IMO, EXT_CLK, CLK_ECO, または CLK_LPECO から選択されます。
CSV_HF13	CLK_HF7	CLK_REF_HF	CLK_REF_HF は CLK_IMO, EXT_CLK, CLK_ECO, または CLK_LPECO から選択されます。
CSV_REF	CLK_REF_HF	ILO0(CLK_ILO0)	–
CSV_ILO	ILO0 (CLK_ILO0)	CLK_LF	CLK_LF は WCO, ILO0, ILO1, ECO_Prescaler, または LPECO_Prescaler から選択されます。
CSV_LF	CLK_LF	ILO0(CLK_ILO0)	–

7 用語集

Table 36 用語集

用語	説明
AUDIOSS	Audio subsystem。詳細は TRAVEO™ T2G **achitecture TRM** の「Audio subsystem」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 **achitecture TRM** の 「CAN FD controller」を参照してください。
CLK_FAST_0	Fast clock。CLK_FAST は CM7 と CPUSS fast infrastructure に使用されます。
CLK_FAST_1	Fast clock。CLK_FAST は CM7 と CPUSS fast infrastructure に使用されます。
CLK_GR	Group clock。CLK_GR は周辺機能へのクロック入力です。
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_MEM	Memory clock。CLK_MEM は CPUSS fast infrastructure を動作させます。
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_SLOW	Slow clock。CLK_FATST は CM0+と CPUSS slow infrastructure に使用されます。
クロック調整カウンタ	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。
CSV	Clock supervision
ECO	External cystal ocillator
EXT_CLK	External cock
FLL	周波数ロックループ
FPU	浮動小数点ユニット
ILO	Internal low-speed ocillators
IMO	Internal main ocillator
LIN	Local Interconnect Network。詳細は TRAVEO™ T2G **achitecture TRM** の「Local Interconnect Network (LIN)」を参照してください。
LPECO	Low-power external crystal oscillator
Peripheral clock divider	周辺クロック分周器は周辺機能を使用するためのクロックを動作させます。
PLL#0	Phase locked loop。 この PLL は SSCG と Fractional operation を搭載していません。
PLL#1	Phase locked loop。 この PLL は SSCG と Fractional operation を搭載していません。
PLL#2	Phase locked loop。 この PLL は SSCG と Fractional operation を搭載しています。
PLL#3	Phase locked loop。 この PLL は SSCG と Fractional operation を搭載しています。
SAR ADC	Successive approximation register analog-to-digital converter。詳細は TRAVEO™ T2G **achitecture TRM** の「SAR ADC」を参照してください。
SCB	Serial communications block。詳細は TRAVEO™ T2G **achitecture TRM** の「Serial communications block (SCB)」を参照してください。
SMIF	Serial memory interface。詳細は TRAVEO™ T2G **achitecture TRM** の「Serial memory interface」を参照してください。

用語集

用語	説明
TCPWM	Timer, counter, and pulse width modulator。詳細は TRAVEO™ T2G **achitecture TRM** の「Timer, counter, and PWM」を参照してください。
VIDEOSS	Video subsystem。詳細は TRAVEO™ T2G **achitecture TRM** の「Video subsystem」を参照してください。
WCO	Watch crystal oscillator

関連ドキュメント

以下は TRAVEO™ T2G ファミリの Datasheet および Technical reference manual です。これらドキュメントの入手については**テクニカルサポート**に連絡してください。

[1] Device datasheets

- CYT4DN datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family (Doc No. 002-24601)
- CYT3DL datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family (Doc No. 002-27763)

[2] Technical reference manuals

- Cluster 2D ファミリ
  - TRAVEO™ T2G automotive cluster 2D family architecture technical reference manual (TRM) (Doc No. 002-25800)
  - TRAVEO™ T2G automotive cluster 2D registers technical reference manual (TRM) for CYT4DN (Doc No. 002-25923)
  - TRAVEO™ T2G automotive cluster 2D registers technical reference manual (TRM) for CYT3DL (Doc No. 002-29854)

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
**	2020-09-18	このドキュメントは英語版 002-26071 Rev.**を翻訳した日本語版 002-31003 Rev.**です。
*A	2022-02-17	このドキュメントは英語版 002-26071 Rev.*B を翻訳した日本語版 002-31003 Rev.*A です。

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

Edition 2022-02-17
Published by
Infineon Technologies AG
81726 Munich, Germany

© 2022 Infineon Technologies AG.
All Rights Reserved.

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

Document reference
002-31003 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。