Clock configuration setup in TRAVEO™ T2G family CYT4D series MCUs

About this document

Scope and purpose

AN226071 describes how to set up the various clock sources in TRAVEO™ T2G family CYT4D series MCUs and provides examples including configuring PLL/FLL and calibrating the ILO.

Associated part family

TRAVEO™ T2G family CYT4D series automotive microcontrollers

Intended audience

This document is intended for users who use the clock configuration setup in TRAVEO™ T2G family CYT4D series MCUs.

Table of contents

Table of contents

1 Introduction

TRAVEO™ T2G family MCUs, targeted at automotive systems such as instrument clusters and head-up display (HUD), have a 2D graphics engine, sound processing, 32-bit automotive microcontrollers based on the Arm® Cortex®-M7 processor with FPU (single and dual precision), and manufactured on an advanced 40-nm process technology. These products enable a secure computing platform, and incorporate Infineon low-power flash memory along with multiple high-performance analog and digital functions.

The TRAVEO™ T2G clock system supports high-, and low-speed clocks using both internal and external clock sources. One of the typical use case for clock input is internal real-time clock (RTC). The TRAVEO™ T2G MCU supports phase-locked loop (PLL) and frequency-locked loop (FLL) to generate clocks that operate the internal circuit at a high speed.

The TRAVEO™ T2G MCU also supports the function to monitor clock operation and to measure the clock difference of each clock with reference to a known clock.

To know more on the functionality described and terminology used in this application note, see the “Clocking system” chapter in the architecture technical reference manual (TRM).

In this document, TRAVEO™ T2G family MCUs refer to CYT4D series.

2 Clock system for TRAVEO™ T2G family MCUs

2.1 Overview of the clock system

The clock system in this series of MCUs is divided into two blocks. One block selects the clock resources (such as external oscillation and internal oscillation) and multiplies the clock (using FLL and PLL). The other block distributes and divides clocks to the CPU cores and other peripheral functions. However, there are some exceptions where the RTC can connect directly to a clock resource.

Figure 1 shows the overview of the clock system structure.

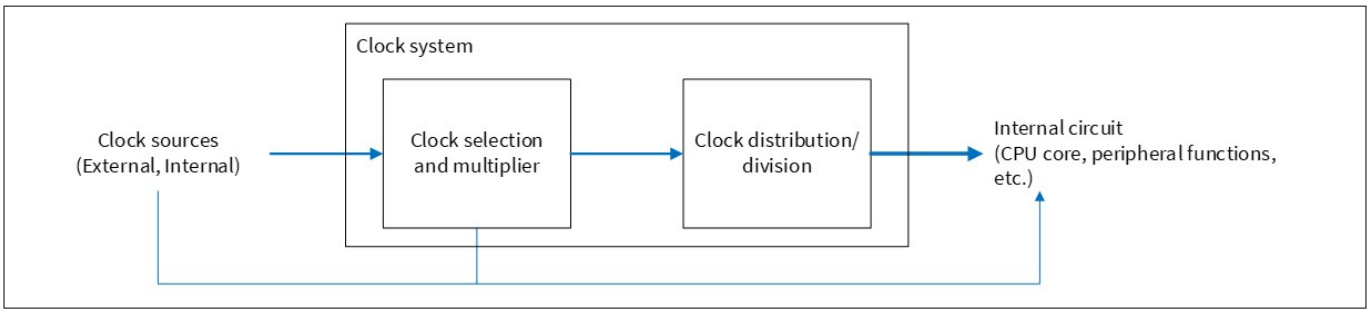

Figure 1 Overview of the clock system structure

2.2 Clock resources

The MCUs support two types of resource inputs: internal and external. Each of these internally support three types of clocks respectively.

- Internal clock sources (All these clocks are enabled by default):
  - Internal main oscillator (IMO): This is a built-in clock with a frequency of 8 MHz (TYP).
  - Internal low-speed oscillator 0 (ILO0): This is a built-in clock with a frequency of 32 kHz (TYP).
  - Internal low-speed oscillator 1 (ILO1): ILO1 has the same function as ILO0, but ILO1 can monitor the clock of ILO0.
- External clock sources (All these clocks are disabled by default):
  - External crystal oscillator (ECO): This clock uses an external oscillator whose input frequency range is between 3.988 MHz and 33.34 MHz.
  - Watch crystal oscillator (WCO): This also uses an external oscillator whose frequency is stable at 32.768 kHz, mainly used by the RTC module.
  - External clock (EXT_CLK): The EXT_CLK is a 0.25 MHz to 100 MHz range clock that can be sourced from a signal on a designed I/O pin. This clock can be used as the source clock for either PLL or FLL, or can be used directly by the high-frequency clocks.
  - Low-power external crystal oscillator (LPECO): This clock uses an external oscillator. The input frequency range is between 4 MHz and 8 MHz. The LPECO can be regarded as an ECO operating in low-power mode.

For more details on functions such as IMO, PLL, and so on, and numerical values such as frequency, see the TRAVEO™ T2G architecture TRM and the datasheet.

2.3 Clock system functions

Figure 2 shows the details of the clock selection and multiplier block. This block generates root frequency clocks CLK_HF0 to CLK_HF13 from the clock resources. This block has a capability to select one of the supported clock resources, FLL, and PLL to generate the required high-speed clocks. These MCUs support two

2 Clock system for TRAVEO™ T2G family MCUs

types of PLLs: PLL without spread spectrum clock generation (SSCG) and fractional operation (PLL200#x), and PLL with SSCG and fractional operation (PLL400#x).

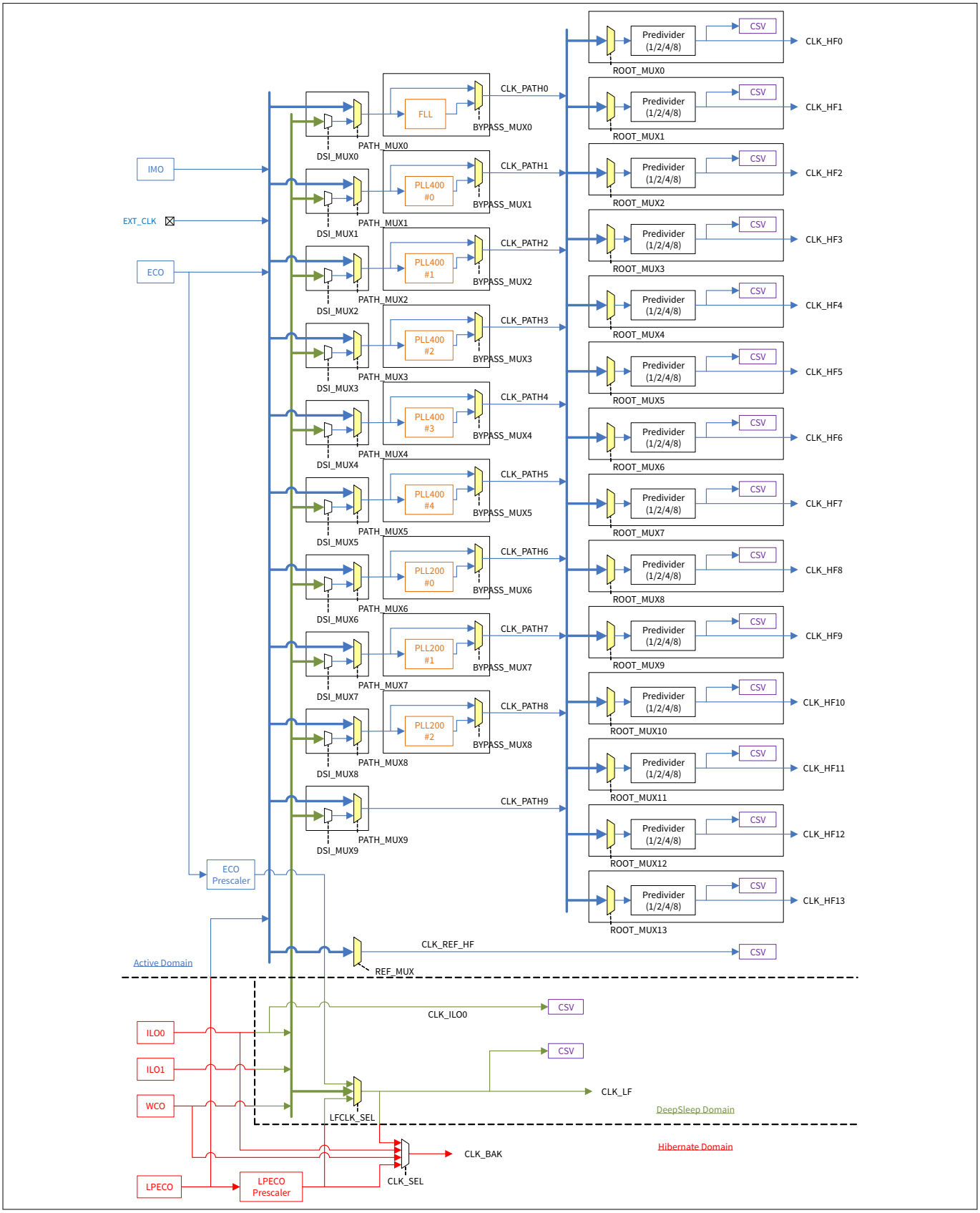

Figure 2 Block diagram

2 Clock system for TRAVEO™ T2G family MCUs

Active domain	Region of operation in only Active power mode
DeepSleep domain	Region of operation in only Active and DeepSleep modes
Hibernate domain	Region of operation in all power modes
ECO prescaler	Divides the ECO and creates a clock that can be used with the CLK_LF clock. The division function has a 10-bit integer divider and 8-bit fractional divider.
LPECO prescaler	Divides the LPECO and creates a clock that can be used with the CLK_BAK. The division function has a 10-bit integer divider and 8-bit fractional divider.
DSI_MUX	Selects a clock from ILO0, ILO1, and WCO
PATH_MUX	Selects a clock from IMO, ECO, EXT_CLK, LPECO, and DSI_MUX output
CLK_PATH	CLK_PATHx 0 through 9 are used as the input sources for high-frequency clocks.
CLK_HF	CLK_HFx 0 through 13 are recognized as high-frequency clocks.
FLL	Generates the high-frequency clock
PLL	Generates the high-frequency clock. There are two kinds of PLL: PLL200 and PLL400. PLL200 is with SSCG and fractional operation and PLL400 is with SSCG and fractional operation.
BYPASS_MUX	Selects the clock to be output to the CLK_PATH. In the case of FLL, the clock that can be selected is either FLL output or clock input to FLL.
ROOT_MUX	Selects the clock source of the CLK_HFx. The clocks that can be selected are CLK_PATHs 0 through 9.
Predivider	The predivider (divided by 1, 2, 4, or 8) is available to divide the selected CLK_PATH.
REF_MUX	Selects the CLK_REF_HF clock source
CLK_REF_HF	Used to monitor the CSV of the CLK_HF
LFCLK_SEL	Selects the CLK_LF clock source
CLK_LF	MCWDT source clock
CLK_SEL	Selects the clock to be input to the RTC
CLK_BAK	Mainly input to the RTC
CSV	Clock supervision to monitor the clock operation

Figure 3 shows the distribution of the CLK_HF0.

The CLK_HF0 is the root clock for the CPU subsystem (CPUSS) and peripheral clock dividers. For details on Figure 3, see the architecture TRM and datasheet.

2 Clock system for TRAVEO™ T2G family MCUs

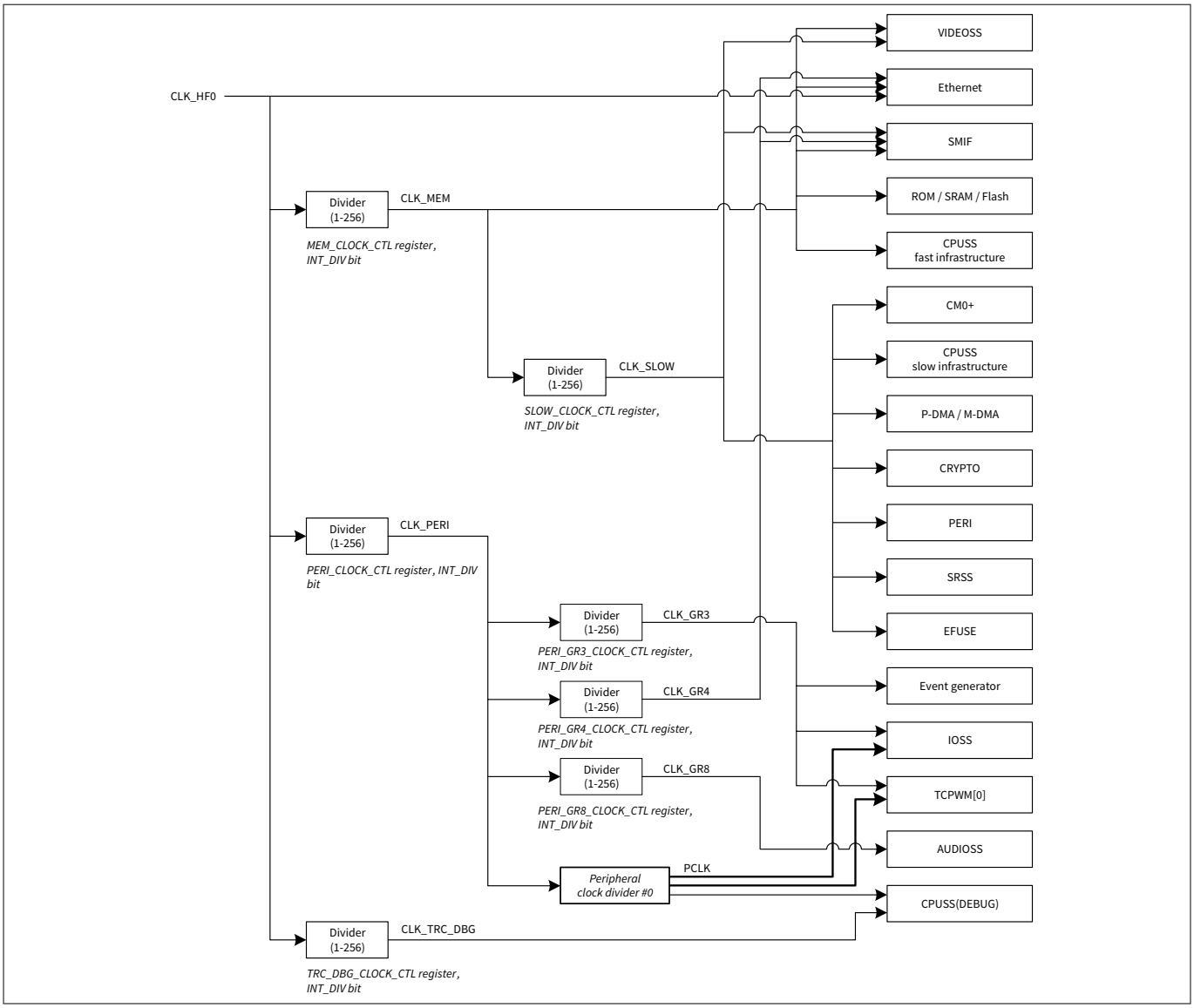

Figure 3 Block diagram for CLK_HF0

CLK_MEM	Clock input to the CPUSS of the fast infrastructure, Ethernet, and serial memory interface (SMIF)
CLK_PERI	Clock source for the CLK_GR and peripheral clock divider
CLK_SLOW	Clock input to the CPUSS of Cortex®-M0+ and slow infrastructure, SMIF, and VIDEOSS
CLK_GR	Clock input to peripheral functions. The CLK_GR is grouped by the clock gater. CLK_GR has six groups.
PCLK	Peripheral clock used in peripheral functions. The PCLK can be configured each channel of IPs independently and select one divider to generate the PCLK.
CLK_TRC_DBG	Clock input to the CPUSS (DEBUG).
Divider	Divider has a function to divide each clock. It can be configured from 1 division to 256 divisions.

Figure 4 shows details of the peripheral clock divider #0.

These MCUs need a clock to each peripheral unit (say, the serial communication block (SCB), the timer, counter, PWM (TCPWM), and so on) and its respective channel. These clocks are controlled by their respective dividers.

2 Clock system for TRAVEO™ T2G family MCUs

This peripheral clock divider #0 has many peripheral clock dividers to generate the peripheral clock (PCLK). See the datasheet for the number of dividers. The output of any of these dividers can be routed to any peripheral. Note that dividers already in use cannot be used for other peripherals or channels.

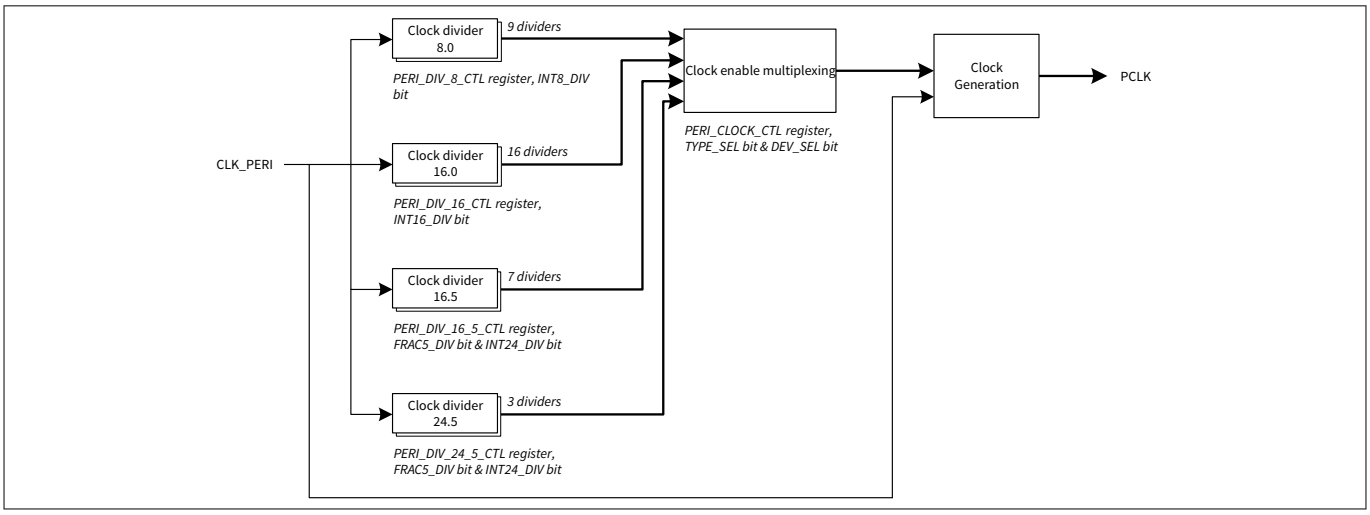

Figure 4 Block diagram for the peripheral clock divider #0

Clock divider8.0	Divides a clock by 8
Clock divider16.0	Divides a clock by 16
Clock divider16.5	Divides a clock by 16.5
Clock divider24.5	Divides a clock by 24.5
Clock enable multiplexing	Enables the signal output from the clock divider
Clock generator	Divides the CLK_PERI based on the clock divider

Figure 5 shows the distribution of the CLK_HF1.

The CLK_HF1 is a clock source of the CLK_FAST_0 and CLK_FAST_1. The clock distribution of the CLK_HF1 is shown in Figure 5. The CLK_FAST_0 and CLK_FAST_1 are the input sources for CM7_0 and CM7_1 respectively.

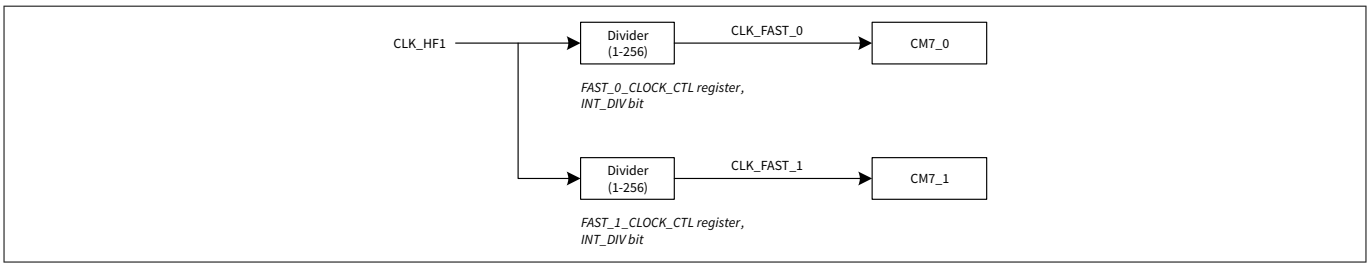

Figure 5 Block diagram for the CLK_HF1

Figure 6 shows the distribution of the CLK_HF2, which is a clock source for the CLK_GR and PCLK.

2 Clock system for TRAVEO™ T2G family MCUs

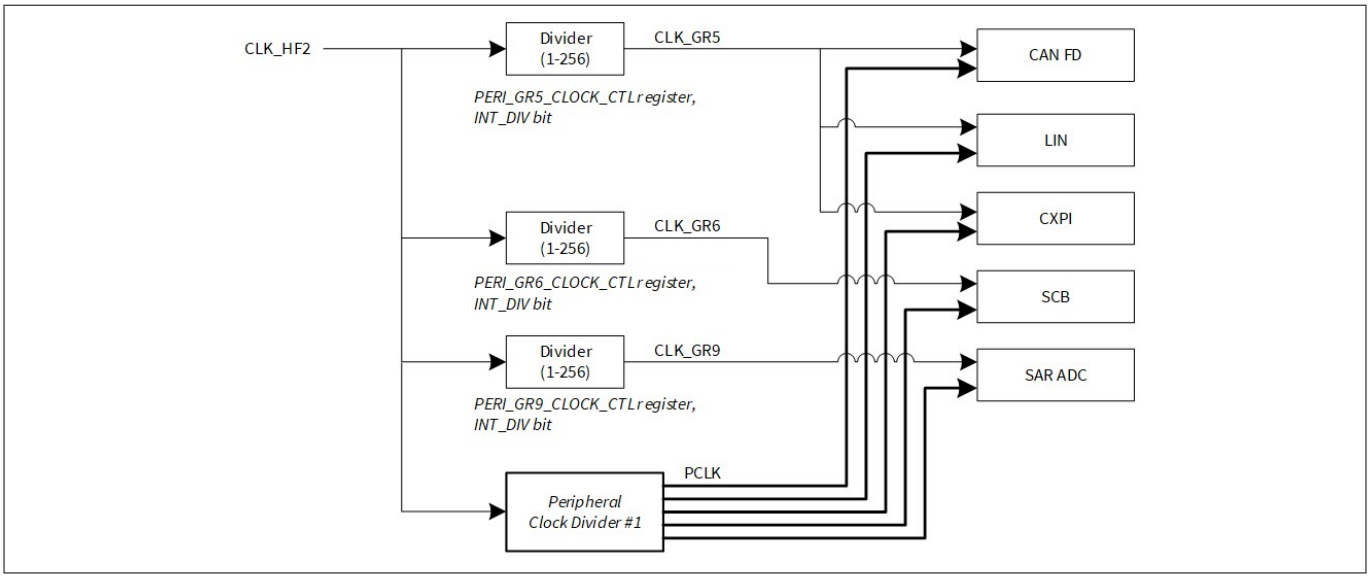

Figure 6 Block diagram for the CLK_HF2

Figure 7 shows details of the peripheral clock divider #1.

The peripheral clock divider #1 has many peripheral clock dividers to generate the PCLK. See the datasheet for the number of dividers. The output of any of these dividers can be routed to any peripheral. Note that the dividers already in use cannot be used for other peripherals or channels.

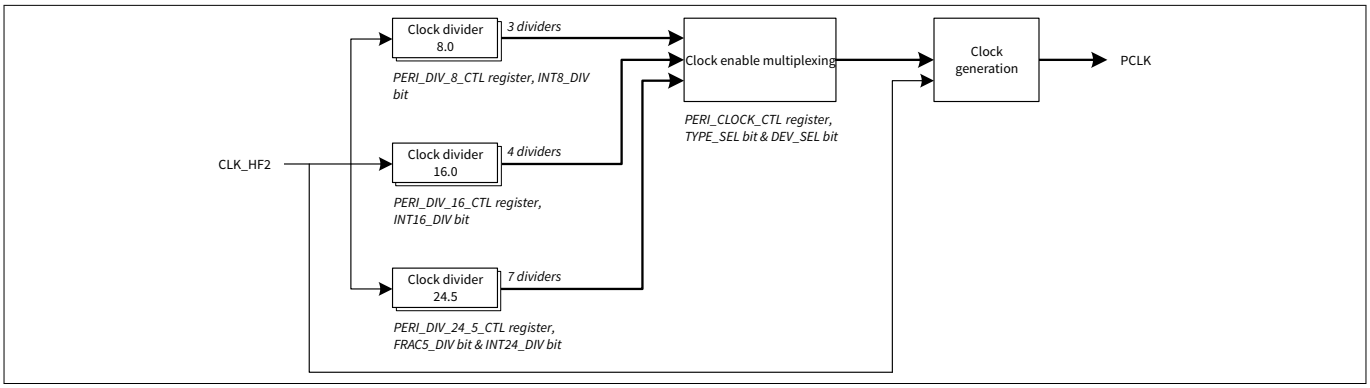

Figure 7 Block diagram for the peripheral clock divider #1

Clock divider8.0	Divides a clock by 8
Clock divider16.0	Divides a clock by 16
Clock divider24.5	Divides a clock by 24.5
Clock enable multiplexing	Enables the signal output from the clock divider
Clock generator	Divides the CLK_PERI based on the clock divider

Figure 8 shows the distribution of the CLK_HF3, CLK_HF4, CLK_HF5, CLK_HF6, CLK_HF7, CLK_HF8, CLK_HF9, CLK_HF10, CLK_HF11, and CLK_HF12. For details on these functions in Figure 8, see the architecture TRM.

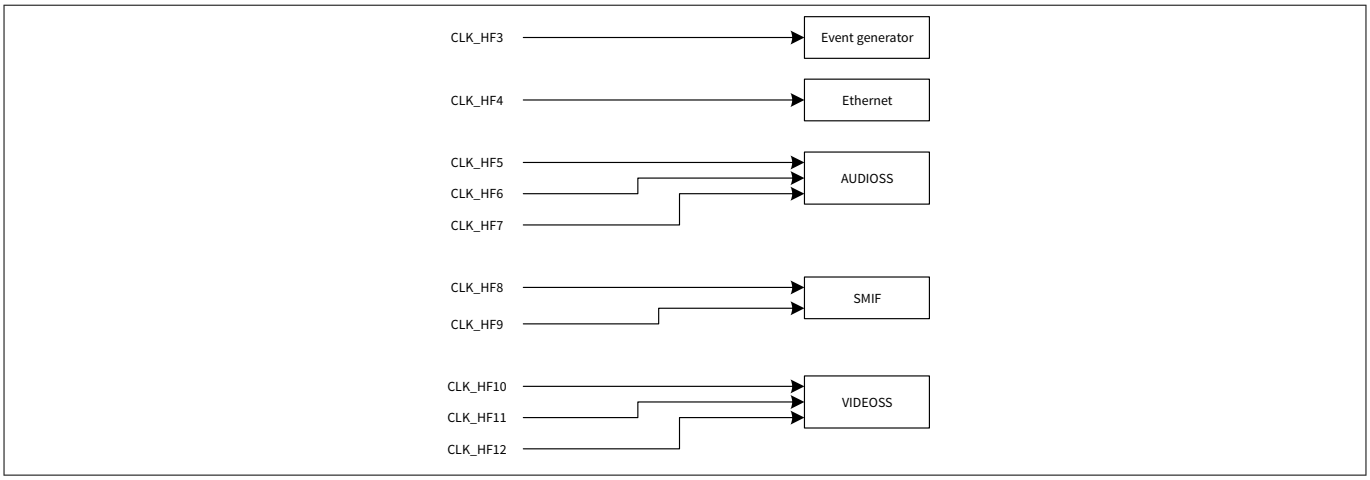

Figure 8 Block diagram for the CLK_HFx (x = 3 to 12)

The CLK_HF13 is dedicated to the CSV. See the architecture TRM for the CSV description.

2.4 Basic clock system settings

This section describes how to configure the clock system based on a use case using the sample driver library (SDL) provided by Infineon. The code snippets in this application note are part of the SDL. See Other references.

The SDL has a configuration part and a driver part. The configuration part configures the parameter values for the desired operation. The driver part configures each register based on the parameter values in the configuration part. You can configure the configuration part according to your system.

3 Configuration of the clock resources

3.1 Setting the ECO

The ECO is disabled by default and needs to be enabled for usage. Also, trimming is necessary to use the ECO. This device can configured with the trimming parameters that control the oscillator according to crystal unit and ceramic resonator. The method to determine the parameters differs between the crystal unit and ceramic resonator. See the Setting ECO parameters section in the TRAVEO™ T2G user guide for more information.

Figure 9 shows the ECO setting steps.

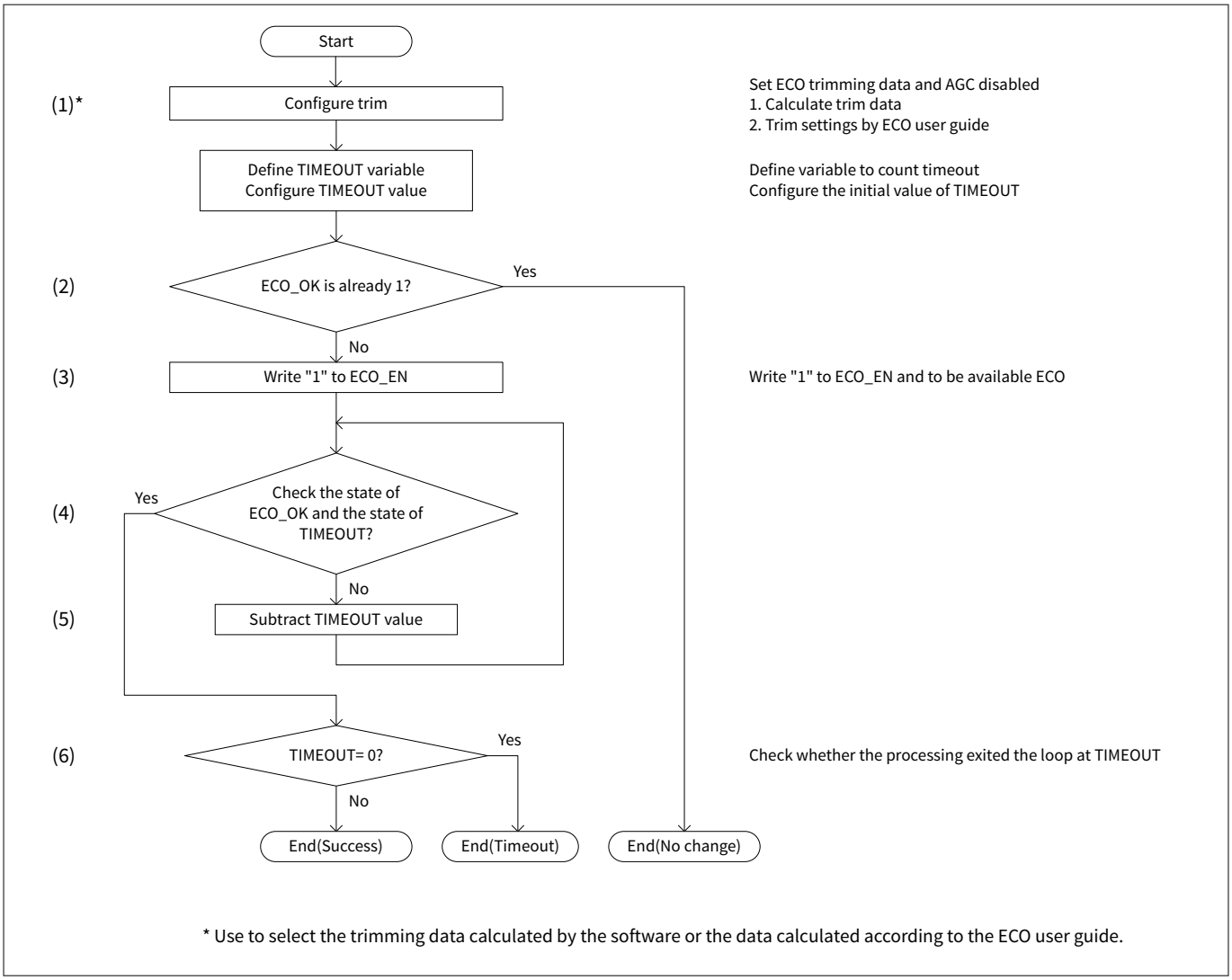

Figure 9 Enabling the ECO

3.1.1 Use case

- Oscillator to use: Crystal unit
- Fundamental frequency: 16 MHz
- Maximum drive level: 300.0 µW
- Equivalent series resistance: 150.0 ohm
- Shunt capacitance: 0.530 pF
- Parallel load capacitance: 8.000 pF

- Crystal unit vendor's recommended value of negative resistance: 1500 ohm
- Automatic gain control: OFF

***Note**: These values are decided in consultation with the Crystal unit vendor.*

3.1.2 Configuration

Table 1 lists the parameters and Table 2 lists the functions of the configuration part of in the SDL for ECO trim settings.

Table 1 List of ECO trim settings parameters

Parameters	Description	Value
CLK_ECO_CONFIG2.WDTRIM	Watchdog trim Calculated from **Setting ECO parameters in TRAVEO™ T2G user guide**	7ul
CLK_ECO_CONFIG2.ATRIM	Amplitude trim Calculated from **Setting ECO parameters in TRAVEO™ T2G user guide**	0ul
CLK_ECO_CONFIG2.FTRIM	Filter trim of 3rd harmonic oscillation Calculated from **Setting ECO parameters in TRAVEO™ T2G user guide**	3ul
CLK_ECO_CONFIG2.RTRIM	Feedback resistor trim Calculated from **Setting ECO parameters in TRAVEO™ T2G user guide**	3ul
CLK_ECO_CONFIG2.GTRIM	Startup time of the gain trim Calculated from **Setting ECO parameters in TRAVEO™ T2G user guide**	3ul
CLK_ECO_CONFIG.AGC_EN	Automatic gain control (AGC) disabled Calculated from **Setting ECO parameters in TRAVEO™ T2G user guide**	0ul [OFF]
WAIT_FOR_STABILIZATION	Waiting for stabilization	10000ul
PLL_400M_0_PATH_NO	PLL number for PLL_400M_0	1ul
PLL_400M_1_PATH_NO	PLL number for PLL_400M_1	2ul
PLL_200M_0_PATH_NO	PLL number for PLL_200M_0	3ul
PLL_200M_1_PATH_NO	PLL number for PLL_200M_1	4ul
CLK_FREQ_ECO	Source clock frequency	16000000ul

(table continues...)

Table 1 (continued) List of ECO trim settings parameters

Parameters	Description	Value
SUM_LOAD_SHUNT_CAP_IN_PF	Sum of load shunt capacity (pF)	17ul
ESR_IN_OHM	Equivalent series resistance (ESR) (ohm)	250ul
MAX_DRIVE_LEVEL_IN_UW	Maximum drive level (uW)	100ul
MIN_NEG_RESISTANCE	Minimum negative resistance	5 * ESR_IN_OHM

Table 2 List of ECO trim settings functions

Functions	Description	Value
`Cy_WDT_Disable()`	Disable the watchdog timer	–
`Cy_SysClk_FllDisableSequence(Wait Cycle)`	Disable the FLL	Wait cycle = WAIT_FOR_STABILIZATION
`Cy_SysClk_Pll400MDisable(PLL Number)`	Disable the PLL400M_0	PLL number = PLL_400M_0_PATH_NO
	Disable the PLL400M_1	PLL number = PLL_400M_1_PATH_NO
`Cy_SysClk_PllDisable(PLL Number)`	Disable the PLL200M_0	PLL number = PLL_200M_0_PATH_NO
	Disable the PLL200M_1	PLL number = PLL_200M_1_PATH_NO
`AllClockConfiguration()`	Clock configuration	–
`Cy_SysClk_EcoEnable(Timeout value)`	Set ECO enable and timeout value	Timeout value = WAIT_FOR_STABILIZATION
`Cy_SysLib_DelayUs(Wait Time)`	Delay by the specified number of microseconds	Wait time = 1u (1us)

3.1.3 Sample code for initial ECO configuration

General configuration of ECO settings provides a sample code.

The following description will help you understand the register notation of the driver part of the SDL:

- SRSS->unCLK_ECO_CONFIG.stcField.u1ECO_EN is the SRSS_CLK_ECO_CONFIG.ECO_EN mentioned in the Register TRM. Other registers are also described in the same manner.
- Performance improvement measures

3 Configuration of the clock resources

To improve the performance of register setting, the SDL writes a complete 32-bit data to the register. Each bit field is generated in advance in a bit-writable buffer and written to the register as the final 32-bit data.

```
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;
```

See `cyip_srss_v2.h` under `hdr/rev_x/ip` for more information on the union and structure representation of registers.

3 Configuration of the clock resources

General configuration of ECO settings

```
:
/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul) /** Define the TIMEOUT variable.**/
 :
#define CLK_FREQ_ECO       (16000000ul) /**Define the oscillator parameters to use for software
calculation.**/
 :
#define PLL_400M_0_PATH_NO   (1ul) /** Define the PLL number.**/
#define PLL_400M_1_PATH_NO   (2ul) /** Define the PLL number.**/
#define PLL_200M_0_PATH_NO   (3ul) /** Define the PLL number.**/
#define PLL_200M_1_PATH_NO   (4ul) /** Define the PLL number.**/
:
#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(); /** Watchdog timer disable **/
 :
    /* Disable Fll */
    CY_ASSERT(Cy_SysClk_FllDisableSequence(WAIT_FOR_STABILIZATION) == CY_SYSCLK_SUCCESS); /**
Disable the FLL.**/



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

    /* Enable interrupt */
    __enable_irq();

    /* Set Clock Configuring registers */
    AllClockConfiguration(); /** Trim and ECO setting. See AllClockConfiguration() function.**/
 :
    /* Please ensure output clock frequency using oscilloscope */
```

```
    for(;;);
}
```

AllClockConfiguration() function

```
static void AllClockConfiguration(void)
{
 :
    /*****  ECO setting ******/
        cy_en_sysclk_status_t ecoStatus; /** (1)-1. Trim settings for software calculation. See
Cy_SysClk_EcoConfigureWithMinRneg() function.**/
        ecoStatus = Cy_SysClk_EcoConfigureWithMinRneg( /** (1)-1. Trim settings for software
calculation. See Cy_SysClk_EcoConfigureWithMinRneg() function.**/
                            CLK_FREQ_ECO, /** (1)-1. Trim settings for software calculation. See
Cy_SysClk_EcoConfigureWithMinRneg() function.**/
                            SUM_LOAD_SHUNT_CAP_IN_PF, /** (1)-1. Trim settings for software
calculation. See Cy_SysClk_EcoConfigureWithMinRneg() function.**/
                            ESR_IN_OHM, /** (1)-1. Trim settings for software calculation. See
Cy_SysClk_EcoConfigureWithMinRneg() function.**/
                            MAX_DRIVE_LEVEL_IN_UW, /** (1)-1. Trim settings for software
calculation. See Cy_SysClk_EcoConfigureWithMinRneg() function.**/
                            MIN_NEG_RESISTANCE
                            ); /** (1)-1. Trim settings for software calculation. See Code
Listing 4.**/
        CY_ASSERT(ecoStatus == CY_SYSCLK_SUCCESS);
    {
     SRSS->unCLK_ECO_CONFIG2.stcField.u3WDTRIM = 7ul; /** (1)-2. Trim settings according to the
ECO user guide**/
     SRSS->unCLK_ECO_CONFIG2.stcField.u4ATRIM  = 0ul; /** (1)-2. Trim settings according to the
ECO user guide**/
     SRSS->unCLK_ECO_CONFIG2.stcField.u2FTRIM  = 3ul; /** (1)-2. Trim settings according to the
ECO user guide**/
     SRSS->unCLK_ECO_CONFIG2.stcField.u2RTRIM  = 3ul; /** (1)-2. Trim settings according to the
ECO user guide**/
     SRSS->unCLK_ECO_CONFIG2.stcField.u3GTRIM  = 0ul; /** (1)-2. Trim settings according to the
ECO user guide**/
     SRSS->unCLK_ECO_CONFIG.stcField.u1AGC_EN  = 0ul; /** (1)-2. Trim settings according to the
ECO user guide**/

        ecoStatus = Cy_SysClk_EcoEnable(WAIT_FOR_STABILIZATION); /** ECO enable. See
Cy_SysClk_EcoEnable() function.**/
        CY_ASSERT(ecoStatus == CY_SYSCLK_SUCCESS);
    }
 :
    return;
}
```

Either (1)-1 or (1)-2 can be used.

Comment out or delete unused code snippets in (1)-1 or (1)-2.

3 Configuration of the clock resources

Cy_SysClk_EcoEnable() function

```
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 */ /** (2) Check if
ECO_OK is already enabled**/

    {
        return CY_SYSCLK_INVALID_STATE;
    }

    /* first set ECO enable */
    SRSS->unCLK_ECO_CONFIG.stcField.u1ECO_EN = 1ul; /* 1 = enable */ /** (3) Write “1” to the
ECO_EN bit, and make ECO available.**/

    /* now do the timeout wait for ECO_STATUS, bit ECO_OK */
    for (;
         (SRSS->unCLK_ECO_STATUS.stcField.u1ECO_OK == 0ul) &&(timeoutus != 0ul); /** (4) Check
the state of ECO_OK and the state of TIMEOUT.**/
         timeoutus--) /** (5) Subtract the TIMEOUT value.**/
    {
        Cy_SysLib_DelayUs(1u); /** Wait for 1 us.**/
    }

    rtnval = ((timeoutus == 0ul) ? CY_SYSCLK_TIMEOUT : CY_SYSCLK_SUCCESS); /** (6) Check
whether processing exited the loop at TIMEOUT.**/
    return rtnval;
}
```

3 Configuration of the clock resources

Cy_SysClk_EcoConfigureWithMinRneg() function

```
cy_en_sysclk_status_t Cy_SysClk_EcoConfigureWithMinRneg(uint32_t freq, uint32_t cSum, uint32_t
esr, uint32_t driveLevel, uint32_t minRneg) /** Trim calculation by software **/
{
    /* 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); /** Get the Atrim value. See
Cy_SysClk_SelectEcoAtrim () function.**/
    if(atrim == CY_SYSCLK_INVALID_TRIM_VALUE)
    {
        return(CY_SYSCLK_BAD_PARAM);
    }

    agcen = Cy_SysClk_SelectEcoAGCEN(maxAmplitude); /** Get the AGC enable setting. See
Cy_SysClk_SelectEcoAGCEN() function.**/
    if(agcen == CY_SYSCLK_INVALID_TRIM_VALUE)
    {
        return(CY_SYSCLK_BAD_PARAM);
    }

    wdtrim = Cy_SysClk_SelectEcoWDtrim(maxAmplitude); /** Get the Wdtrim value. See
Cy_SysClk_SelectEcoWDtrim() function.**/
    if(wdtrim == CY_SYSCLK_INVALID_TRIM_VALUE)
    {
        return(CY_SYSCLK_BAD_PARAM);
    }


    gtrim  = Cy_SysClk_SelectEcoGtrim(gm_min); /** Get the Gtrim value. See
Cy_SysClk_SelectEcoGtrim() function.**/
    if(gtrim == CY_SYSCLK_INVALID_TRIM_VALUE)
    {
        return(CY_SYSCLK_BAD_PARAM);
    }

    rtrim  = Cy_SysClk_SelectEcoRtrim(freqMHz); /** Get then Rtrim value. See
```

3 Configuration of the clock resources

```
Cy_SysClk_SelectEcoRtrim() function.**/
    if(rtrim == CY_SYSCLK_INVALID_TRIM_VALUE)
    {
        return(CY_SYSCLK_BAD_PARAM);
    }

    ftrim  = Cy_SysClk_SelectEcoFtrim(atrim); /** Get the Ftrim value. See
Cy_SysClk_SelectEcoFtrim() function.**/

    /* 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);
}
```

3 Configuration of the clock resources

Cy_SysClk_SelectEcoAtrim () function

```
__STATIC_INLINE uint32_t Cy_SysClk_SelectEcoAtrim(float32_t maxAmplitude) /** Get the Atrim
value **/
{
    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)
    {
        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);
    }
}
```

Cy_SysClk_SelectEcoAGCEN() function

```
__STATIC_INLINE uint32_t Cy_SysClk_SelectEcoAGCEN(float32_t maxAmplitude) /** Get the AGC
enable setting.**/
{
    if((0.50f <= maxAmplitude) && (maxAmplitude < 1.10f))
    {
        return(0x01ul);
    }
    else if(1.10f <= maxAmplitude)
    {
        return(0x00ul);
    }
    else
    {
        return(CY_SYSCLK_INVALID_TRIM_VALUE);
    }
}
```

3 Configuration of the clock resources

Cy_SysClk_SelectEcoWDtrim() function

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

3 Configuration of the clock resources

Cy_SysClk_SelectEcoGtrim() function

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

Cy_SysClk_SelectEcoRtrim() function

```
__STATIC_INLINE uint32_t Cy_SysClk_SelectEcoRtrim(float32_t freqMHz) /** Get the Rtrim value.**/
{
    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);
    }
}
```

Cy_SysClk_SelectEcoFtrim() function

```
__STATIC_INLINE uint32_t Cy_SysClk_SelectEcoFtrim(uint32_t atrim) /** Get the Ftrim value.**/
{
    return(0x03ul);
}
```

3.2 Setting WCO

3.2.1 Operation overview

The WCO is disabled by default. Accordingly, the WCO cannot be used unless it is enabled. Figure 10 shows how to configure registers for enabling the WCO.

To disable the WCO, write '0' to the WCO_EN bit of the BACKUP_CTL register.

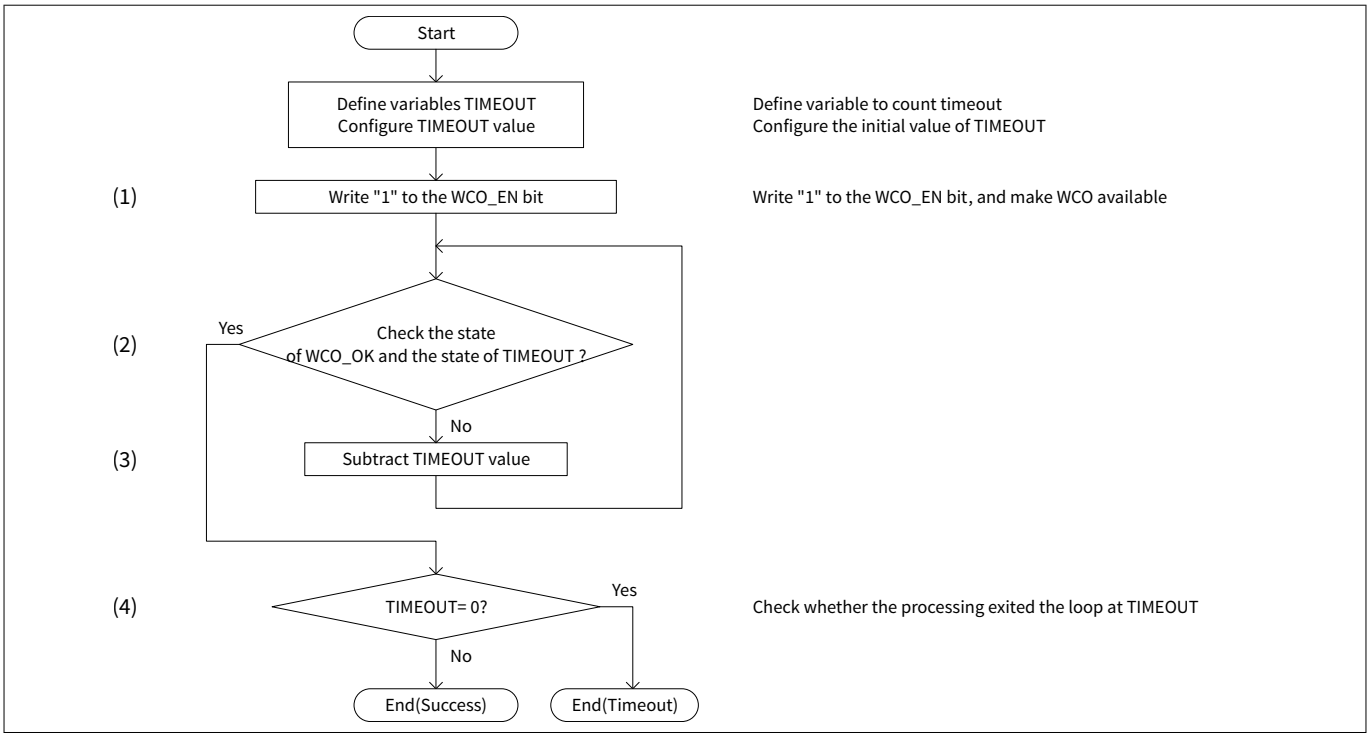

Figure 10 Enabling WCO

3.2.2 Configuration

Table 3 lists the parameters and Table 4 lists the functions of the configuration part of in the SDL for WCO settings.

Table 3 List of WCO settings parameters

Parameters	Description	Value
WAIT_FOR_STABILIZATION	Waiting for stabilization	10000ul
PLL_400M_0_PATH_NO	PLL number for PLL_400M_0	1ul
PLL_400M_1_PATH_NO	PLL number for PLL_400M_1	2ul
PLL_200M_0_PATH_NO	PLL number for PLL_200M_0	3ul
PLL_200M_1_PATH_NO	PLL number for PLL_200M_1	4ul

Table 4 List of WCO settings functions

Functions	Description	Value
`Cy_WDT_Disable()`	Disable the watchdog timer	–
`Cy_SysClk_FllDisableSequence(Wait Cycle)`	Disable FLL	Wait Cycle = WAIT_FOR_STABILIZATION
`Cy_SysClk_Pll400MDisable(PLL Number)`	Disable PLL400M_0	PLL number = PLL_400M_0_PATH_NO
	Disable PLL400M_1	PLL number = PLL_400M_1_PATH_NO

(table continues...)

Table 4 (continued) List of WCO settings functions

Functions	Description	Value
`Cy_SysClk_PllDisable(PLL Number)`	Disable the PLL200M_0	PLL number = PLL_200M_0_PATH_NO
	Disable the PLL200M_1	PLL number = PLL_200M_1_PATH_NO
`AllClockConfigurationw()`	Clock configuration	–
`Cy_SysClk_WcoEnable(Timeout value)`	Set the WCO enable and timeout value	Timeout value = WAIT_FOR_STABILIZATION
`Cy_SysLib_DelayUs(Wait Time)`	Delay by the specified number of microseconds	Wait time = 1u (1us)

3.2.3 Sample code for the initial configuration of WCO settings

See the below code listing for sample settings.

- General configuration of WCO settings
- AllClockConfiguration () function
- Cy_Sysclk_WcoEnable() function

3 Configuration of the clock resources

General configuration of WCO settings

```
:
/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul) /** Define the TIMEOUT variable.**/
:
#define PLL_400M_0_PATH_NO   (1ul) /** Define the PLL number.**/
#define PLL_400M_1_PATH_NO   (2ul) /** Define the PLL number.**/

#define PLL_200M_0_PATH_NO   (3ul) /** Define the PLL number.**/

#define PLL_200M_1_PATH_NO   (4ul) /** Define the PLL number.**/
:
static void AllClockConfiguration(void);
:
int main(void)
{
    /* disable watchdog timer */
    Cy_WDT_Disable(); /** Watchdog timer disable **/
:
    /* Disable Fll */
    CY_ASSERT(Cy_SysClk_FllDisableSequence(WAIT_FOR_STABILIZATION) == CY_SYSCLK_SUCCESS); /**
Disable the FLL **/

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

    /* Enable interrupt */
    __enable_irq();

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

AllClockConfiguration () function

```
static void AllClockConfiguration(void)
{
:
    /***** WCO setting ******/
    {
        cy_en_sysclk_status_t wcoStatus;
        wcoStatus = Cy_SysClk_WcoEnable(WAIT_FOR_STABILIZATION*10ul) /** WCO enable. See
Cy_Sysclk_WcoEnable() function.**/;

        CY_ASSERT(wcoStatus == CY_SYSCLK_SUCCESS);
    }

   return;
}
```

Cy_Sysclk_WcoEnable() function

```
:
__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; /** (1) Write “1” to the WCO_EN bit, and make the
WCO available.**/

    /* now do the timeout wait for STATUS, bit WCO_OK */
    for (; (Cy_SysClk_WcoOkay() == false) && (timeoutus != 0ul); timeoutus--) /** (2) Check the
state of WCO_OK and the state of TIMEOUT.**/ /** (3) Subtract the TIMEOUT value.**/

    {
        Cy_SysLib_DelayUs(1u); /** Wait for 1 us.**/
    }
    if (timeoutus != 0ul) /** (4) Check whether the processing exited the loop at TIMEOUT.**/
    {
        rtnval = CY_SYSCLK_SUCCESS;
    }

    return (rtnval);
}
```

3.3 Configuring IMO

By default, the IMO is enabled so that all functions work properly. The IMO will automatically be disabled during DeepSleep, Hibernate, and XRES modes. Therefore, it is not required to set the IMO explicitly.

3.4 Configuring ILO0/ILO1

The ILO0 is enabled by default.

Note that the ILO0 is used as the operating clock of the watchdog timer (WDT). Therefore, if the ILO0 is disabled, it is necessary to disable the WDT. To disable the ILO0, write '0b01' to the WDT_LOCK bit of the WDT_CTL register, and then write '0b00' to the ENABLE bit of the CLK_ILO0_CONFIG register.

The ILO1 is disabled by default. To enable the ILO1, write '0b01' to the ENABLE bit of the CLK_ILO1_CONFIG register.

3.5 Setting the LPECO

The LPCEO is disabled by default. The LPECO cannot be used unless it is enabled. Figure 11 shows how to configure registers for enabling the LPECO. To disable the LPECO, write '0' to the LPECO_EN bit of the BACKUP_LPECO_CTL register.

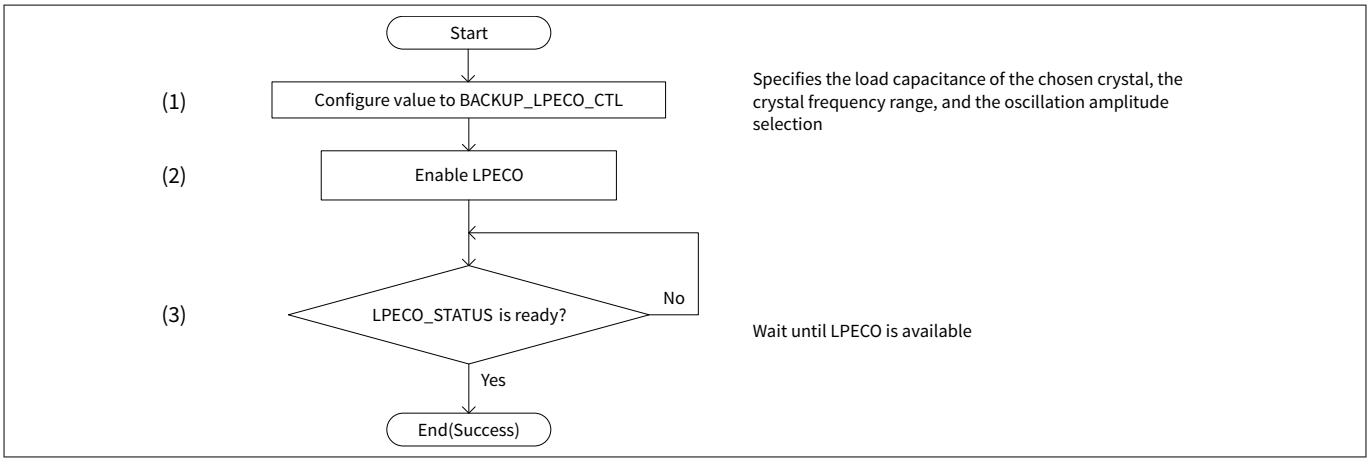

Figure 11 LPECO configuration

3.5.1 Use case

- Oscillator to use: Crystal unit
- Fundamental frequency: 8 MHz

***Note**: These values are decided in consultation with the crystal unit vendor.*

Table 5 lists the parameters and Table 6 lists the functions of the configuration part of in the SDL for LPECO settings.

Table 5 List of LPECO settings parameters

Parameters	Description	Value
WAIT_FOR_STABILIZATION	Waiting for stabilization	10000ul
CLK_FREQ_LPECO	Source clock frequency	8000000ul
CY_SYSCLK_BAK_LPECO_LCAP_5TO10PF	Backup domain LPECO load is in the range of 5 pF to 10 pF.	0ul
CY_SYSCLK_BAK_LPECO_FREQ_6TO8MHZ	Backup domain LPECO frequency is in the range of 6 MHz to 8 MHz.	1ul
CY_SYSCLK_BAK_LPECO_AMP_MAX_1P35V	Backup domain LPECO maximum oscillation amplitude is 1.35 V.	0ul

Table 6 List of LPECO settings functions

Functions	Description	Value
`Cy_WDT_Disable()`	Disable the watchdog timer.	–
`Cy_SysClk_ClkBak_LPECO_SetLoadCap(range)`	Set the load capacitance range for the LPECO crystal.	CY_SYSCLK_BAK_LPECO_LCAP_5TO10PF
`Cy_SysClk_ClkBak_LPECO_SetFrequency(range)`	Set the frequency range for the LPECO crystal.	CY_SYSCLK_BAK_LPECO_FREQ_6TO8MHZ
`Cy_SysClk_ClkBak_LPECO_SetAmplitude(value)`	Set the maximum oscillation amplitude value for the LPECO crystal.	CY_SYSCLK_BAK_LPECO_AMP_MAX_1P35V
`Cy_SysClk_ClkBak_LPECO_Enable()`	Enable the LPECO.	–
`Cy_SysClk_ClkBak_LPECO_Ready()`	Return the status of LPECO stabilization.	–

3.5.2 Sample code for the initial configuration of LPECO settings

See the below code listing for sample code.

- General configuration of LPECO settings
- AllClockConfiguration () function
- Cy_SysClk_ClkBak_LPECO_SetLoadCap () function
- Cy_SysClk_ClkBak_LPECO_SetFrequency () function
- Cy_SysClk_ClkBak_LPECO_SetAmplitude () function
- Cy_SysClk_ClkBak_LPECO_Enable () function
- Cy_SysClk_ClkBak_LPECO_Ready () function

3 Configuration of the clock resources

General configuration of LPECO settings

```
:
/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul) /** Define the TIMEOUT variable.**/
 :
#define CLK_FREQ_LPECO       (8000000ul) /** Define oscillator parameters to use for software
calculation.**/
 :
static void AllClockConfiguration(void);
:
int main(void)
{
    /* disable watchdog timer */
    Cy_WDT_Disable();
:
    /* Enable interrupt */
    __enable_irq();

    /* Set Clock Configuring registers */
    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(;;);
}
```

3 Configuration of the clock resources

AllClockConfiguration () function

```
static void AllClockConfiguration(void)
{
:
#ifdef LPECO_ENABLE
    /***** LPECO setting ******/
    {
        Cy_SysClk_ClkBak_LPECO_SetLoadCap(CY_SYSCLK_BAK_LPECO_LCAP_5TO10PF); /** (1) Configure
the value to BACKUP_LPECO_CTL. See Cy_SysClk_ClkBak_LPECO_SetLoadCap () function,
Cy_SysClk_ClkBak_LPECO_SetFrequency () function and Cy_SysClk_ClkBak_LPECO_SetAmplitude ()
function.**/
        Cy_SysClk_ClkBak_LPECO_SetFrequency(CY_SYSCLK_BAK_LPECO_FREQ_6TO8MHZ); /** (1)
Configure the value to BACKUP_LPECO_CTL. See Cy_SysClk_ClkBak_LPECO_SetLoadCap () function,
Cy_SysClk_ClkBak_LPECO_SetFrequency () function and Cy_SysClk_ClkBak_LPECO_SetAmplitude ()
function.**/
        Cy_SysClk_ClkBak_LPECO_SetAmplitude(CY_SYSCLK_BAK_LPECO_AMP_MAX_1P35V); /** (1)
Configure the value to BACKUP_LPECO_CTL. See Cy_SysClk_ClkBak_LPECO_SetLoadCap () function,
Cy_SysClk_ClkBak_LPECO_SetFrequency () function and Cy_SysClk_ClkBak_LPECO_SetAmplitude ()
function.**/

        Cy_SysClk_ClkBak_LPECO_Enable(); /** (2) Enable the LPECO. See
Cy_SysClk_ClkBak_LPECO_Enable () function.**/
        while(Cy_SysClk_ClkBak_LPECO_Ready() == false); /** (3) Wait until the LPECO is
available. See Cy_SysClk_ClkBak_LPECO_Ready () function.**/
    }
:
#endif
:
   return;
}
```

Cy_SysClk_ClkBak_LPECO_SetLoadCap () function

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

Cy_SysClk_ClkBak_LPECO_SetFrequency () function

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

3 Configuration of the clock resources

Cy_SysClk_ClkBak_LPECO_SetAmplitude () function

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

Cy_SysClk_ClkBak_LPECO_Enable () function

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

Cy_SysClk_ClkBak_LPECO_Ready () function

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

4 Configuration of the FLL and PLL

This section describes the configuring of the FLL and PLL in the clock system.

4.1 Setting FLL

4.1.1 Operation overview

The FLL must be set before using it. The FLL has a current-controlled oscillator (CCO); the output frequency of this CCO is controlled by adjusting the trim of the CCO. Figure 12 shows the steps to configure the FLL.

Figure 12 Procedure for setting the FLL

For details of FLL and FLL setting registers, see the architecture TRM and register TRM.

4.1.2 Use case

- Input clock frequency: 16 MHz
- Output clock frequency: 100 MHz

4.1.3 Configuration

Table 7 lists the parameters and Table 8 lists the functions of the configuration part of in the SDL for FLL settings.

Table 7 List of FLL settings parameters

Parameters	Description	Value
WAIT_FOR_STABILIZATION	Waiting for stabilization	10000ul
FLL_PATH_NO	FLL number	0u
FLL_TARGET_FREQ	FLL target frequency	100000000ul (100 MHz)
CLK_FREQ_ECO	Source clock frequency	16000000ul (16 MHz)
PATH_SOURCE_CLOCK_FREQ	Source clock frequency	CLK_FREQ_ECO
CY_SYSCLK_FLLPLL_OUTPUT_AUTO	FLL output mode CY_SYSCLK_FLLPLL_OUTPUT_AUTO: Automatic using the lock indicator. CY_SYSCLK_FLLPLL_OUTPUT_LOCKED_OR_NOTHING: Similar to AUTO, except that the clock is gated off when unlocked. CY_SYSCLK_FLLPLL_OUTPUT_INPUT: Select FLL reference input (bypass mode) CY_SYSCLK_FLLPLL_OUTPUT_OUTPUT: Select the FLL output. Ignores the lock indicator. See SRSS_CLK_FLL_CONFIG3 in the register TRM for details.	0ul

Table 8 List of FLL settings functions

Functions	Description	Value
`AllClockConfiguration()`	Clock configuration	–
`Cy_SysClk_FllConfigureStandard(inputFreq, outputFreq, outputMode)`	inputFreq: Input frequency	inputFreq = PATH_SOURCE_CLOCK_FREQ,
	outputFreq: Output frequency	outputFreq = FLL_TARGET_FREQ,
	outputMode: FLL output mode	outputMode = CY_SYSCLK_FLLPLL_OUTPUT_AUTO
`Cy_SysClk_FllEnable(Timeout value)`	Set FLL enable and timeout value	Timeout value = WAIT_FOR_STABILIZATION
`Cy_SysLib_DelayUs(Wait Time)`	Delay by the specified number of microseconds	Wait time = 1u (1us)

4.1.4 Sample code for the initial configuration of FLL settings

See the below code listing for sample code.

- General configuration of FLL settings
- AllClockConfiguration() function
- Cy_SysClk_FllConfigureStandard() function
- Cy_SysClk_FllManualConfigure() function
- Cy_SysClk_FllEnable() function

General configuration of FLL settings

```
/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul) /** Define the TIMEOUT variable.**/
:
#define FLL_TARGET_FREQ  (100000000ul) /** Define the FLL target frequency.**/
#define CLK_FREQ_ECO     (16000000ul) /** Define the FLL input frequency.**/
#define PATH_SOURCE_CLOCK_FREQ CLK_FREQ_ECO
:
#define FLL_PATH_NO        /** Define the FLL number.**/
:

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

4 Configuration of the FLL and PLL

AllClockConfiguration() function

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

        fllStatus = Cy_SysClk_FllEnable(WAIT_FOR_STABILIZATION); /** FLL enable. See
Cy_SysClk_FllEnable() function.**/
        CY_ASSERT((fllStatus == CY_SYSCLK_SUCCESS) || (fllStatus == CY_SYSCLK_TIMEOUT));
:
    }
   return;
}
```

4 Configuration of the FLL and PLL

Cy_SysClk_FllConfigureStandard() function

```
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 */ /** (1) Check if the FLL is already enabled.**/
    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 */ /** Check the FLL output range.**/
    {
        return(CY_SYSCLK_INVALID_STATE);
    }
    else if (((float32_t)outputFreq / (float32_t)inputFreq) < 2.2f) /* check output/input
frequency ratio */ /** Check the FLL 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. */ /** FLL parameter
calculation.**/
    config.enableOutputDiv = false;

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

    /* 3. Compute the CCO range value from the CCO frequency */
    if(ccoFreq >= CY_SYSCLK_FLL_CCO_BOUNDARY4_FREQ)
    {
        config.ccoRange = CY_SYSCLK_FLL_CCO_RANGE4;
    }
    else if(ccoFreq >= CY_SYSCLK_FLL_CCO_BOUNDARY3_FREQ)
```

```
    {
        config.ccoRange = CY_SYSCLK_FLL_CCO_RANGE3;
    }
    else if(ccoFreq >= CY_SYSCLK_FLL_CCO_BOUNDARY2_FREQ)
    {
        config.ccoRange = CY_SYSCLK_FLL_CCO_RANGE2;
    }
    else if(ccoFreq >= CY_SYSCLK_FLL_CCO_BOUNDARY1_FREQ)
    {
        config.ccoRange = CY_SYSCLK_FLL_CCO_RANGE1;
    }
    else
    {
        config.ccoRange = CY_SYSCLK_FLL_CCO_RANGE0;
    }

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

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

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

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

4 Configuration of the FLL and PLL

```
                break;
            }
        }

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

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


        /* configure FLL based on calculated values */
        cy_en_sysclk_status_t returnStatus;
        returnStatus = Cy_SysClk_FllManualConfigure(&config); /** Set FLL registers. See
    Cy_SysClk_FllManualConfigure() function.**/

        return (returnStatus);
    }
```

4 Configuration of the FLL and PLL

Cy_SysClk_FllManualConfigure() function

```
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 */ /** (1) Check if the FLL is already enabled.**/
    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 */ /** (2) FLL configuration **/
    {
        /* 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; /** Set the
CLK_FLL_CONFIG register.**/
        tempConfg.stcField.u18FLL_MULT      = config->fllMult; /** Set the CLK_FLL_CONFIG
register.**/
        tempConfg.stcField.u1FLL_OUTPUT_DIV = (uint32_t)(config->enableOutputDiv); /** Set the
CLK_FLL_CONFIG register.**/
        SRSS->unCLK_FLL_CONFIG.u32Register  = tempConfg.u32Register; /** Set the CLK_FLL_CONFIG
register.**/

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

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

4 Configuration of the FLL and PLL

```
CLK_FLL_CONFIG3 register.**/
        SRSS->unCLK_FLL_CONFIG3.u32Register   = tempConfg3.u32Register; /** Set the
CLK_FLL_CONFIG3 register.**/

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

    return (returnStatus);
}
```

4 Configuration of the FLL and PLL

Cy_SysClk_FllEnable() function

```
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; /** (3) Enable the CCO.**/

    /* Wait until CCO is ready */
    while(SRSS->unCLK_FLL_STATUS.stcField.u1CCO_READY == 0ul) /** (4) Wait until the CCO is
available.**/
    {
        if(timeoutus == 0ul) /** (5) Check the timeout.**/
        {
            /* If cco ready doesn't occur, FLL is stopped. */
            Cy_SysClk_FllDisable(); /** FLL disabled if a timeout occurs.**/
            return(CY_SYSCLK_TIMEOUT);
        }
        Cy_SysLib_DelayUs(1u); /** Wait for 1 us.**/
        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; /** (6) Enable the FLL.**/

    /* now do the timeout wait for FLL_STATUS, bit LOCKED */
    while(SRSS->unCLK_FLL_STATUS.stcField.u1LOCKED == 0ul) /** (7) Wait until the FLL is
locked.**/
    {
        if(timeoutus == 0ul) /** (8) Check the timeout.**/
        {
            /* If lock doesn't occur, FLL is stopped. */
            Cy_SysClk_FllDisable();
            return(CY_SYSCLK_TIMEOUT); /** FLL disabled if timeout occurs.**/
        }
        Cy_SysLib_DelayUs(1u); /** Wait for 1 us.**/
        timeoutus--;
    }

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

    return(CY_SYSCLK_SUCCESS);
}
```

4.2 Setting PLL

The PLL must be configured before using it. Figure 13 shows the steps to configure PLL400 and PLL200. For details on PLL400 and PLL200, see the architecture TRM.

4 Configuration of the FLL and PLL

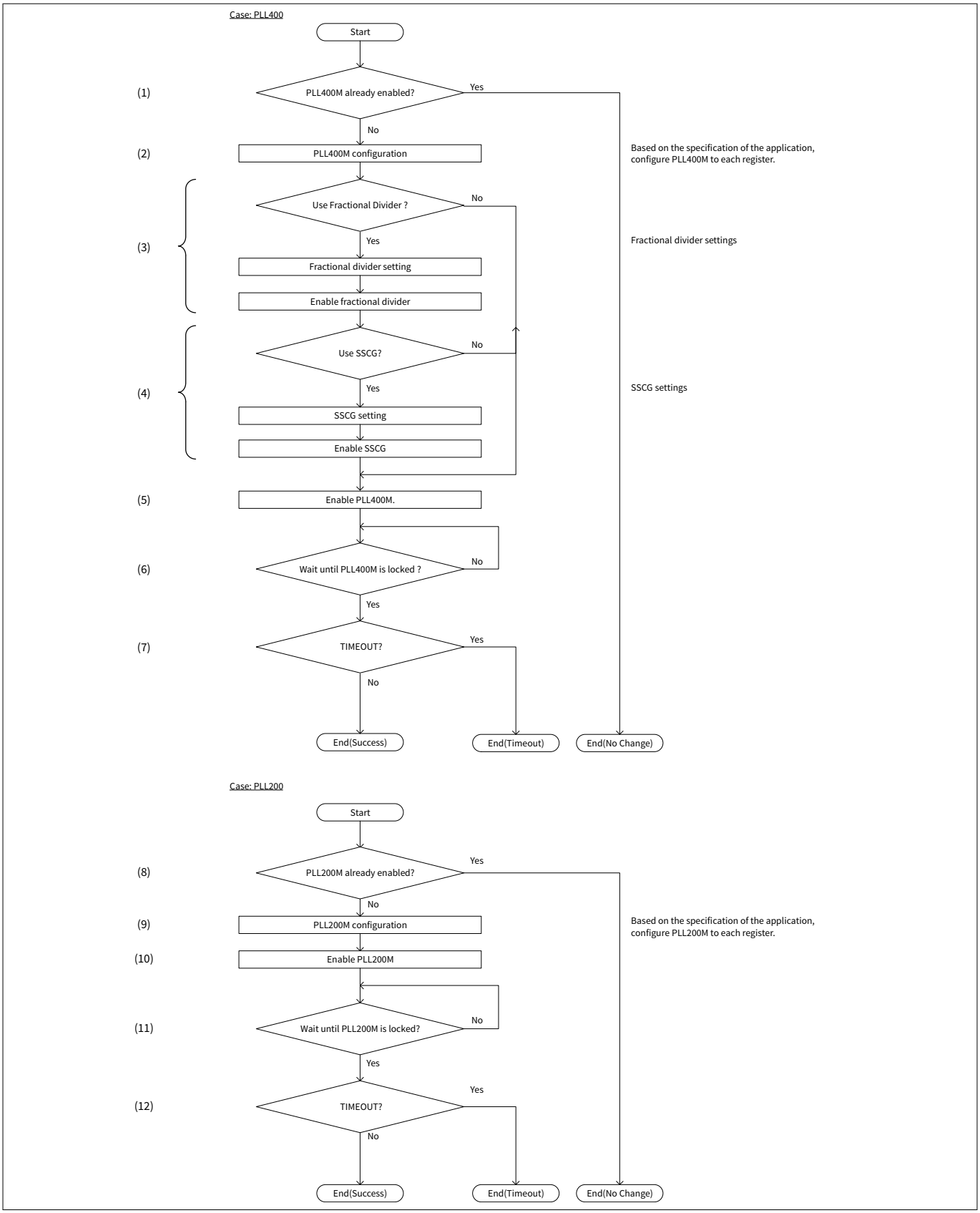

Figure 13 Procedure for configuring the PLL

4.2.1 Use case

- Input clock frequency: 16.000 MHz
- Output clock frequency:
  250.000 MHz (PLL400 #0)
  196.608 MHz (PLL400 #1)
  160.000 MHz (PLL200 #0)
  80.000 MHz (PLL200 #1)
- Fractional divider:
  Disable (PLL400 #0)
  Enable (PLL400 #1)
- SSCG:
  Enable (PLL400 #0)
  Disable (PLL400 #1)
- SSCG dithering:
  Enable (PLL400 #0)
  Disable (PLL400 #1)
- SSCG modulation depth: -2.0% (PLL400)
- SSCG modulation rate: Divide 512 (PLL400)
- LF mode: 200 MHz to 400 MHz (PLL200)

4.2.2 Configuration

Table 9 and Table 11 list parameters of the PLL (400/200); Table 10 and Table 12 list functions of the PLL (400/200) of the configuration part of in the SDL for PLL (400/200) settings.

Table 9 List of PLL 400 settings parameters

Parameters	Description	Value
PLL400_0_TARGET_FREQ	PLL400 #0 target frequency	250 MHz (250000000ul)
PLL400_1_TARGET_FREQ	PLL400 #1 target frequency	196.608 MHz (196608000ul)
WAIT_FOR_STABILIZATION	Waiting for stabilization	10000ul
PLL400_0_PATH_NO	PLL400 #0 number	1u
PLL400_1_PATH_NO	PLL400 #1 number	2u
CLK_FREQ_ECO	ECO clock frequency	16000000ul (16 MHz)
PATH_SOURCE_CLOCK_FREQ	PATH source clock frequency	CLK_FREQ_ECO

(table continues...)

Table 9 (continued) List of PLL 400 settings parameters

Parameters	Description	Value
CY_SYSCLK_FLLPLL_OUTPUT_AUTO	FLL output mode CY_SYSCLK_FLLPLL_OUTPUT_AUTO: Automatic using the lock indicator CY_SYSCLK_FLLPLL_OUTPUT_LOCKED_OR_NOTHING: Similar to AUTO, except that the clock is gated off when unlocked CY_SYSCLK_FLLPLL_OUTPUT_INPUT: Selects the FLL reference input (bypass mode) CY_SYSCLK_FLLPLL_OUTPUT_OUTPUT: Selects the FLL output. Ignores then lock indicator. See SRSS_CLK_FLL_CONFIG3 in the register TRM for details.	0ul
pllConfig.inputFreq	Input PLL frequency	PATH_SOURCE_CLOCK_FREQ
pllConfig.outputFreq	Output PLL frequency (PLL400 #0)	PLL400_0_TARGET_FREQ
	Output PLL frequency (PLL400 #1)	PLL400_1_TARGET_FREQ
pllConfig.outputMode	Output mode; 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	Enable the fractional divider (PLL400 #0)	false
	Enable the fractional divider (PLL400 #1)	true
pllConfig.fracDitherEn	Enable dithering operation (PLL400 #0)	false

(table continues...)

Table 9 (continued) List of PLL 400 settings parameters

Parameters	Description	Value
	Enable dithering operation (PLL400 #1)	true
pllConfig.sscgEn	Enable the SSCG (PLL400 #0)	true
	Enable the SSCG (PLL400 #1)	false
pllConfig.sscgDitherEn	Enable SSCG dithering operation (PLL400 #0)	true
	Enable SSCG dithering operation (PLL400 #1)	false
pllConfig.sscgDepth	Set the SSCG modulation depth	CY_SYSCLK_SSCG_DEPTH_MINUS_2_0
pllConfig.sscgRate	Set the SSCG modulation rate	CY_SYSCLK_SSCG_RATE_DIV_512
manualConfig.feedbackDiv	Control bits for the feedback divider	p (Calculated value)
manualConfig.referenceDiv	Control bits for the reference divider	q (Calculated value)
manualConfig.outputDiv	Control bits for the output divider: 0: Illegal (undefined behavior) 1: Illegal (undefined behavior) 2: Divide by 2. Suitable for direct usage as the HFCLK source. ... 16: Divide by 16. Suitable for direct usage as the HFCLK source. >16: Illegal (undefined behavior)	out (Calculated value)
manualConfig.lfMode	VCO frequency range selection: 0: VCO frequency is [200 MHz, 400 MHz] 1: VCO frequency is [170 MHz, 200 MHz)	config->lfMode (Calculated value)
manualConfig.outputMode	Bypass mux located just after PLL output: 0: AUTO 1: LOCKED_OR_NOTHING 2: PLL_REF 3: PLL_OUT	config->outputMode (Calculated value)

Table 10 List of PLL 400 settings functions

Functions	Description	Value
`AllClockConfiguration()`	Clock configuration	–

(table continues...)

Table 10 (continued) List of PLL 400 settings functions

Functions	Description	Value
`Cy_SysClk_Pll400M Configure(PLL Number,PLL Configure)`	Set the PLL path number and configure the PLL (PLL400 #0).	PLL number = PLL400_0_PATH_NO,
		PLL configure = g_pll400_0_Config
	Set the PLL path number and configure the PLL (PLL400 #1).	PLL number = PLL400_1_PATH_NO,
		PLL configure = g_pll400_1_Config
`Cy_SysClk_Pll400M Enable(PLL Number, Timeout value)`	Set the PLL path number and monitor the PLL configuration (PLL400 #0).	PLL number = PLL400_0_PATH_NO,
		Timeout value = WAIT_FOR_STABILIZATION
	Set the PLL path number and monitor the PLL configuration (PLL400 #1).	PLL number = PLL400_1_PATH_NO,
		Timeout value = WAIT_FOR_STABILIZATION
`Cy_SysLib_DelayUs(Wait Time)`	Delays by the specified number of microseconds.	Wait time = 1u (1us)
`Cy_SysClk_PllManualConfigure(PLL Number, PLL Manual Configure)`	Set the PLL path number and manually configure the PLL (PLL400 #0).	PLL number = PLL400_0_PATH_NO,
		PLL manual configure = manualConfig
	Set the PLL path number and manually configure the PLL (PLL400 #1).	PLL number = PLL400_1_PATH_NO,
		PLL manual configure = manualConfig
`Cy_SysClk_GetPll400MNo (Clkpath, PllNo)`	Return the PLL number according to the input PATH number (PLL400 #0).	Clkpath = 1u
		PllNo = 0u
	Return the PLL number according to the input PATH number (PLL400 #1).	Clkpath = 2u
		PllNo = 1u
`Cy_SysClk_PllCaluc Dividers()`	Calculate the appropriate divider settings according to the PLL input/ output frequency.	
`Cy_SysClk_Pll400M Enable(PLL Number,Timeout value)`	Set the PLL path number and monitor the PLL configuration (PLL400 #0).	PLL number = PLL400_0_PATH_NO,
		Timeout value = WAIT_FOR_STABILIZATION
	Set the PLL path number and monitor the PLL configuration (PLL400 #1).	PLL number = PLL400_1_PATH_NO,
		Timeout value = WAIT_FOR_STABILIZATION

Table 11 List of PLL 200 settings parameters

Parameters	Description	Value
PLL200_0_TARGET_FREQ	PLL200 #0 target frequency	160 MHz (160000000ul)
PLL200_1_TARGET_FREQ	PLL200 #1 target frequency	80 MHz (80000000ul)

(table continues...)

Table 11 (continued) List of PLL 200 settings parameters

Parameters	Description	Value
WAIT_FOR_STABILIZATION	Waiting for stabilization	10000ul
PLL200_0_PATH_NO	PLL200 #0 number	3u
PLL200_1_PATH_NO	PLL200 #1 number	4u
PATH_SOURCE_CLOCK_FREQ	PATH source clock frequency	16000000ul (16 MHz)
pllConfig.inputFreq	Input PLL frequency	PATH_SOURCE_CLOCK_FREQ
pllConfig.outputFreq	Output PLL frequency (PLL200 #0)	PLL200_0_TARGET_FREQ
	Output PLL frequency (PLL200 #1)	PLL200_1_TARGET_FREQ
pllConfig.lfMode	PLL LF mode: 0: VCO frequency is [200 MHz, 400 MHz] 1: VCO frequency is [170 MHz, 200 MHz]	0u (VCO frequency is 320 MHz)
pllConfig.outputMode	Output mode: 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	Control bits for the feedback divider	p (Calculated value)
manualConfig.referenceDiv	Control bits for the reference divider	q (Calculated value)
manualConfig.outputDiv	Control bits for the output divider: 0: Illegal (undefined behavior) 1: Illegal (undefined behavior) 2: Divide by 2. Suitable for direct usage as the HFCLK source. ... 16: Divide by 16. Suitable for direct usage as the HFCLK source. >16: Illegal (undefined behavior)	out (Calculated value)

(table continues...)

Table 11 (continued) List of PLL 200 settings parameters

Parameters	Description	Value
manualConfig.lfMode	VCO frequency range selection: 0: VCO frequency is [200 MHz, 400 MHz] 1: VCO frequency is [170 MHz, 200 MHz)	config->lfMode (Calculated value)
manualConfig.outputMode	Bypass mux located just after PLL output: 0: AUTO 1: LOCKED_OR_NOTHING 2: PLL_REF 3: PLL_OUT	config->outputMode (Calculated value)
manualConfig.fracDiv	Fractional divider value	config->fracDiv (Calculated value)

Table 12 List of PLL 200 settings functions

Functions	Description	Value
`AllClockConfiguration()`	Clock configuration	–
`Cy_SysClk_PllConfigure (PLL Number,PLL Configure)`	Set the PLL path number and configure the PLL (PLL200 #0).	PLL number = PLL200_0_PATH_NO, PLL configure = g_pll200_0_Config
	Set the PLL path number and configure the PLL (PLL200 #1).	PLL number = PLL200_1_PATH_NO, PLL configure = g_pll200_1_Config
`Cy_SysLib_DelayUs(Wait Time)`	Delay by the specified number of microseconds.	Wait time = 1u (1us)
`Cy_SysClk_PllManual Configure(PLL Number, PLL Manual Configure)`	Set the PLL path number and manually configure the PLL (PLL200 #0).	PLL number = PLL200_0_PATH_NO, PLL manual configure = manualConfig
	Set the PLL path number and manually configure the PLL (PLL200 #1).	PLL number = PLL200_1_PATH_NO, PLL manual configure = manualConfig
`Cy_SysClk_GetPllNo (Clkpath, PllNo)`	Return the PLL number according to the input PATH number (PLL200 #0).	Clkpath = 3u PllNo = 0u
	Return the PLL number according to the input PATH number (PLL200 #1).	Clkpath = 4u PllNo = 1u
`Cy_SysClk_PllCaluc Dividers(InputFreq, OutputFreq,PLLlimit,FracBitNum,RefDiv,OutputDiv,FeedBackFracDiv)`	Calculate the appropriate divider settings according to the PLL input/output frequency.	InputFreq = PATH_SOURCE_CLOCK_ FREQ

(table continues...)

Table 12 (continued) List of PLL 200 settings functions

Functions	Description	Value
		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)`	Set the PLL path number and monitor the PLL configuration (PLL200 #0).	PLL number = PLL200_0_PATH_NO,
		Timeout value = WAIT_FOR_STABILIZATION
	Set the PLL path number and monitor the PLL configuration (PLL200 #1).	PLL number = PLL200_1_PATH_NO,
		Timeout value = WAIT_FOR_STABILIZATION

4.2.3 Sample code for the initial PLL configuration

See the below Code Listings to show the sample code for the PLL400 #0 example.

- General configuration of PLL 400 #0 settings
- AllClockConfiguration() function
- Cy_SysClk_Pll400MConfigure() function
- Cy_SysClk_Pll400MManualConfigure() function
- Cy_SysClk_GetPll400MNo() function
- Cy_SysClk_PllCalucDividers() function
- Cy_SysClk_Pll400MEnable() function

4 Configuration of the FLL and PLL

See the below Code Listings to show the sample code for the PLL200 #0 example.

- General configuration of PLL 200 #0 settings
- AllClockConfiguration() function
- Cy_SysClk_PllConfigure() function
- Cy_SysClk_PllManualConfigure() function
- Cy_SysClk_GetPllNo() function
- Cy_SysClk_PllCalucDividers() function
- Cy_SysClk_PllEnable() function

General configuration of PLL 400 #0 settings

```
:
#define PLL400_0_TARGET_FREQ      (250000000ul) /** PLL target frequency**/
#define PLL400_1_TARGET_FREQ      (196608000ul) /** PLL target frequency**/
:
/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul) /** Define the TIMEOUT variable.**/
:
#define PLL_400M_0_PATH_NO   (1ul) /** Define the PLL number.**/
#define PLL_400M_1_PATH_NO   (2ul)/** Define the PLL number.**/
#define PLL_200M_0_PATH_NO   (3ul)/** Define the PLL number.**/
#define PLL_200M_1_PATH_NO   (4ul)/** Define the PLL number.**/
#define BYPASSED_PATH_NO     (5ul)/** Define the PLL number.**/
 :
/*** Parameters for Clock Configuration ***/
cy_stc_pll_400M_config_t g_pll400_0_Config = /** PLL400 #0 configuration.**/
{
    .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() /** PLL400 #0 setting. See AllClockConfiguration() function**/;
:
    /* Please check clock output using oscilloscope after CPU reached here. */
    for(;;);
}
```

4 Configuration of the FLL and PLL

AllClockConfiguration() function

```
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); /** PLL400 configuration. See
Cy_SysClk_Pll400MConfigure() function**/

        status = Cy_SysClk_Pll400MEnable(PLL_400M_0_PATH_NO, WAIT_FOR_STABILIZATION);
        CY_ASSERT(status == CY_SYSCLK_SUCCESS); /** PLL400 enable. See
Cy_SysClk_Pll400MEnable() function**/
:
   }
    return;
}
```

4 Configuration of the FLL and PLL

Cy_SysClk_Pll400MConfigure() function

```
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); /** Check for valid
clock path and PLL400 number. See Cy_SysClk_GetPll400MNo() function**/
    if(status != CY_SYSCLK_SUCCESS)
    {
        return(status);
    }

    if (SRSS->CLK_PLL400M[pllNo].unCONFIG.stcField.u1ENABLE != 0ul) /* 1 = enabled */ /** (1)
Check if PLL400 is already 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,
                                        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); /** PLL400 manual
configuration. See Cy_SysClk_Pll400MManualConfigure() function**/
    return (status);
}
```

4 Configuration of the FLL and PLL

Cy_SysClk_Pll400MManualConfigure() function

```
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); /** Get the PLL400
PATH number. See Cy_SysClk_GetPll400MNo() function.**/
    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);
    }

    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) /** (2) PLL400 configuration**/
    {
        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; /** (3) Fractional
divider settings**/
    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; /** (4)
SSCG settings**/
    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);
}
```

Cy_SysClk_GetPll400MNo() function

```
__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);
    }

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

Cy_SysClk_PllCalucDividers() function

```
__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;
        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;
                *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);
    }
}
```

Cy_SysClk_Pll400MEnable() function

```
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; /** (5) Enable PLL400**/

    /* now do the timeout wait for PLL_STATUS, bit LOCKED */
    for (; (SRSS->CLK_PLL400M[pllNo].unSTATUS.stcField.u1LOCKED == 0ul) && /** (6) Wait until
the PLL400 is locked.**/
           (timeoutus != 0ul); /** (7) Check the timeout.**/
         timeoutus--)
    {
        Cy_SysLib_DelayUs(1u); /** Wait for 1 us.**/
    }

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

    return (status);
}
```

4 Configuration of the FLL and PLL

General configuration of PLL 200 #0 settings

```
:
#define PLL200_0_TARGET_FREQ      (160000000ul /** PLL target frequency**/
#define PLL200_1_TARGET_FREQ      (80000000ul) /** PLL target frequency**/
:
/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul) /** Define the TIMEOUT variable**/
:
#define PLL_400M_0_PATH_NO   (1ul)/** Define the PLL number.**/
#define PLL_400M_1_PATH_NO   (2ul)/** Define the PLL number.**/
#define PLL_200M_0_PATH_NO   (3ul)/** Define the PLL number.**/
#define PLL_200M_1_PATH_NO   (4ul)/** Define the PLL number.**/
#define BYPASSED_PATH_NO     (5ul)/** Define the PLL number.**/
 :
/*** Parameters for Clock Configuration ***/
cy_stc_pll_config_t g_pll200_0_Config = /** PLL200 #0 configuration**/
{
    .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();/** PLL200 #0 setting. See AllClockConfiguration() function.**/
:
    /* Please check clock output using oscilloscope after CPU reached here. */
    for(;;);
}
```

4 Configuration of the FLL and PLL

AllClockConfiguration() function

```
static void AllClockConfiguration(void)
{
:
    /***** PLL200M#0(PATH3) source setting ******/
    {
:
        status = Cy_SysClk_PllConfigure(PLL_200M_0_PATH_NO , &g_pll200_0_Config);/** PLL200
configuration. See Cy_SysClk_PllConfigure() function **/
        CY_ASSERT(status == CY_SYSCLK_SUCCESS);

        status = Cy_SysClk_PllEnable(PLL_200M_0_PATH_NO, WAIT_FOR_STABILIZATION);/** PLL200
enable. See Cy_SysClk_PllEnable() function.**/
        CY_ASSERT(status == CY_SYSCLK_SUCCESS);
:
    }
    return;
}
```

4 Configuration of the FLL and PLL

Cy_SysClk_PllConfigure() function

```
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 */ /**(8) Check
if PLL200 is already 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, /** PLL200 calculation for
dividers settings. See Cy_SysClk_PllCalucDividers() function**/
                                        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); /** PLL200 manual
configuration. See Cy_SysClk_PllManualConfigure() function**/
    return (status);

}
```

4 Configuration of the FLL and PLL

Cy_SysClk_PllManualConfigure() function

```
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) /**(9) PLL200 configuration**/
    {
        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);
}
```

Cy_SysClk_GetPllNo() function

```
__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);
}
```

Cy_SysClk_PllCalucDividers() function

```
__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;
        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;
                *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);
```

```
    }
}
```

Cy_SysClk_PllEnable() function

```
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; /** (10) Enable PLL200**/

    /* now do the timeout wait for PLL_STATUS, bit LOCKED */
    for (; (SRSS->unCLK_PLL_STATUS[pllNo].stcField.u1LOCKED == 0ul) && /** (11) Wait until
PLL200 is locked.**/
           (timeoutus != 0ul); /** (12) Check the timeout.**/
         timeoutus--)
    {
        Cy_SysLib_DelayUs(1u); /** Wait for 1 us.**/
    }

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

    return (status);
}
```

5 Configuring the internal clock

This section describes how to configure the internal clocks as part of the clock system.

5.1 Configuring CLK_PATHx

The CLK_PATHx is used as the input source for the root clock, CLK_HFx. The CLK_PATHx can select all clock resources including the FLL and PLL using DSI_MUX and PATH_MUX. The CLK_PATH9 cannot select the FLL and PLL, but other clock resources can be selected.

Table 13 shows the relationship between the FLL/PLLs and CLK_PATHx.

Table 13 Relationship between the FLL/PLLs and PATHx

FLL/PLLs	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 and PLL cannot be selected)	CLK_PATH9

Figure 14 shows a generation diagram for CLK_PATH.

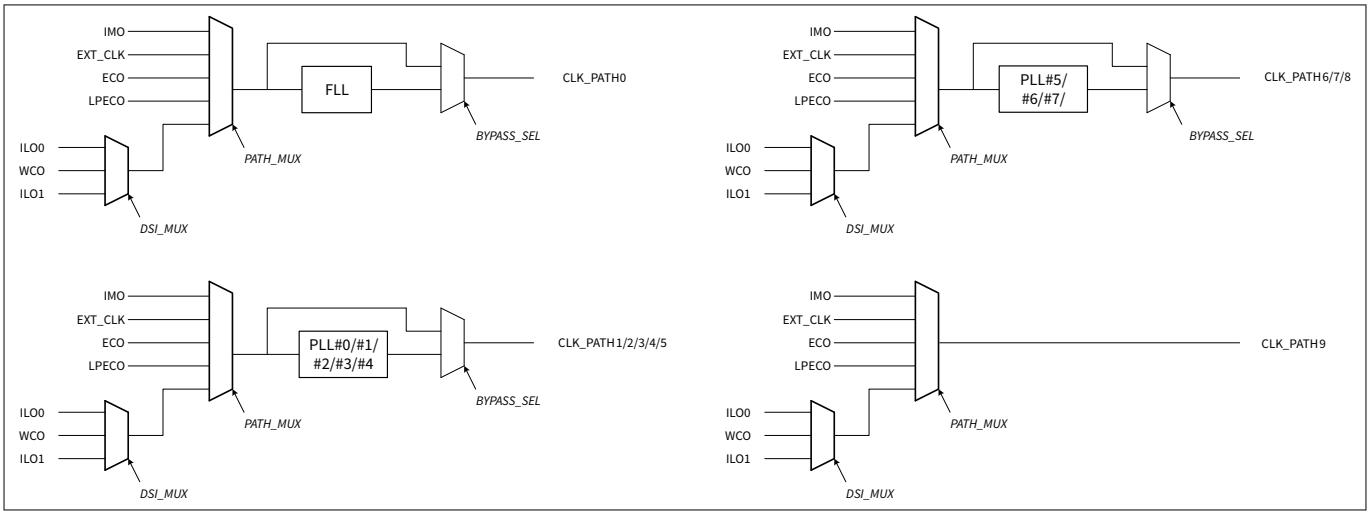

Figure 14 Generation diagram for the CLK_PATH

To configure the CLK_PATHx, you must configure the DSI_MUX and PATH_MUX. The BYPASS_MUX is also required for the CLK_PATHx. Table 14 shows the registers required for configuring the CLK_PATHx. See the architecture TRM for details.

Table 14 Configuring CLK_PATHx

Register name	Bit name	Value	Selected clock and item
CLK_PATH_SELECT	PATH_MUX[2:0]	0 (Default)	IMO
		1	EXT_CLK
		2	ECO
		4	DSI_MUX
		5	LPECO
		other	Reserved. Do not use.
CLK_DSI_SELECT	DSI_MUX[4:0]	16	ILO0
		17	WCO
		20	ILO1
		Other	Reserved. Do not use.
CLK_FLL_CONFIG3	BYPASS_SEL[29:28]	0 (Default)	AUTO[1)]
		1	LOCKED_OR_NOTHING[2)]
		2	FLL_REF (bypass mode)[3)]
		3	FLL_OUT[3)]
CLK_PLL_CONFIG	BYPASS_SEL[29:28]	0 (Default)	AUTO[1)]
		1	LOCKED_OR_NOTHING[2)]
		2	PLL_REF (bypass mode)[3)]
		3	PLL_OUT[3)]

1) Switching automatically according to the locked state.
2) The clock is gated off when unlocked
3) In this mode, the lock state is ignored.

5.2 Configuring CLK_HFx

The CLK_HFx (x = 0 to 13) can be selected from CLK_PATHy (y= 0 to 9). A predivider is available to divide the selected CLK_PATHx. The CLK_HF0 is always enabled because it is the source clock for the CPU cores. It is possible to disable CLK_HFx.

To enable the CLK_HFx, write '1' to the ENABLE bit of each CLK_ROOT_SELECT register. To disable CLK_HFx, write '0' to the ENABLE bit of each CLK_ROOT_SELECT register.

The ROOT_DIV bit of the CLK_ROOT register configures the predivider values from the following options: no division, divide by 2, divide by 4, and by 8.

Figure 15 shows the details of the ROOT_MUX and predivider.

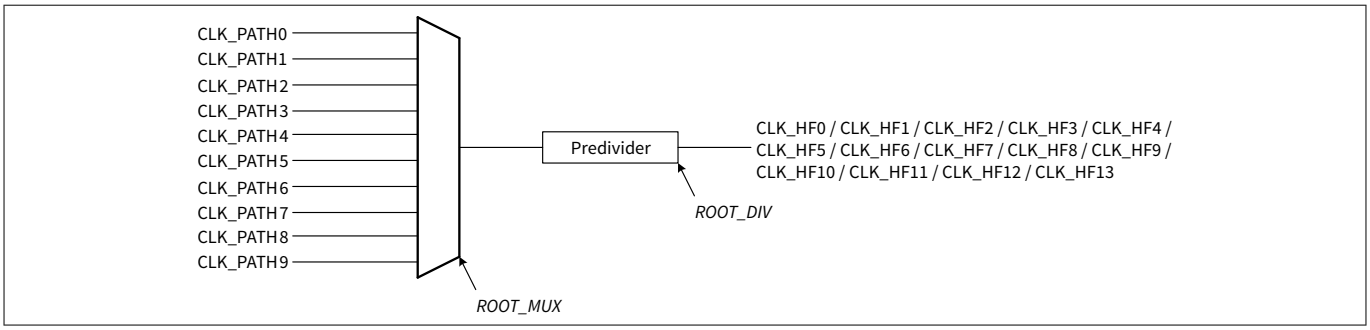

Figure 15 ROOT_MUX and predivider

Table 15 shows the registers required for the CLK_HFx. See the architecture TRM for details.

Table 15 Configuring of the CLK_HFx

Register name	Bit name	Value	Selected item
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
		Other	Reserved. Do not use.
CLK_ROOT_SELECT	ROOT_DIV[1:0]	0	No division
		1	Divide clock by 2
		2	Divide clock by 4
		3	Divide clock by 8

5.3 Configuring the CLK_LF

The CLK_LF can be selected from one of the possible sources WCO, ILO0, ILO1, ECO_Prescaler, and LPECO_Prescaler. The CLK_LF cannot be configured when the WDT_LOCK bit in the WDT_CLTL register is disabled because the CLK_LF can select the ILO0 that is the input clock for the WDT.

Figure 16 shows the details of LFCLK_SEL that configure the CLK_LF.

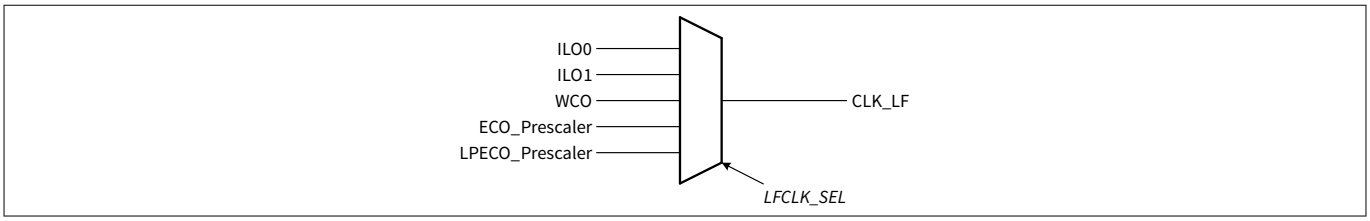

Figure 16 LFCLK_SEL

Table 16 shows the registers required for the CLK_LF. See the architecture TRM for details.

Table 16 Configuring of the CLK_LF

Register name	Bit name	Value	Selected item
CLK_SELECT	LFCLK_SEL[2:0]	0 (Default)	ILO0
		1	WCO
		4	ILO1
		5	ECO_Prescaler
		6	LPECO_Prescaler
		other	Reserved. Do not use.

5.4 Configuring CLK_FAST_0/CLK_FAST_1

CLK_FAST_0 and CLK_FAST_1 are generated by dividing CLK_HF1 by (x+1). When configuring the CLK_FAST_0 and CLK_FAST_1, configure a value (x = 0 to 255) divided by the FRAC_DIV bit and INT_DIV bit of the CPUSS_FAST_0_CLOCK_CTL register and CPUSS_FAST_1_CLOCK_CTL register.

5.5 Configuring CLK_MEM

The CLK_MEM is generated by dividing the CLK_HF0; its frequency is configured by the value obtained by dividing the CLK_HF0 by (x+1). When configuring the CLK_MEM, configure a value (x= 0 to 255) divided by the INT_DIV bit of the CPUSS_MEM_CLOCK_CTL register.

5.6 Configuring CLK_PERI

The CLK_PERI is the clock input to the peripheral clock divider and CLK_GR. The CLK_PERI is generated by dividing the CLK_HF0; its frequency is configured by the value obtained by dividing CLK_HF0 by (x+1). When configuring the CLK_PERI, configure a value (x= 0 to 255) divided by the INT_DIV bit of the CPUSS_PERI_CLOCK_CTL register.

5.7 Configuring CLK_SLOW

The CLK_SLOW is generated by dividing the CLK_MEM; its frequency is configured by the value obtained by dividing CLK_MEM by (x+1). After configuring the CLK_MEM, configure a value divided (x= 0 to 255) by the INT_DIV bit of the CPUSS_SLOW_CLOCK_CTL register.

5.8 Configuring CLK_GR

The clock source of the CLK_GP is the CLK_PERI in groups 3, 4, 8 and CLK_HF2 in groups 5, 6, 9. Groups 3, 4, 8 are clocks divided by CLK_PERI. To generate CLK_GR3, CLK_GR4, and CLK_GR8, write the division value (from 1 to 255) to divide the INT8_DIV bit of the CPUSS_PERI_GRx_CLOCK_CTL register.

5.9 Configuring PCLK

The PCLK is a clock that activates each peripheral function. Peripheral clock dividers have a function to divide the CLK_PERI and generate a clock to be supplied to each peripheral function. For assignment of the peripheral clocks, see the “Peripheral clocks” section in the Datasheet.

Figure 17 shows the steps to configure peripheral clock dividers. See the architecture TRM for details.

5 Configuring the internal clock

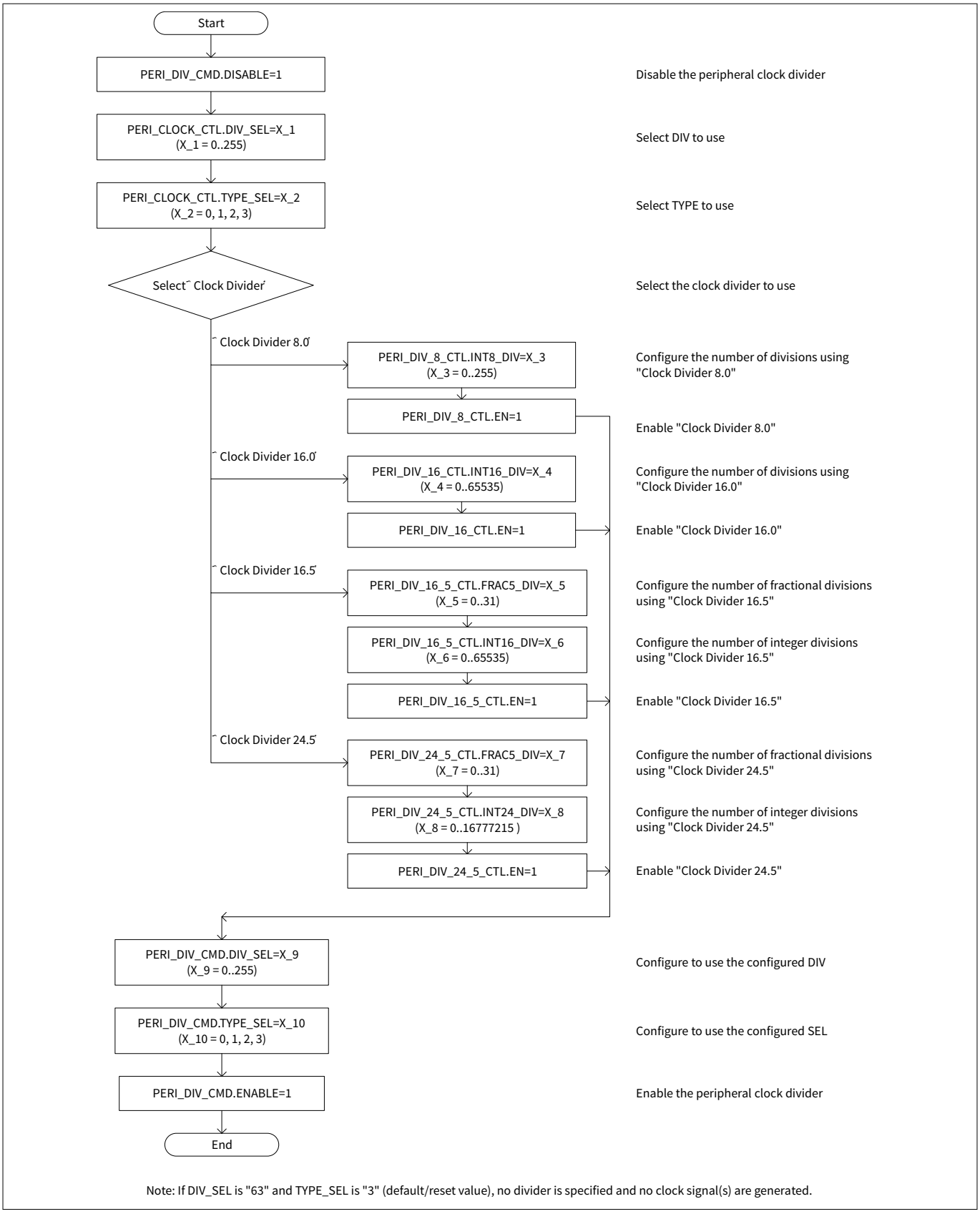

Figure 17 Procedure for configuring to generate the PCLK

5.9.1 Example of PCLK setting

5.9.1.1 Use case

- Input clock frequency: 80 MHz
- Output clock frequency: 2 MHz
- Divider type: Clock divider 16.0
- Used divider: Clock divider 16.0#0
- Peripheral clock output number: 38 (TCPWM0, Group#0, Counter#0)

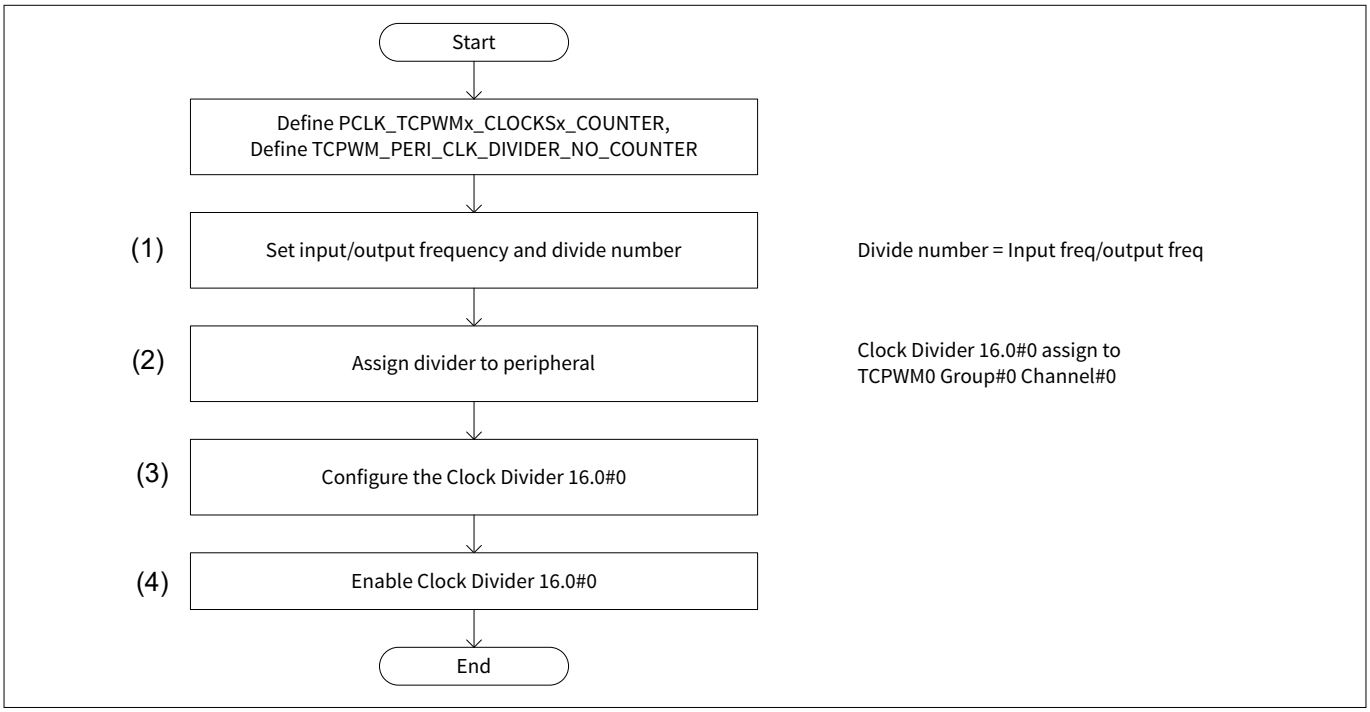

Figure 18 Example procedure for setting the PCLK

5.9.1.2 Configuration

Table 17 lists the parameters and Table 18 lists the functions of the configuration part of in the SDL for the PCLK (example of the TCPWM timer) settings.

Table 17 List of PCLK (example of the TCPWM timer) parameters

Parameters	Description	Value
PCLK_TCPWMx_CLOCKSx_COUNTER	PCLK of TCPWM0	PCLK_TCPWM0_CLOCKS0 = 38ul
TCPWM_PERI_CLK_DIVIDER_NO_COUNTER	Number of dividers to be used	0ul

(table continues...)

Table 17 (continued) List of PCLK (example of the TCPWM timer) parameters

Parameters	Description	Value
CY_SYSCLK_DIV_16_BIT	Divider type: CY_SYSCLK_DIV_8_BIT = 0u, 8 bit divider CY_SYSCLK_DIV_16_BIT = 1u, 16 bit divider CY_SYSCLK_DIV_16_5_BIT = 2u, 16.5 bit fractional divider CY_SYSCLK_DIV_24_5_BIT = 3u, 24.5 bit fractional divider	1ul
periFreq	Peripheral clock frequency	80000000ul (80 MHz)
targetFreq	Target clock frequency	2000000ul (2 MHz)
divNum	Divide number	periFreq/targetFreq

Table 18 List of PCLK (Example of the TCPWM timer) settings functions

Functions	Description	Value
`Cy_SysClk_PeriphAssign Divider(IPblock dividerType, dividerNum)`	Assign a programmable divider to a selected IP block (such as a TCPWM)	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)`	Set the peripheral divider	dividerType, = CY_SYSCLK_DIV_16_BIT dividerNum = TCPWM_PERI_CLK_DIVIDER_NO_COUNTER dividerValue = divNum-1ul
`Cy_SysClk_PeriphEnable Divider(dividerType, dividerNum)`	Enable the peripheral divider	dividerType, = CY_SYSCLK_DIV_16_BIT dividerNum = TCPWM_PERI_CLK_DIVIDER_NO_COUNTER

5.9.2 Sample code for the initial configuration of PCLK settings (example of the TCPWM timer)

See the below code listing for sample code.

- General configuration of PCLK (example of the TCPWM timer) settings
- Cy_SysClk_PeriphAssignDivider() function

5 Configuring the internal clock

- Cy_SysClk_PeriphSetDivider() function
- Cy_SysClk_PeriphEnableDivider() function

General configuration of PCLK (example of the TCPWM timer) settings

```
:
#define PCLK_TCPWMx_CLOCKSx_COUNTER       PCLK_TCPWM0_CLOCKS0     /** Define
PCLK_TCPWMx_CLOCKSx_COUNTER, Define TCPWM_PERI_CLK_DIVIDER_NO_COUNTER.**/
#define TCPWM_PERI_CLK_DIVIDER_NO_COUNTER 0u
:
int main(void)
{
    SystemInit();

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

    uint32_t periFreq = 80000000ul; /** (1) Set the input/output frequency and divide number.**/
    uint32_t targetFreq = 2000000ul;
    uint32_t divNum = (periFreq / targetFreq); /** Calculation of division **/

    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); /** Peripheral divider assign setting. See
Cy_SysClk_PeriphAssignDivider() function.**/
    /* Sets the 16-bit divider */
    Cy_SysClk_PeriphSetDivider(CY_SYSCLK_DIV_16_BIT, TCPWM_PERI_CLK_DIVIDER_NO_COUNTER,
(divNum-1ul)); /** Peripheral divider setting. See Cy_SysClk_PeriphSetDivider() function.**/

    Cy_SysClk_PeriphEnableDivider(CY_SYSCLK_DIV_16_BIT, TCPWM_PERI_CLK_DIVIDER_NO_COUNTER); /**
Peripheral divider enable setting. See Cy_SysClk_PeriphEnableDivider() function.**/


    for(;;);
}
```

5 Configuring the internal clock

Cy_SysClk_PeriphAssignDivider() function

```
__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; /**
(2) Assign the divider to the peripheral.**/
    tempCLOCK_CTL_RegValue.stcField.u2TYPE_SEL = dividerType; /** (2) Assign the divider to the
peripheral.**/
    tempCLOCK_CTL_RegValue.stcField.u8DIV_SEL  = dividerNum; /** (2) Assign the divider to the
peripheral.**/
    PERI->unCLOCK_CTL[ipBlock].u32Register     = tempCLOCK_CTL_RegValue.u32Register; /** (2)
Assign the divider to the peripheral.**/

    return CY_SYSCLK_SUCCESS;

}
```

Cy_SysClk_PeriphSetDivider() function

```
__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; /** (3)
Division setting to the clock divider 16.0#0.**/
        :
    }
    else
    { /* return bad parameter */
        return CY_SYSCLK_BAD_PARAM;
    }

    return CY_SYSCLK_SUCCESS;
}
```

Cy_SysClk_PeriphEnableDivider() function

```
__STATIC_INLINE cy_en_sysclk_status_t Cy_SysClk_PeriphEnableDivider(cy_en_divider_types_t
dividerType, uint32_t dividerNum) /** (4) Enable clock divider 16#0.**/
{
:
    /* specify the divider, make the reference = clk_peri, and enable the divider */
    un_PERI_DIV_CMD_t tempDIV_CMD_RegValue;
    tempDIV_CMD_RegValue.u32Register            = PERI->unDIV_CMD.u32Register;
    tempDIV_CMD_RegValue.stcField.u1ENABLE      = 1ul;
    tempDIV_CMD_RegValue.stcField.u2PA_TYPE_SEL = 3ul;
    tempDIV_CMD_RegValue.stcField.u8PA_DIV_SEL  = 0xFFul;
    tempDIV_CMD_RegValue.stcField.u2TYPE_SEL    = dividerType; /** Set the divider type
select.**/
    tempDIV_CMD_RegValue.stcField.u8DIV_SEL     = dividerNum; /** Set the divider number.**/
    PERI->unDIV_CMD.u32Register                 = tempDIV_CMD_RegValue.u32Register;

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

    return CY_SYSCLK_SUCCESS;
}
```

5.10 Setting ECO_Prescaler

The ECO_Prescaler divides the ECO, and creates a clock that can be used with the CLK_LF. The division function has a 10-bit integer divider and 8-bit fractional divider.

Figure 19 shows the steps to enable the ECO_Prescaler. For details on the ECO_Prescaler, see the architecture TRM and register TRM.

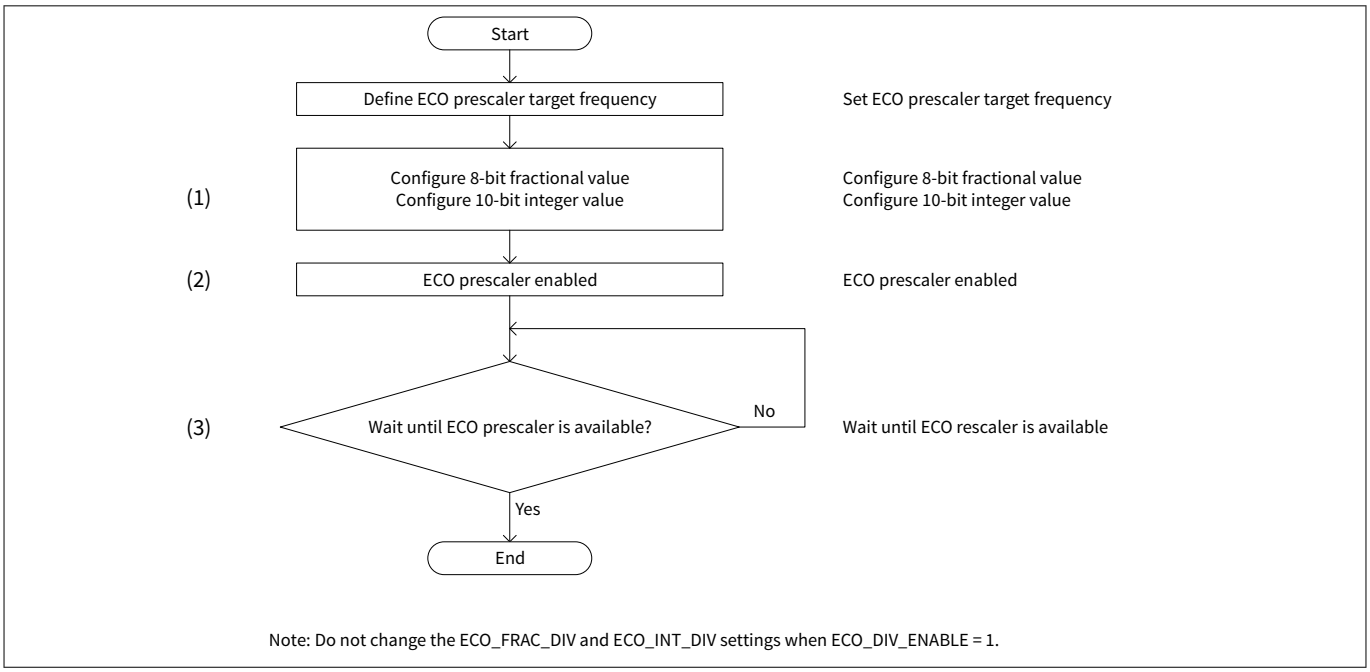

Figure 19 Enabling the ECO_Prescaler

Figure 20 shows the flow to disable the ECO_Prescaler. For details on the ECO_Prescaler, see the architecture TRM.

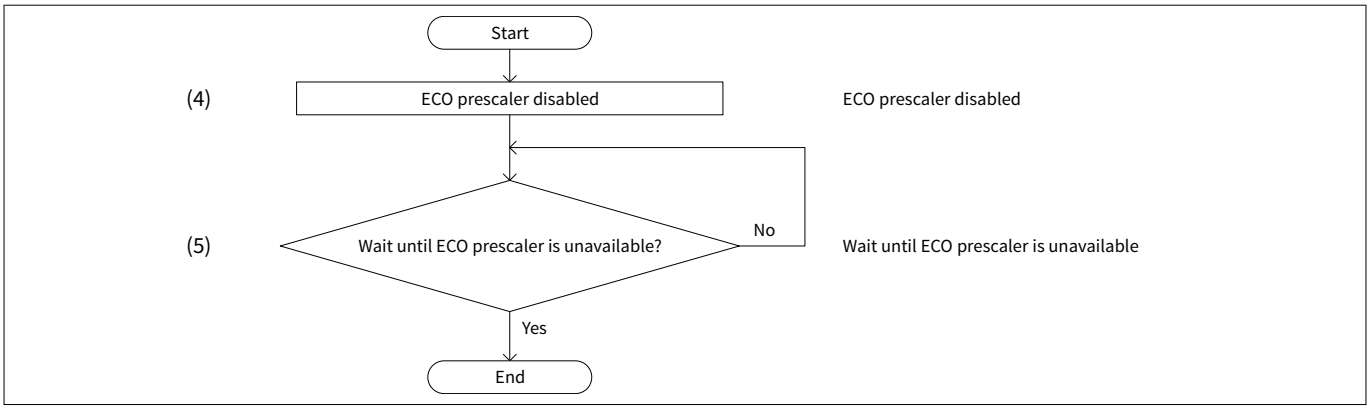

Figure 20 Disabling the ECO_Prescaler

5.10.1 Use case

- Input clock frequency: 16 MHz
- ECO prescaler target frequency: 1.234567 MHz

5.10.2 Configuration

Table 19 lists the parameters and Table 20 lists the functions of the configuration part of in the SDL for ECO prescaler settings.

Table 19 List of ECO prescaler settings parameters

Parameters	Description	Value
ECO_PRESCALER_TARGET_FREQ	ECO prescaler target frequency	1234567ul
WAIT_FOR_STABILIZATION	Waiting for stabilization	10000ul
CLK_FREQ_ECO	ECO clock frequency	16000000ul (16 MHz)
PATH_SOURCE_CLOCK_FREQ	PATH source clock frequency	CLK_FREQ_ECO

Table 20 List of ECO prescaler setting functions

Functions	Description	Value
`AllClockConfiguration()`	Clock configuration	–
`Cy_SysClk_SetEcoPrescale(Inclk, Targetclk)`	Set the ECO frequency and target frequency.	Inclk = PATH_SOURCE_CLOCK_FREQ,
		Targetclk = ECO_PRESCALER_TARGET_FREQ
`Cy_SysClk_EcoPrescaleEnable(Timeout value)`	Set the ECO prescaler enable and timeout value	Timeout value = WAIT_FOR_STABILIZATION
`Cy_SysClk_SetEcoPrescaleManual (divInt, divFact)`	divInt: 10-bit integer value allows for ECO frequencies divFrac: 8-bit fractional value	–
`Cy_SysClk_GetEcoPrescaleStatus`	Check the prescaler status.	–

5.10.3 Sample code for the initial configuration of ECO prescaler settings

See the below Code Listings for sample code.

- General configuration of ECO prescaler settings
- AllClockConfiguration() function
- Cy_SysClk_SetEcoPrescale() function
- Cy_SysClk_SetEcoPrescaleManual() function
- Cy_SysClk_EcoPrescaleEnable() function
- Cy_SysClk_GetEcoPrescaleStatus() function
- Cy_SysClk_EcoPrescaleDisable() function

General configuration of ECO prescaler settings

```
#define ECO_PRESCALER_TARGET_FREQ (1234567ul) /** Define the ECO prescaler target frequency.**/
#define CLK_FREQ_ECO       (16000000ul) /** Define the ECO clock frequency.**/
#define PATH_SOURCE_CLOCK_FREQ CLK_FREQ_ECO

/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul) /** Define waiting for stabilization.**/


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

5 Configuring the internal clock

AllClockConfiguration() function

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

        ecoPreStatus = Cy_SysClk_SetEcoPrescale(CLK_FREQ_ECO, ECO_PRESCALER_TARGET_FREQ); /**
ECO prescaler setting. See AllClockConfiguration() function.**/
        CY_ASSERT(ecoPreStatus == CY_SYSCLK_SUCCESS);

        ecoPreStatus = Cy_SysClk_EcoPrescaleEnable(WAIT_FOR_STABILIZATION); /** ECO prescaler
enable. See Cy_SysClk_EcoPrescaleEnable() function.**/
        CY_ASSERT(ecoPreStatus == CY_SYSCLK_SUCCESS);
    }

   return;
}
```

Cy_SysClk_SetEcoPrescale() function

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

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

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

    fixedPointDivNum = (uint32_t)fixedPointDivNum64;

    Cy_SysClk_SetEcoPrescaleManual(
                                   (((fixedPointDivNum & 0x0003FF00ul) >> 8ul) - 1ul),
                                   (fixedPointDivNum & 0x000000FFul)
                                   ); /** Configure the ECO prescaler. See
Cy_SysClk_SetEcoPrescaleManual() function**/

    return CY_SYSCLK_SUCCESS;
}
```

Cy_SysClk_SetEcoPrescaleManual() function

```
__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; /** (1)
Configure the ECO prescaler.**/
    tempRegEcoPrescale.stcField.u10ECO_INT_DIV = divInt;
    tempRegEcoPrescale.stcField.u8ECO_FRAC_DIV = divFract;
    SRSS->unCLK_ECO_PRESCALE.u32Register       = tempRegEcoPrescale.u32Register;

    return;
}
```

Cy_SysClk_EcoPrescaleEnable() function

```
cy_en_sysclk_status_t Cy_SysClk_EcoPrescaleEnable(uint32_t timeoutus)
{
    // Send enable command
    SRSS->unCLK_ECO_CONFIG.stcField.u1ECO_DIV_ENABLE = 1ul; /**(2) Enable the ECO prescaler.**/

    // Wait eco prescaler get enabled
    while(CY_SYSCLK_ECO_PRESCALE_ENABLE != Cy_SysClk_GetEcoPrescaleStatus()) /**(3) Wait until
the ECO prescaler is available.**/
    {
        if(0ul == timeoutus)
        {
            return CY_SYSCLK_TIMEOUT;
        }

        Cy_SysLib_DelayUs(1u);

        timeoutus--;
    }

    return CY_SYSCLK_SUCCESS;
}
```

Cy_SysClk_GetEcoPrescaleStatus() function

```
__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);
/** Check the prescaler status.**/
}
```

If you want to disable the ECO prescaler, set the wait time in the same way as the function above, and then call the next function.

Cy_SysClk_EcoPrescaleDisable() function

```
cy_en_sysclk_status_t Cy_SysClk_EcoPrescaleDisable(uint32_t timeoutus)
{
    // Send disable command
    SRSS->unCLK_ECO_CONFIG.stcField.u1ECO_DIV_DISABLE = 1ul; /** (4) Disable the ECO
prescaler.**/

    // Wait eco prescaler actually get disabled
    while(CY_SYSCLK_ECO_PRESCALE_DISABLE != Cy_SysClk_GetEcoPrescaleStatus()) /** (5) Wait
until the ECO prescaler is unavailable.**/
    {
        if(0ul == timeoutus)
        {
            return CY_SYSCLK_TIMEOUT;
        }

        Cy_SysLib_DelayUs(1u);

        timeoutus--;
    }

    return CY_SYSCLK_SUCCESS;
}
```

5.11 Configuring the LPECO_Prescaler

The LPECO_Prescaler divides the LPECO. The division function has a 10-bit integer divider and 8-bit fractional divider.

Figure 21 shows the steps to enable the LPECO_Prescaler. For details on the LPECO_Prescaler, see the architecture TRM.

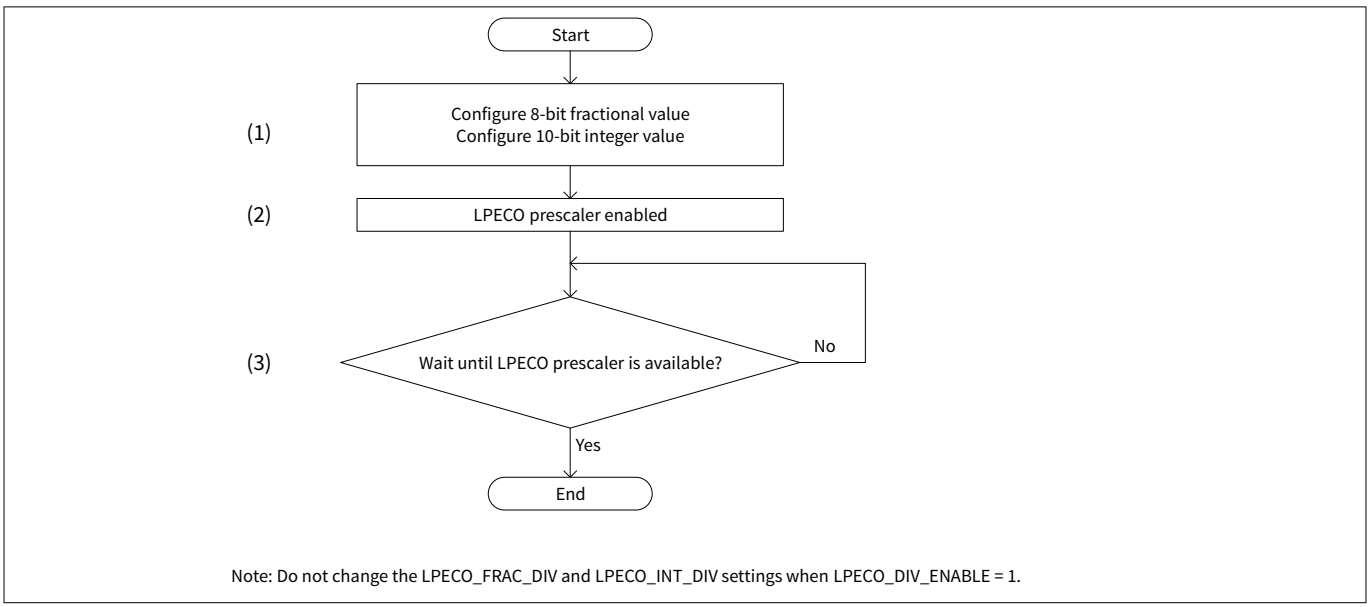

Figure 21 Enabling the LPECO_Prescaler

Figure 22 shows the steps to disable the LPECO_Prescaler. For details on the LPECO_Prescaler, see the architecture TRM.

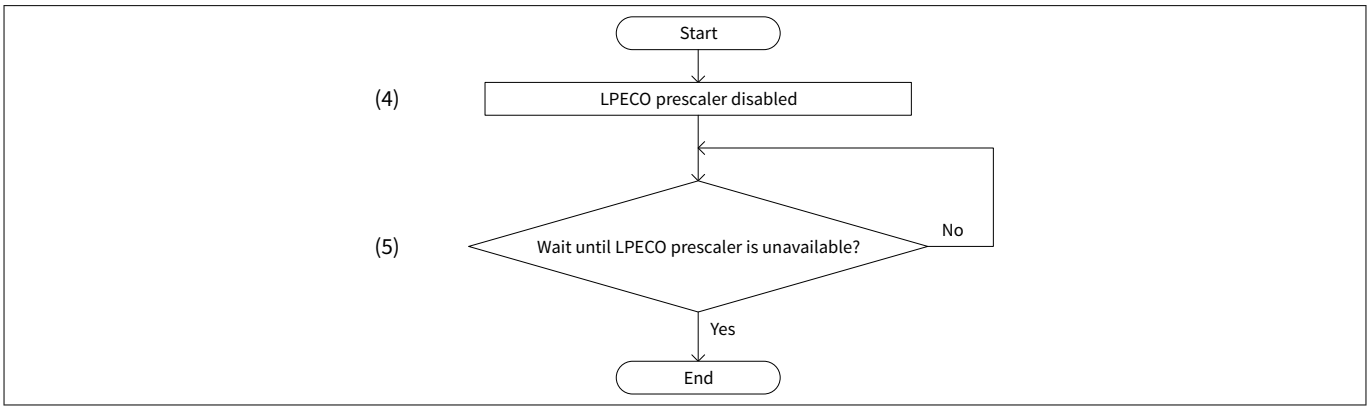

Figure 22 Disabling the LPECO_Prescaler

5.11.1 Use case

- Input clock frequency: 8 MHz
- LPECO prescaler target frequency: 1.234567 MHz

5.11.2 Configuration

Table 21 lists the parameters and Table 22 lists the functions of the configuration part of in the SDL for LPECO prescaler settings.

Table 21 List of LPECO prescaler settings parameters

Parameters	Description	Value
LPECO_PRESCALER_TARGET_FREQ	ECO prescaler target frequency	1234567ul
WAIT_FOR_STABILIZATION	Waiting for stabilization	10000ul
CLK_FREQ_LPECO	ECO clock frequency	8000000ul (8 MHz)
PATH_SOURCE_CLOCK_FREQ	PATH source clock frequency	CLK_FREQ_LPECO

Table 22 List of LPECO prescaler settings functions

Functions	Description	Value
`AllClockConfiguration()`	Clock configuration	–
`Cy_SysClk_ClkBak_LPECO_SetPrescale(frac,int)`	Prescaler integer and fractional divider to generate 32.768 kHz from the LPECO	frac = fixedPointDivNum & 0x000000FFul int = (fixedPointDivNum & 0x0003FF00ul) >> 8ul) - 1ul
`Cy_SysClk_ClkBak_LPECO_EnableDivider(divInt,divFract)`	Set the prescaler enable for the LPECO.	divInt = 0x3FF divFract = 0xFF
`Cy_SysClk_ClkBak_LPECO_PrescalerOkay()`	Return the status from the LPECO after setting the prescaler divider.	–

5.11.3 Sample code for the initial configuration of LPECO prescaler settings

See the below Code Listings for sample code.

- General configuration of LPECO prescaler settings
- AllClockConfiguration() function
- Cy_SysClk_ClkBak_LPECO_SetPrescale () function
- Cy_SysClk_ClkBak_LPECO_SetPrescaleManual() function
- Cy_SysClk_ClkBak_LPECO_PrescaleEnable() function

General configuration of LPECO prescaler settings

```
#define LPECO_PRESCALER_TARGET_FREQ (1234567ul) /** Define the LPECO prescaler target
frequency.**/
#define CLK_FREQ_LPECO       (8000000ul) /** Define the LPECO clock frequency.**/
#define PATH_SOURCE_CLOCK_FREQ CLK_FREQ_LPECO

/** Wait time definition **/
#define WAIT_FOR_STABILIZATION (10000ul) /** Define the TIMEOUT variable.**/


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

AllClockConfiguration() function

```
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); /** LPECO prescaler setting. See
Cy_SysClk_ClkBak_LPECO_SetPrescale () function.**/

        lpecoPreStatus = Cy_SysClk_ClkBak_LPECO_PrescaleEnable(WAIT_FOR_STABILIZATION);
        CY_ASSERT(lpecoPreStatus == CY_SYSCLK_SUCCESS); /** LPECO prescaler enable. See
Cy_SysClk_ClkBak_LPECO_SetPrescaleManual() function.**/
    }
:
    return;
}

}
```

Cy_SysClk_ClkBak_LPECO_SetPrescale () function

```
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); /** LPECO prescaler setting. See
Cy_SysClk_ClkBak_LPECO_SetPrescale () function**/

        lpecoPreStatus = Cy_SysClk_ClkBak_LPECO_PrescaleEnable(WAIT_FOR_STABILIZATION);
        CY_ASSERT(lpecoPreStatus == CY_SYSCLK_SUCCESS); /** LPECO prescaler enable. See
Cy_SysClk_ClkBak_LPECO_SetPrescaleManual() function**/
    }
:
    return;
}

}
```

Cy_SysClk_ClkBak_LPECO_SetPrescaleManual() function

```
__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( /** Configure the LPECO prescaler.
See .Cy_SysClk_ClkBak_LPECO_SetPrescaleManual() function**/
                                            (((fixedPointDivNum & 0x0003FF00ul) >> 8ul) - 1ul),
                                            (fixedPointDivNum & 0x000000FFul)
                                           );

    return CY_SYSCLK_SUCCESS;
}
```

Cy_SysClk_ClkBak_LPECO_SetPrescaleManual() function

```
__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; /** (1) Configure the 8-
bit fractional value and 10-bit integer value.**/
        BACKUP->unLPECO_PRESCALE.stcField.u10LPECO_INT_DIV = intDiv;
    }
}
```

5 Configuring the internal clock

Cy_SysClk_ClkBak_LPECO_PrescaleEnable() function

```
__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; /** (2) Enable the LPECO
prescaler.**/

    // Wait for eco prescaler to get enabled
    while(CY_SYSCLK_ECO_PRESCALE_ENABLE != Cy_SysClk_ClkBak_LPECO_PrescalarOkay()) /** (3) Wait
until the LPECO prescaler is available.**/
    {
        if(0ul == timeoutus)
        {
            return CY_SYSCLK_TIMEOUT;
        }

        Cy_SysLib_DelayUs(1u);

        timeoutus--;
    }

    return CY_SYSCLK_SUCCESS;
}
```

Cy_SysClk_ClkBak_LPECO_PrescalerOkay() function

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

To disable the LPECO prescaler, set the wait time in the same way as the function above, and then call the next function.

5 Configuring the internal clock

Cy_SysClk_ClkBak_LPECO_PrescaleDisable() function

```
__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; /** (4) Disable the LPECO prescaler.**/

    // Wait for eco prescaler to get enabled
    while(BACKUP->unLPECO_PRESCALE.stcField.u1LPECO_DIV_ENABLED == 1) /** (5) Wait until the
LPECO prescaler is unavailable.**/
    {
        if(0ul == timeoutus)
        {
            return CY_SYSCLK_TIMEOUT;
        }

        Cy_SysLib_DelayUs(1u);

        timeoutus--;
    }

    return CY_SYSCLK_SUCCESS;
}
```

6 Supplementary information

6.1 Input clocks in peripheral functions

Table 23 to Table 31 list the clock input to each peripheral function. For detailed values of PCLK, see the "Peripheral clocks" section of the datasheet.

Table 23 Clock input to TCPWM[0]

Peripheral function	Operation clock	Channel clock
TCPWM[0]	CLK_GR3 (Group 3)	PCLK (PCLK_TCPWM0_CLOCKSx, x = 0 to 37)
		PCLK (PCLK_TCPWM0_CLOCKSy, y = 256 to 267)
		PCLK (PCLK_TCPWM0_CLOCKSz, z =512–543)

Table 24 Clock input to CAN FD

Peripheral function	Operation clock (clk_sys (hclk))	Channel clock (clk_can (cclk))
CAN FD0	CLK_GR5 (Group 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 Clock input to LIN

Peripheral function	Operation clock	Channel clock (clk_lin_ch)
LIN	CLK_GR5 (Group 5)	Ch0: PCLK (PCLK_LIN_CLOCK_CH_EN0)
		Ch1: PCLK (PCLK_LIN_CLOCK_CH_EN1)

Table 26 Clock input to SCB

Peripheral function	Operation clock	Channel clock
SCB0	CLK_GR6 (Group 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)

(table continues...)

6 Supplementary information

Table 26 (continued) Clock input to SCB

Peripheral function	Operation clock	Channel 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 Clock input to SAR ADC

Peripheral function	Operation clock	Unit clock
SAR ADC	CLK_GR9 (Group 9)	Unit0: PCLK (PCLK_PASS_CLOCK_SAR0)

Table 28 Clock input to CXPI

Peripheral function	Operation clock	Channel clock
CXPI	CLK_GR5 (Group 5)	PCLK (PCLK_CXPI0_CLOCK_CH_EN0)
		PCLK (PCLK_CXPI0_CLOCK_CH_EN1)

Table 29 Clock input to SMIF

Peripheral function	"clk_slow" domain (XIP AHB-Lite Interface0)	"clk_mem" Domain (XIP AHB interface)	"clk_sys" domain (MMIO AHB-Lite interface)	"clk_if" domain
SMIF	clk_slow	clk_mem	CLK_GR4	CLK_HF6

Table 30 Clock input to Ethernet

Peripheral function	"clk_sys" domain	"clk_mem" domain	Internal reference clock	Time Stamp Unit (TSU) clock
Ethernet	CLK_GR4	clk_mem	CLK_HF4	CLK_HF5

Table 31 Clock input to Sound subsystem

Peripheral function	clk_sys_i2s	clk_audio_i2s
Sound Subsystem #0 (TDM, SG, PWM, MIXER, DAC)	CLK_GR8	CLK_HF5
Sound Subsystem #1 (TDM, SG, PWM, MIXER)	CLK_GR8	CLK_HF6
Sound Subsystem #2 (TDM, SG, PWM)	CLK_GR8	CLK_HF7

***Note**: For details on Ethernet clocks, see the architecture TRM.*

6.2 Use case of the clock calibration counter function

The clock calibration counter has two counters that can be used to compare the frequency of two clock sources. All clock sources are available as a source for these two clocks. For details, see the architecture TRM.

Use the following procedure to calibrate using the clock calibration counter:

1. Calibration Counter1 counts clock pulses from Calibration Clock1 (the high-accuracy clock used as the reference clock). It counts downwards.
2. Calibration Counter2 counts clock pulses from Calibration Clock2 (measurement clock). It counts upwards.
3. When Calibration Counter1 reaches 0, Calibration Counter2 stops counting upwards, and its value can be read.
4. The frequency of Calibration Counter2 can be obtained by using its value and the following equation:

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

Equation 1

Figure 23 shows an example of the clock calibration counter function when ILO0 and ECO are used. ILO0 and ECO must be enabled. See See ILO0 and ECO for Configuring ILO0/ILO1and Setting the ECO.

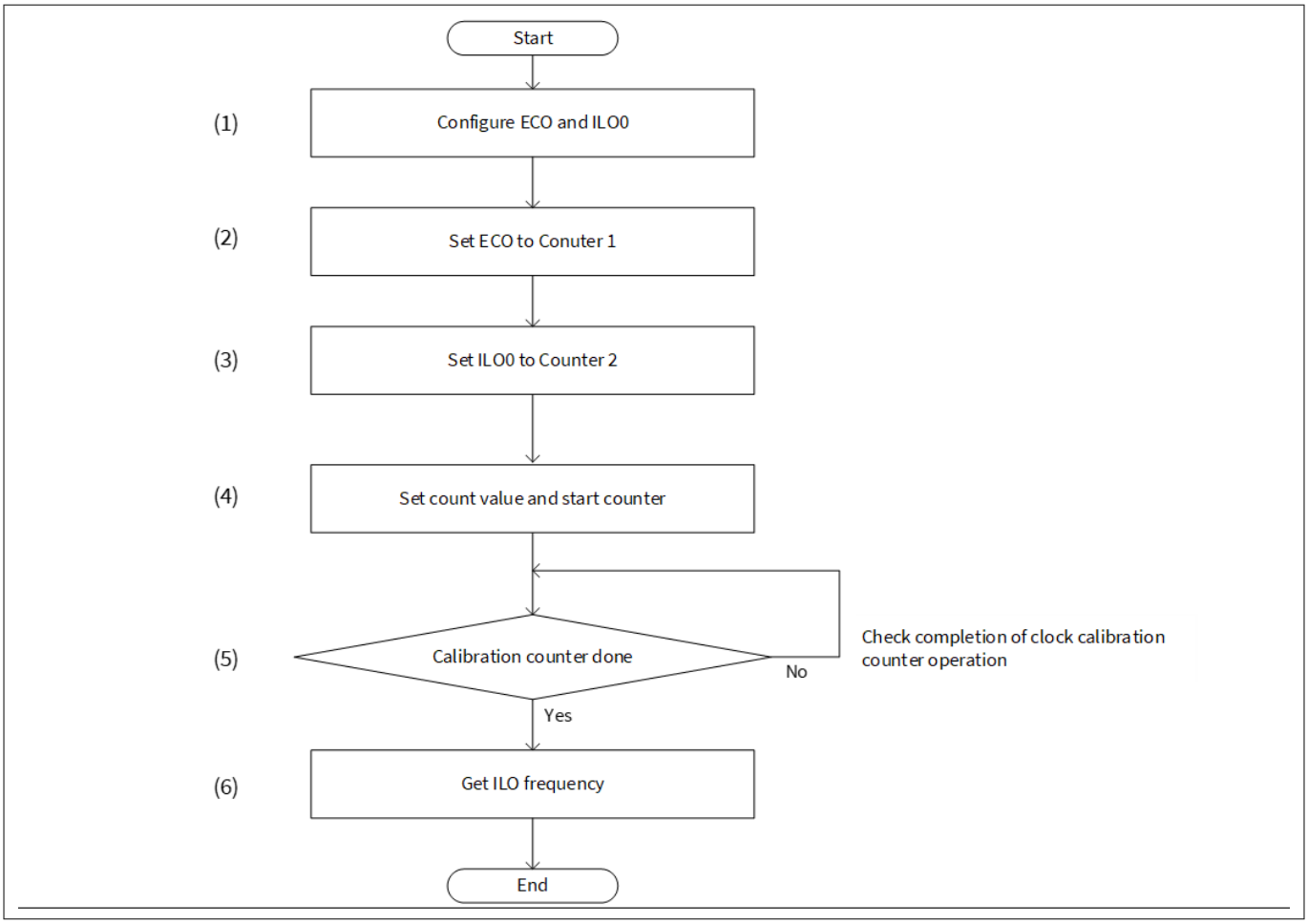

Figure 23 Example of the clock calibration counter with ILO0 and ECO

6.2.1 Use case

- Measurement clock: ILO0 clock frequency 32.768 kHz
- Reference clock: ECO clock frequency 16 MHz
- Reference clock count value: 40000ul

6.2.2 Configuration

Table 32 lists the parameters and Table 33 lists the functions of the configuration part of in the SDL for the clock calibration counter with ILO0 and ECO settings.

Table 32 List of clock calibration counter with ILO0 and ECO settings parameters

Parameters	Description	Value
ILO_0	Define the ILO_0 setting parameter	0ul
ILO_1	Define the ILO_1 setting parameter	1ul
ILONo	Define the measurement clock.	ILO_0
clockMeasuredInfo[].name	Measurement clock	CY_SYSCLK_MEAS_CLK_ILO0 = 1ul
clockMeasuredInfo[].measuredFreq	Store the measurement clock frequency.	–
counter1	Reference clock count value	40000ul
CLK_FREQ_ECO	ECO clock frequency	16000000ul (16 MHz)

Table 33 List of clock calibration counter with ILO0 and ECO settings functions

Functions	Description	Value
`GetILOClockFreq()`	Get the ILO 0 frequency.	–
`Cy_SysClk_StartClk MeasurementCounters (clk1, count1,clk2)`	Set and start calibration.	[Set the counter]
	Clk1: Reference clock	clk1 = CY_SYSCLK_MEAS_CLK_ECO = 0x101ul
	Count1: Measurement period	count1 = counter1
	Clk2: measurement clock	clk2 = clockMeasuredInfo[].name
`Cy_SysClk_ClkMeasurement CountersDone()`	Check if counter measurement is done.	–
`Cy_SysClk_ClkMeasurement CountersGetFreq (MesauredFreq, refClkFreq)`	Get the measurement clock frequency	
	MesauredFreq: Stored measurement clock frequency	MesauredFreq = clockMeasuredInfo[].measuredFreq
	refClkFreq: Reference clock frequency	refClkFreq = CLK_FREQ_ECO

6.2.3 Sample code for the initial configuration of the clock calibration counter with ILO0 and ECO settings

See the below code listing for sample code.

6 Supplementary information

- General configuration of the clock calibration counter with ILO0 and ECO settings
- GetILOClockFreq() function
- Cy_SysClk_StartClkMeasurementCounters() function
- Cy_SysClk_ClkMeasurementCountersDone() function
- Cy_SysClk_ClkMeasurementCountersGetFreq() function

General configuration of the clock calibration counter with ILO0 and ECO settings

```
#define CY_SYSCLK_DIV_ROUND(a, b) (((a) + ((b) / 2ull)) / (b)) /** Define the
CY_SYSCLK_DIV_ROUND function.**/

#define ILO_0   0ul /** Define the measurement clock (ILO0).**/
#define ILO_1   1ul
#define ILONo   ILO_0
#define CLK_FREQ_ECO            (16000000ul)

int32_t ILOFreq;

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

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

    /* Set Clock Configuring registers */
    AllClockConfiguration(); /** (1) ECO and ILO0 setting. See Setting the ECO and Configuring
ILO0/ILO1.**/

    /* return: Frequency of ILO */
    ILOFreq = GetILOClockFreq(); /** Get the clock frequency. See GetILOClockFreq() function.**/

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

6 Supplementary information

GetILOClockFreq() function

```
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)) /** Check the ECO status.**/
    {
       while(1);
    }

    cy_en_sysclk_status_t status;
    status = Cy_SysClk_StartClkMeasurementCounters(CY_SYSCLK_MEAS_CLK_ECO, counter1,
clockMeasuredInfo[0].name); /** Start the clock measurement counter. See
Cy_SysClk_StartClkMeasurementCounters() function.**/
    CY_ASSERT(status == CY_SYSCLK_SUCCESS);

    while(Cy_SysClk_ClkMeasurementCountersDone() == false); /** Check if counter measurement is
done. See Cy_SysClk_ClkMeasurementCountersDone() function.**/

    status = Cy_SysClk_ClkMeasurementCountersGetFreq(&clockMeasuredInfo[0].measuredFreq,
CLK_FREQ_ECO);
    CY_ASSERT(status == CY_SYSCLK_SUCCESS); /** Get the ILO frequency. See
Cy_SysClk_ClkMeasurementCountersGetFreq() function.**/

:

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

6 Supplementary information

Cy_SysClk_StartClkMeasurementCounters() function

```
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; /** (2) Setting the
reference clock (ECO).**/
:
SRSS->unCLK_OUTPUT_SLOW.stcField.u4SLOW_SEL1 = (uint32_t)clock2; /** (3) Setting the
measurement clock (ILO0).**/
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; /** (4) Set the count value and
start the counter.**/
:
    return (rtnval);
}
```

Cy_SysClk_ClkMeasurementCountersDone() function

```
__STATIC_INLINE bool Cy_SysClk_ClkMeasurementCountersDone(void)
{
   return (bool)(SRSS->unCLK_CAL_CNT1.stcField.u1CAL_COUNTER_DONE); /* 1 = done */ /** (5)
Check the completion of lock calibration counter operation.**/
}
```

Cy_SysClk_ClkMeasurementCountersGetFreq() function

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

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

    volatile uint64_t counter2Value = (uint64_t)SRSS-
>unCLK_CAL_CNT2.stcField.u24CAL_COUNTER2; /** Get the ILO 0 count value.**/

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

    *measuredFreq = CY_SYSCLK_DIV_ROUND(counter2Value * (uint64_t)refClkFreq,
(uint64_t)clk1Count1 ); /** (6) Get the ILO 0 frequency.**/

    return(CY_SYSCLK_SUCCESS);
}
```

6.2.4 ILO0 calibration using the clock calibration counter function

The ILO frequency is determined during manufacturing. Because the ILO frequency varies according to voltage and temperature conditions, the ILO frequency can be updated in the field. The ILO frequency trim can be updated using the ILOx_FTRIM bit of the CLK_TRIM_ILOx_CTL register. The initial value of the ILOx_FTRIM bit is 0x2C. Increasing the value of this bit by 0x01 increases the frequency by 1.5% (typical); decreasing this bit value by 0x01 decreases the frequency by 1.5% (typical). The CLK_TRIM_ILO0_CTL register is protected by WDT_CTL.WDT_LOCK. For the specification of the WDT_CTL register, see the “Watchdog timer” section of the architecture TRM.

Figure 24 shows an example flow of ILO0 calibration using the clock calibration counter and the CLK_TRIM_ILOx_CTL register.

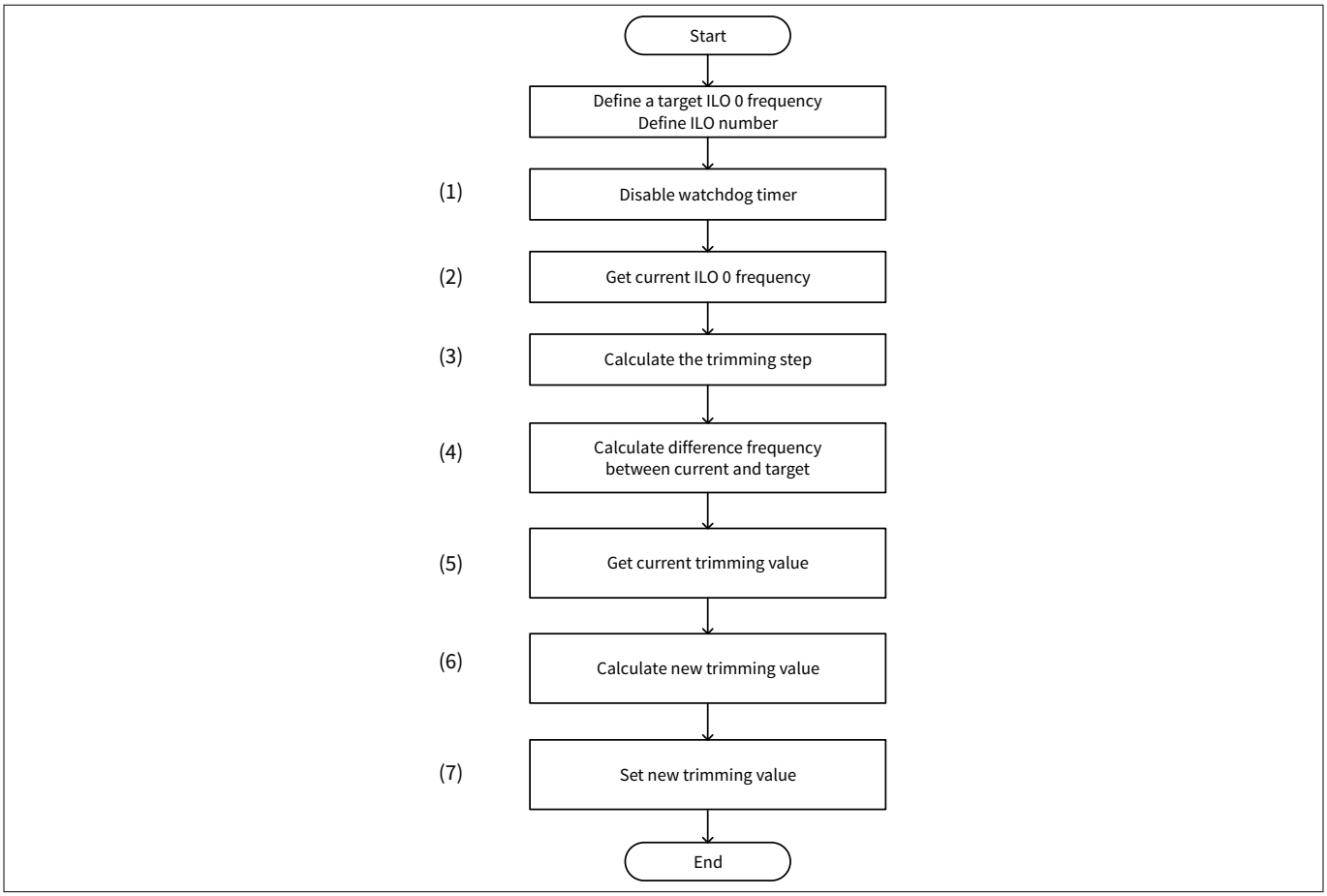

Figure 24 ILO0 calibration

6.2.4.1 Configuration

Table 34 lists the parameters and Table 35 lists the functions of the configuration part of in the SDL for ILO0 calibration using clock calibration counter settings.

Table 34 List of ILO0 calibration using clock calibration counter settings parameters

Parameters	Description	Value
CY_SYSCLK_ILO_TARGET_FREQ	ILO target frequency	32768ul (32.768 kHz)
ILO_0	Define the ILO_0 setting parameter	0ul
ILO_1	Define the ILO_1 setting parameter	1ul
ILONo	Define the measurement clock	ILO_0
iloFreq	Current ILO 0 frequency stored	–

Table 35 List of ILO0 calibration using clock calibration counter settings functions

Functions	Description	Value
`Cy_WDT_Disable()`	WDT disable	–
`Cy_WDT_Unlock()`	Unlock the watchdog timer.	–
`GetILOClockFreq()`	Get the current ILO 0 frequency.	–

(table continues...)

Table 35 (continued) List of ILO0 calibration using clock calibration counter settings functions

Functions	Description	Value
`Cy_SysClk_IloTrim (iloFreq, iloNo)`	Set the trim:	–
	iloFreq: Current ILO 0 frequency	iloFreq: iloFreq
	iloNo: Trimming ILO number	iloNo: ILONo

6.2.4.2 Sample code for the initial configuration of ILO0 calibration using clock calibration counter settings

See the below Code Listings for sample code.

- General configuration of ILO0 calibration using clock calibration counter settings
- Cy_SysClk_IloTrim() function

General configuration of ILO0 calibration using clock calibration counter settings

```
#define CY_SYSCLK_DIV_ROUND(a, b) (((a) + ((b) / 2ull)) / (b)) /** Define the
CY_SYSCLK_DIV_ROUND function.**/

#define CY_SYSCLK_ILO_TARGET_FREQ  32768ul /** Define the target ILO 0 frequency.**/

#define ILO_0   0
#define ILO_1   1
#define ILONo   ILO_0 /** Define the ILO 0 number.**/

int32_t iloFreq;

int main(void)
{

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

    Cy_WDT_Disable(); /** (1) Watchdog timer disable**/
:
    /* return: Frequency of ILO */
    ILOFreq = GetILOClockFreq(); /**(2) Get the current ILO 0 frequency. See GetILOClockFreq()
function.**/
:
    /* Must unlock WDT befor update Trim */
    Cy_WDT_Unlock(); /** (1) Watchdog timer unlock**/
    Trim_diff = Cy_SysClk_IloTrim(ILOFreq,ILONo); /** Trimming the ILO 0. See
Cy_SysClk_IloTrim() function.**/
:
    for(;;);

}
```

6 Supplementary information

Cy_SysClk_IloTrim() function

```
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); /** (3) Calculate the trimming step.**/

    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); /**
(4) Calculate the diff between current and target.**/

    if (diff >= trimStep) /** Check if diff is greater than the trimming step. **/
    {
        if(iloNo == 0u)
        {
            curTrim = SRSS->unCLK_TRIM_ILO0_CTL.stcField.u6ILO0_FTRIM; /** (5) Read the current
trimming value.**/
        }
        else
        {
            curTrim = SRSS->unCLK_TRIM_ILO1_CTL.stcField.u6ILO1_FTRIM; /** (5) Read the current
trimming value.**/
        }

        if (iloFreq > CY_SYSCLK_ILO_TARGET_FREQ) /** Check if the current frequency is smaller
than the target frequency.**/
        { /* iloFreq is too high. Reduce the trim value */
            newTrim = curTrim - CY_SYSCLK_DIV_ROUND(iloFreq - CY_SYSCLK_ILO_TARGET_FREQ,
trimStep);
        } /** (6) Calculate the new trim value.**/
        else
        { /* iloFreq too low. Increase the trim value. */
            newTrim = curTrim + CY_SYSCLK_DIV_ROUND(CY_SYSCLK_ILO_TARGET_FREQ - iloFreq,
trimStep);
        } /** (6) Calculate the new trim value.**/

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




            {
                return(CY_SYSCLK_INVALID_STATE);
            }
            SRSS->unCLK_TRIM_ILO0_CTL.stcField.u6ILO0_FTRIM = newTrim; /** (7) Set the new
```

```
trimming value.**/
        }
        else
        {
            SRSS->unCLK_TRIM_ILO1_CTL.stcField.u6ILO1_FTRIM = newTrim; /** (7) Set the new
trimming value.**/
        }
    }
    return (int32_t)(curTrim - newTrim);
}
```

6.3 CSV diagram and relationship of the monitored clock and reference clocks

Figure 25 shows the clock diagram with the monitored clock and reference clock of CSV. Table 36 shows the relationship between the monitored clock and reference clock.

6 Supplementary information

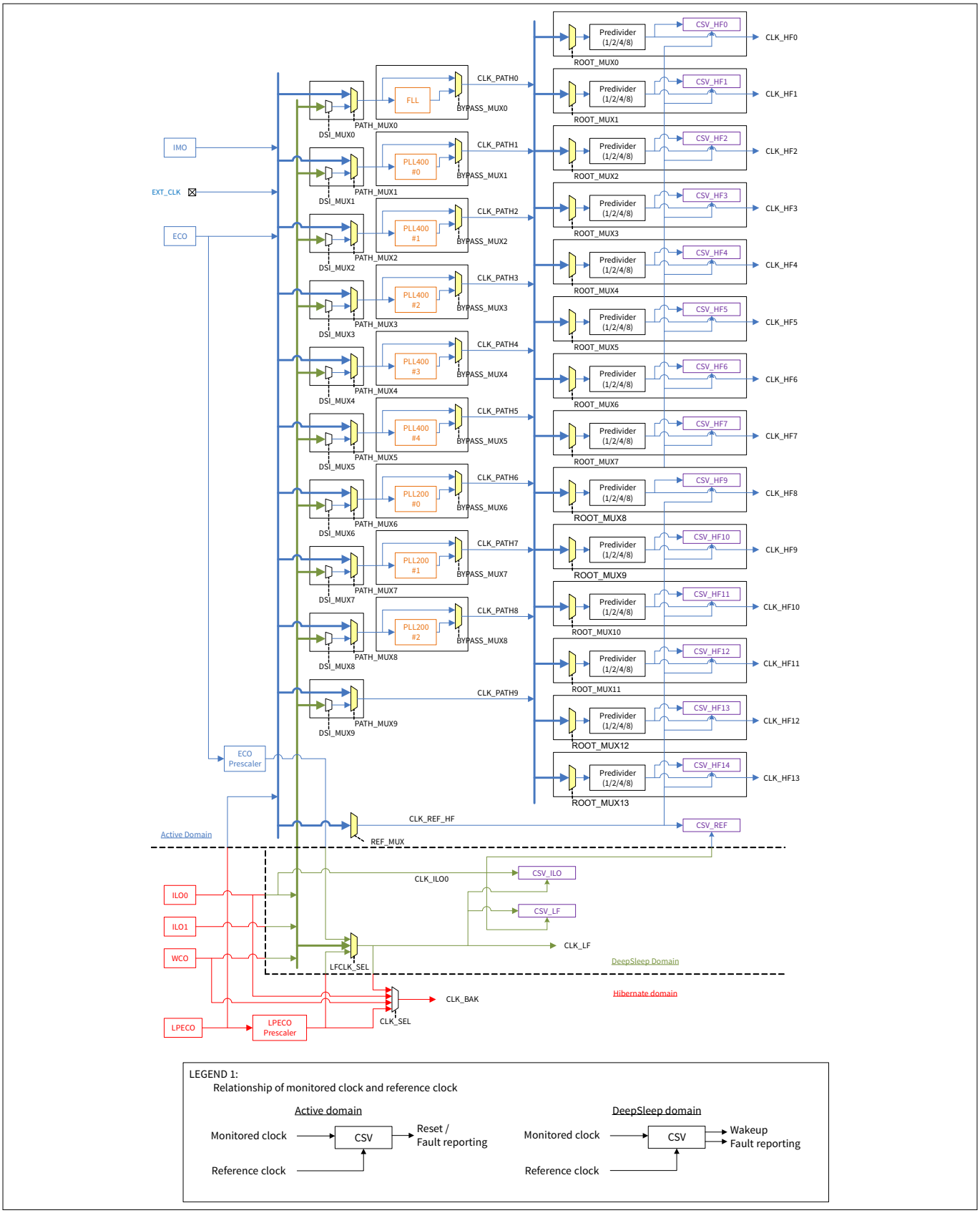

Figure 25 CSV diagram

6 Supplementary information

Table 36 Monitored clock and reference clock

CSV components	Monitor clock	Reference clock	Note
CSV_HF0	CLK_HF0	CLK_REF_HF	CLK_REF_HF is selected from the CLK_IMO, EXT_CLK, CLK_ECO, or CLK_LPECO.
CSV_HF1	CLK_HF1	CLK_REF_HF	CLK_REF_HF is selected from the CLK_IMO, EXT_CLK, CLK_ECO, or CLK_LPECO.
CSV_HF2	CLK_HF2	CLK_REF_HF	CLK_REF_HF is selected from the CLK_IMO, EXT_CLK, CLK_ECO, or CLK_LPECO.
CSV_HF3	CLK_HF3	CLK_REF_HF	CLK_REF_HF is selected from the CLK_IMO, EXT_CLK, CLK_ECO, or CLK_LPECO.
CSV_HF4	CLK_HF4	CLK_REF_HF	CLK_REF_HF is selected from the CLK_IMO, EXT_CLK, CLK_ECO, or CLK_LPECO.
CSV_HF5	CLK_HF5	CLK_REF_HF	CLK_REF_HF is selected from the CLK_IMO, EXT_CLK, CLK_ECO, or CLK_LPECO.
CSV_HF6	CSV_HF6	CLK_REF_HF	CLK_REF_HF is selected from the CLK_IMO, EXT_CLK, CLK_ECO, or CLK_LPECO.
CSV_HF7	CLK_HF7	CLK_REF_HF	CLK_REF_HF is selected from the CLK_IMO, EXT_CLK, CLK_ECO, or CLK_LPECO.
CSV_HF8	CLK_HF7	CLK_REF_HF	CLK_REF_HF is selected from the CLK_IMO, EXT_CLK, CLK_ECO, or CLK_LPECO.
CSV_HF9	CLK_HF7	CLK_REF_HF	CLK_REF_HF is selected from the CLK_IMO, EXT_CLK, CLK_ECO, or CLK_LPECO.
CSV_HF10	CLK_HF7	CLK_REF_HF	CLK_REF_HF is selected from the CLK_IMO, EXT_CLK, CLK_ECO, or CLK_LPECO.
CSV_HF11	CLK_HF7	CLK_REF_HF	CLK_REF_HF is selected from the CLK_IMO, EXT_CLK, CLK_ECO, or CLK_LPECO.
CSV_HF12	CLK_HF7	CLK_REF_HF	CLK_REF_HF is selected from CLK_IMO, EXT_CLK, CLK_ECO, or CLK_LPECO.
CSV_HF13	CLK_HF7	CLK_REF_HF	CLK_REF_HF is selected from the CLK_IMO, EXT_CLK, CLK_ECO, or CLK_LPECO.
CSV_REF	CLK_REF_HF	ILO0(CLK_ILO0)	–
CSV_ILO	ILO0 (CLK_ILO0)	CLK_LF	CLK_LF is selected from the WCO, ILO0, ILO1, ECO_Prescaler, or LPECO_Prescaler.
CSV_LF	CLK_LF	ILO0(CLK_ILO0)	–

7 Glossary

Table 37 Glossary

Terms	Description
CAN FD	CAN FD is the CAN with flexible data rate, and CAN is the controller area network. See the “CAN FD controller” chapter of TRAVEO™ T2G architecture TRM for details.
CLK_FAST_0	Fast clock. The CLK_FAST is used for the CM7 and CPUSS fast infrastructure.
CLK_FAST_1	Fast clock. The CLK_FAST is used for the CM7 and CPUSS fast infrastructure.
CLK_GR	Group clock. The CLK_GR is the clock input to peripheral functions.
CLK_HF	High-frequency clock. The CLK_HF derives both CLK_FAST and CLK_SLOW. CLK_HF, CLK_FAST, and CLK_SLOW are synchronous to each other.
CLK_MEM	Memory clock. The CLK_MEM clocks the CPUSS fast infrastructure.
CLK_PERI	Peripheral clock. The CLK_PERI is the clock source for CLK_SLOW, CLK_GR and peripheral clock divider.
CLK_SLOW	Slow clock. The CLK_FAST is used for the CM0+ and CPUSS slow infrastructure.
Clock Calibration Counter	Clock calibration counter has a function to calibrate the clock using two clocks.
CSV	Clock supervision
CXPI	Clock extension peripheral interface. See the “Clock extension peripheral interface (CXPI)” chapter of the TRAVEO™ T2G architecture TRM for details.
ECO	External crystal oscillator
EXT_CLK	External clock
FLL	Frequency-locked loop
FPU	Floating point unit
ILO	Internal low-speed oscillators
IMO	Internal main oscillator
LIN	Local interconnect network. See the “Local interconnect network (LIN)” chapter of the TRAVEO™ T2G architecture TRM for details.
LPECO	Low-power external crystal oscillator
Peripheral clock divider	The peripheral clock divider derives a clock for use in each peripheral function.
PLL#0	Phase-locked loop. This PLL is not implemented with SSCG and fractional operation.
PLL#1	Phase-locked loop. This PLL is not implemented with SSCG and fractional operation.
PLL#2	Phase-locked loop. This PLL is implemented with SSCG and fractional operation.
PLL#3	Phase-locked loop. This PLL is implemented with SSCG and fractional operation.

(table continues...)

7 Glossary

Table 37 (continued) Glossary

Terms	Description
SAR ADC	Successive approximation register analog-to-digital converter. See the “SAR ADC” chapter of the TRAVEO™ T2G architecture TRM for details.
SCB	Serial communications block. See the “Serial communications block (SCB)” chapter of the TRAVEO™ T2G architecture TRM for details.
SMIF	Serial memory interface. See the “Serial memory interface” chapter of the TRAVEO™ T2G architecture TRM for details.
Sound subsystem	See the “Sound subsystem” chapter of TRAVEO™ T2G architecture TRM for details.
TCPWM	Timer, counter, and pulse width modulator. See the “Timer, counter, and PWM” chapter of the TRAVEO™ T2G architecture TRM for details.
VIDEOSS	Video subsystem. See the “Video subsystem” chapter of the TRAVEO™ T2G architecture TRM for details.
WCO	Watch crystal oscillator

References

The following are the TRAVEO™ T2G family series datasheets and technical reference manuals. Contact Technical Support to obtain these documents.

[1] Device datasheets:

- CYT4DN datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family
- CYT3DL datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family

[2] Technical reference manuals:

- Cluster 2D family
  - TRAVEO™ T2G automotive cluster 2D family architecture technical reference manual (TRM)
  - TRAVEO™ T2G automotive cluster 2D registers technical reference manual (TRM) for CYT4DN
  - TRAVEO™ T2G automotive cluster 2D registers technical reference manual (TRM) for CYT3DL

Other references

A sample driver library (SDL) including startup as sample software to access various peripherals is provided.

The SDL also serves as a reference, to customers, for drivers that are not covered by the official AUTOSAR products.

The SDL cannot be used for production purposes as it does not qualify to any automotive standards. The code snippets in this application note are part of the SDL. Contact Technical Support to obtain the SDL.

Revision history

Document version	Date of release	Description of changes
**	2019-12-12	New application note.
*A	2021-06-07	Updated to Infineon template.
*B	2021-09-07	Updated Configuration of the Clock Resources: Added flowchart and example codes in all instances. Updated Configuration of FLL and PLL: Added flowchart and example codes in all instances. Updated Configuration of the Internal Clock: Added flowchart and example codes in all instances. Removed “Example for Configuring Internal Clock”.
*C	2025-03-19	Corrected PCLK output number of TCPWM0. Added Table for Clock input to Ethernet. Revised Table for Clock input to Sound subsystem. Renamed AUDIOSS(Audio subsystem) to Sound subsystem. Revised Glossary. Added links to References. Template update.

Trademarks

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

Edition 2025-03-19

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-kny1739178854858

Important notice

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.