Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

AN220208

Clock Configuration Setup in Traveo II Body Entry Family

Author: Go Shimada

Associated Part Family: Traveo™ II Family CYT2B Series

Related Documents: see Related Documents

AN220208 describes how to set clock sources and PLL/FLL in Traveo™ II family CYT2B series MCUs. AN220208 also provides examples for setting PLL/FLL, how to calibrate ILO, and supplementary information.

Contents

1 Introduction

Traveo II family MCUs, targeted at automotive systems such as body control units, are 32-bit automotive microcontrollers based on the Arm® Cortex®-M4 processor with FPU and manufactured on an advanced 40-nm process. These products enable a secure computing platform, and incorporate Cypress' low-power flash memory along with multiple high-performance analog and digital functions.

Traveo II Clock System supports both the internal and external clock sources, and supports high-speed clock using PLL and FLL. Traveo II Clock System also supports low-speed clock with internal and external clock. The clock source can also use external oscillator, and Traveo II supports clock input mainly used for RTC.

Traveo II also supports the function to monitor clock operation and to measure the clock difference of each clock.

To understand 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 II family MCU refers to the Body Entry or the CYT2B series.

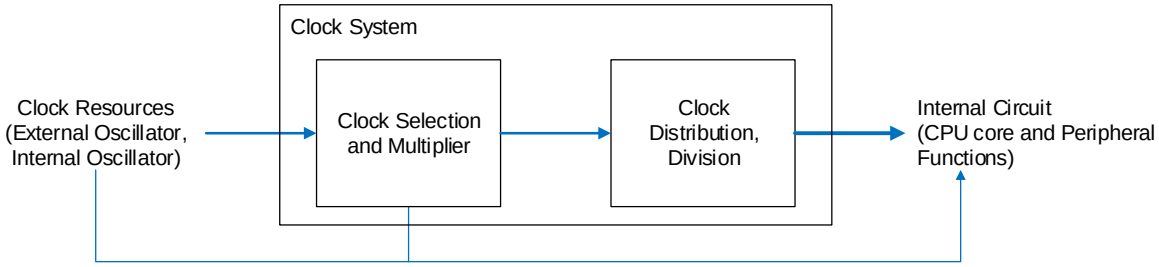
2 Clock System for Traveo II Family MCUs

2.1 Overview of the Clock System

The clock system in this CYT2B series MCU can be divided into two blocks. One block selects the clock resources such as external oscillator and internal oscillator, and multiplies the clock using FLL and PLL. The other block distributes and divides clocks to the CPU core, and peripheral functions. However, there are some exceptions such as RTC, that connect directly from a clock resource to a peripheral circuit.

Figure 1 shows the overview of the clock system structure.

Figure 1. Overview of the Clock System Structure



2.2 Clock Resources

Two kinds of clock resources, internal clock and external clock sources are input to the clock system of this MCU series. There are three types of internal clock and external clock sources:

- Internal clock sources:
  - IMO: Internal Main Oscillator. The IMO is a built-in clock, and its frequency is 8 MHz (TYP). IMO is enabled by default.
  - ILO0: Internal Low-speed Oscillator 0. ILO0 is a built-in clock, and its frequency is 32 kHz (TYP). ILO0 is enabled by default.
  - ILO1: Internal Low-speed Oscillator 1. ILO1 has the same function as ILO0, but ILO1 is available to monitor the clock of ILO0. ILO1 is enabled by default.
- External clock sources:
  - ECO: External Crystal Oscillator. This clock uses an external crystal. Input frequency range is between 3.988 MHz and 33.34 MHz. ECO is disabled by default.
  - WCO: Watch Crystal Oscillator. The WCO is mainly used in RTC. Use a clock frequency of 32.768 kHz. WCO is disabled by default.
  - EXT_CLK: External Clock. The EXT_CLK is a 0.25 MHz to 100 MHz range clock that can be sourced from a signal on a dedicated I/O pin. This clock can be used as the source clock for either PLL or FLL, or can be used directly as the high-frequency clock. EXT_CLK is disabled by default.

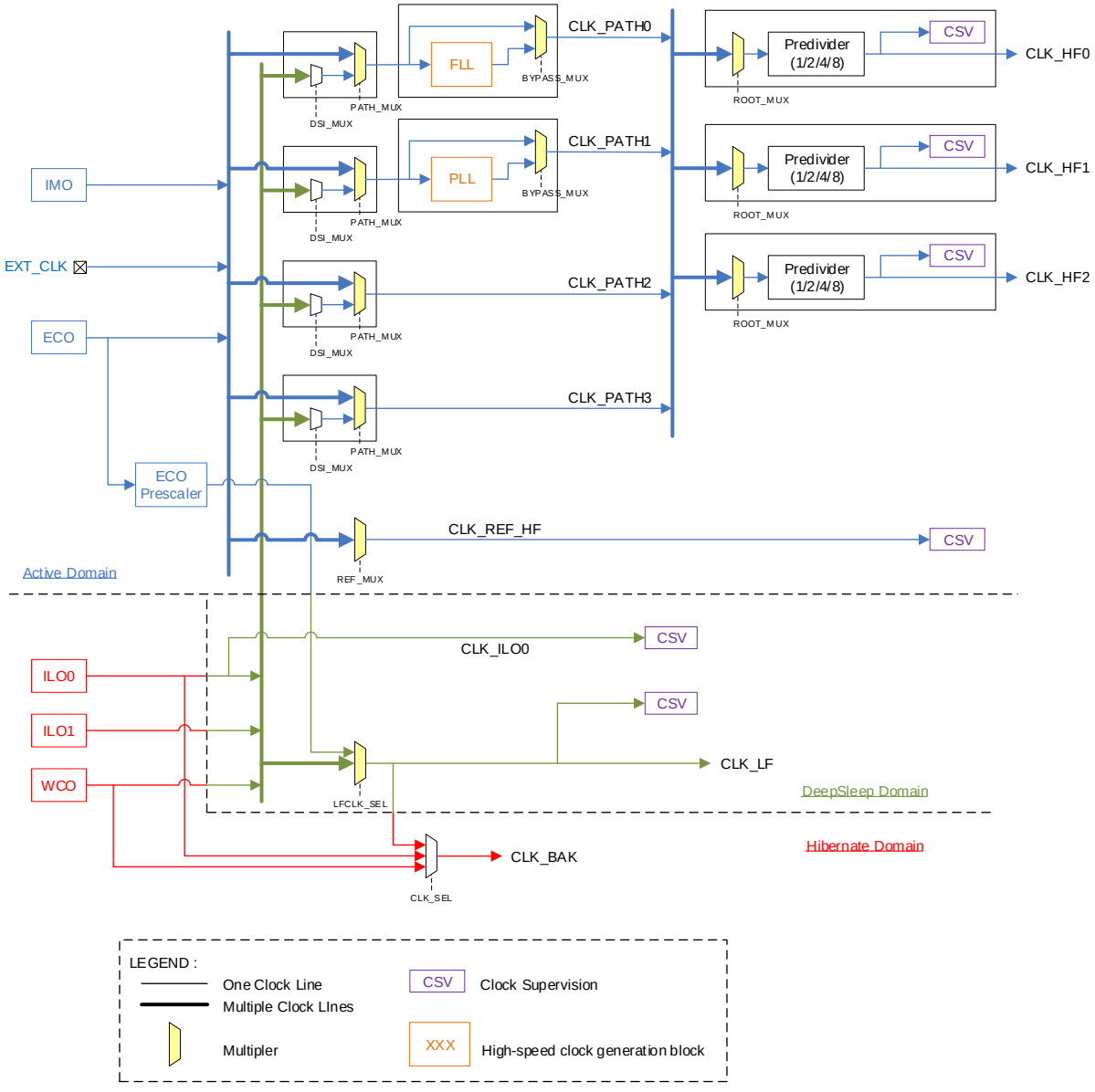
For more details on functions such as IMO, PLL, and so on, and numerical values such as frequency, see the Traveo II Architecture TRM and the Datasheet.

2.3 Functions of Clock System

This section explains the functions of the clock system.

Figure 2 shows the details of the Clock Selection and Multiplier block shown in Figure 1. This block generates CLK_HF0, CLK_HF1, and CLK_HF2 from the clock resources. CLK_HF0, CLK_HF1, and CLK_HF2 are the base clocks for operating this CYT2B series MCU. This block also selects the clock resources, and FLL and PLL to generate high-speed clock.

Figure 2. Block Diagram



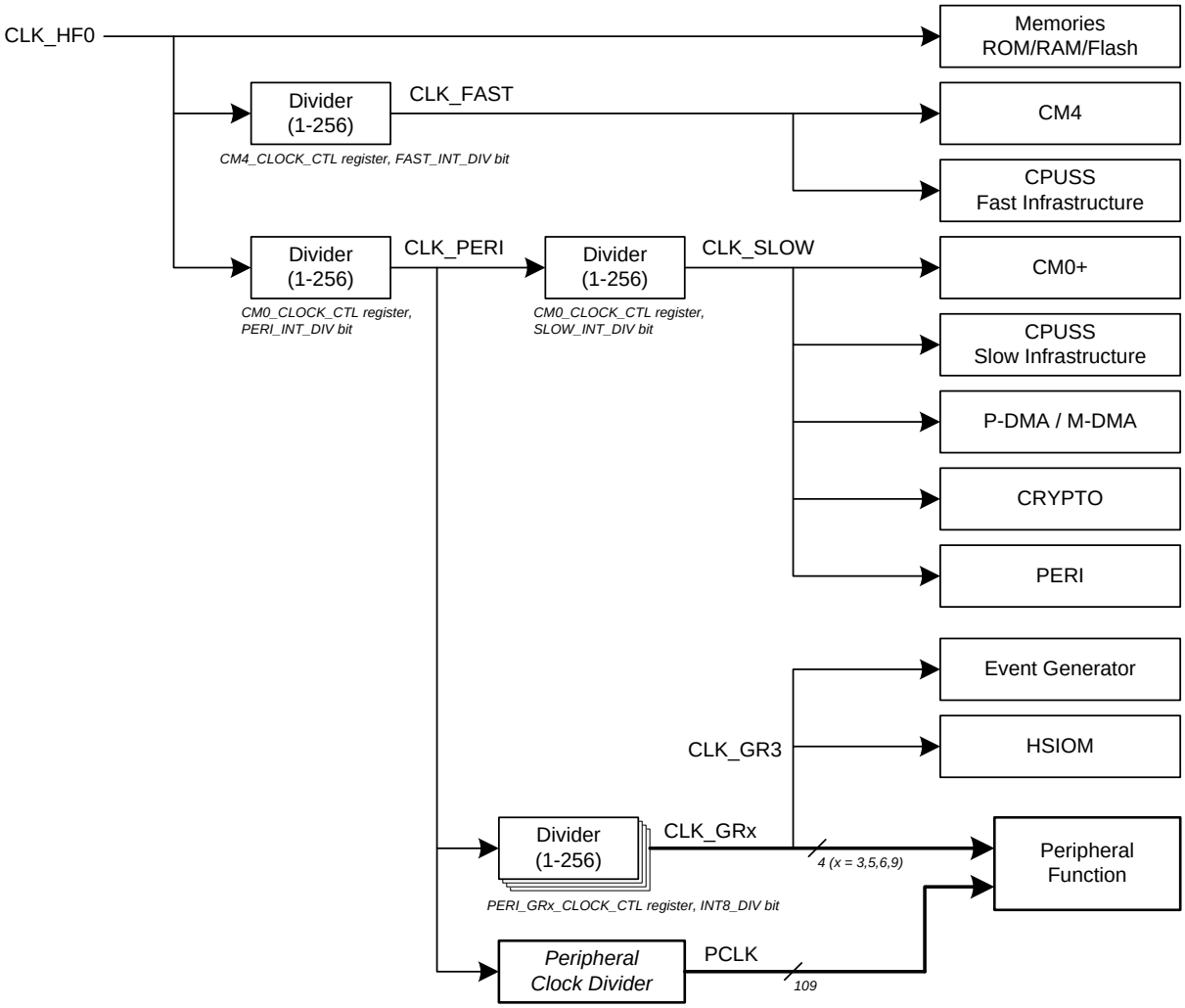
Active Domain	Active Domain is the region for operating only during active power mode.
DeepSleep Domain	DeepSleep Domain is the region for operating only during Active mode and DeepSleep mode.
Hibernate Domain	Hibernate Domain is the region for operating in all Power modes.

ECO Prescaler	ECO_Prescaler divides the ECO and creates a clock that can be used with the LFCLK clock. The division function has a 10-bit integer divider and an 8-bit fractional divider.
DSI_MUX	DSI_MUX has a function to select a clock from ILO0, ILO1, and WCO.
PATH_MUX	PATH_MUX has a function to select a clock from IMO, ECO, EXT_CLK and DSI_MUX outputs.
CLK_PATH	CLK_PATH0, CLK_PATH1, CLK_PATH2, and CLK_PATH3 are used as the input sources for CLK_HF0, CLK_HF1 and CLK_HF2.
CLK_HF	CLK_HF0, CLK_HF1, and CLK_HF2 are high-frequency clocks.
FLL	FLL is a Frequency Locked Loop which can generate high-speed clock.
PLL	PLL is a Phase Locked Loop which can generate high-speed clock.
BYPASS_MUX	BYPASS_MUX has a function to select the clock to be the output of CLK_PATH. It can either choose the output of FLL/PLL or bypass them.
ROOT_MUX	ROOT_MUX has a function to the clock source of CLK_HFx. The clocks that can be selected are CLK_PATH0, CLK_PATH1, CLK_PATH2, and CLK_PATH3.
Predivider	The Predivider is available to divide the selected CLK_PATH. 1, 2, 4, and 8 divisions can be selected.
REF_MUX	REF_MUX selects the CLK_REF_HF clock source.
CLK_REF_HF	CLK_REF_HF monitors CSV of CLK_HF.
LFCLK_SEL	LFCLK_SEL selects the CLK_LF clock source or a ECO divided clock too.
CLK_LF	CLK_LF is the MCWDT source clock.
CLK_SEL	CLK_SEL selects the clock to be input to RTC.
CLK_BAK	CLK_BAK is mainly used by RTC.
CSV	CSV is clock supervision, which monitors the operation of the clock. The clocks that can be monitored are CLK_HFs, CLK_REF_HF, ILO0, and CLK_LF.

Figure 3 shows the distribution of CLK_HF0 and the details of the Clock Distribution and Division block shown in Figure 1.

CLK_HF0 is the root clock for the CPU subsystem (CPUSS) and peripheral clock dividers. For the functions shown in the figure, see the Architecture TRM.

Figure 3. Block Diagram for CLK_HF0



CLK_FAST	CLK_FAST is the clock input for CM4 and CPUSS of the fast infrastructure.
CLK_PERI	CLK_PERI is the clock source for CLK_SLOW, CLK_GR, and peripheral clock divider.
CLK_SLOW	CLK_SLOW is the clock input for CM0+ and CPUSS of the slow infrastructure.
CLK_GR	CLK_GR is the clock input to peripheral functions. CLK_GR is grouped by Clock Gater. CLK_GR has six groups.
Divider	Divider divides each clock and can be configured from 1 to 256 divisions.

Figure 4 shows the distribution of CLK_HF1 and the details of “Clock Distribution, Division” block shown in Figure 1.

CLK_HF1 is an input source for the Event Generator that generates interrupts and triggers. These interrupts and triggers route internal CPU and peripheral signals with the GPIO. Event Generator uses not only the clock of CLK_GR3, but also the clock of CLK_HF1. Figure 4 shows the clock distribution of CLK_HF1 is shown.

For more details on the Event Generator, see the Architecture TRM.

Figure 4. Block Diagram for CLK_HF1

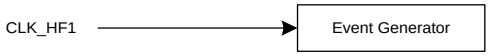


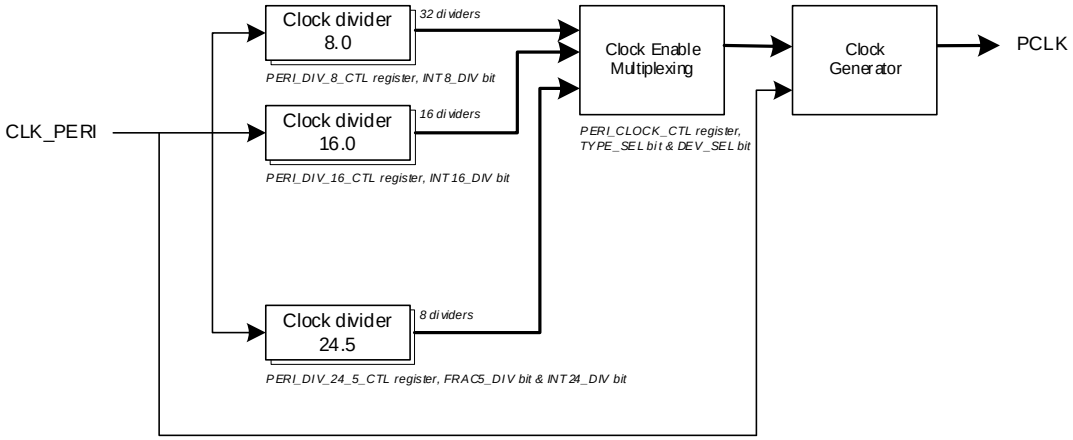
Figure 5 shows the details of the peripheral clock dividers shown in Figure 3.

An operation clock is required for peripheral functions of this CYT2B series MCU, such as the serial communication block (SCB) which is a communication function, and the Timer, Counter, and PWM (TCPWMs) used for waveform output and input signal measurement. These peripherals are clocked by the peripheral clock divider.

This CYT2B series MCU has many peripheral clock dividers to generate PCLK. It has thirty-two 8-bit dividers, sixteen 16-bit dividers, and eight 24.5-bit dividers (24 integer bits, five fractional bits). The output of any of these dividers can be routed to any peripheral.

Figure 5 shows the clock distribution of CLK_PERI. For the functions shown in Figure 5, see the Architecture TRM.

Figure 5. Block Diagram for Peripheral Clock Dividers



Clock divider8.0	Clock divided by 8.
Clock divider16.0	Clock divided by 16.
Clock divider24.5	Clock divided by 24.5.
Clock Enable Multiplexing	Clock Enable Multiplexing enables the signal output from clock divider.
Clock Generator	Clock Generator divides CLK_PERI based on clock divider.

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 Cypress. The code snippets in this application note are part of SDL. See Other References.

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.

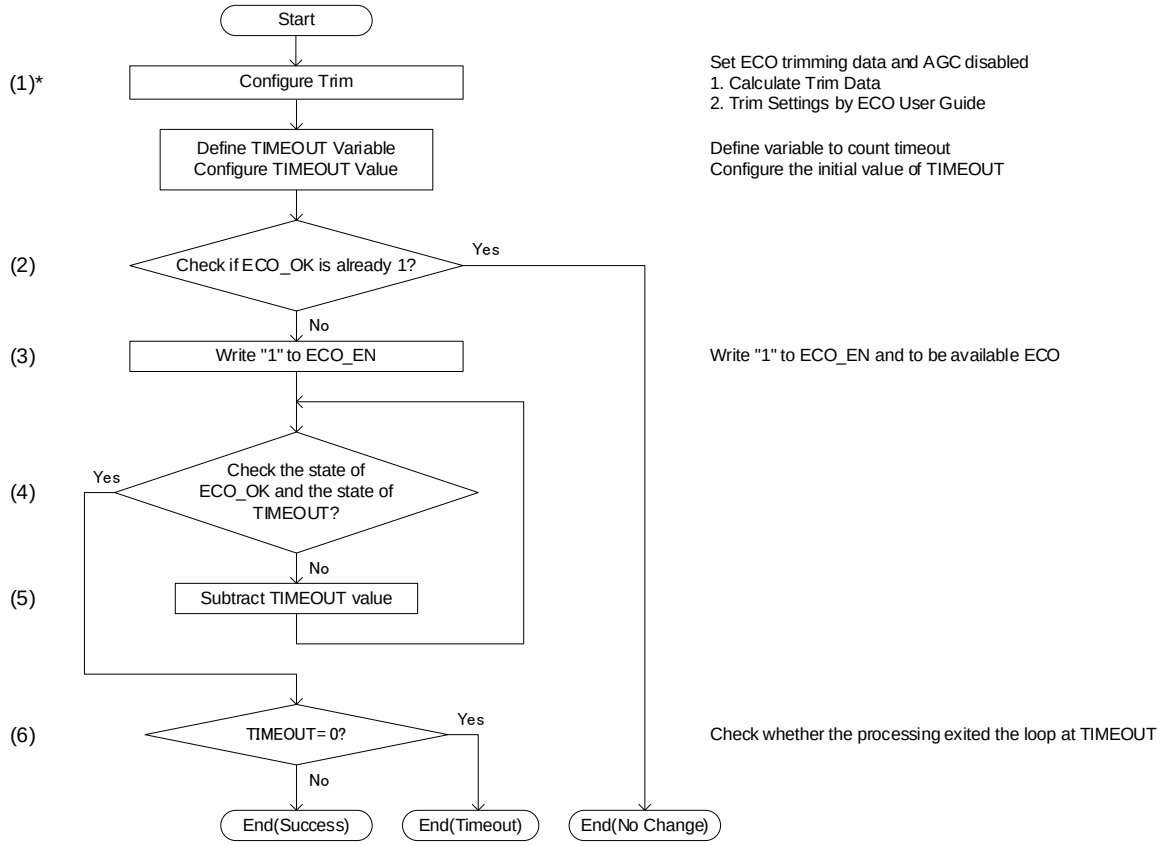
3 Configuring Clock Resources

This section explains how to configure the clock resources.

3.1 Setting 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 set the trimming parameters that control the oscillator according to the crystal unit and ceramic resonator. The method to determine the parameters differs between the crystal unit and ceramic resonator. See the Setting ECO Parameters in Traveo II User Guide for more information.

Figure 6. Enabling ECO



* Use to select the Trimming data that is calculated by software or the data calculated according to the ECO User Guide.

3.1.1 Use Case

- Oscillator to use: Crystal Unit
- Fundamental Frequency: 16 MHz
- Maximum Drive Level: 300.0 uW
- 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 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 II User Guide	7ul
CLK_ECO_CONFIG2.ATRIM	Amplitude trim Calculated from "Setting ECO Parameters" in Traveo II User Guide	0ul
CLK_ECO_CONFIG2.FTRIM	Filter Trim of 3rd harmonic Oscillation Calculated from "Setting ECO Parameters" in Traveo II User Guide	3ul
CLK_ECO_CONFIG2.RTRIM	Feedback resistor trim Calculated from "Setting ECO Parameters" in Traveo II User Guide	3ul
CLK_ECO_CONFIG2.GTRIM	Startup time of the Gain trim Calculated from "Setting ECO Parameters" in Traveo II User Guide	0ul
CLK_ECO_CONFIG.AGC_EN	Automatic Gain Control (AGC) Disabled Calculated from "Setting ECO Parameters" in Traveo II User Guide	0ul [OFF]
WAIT_FOR_STABILIZATION	Waiting for stabilization	10000ul
PLL_PATH_NO	PLL Number	1ul
CLK_FREQ_ECO	Source Clock Frequency	16000000ul
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 Watchdog Timer	-
`Cy_SysClk_FllDisableSequence(Wait Cycle)`	Disable FLL	Wait Cycle = 100ul
`Cy_SysClk_PllDisable(PLL Number)`	Disable PLL	PLL Number = PLL_PATH_NO
`AllClockConfiguration()`	Clock Configuration	-

Functions	Description	Value
`Cy_SysClk_EcoEnable(Timeout value)`	Set ECO Enable and timeout value	Timeout value = WAIT_FOR_STABILIZATION
`Cy_SysLib_DelayUs(Wait Time)`	Delays by the specified number of microseconds	Wait Time = 1u (1us)

3.1.3 Sample Code for Initial Configuration of ECO Settings

There is a sample code as shown Code 1.

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

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

Code 1. General Configuration of ECO Settings

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

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

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

    /* Enable interrupt */
    __enable_irq();

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

Define TIMEOUT Variable

Define oscillator parameters to use for software calculation

Define PLL number

Watchdog Timer disable.

Disable FLL

Disable PLL

Trim and ECO setting. See Code 2.

Code 2. AllClockConfiguration() Function

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

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

(1)-1. Trim settings for software calculation. See Code 4.

(1)-2. Trim settings for ECO User Guide

ECO Enable. See Code 3.

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

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

Code 3. 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 */
    {
        return CY_SYSCLK_INVALID_STATE;
    }

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

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

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

(2) Check if ECO_OK is already enabled.

(3) Write "1" to the ECO_EN bit. And make ECO available

(4) Check the state of ECO_OK and the state of TIMEOUT

(5) Subtract TIMEOUT value

Wait for 1 us.

(6) Check whether the processing exited the loop at TIMEOUT

Code 4. 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)
{
    /* Check if ECO is disabled */
    if(SRSS->unCLK_ECO_CONFIG.stcField.u1ECO_EN == 1ul)
    {
        return(CY_SYSCLK_INVALID_STATE);
    }

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

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

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

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

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

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

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

    ftrim  = Cy_SysClk_SelectEcoFtrim(atrim);

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

    SRSS->unCLK_ECO_CONFIG.stcField.u1AGC_EN = agcen;

    return(CY_SYSCLK_SUCCESS);
}
```

Trim Calculation by software

Get Atrim Value. See Code 5.

Get AGC enable setting. See Code 6.

Get Wdtrim Value. See Code 7.

Get Gtrim Value. See Code 8.

Get Rtrim Value. See Code 9.

Get Ftrim Value. See Code 10.

Set Trim value

Code 5. Cy_SysClk_SelectEcoAtrim () Function

```
__STATIC_INLINE uint32_t Cy_SysClk_SelectEcoAtrim(float32_t maxAmplitude)
{
    if((0.50f <= maxAmplitude) && (maxAmplitude < 0.55f))
    {
        return(0x04ul);
    }
    else if(maxAmplitude < 0.60f)
    {
        return(0x05ul);
    }
    else if(maxAmplitude < 0.65f)
    {
        return(0x06ul);
    }
    else if(maxAmplitude < 0.70f)
    {
        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);
    }
}
```

Get Atrim Value.

Code 6. Cy_SysClk_SelectEcoAGCEN() Function

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

Get AGC enable setting.

Code 7. Cy_SysClk_SelectEcoWDtrim() Function

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

Get Wdtrim Value.

Code 8. Cy_SysClk_SelectEcoGtrim() Function

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

Get Gtrim Value.

Code 9. Cy_SysClk_SelectEcoRtrim() Function

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

Get Rtrim Value.

Code 10. Cy_SysClk_SelectEcoFtrim() Function

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

Get Ftrim Value.

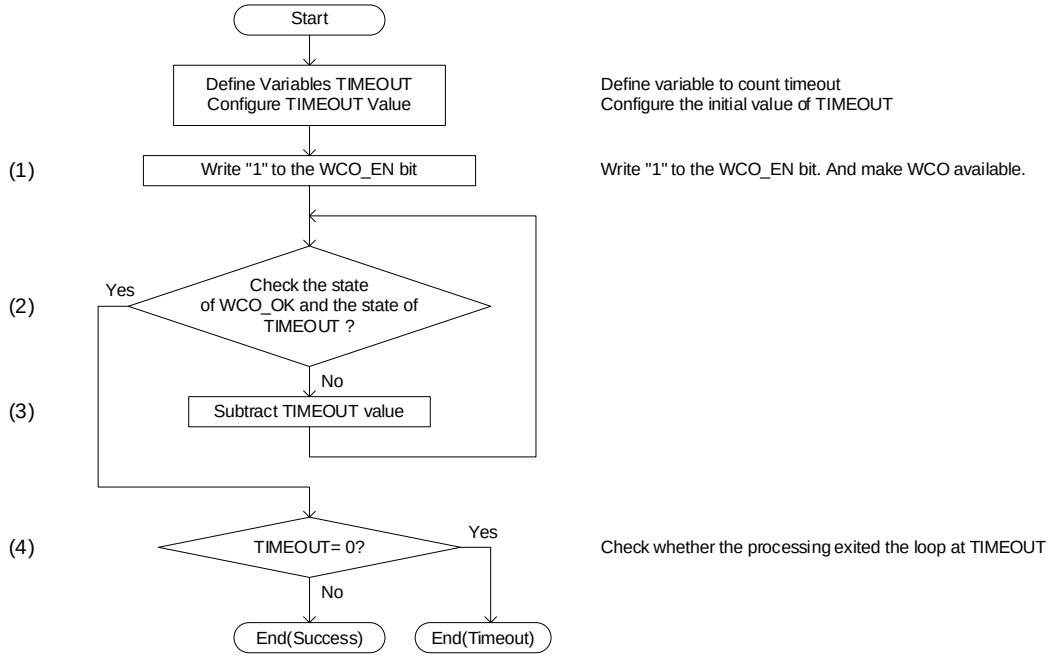
3.2 Setting WCO

3.2.1 Operation Overview

WCO is disabled by default. Accordingly, WCO cannot be used unless it is enabled. Figure 7 shows how to configure registers for enabling WCO.

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

Figure 7. Enabling WCO



3.2.2 Configuration

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

Table 3. List of WCO Settings Parameters

Parameters	Description	Value
WAIT_FOR_STABILIZATION	Waiting for stabilization	10000ul
PLL_PATH_NO	PLL PATH Number	1ul

Table 4. List of WCO Settings Functions

Functions	Description	Value
`Cy_WDT_Disable()`	Disable Watchdog Timer	-
`Cy_SysClk_FllDisableSequence(Wait Cycle)`	Disable FLL	Wait Cycle = 100ul
`Cy_SysClk_PllDisable(PLL Number)`	Disable PLL	PLL Number = PLL_PATH_NO
`AllClockConfiguration()`	Clock Configuration	-
`Cy_SysClk_WcoEnable(Timeout value)`	Set WCO Enable and timeout value	Timeout value = WAIT_FOR_STABILIZATION
`Cy_SysLib_DelayUs(Wait Time)`	Delays by the specified number of microseconds	Wait Time = 1u (1us)

3.2.3 Sample Code for Initial Configuration of WCO Settings

There is a sample code as shown Code 11 to Code 13.

Code 11. General Configuration of WCO Settings

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

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

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

    /* Enable interrupt */
    __enable_irq();

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

Define TIMEOUT Variable

Define PLL number

Watchdog Timer disable.

Disable FLL

Disable PLL

WCO Setting. See Code 12.

Code 12. AllClockConfiguration() Function

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

   return;
}
```

WCO Enable See Code 13.

Code 13. 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;

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

    return (rtnval);
}
```

(1) Write "1" to the WCO_EN bit. And make WCO available

(2) Check the state of WCO_OK and the state of TIMEOUT

Wait for 1 us.

(3) Subtract TIMEOUT Value

(4) Check whether the processing exited the loop at TIMEOUT

3.3 Setting IMO

IMO is enabled by default so that all functions operate properly. IMO will be automatically disabled during Deep Sleep, Hibernate, and XRES. Therefore, you do not need to set IMO.

3.4 Setting ILO0/ILO1

ILO0 and ILO1 are enabled by default.

Note that ILO0 is used as the operating clock for the watchdog timer (WDT). Therefore, if ILO0 is disabled, it is necessary to disable WDT. To disable ILO0, write ‘01b’ to the WDT_LOCK bit of the WDT_CTL register, and then write ‘0’ to the ENABLE bit of the CLK_ILO0_CONFIG register.

4 Configuring FLL and PLL

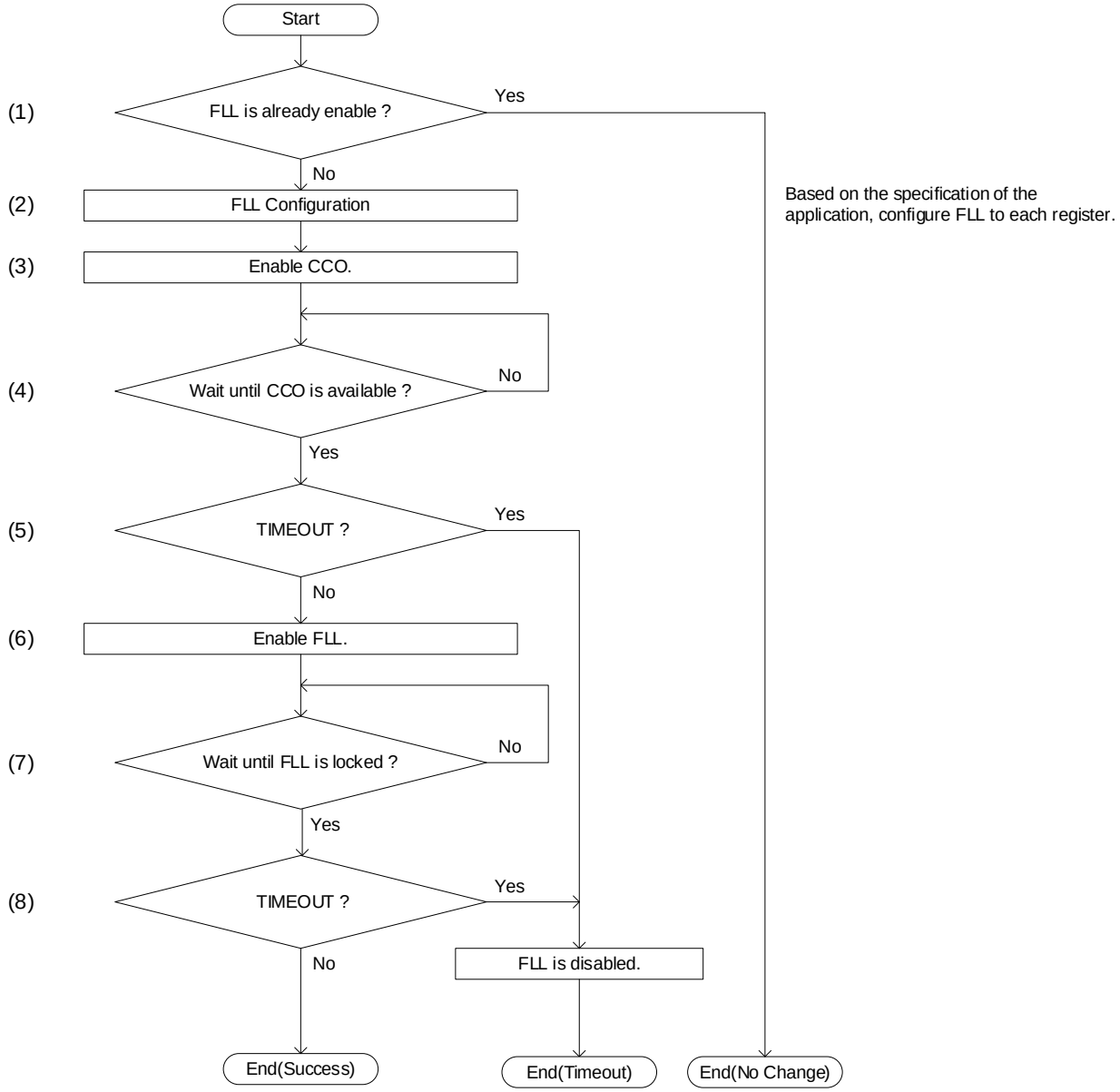
This section shows how to set FLL and PLL in the clock system.

4.1 Setting FLL

4.1.1 Operation Overview

To use FLL, it is necessary to set FLL. FLL has a current-controlled oscillator (CCO), the output frequency of this CCO is controlled by adjusting the trim of the CCO. Figure 8 shows the steps to set FLL.

Figure 8. Procedure for Setting FLL



For details of FLL and FLL setting registers, see the Architecture TRM and Registers TRM.

4.1.2 Use Case

- Input Clock Frequency: 16 MHz (ECO)
- Output Clock Frequency: 100 MHz

4.1.3 Configuration

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

Table 5. List of FLL Settings Parameters

Parameters	Description	Value
WAIT_FOR_STABILIZATION	Waiting for stabilization	10000ul
FLL_PATH_NO	FLL Number	0ul
FLL_TARGET_FREQ	FLL Target Frequency	100000000ul (100 MHz)
CLK_FREQ_ECO	Source Clock Frequency	16000000ul (16 MHz)
PATH_SOURCE_CLOCK_FREQ	FLL Input Frequency	CLK_FREQ_ECO
CY_SYSCLK_FLLPLL_OUTPUT_AUTO	FLL Output Mode CY_SYSCLK_FLLPLL_OUTPUT_AUTO: Automatic using lock indicator. CY_SYSCLK_FLLPLL_OUTPUT_LOCKED_OR_NOTHING: Similar to AUTO, except the clock is gated off when unlocked. CY_SYSCLK_FLLPLL_OUTPUT_INPUT: Select FLL reference input (bypass mode) CY_SYSCLK_FLLPLL_OUTPUT_OUTPUT: Select FLL output. Ignores lock indicator. See SRSS_CLK_FLL_CONFIG3 in Registers TRM for more details.	0ul

Table 6. List of FLL Settings Functions

Functions	Description	Value
`AllClockConfiguration()`	Clock Configuration	-
`Cy_SysClk_FllConfigureStandard(inputFreq, outputFreq, outputMode)`	inputFreq: Input Frequency outputFreq: Output Frequency outputMode: FLL Output Mode	inputFreq = PATH_SOURCE_CLOCK_FREQ, outputFreq = FLL_TARGET_FREQ, 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)`	Delays by the specified number of microseconds	Wait Time = 1u (1us)

4.1.4 Sample Code for Initial Configuration of FLL Settings

There is a sample code as shown Code 14 to Code 18.

Code 14. General Configuration of FLL Settings

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

int main(void)
{
 :
    /* Enable interrupt */
```

Define TIMEOUT Variable

Define FLL Target Frequency.

Define FLL Input Frequency.

Define FLL number

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

FLL setting. See Code 15.

Code 15. 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);
        CY_ASSERT(fllStatus == CY_SYSCLK_SUCCESS);

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

FLL Configuration. See Code 16.

FLL Enable. See Code 18.

Code 16. 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 */
    if (SRSS->unCLK_FLL_CONFIG.stcField.u1FLL_ENABLE != 0ul) /* 1 = enabled */
    {
        return(CY_SYSCLK_INVALID_STATE);
    }
    else if ((outputFreq < CY_SYSCLK_MIN_FLL_OUTPUT_FREQ) || (CY_SYSCLK_MAX_FLL_OUTPUT_FREQ < outputFreq)) /* invalid
output frequency */
    {
        return(CY_SYSCLK_INVALID_STATE);
    }
    else if (((float32_t)outputFreq / (float32_t)inputFreq) < 2.2f) /* check output/input frequency ratio */
    {
        return(CY_SYSCLK_INVALID_STATE);
    }

    /* no error */

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

    cy_stc_fll_manual_config_t config = { 0ul };

    config.outputMode = outputMode;

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

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

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

(1) Check if FLL is already enabled

Check the FLL output range.

Check the FLL frequency ratio.

FLL parameter calculation

```
    {
        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]))
        {
            break;
        }
    }

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

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

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

    return (returnStatus);
}
```

Set FLL registers. See Code 17

Code 17. 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 */
    if (SRSS->unCLK_FLL_CONFIG.stcField.u1FLL_ENABLE != 0ul) /* 1 = enabled */
    {
        returnStatus = CY_SYSCLK_INVALID_STATE;
    }
    else
    { /* return status is OK */
    }

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

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

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

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

    return (returnStatus);
}
```

(1) Check if FLL is already enabled

(2) FLL Configuration

Set CLK_FLL_CONFIG register

Set CLK_FLL_CONFIG2 register

Set CLK_FLL_CONFIG3 register

Set CLK_FLL_CONFIG4 register

Code 18. 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;

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

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

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

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

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

    return(CY_SYSCLK_SUCCESS);
}
```

(3) Enable CCO.

(4) Wait until CCO is available.

(5) Check Timeout.

FLL Disabled if timeout occurs.

Wait for 1 us.

(6) Enable FLL

(7) Wait until FLL is locked.

(8) Check Timeout.

FLL Disabled if timeout occurs.

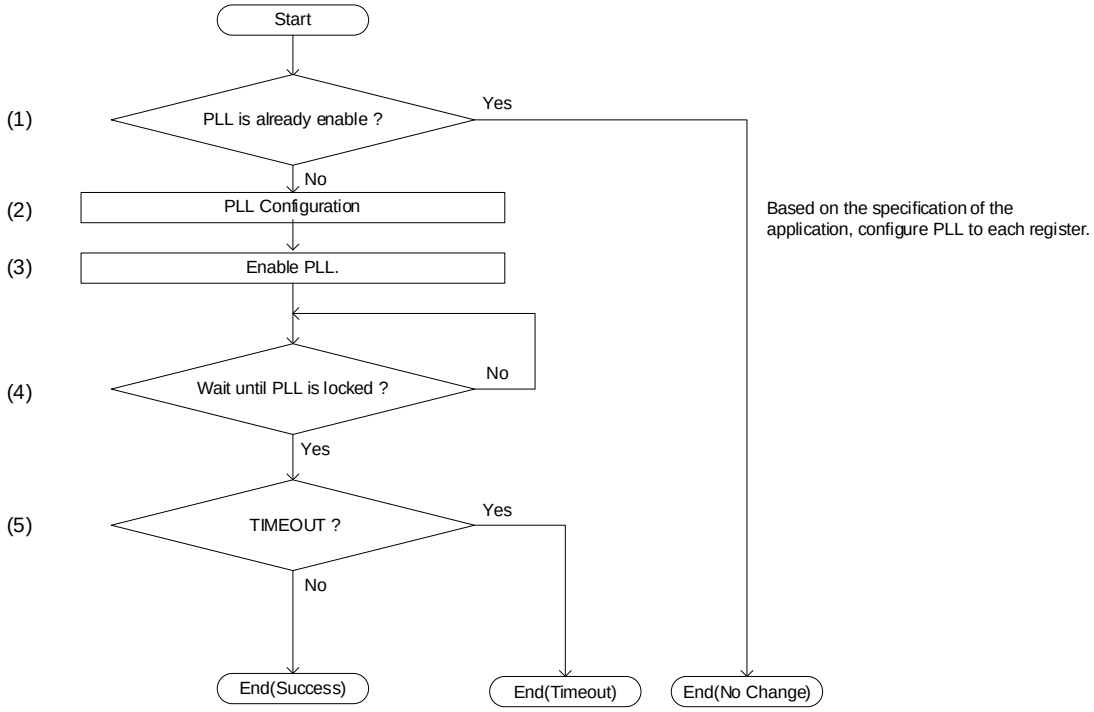
Wait for 1 us.

4.2 Setting PLL

4.2.1 Operation Overview

To use PLL, it is necessary to set PLL. Figure 9 shows the steps to set PLL. For details on PLL, see the Architecture TRM and Registers TRM.

Figure 9. Procedure for Setting PLL



4.2.2 Use case

- Input Clock Frequency: 16 MHz (ECO)
- Output Clock Frequency: 160 MHz
- LF Mode: 200 MHz to 400 MHz (PLL output 320 MHz)

4.2.3 Configuration

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

Table 7. List of PLL Settings Parameters

Parameters	Description	Value
PLL_TARGET_FREQ	PLL Target Frequency	160000000ul (160 MHz)
WAIT_FOR_STABILIZATION	Waiting for stabilization	10000ul
PLL_PATH_NO	PLL Number	1u
CLK_FREQ_ECO	ECO Clock Frequency	16000000ul (16 MHz)
PATH_SOURCE_CLOCK_FREQ	PLL Input Frequency	CLK_FREQ_ECO
CY_SYSCLK_FLLPLL_OUTPUT_AUTO	FLL Output Mode CY_SYSCLK_FLLPLL_OUTPUT_AUTO: Automatic using lock indicator. CY_SYSCLK_FLLPLL_OUTPUT_LOCKED_OR_NOTHING: Similar to AUTO, except the clock is gated off when unlocked. CY_SYSCLK_FLLPLL_OUTPUT_INPUT: Select FLL reference input (bypass mode) CY_SYSCLK_FLLPLL_OUTPUT_OUTPUT: Select FLL output. Ignores lock indicator. See SRSS_CLK_FLL_CONFIG3 in Registers TRM for more details.	0ul
pllConfig.inputFreq	Input PLL Frequency	PATH_SOURCE_CLOCK_FREQ
pllConfig.outputFreq	Outut PLL Frequency	PLL_TARGET_FREQ
pllConfig.lfMode	PLL LF Mode 0: VCO frequency is [200MHz, 400MHz] 1: VCO frequency is [170MHz, 200MHz]	0u (VCO frequency is 320MHz)
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

Table 8. List of PLL Settings Functions

Functions	Description	Value
`AllClockConfiguration()`	Clock Configuration	-
`Cy_SysClk_PllConfigure(PLL Number,PLL Configure)`	Set PLL Path No. and PLL Configure	PLL Number = PLL_PATH_NO, PLL Configure = pllConfig
`Cy_SysClk_PllEnable(PLL Number, Timeout value)`	Set PLL Path No. and Monitor PLL Configure	PLL Number = PLL_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 Configure)`	Set PLL Path No. and PLL Configure	PLL Number = PLL_PATH_NO, PLL Manual Configure = manualConfig

4.2.4 Sample Code for Initial Configuration of PLL settings

There is a sample code as shown Code 19 to Code 23.

Code 19. General Configuration of PLL Settings

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


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

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

Define TIMEOUT Variable

ECO Frequency.

PLL Input Frequency.

PLL Target Frequency.

Define PLL number

PLL Configuration.

PLL setting. See

Code 20. AllClockConfiguration() Function

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

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

   return;
}
```

PLL Configuration. See Code 21.

PLL Enable. See Code 23.

Code 21. Cy_SysClk_PllConfigure() Function

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

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

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

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

Check if the clock path is valid.

(1) Check if PLL is already enabled.

Check the PLL input range.

```
    }

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

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

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

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

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

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

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

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

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

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

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

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

Check the PLL output range.

PLL parameter calculation

Set PLL registers. See Code 22.

Code 22. 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 */
    if ((clkPath == 0ul) || (clkPath > SRSS_NUM_PLL)) /* invalid clock path number */
    {
        return(CY_SYSCLK_BAD_PARAM);
    }

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

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

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

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

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

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

    return (CY_SYSCLK_SUCCESS);
}
```

(2) PLL Configuration

Set CLK_PLL_CONFIG register

Code 23. Cy_SysClk_PllEnable() Function

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

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

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

(3) Enable PLL

(4) Wait until PLL is locked.

(5) Check Timeout.

Wait for 1 us.

5 Configuring Internal Clock

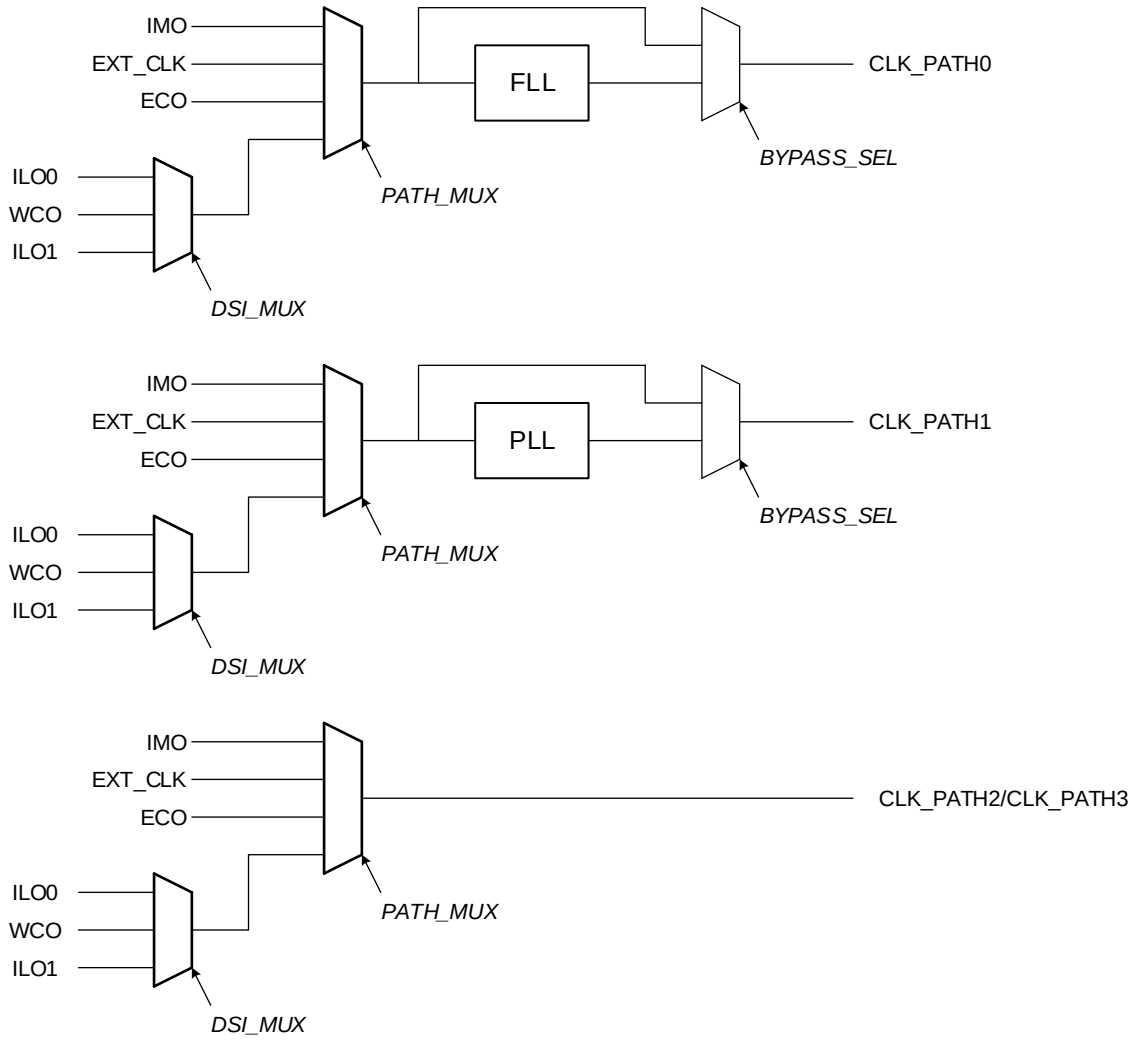
This section explains how to set the internal clock which appears such as a CLK_HF0 and CLK_FAST in the clock system.

5.1 Setting CLK_PATH0, CLK_PATH1, CLK_PATH2, and CLK_PATH3

CLK_PATH0, CLK_PATH1, CLK_PATH2, and CLK_PATH3 are used as the input sources for CLK_HF0, CLK_HF1, and CLK_HF2. CLK_PATH0 and CLK_PATH1 can select all clock resources including FLL and PLL using DSI_MUX and PATH_MUX. CLK_PATH2 and CLK_PATH3 cannot select FLL and PLL, but other clock resources can be selected.

Figure 10 shows the generation block diagram of CLK_PATH0, CLK_PATH1, CLK_PATH2, and CLK_PATH3.

Figure 10. Generation Block for CLK_PATH0, CLK_PATH1, CLKPATH2, and CLK_PATH3



To set CLK_PATH0, CLK_PATH1, CLK_PATH2, and CLK_PATH3, it is necessary to configure DSI_MUX and PATH_MUX. BYPASS_SEL is also required for CLK_PATH0 and CLK_PATH1. Table 9 shows the registers necessary for CLK_PATH. See the Architecture TRM and Registers TRM for more details.

Table 9. Configuring CLK_PATH0, CLK_PATH1, and CLK_PATH2

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
		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[4]
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[4]

5.2 Setting CLK_HF

CLK_HF0, CLK_HF1, and CLK_HF2 can be selected from CLK_PATH0, CLK_PATH1, CLK_PATH2, and CLK_PATH3. A predivider is available to divide the selected CLK_PATH0, CLK_PATH1, CLK_PATH2, and CLK_PATH3. CLK_HF0 is always enabled because it is the source clock for the CPU. It is possible to disable CLK_HF1 and CLK_HF2.

To enable CLK_HF1, write ’1’ to the ENABLE bit of the CLK_ROOT_SELECT register. To disable CLK_HF1 and CLK_HF2, write ‘0’ to the ENABLE bit of the CLK_ROOT_SELECT register.

CLK_PATH0 is the clock output from FLL. CLK_PATH1 is the clock output from PLL. CLK_PATH2 and CLK_PATH3 are the source clock selected by PATH_MUX and DSI_MUX. The ROOT_DIV bit of the CLK_ROOT register sets the predivider values from the options: no division, divide by 2, divide by 4, and or by 8. Figure 11 shows the details of ROOT_MUX and the predivider.

Figure 11. ROOT_MUX and Predivider

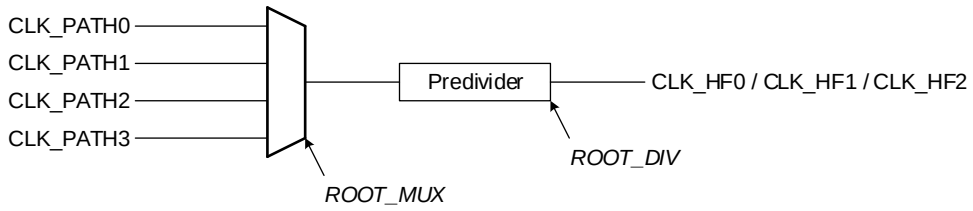


Table 10 shows the registers necessary for CLK_HF0, CLK_HF1, and CLK_HF2. See Architecture TRM and Registers TRM.

[1] Switching automatically according to locked state.

[2] The clock is gated OFF, when unlocked.

[3] In this mode, lock state is ignored.

[4] In this mode, lock state is ignored.

Table 10. Configuring CLK_HF0 and CLK_HF1

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
		Other	Reserved. Do not use.
CLK_ROOT_SELECT	ROOT_DIV[5:4]	0	No division
		1	Divide clock by 2
		2	Divide clock by 4
		3	Divide clock by 8

5.3 Setting CLK_LF

CLK_LF can be selected from WCO, ILO0, ILO1, and ECO_Prescaler. CLK_LF cannot be set when the WDT_LOCK bit in the WDT_CLTL register is disabled, because CLK_LF can select ILO0.

Figure 12 shows the details of LFCLK_SEL.

Figure 12. LFCLK_SEL

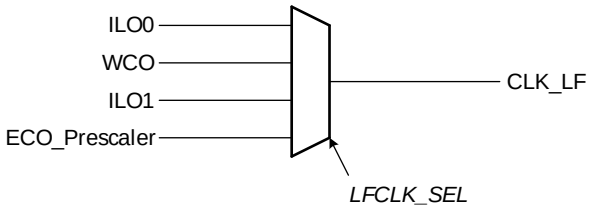


Table 11 shows the registers necessary for CLK_LF. See the Architecture TRM and Registers TRM for more details.

Table 11. Configuring CLK_LF

Register Name	Bit Name	Value	Selected Item
CLK_SELECT	LFCLK_SEL[2:0]	0	ILO0
		1	WCO
		5	ILO1
		6	ECO_Prescaler
		Other	Reserved. Do not use.

5.4 Setting CLK_FAST

CLK_FAST is generated by dividing CLK_HF0 by (x+1). When configuring CLK_FAST, configure value (x = 0..255) to be divided by the FAST_INT_DIV bit of the CM4_CLOCK_CTL register.

5.5 Setting CLK_PERI

CLK_PERI is the clock input to peripheral clock divider. CLK_PERI is generated by dividing CLK_HF0; its frequency is configured by the value obtained by dividing CLK_HF0 by (x+1). When configuring CLK_PERI, configure value (x = 0..255) to be divided by the PERI_INT_DIV bit of the CM0_CLOCK_CTL register.

5.6 Setting CLK_SLOW

CLK_SLOW is generated by dividing CLK_PERI; its frequency is configured by the value obtained by dividing CLK_PERI by (x+1). After configuring CLK_PERI, configure value (x = 0..255) to be divided by the SLOW_INT_DIV bit of the CM0_CLOCK_CTL register.

5.7 Setting CLK_GR

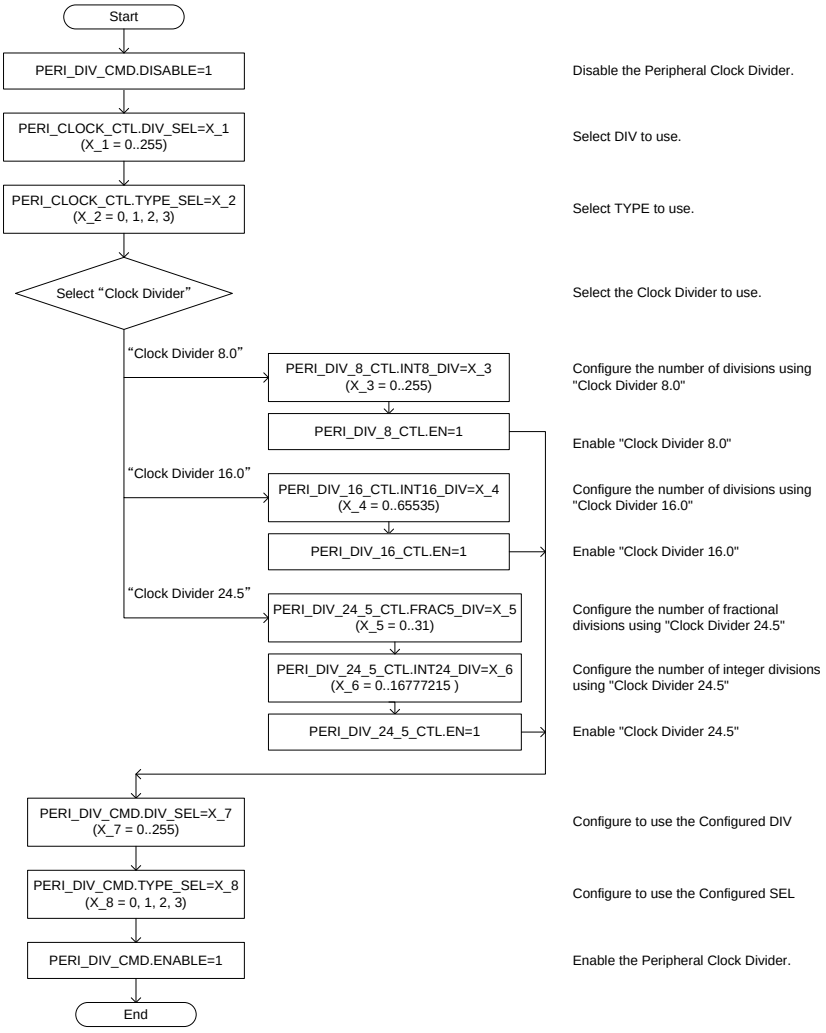
The clock source of CLK_GP is CLK_SLOW in Groups 1 and 2, and CLK_PERI in Groups 3, 5, 6, and 9. Groups 3, 5, 6, and 9 are clocks divided by CLK_PERI. To generate CLK_GR, write the division value (from 1 to 255) to divide the CLOCK_CTL bit of the PERI_GR_CLOCK_CTL register.

5.8 Setting PCLK

Peripheral Clock (PCLK) is a clock that activates each peripheral function. Peripheral clock dividers divide 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 13 shows the steps to set peripheral clock dividers. See the Architecture TRM for more details.

Figure 13. Procedure to Set Generate PCLK



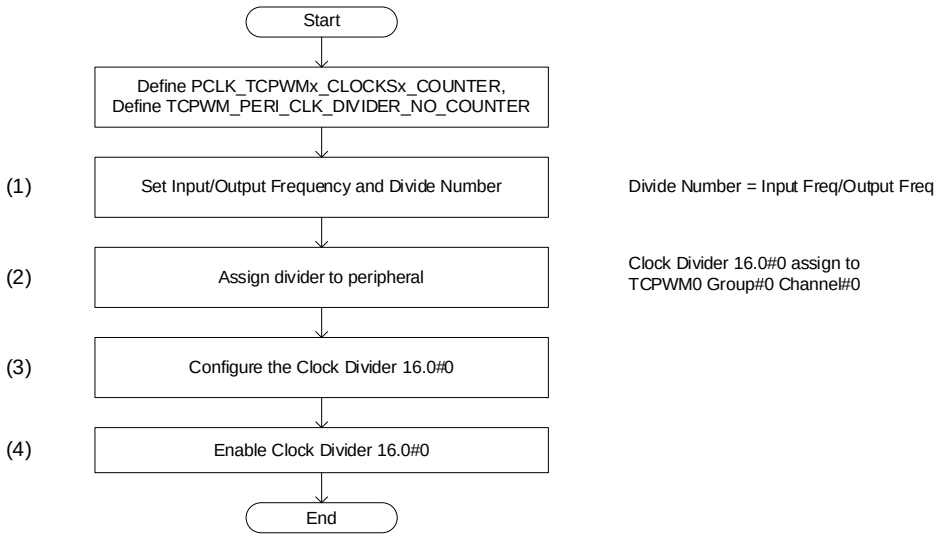
Note: If DIV_SEL is "63" and TYPE_SEL is "3" (default/reset value), no divider is specified and no clock signal(s) are generated.

5.8.1 Example of PCLK Setting

5.8.1.1 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: 31 (TCPWM0, Group#0, Counter#0)

Figure 14. Example Procedure for Setting PCLK



5.8.1.2 Configuration

Table 12 lists the parameters and Table 13 lists the functions of the configuration part of in SDL for PCLK (Example of the TCPWM Timer) settings.

Table 12. List of PCLK (Example of the TCPWM Timer) Settings Parameters

Parameters	Description	Value
PCLK_TCPWMx_CLOCKSx_COUNTER	PCLK of TCPWM0	PCLK_TCPWM0_CLOCKS0 = 31ul
TCPWM_PERI_CLK_DIVIDER_NO_COUNTER	Number of dividers to be used	0ul
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 13. List of PCLK (Example of the TCPWM Timer) Settings Functions

Functions	Description	Value
`Cy_SysClk_PeriphAssignDivider(IPblock,dividerType, dividerNum)`	Assigns 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_PeriphSetDivider(dividerType,dividerNum, dividerValue)`	Set Peripheral Divider	dividerType, = CY_SYSCLK_DIV_16_BIT dividerNum = TCPWM_PERI_CLK_DIVIDER_NO_COUNTER dividerValue = divNum-1ul
`Cy_SysClk_PeriphEnableDivider(dividerType,dividerNum)`	Enable Peripheral Divider	dividerType, = CY_SYSCLK_DIV_16_BIT dividerNum = TCPWM_PERI_CLK_DIVIDER_NO_COUNTER

5.8.2 Sample Code for Initial Configuration of PCLK Settings (Example of the TCPWM Timer)

There is a sample code as shown Code 24 to Code 27.

Code 24. General Configuration of PCLK (Example of the TCPWM Timer) Settings

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

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

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

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

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


    for(;;);
}
```

Define PCLK_TCPWMx_CLOCKSx_COUNTER, Define TCPWM_PERI_CLK_DIVIDER_NO_COUNTER

(1) Set Input/Output Frequency and Divide Number

Calculation of division

Peripheral Divider Assign setting. See Code 25.

Peripheral Divider Enable setting. See Code 27

Peripheral Divider setting. See Code 26

Code 25. 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;
    tempCLOCK_CTL_RegValue.stcField.u2TYPE_SEL = dividerType;
    tempCLOCK_CTL_RegValue.stcField.u8DIV_SEL  = dividerNum;
    PERI->unCLOCK_CTL[ipBlock].u32Register     = tempCLOCK_CTL_RegValue.u32Register;

    return CY_SYSCLK_SUCCESS;

}
```

(2) Assign Divider to Peripheral

Code 26. 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;
        :
    }
    else
    { /* return bad parameter */
        return CY_SYSCLK_BAD_PARAM;
    }

    return CY_SYSCLK_SUCCESS;
}
```

(3) Division Setting to Clock Divider 16.0#0

Code 27. Cy_SysClk_PeriphEnableDivider() Function

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

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

    return CY_SYSCLK_SUCCESS;
}
```

(4) Enable Clock Divider 16#0

Set divider Type Select

Set divider number

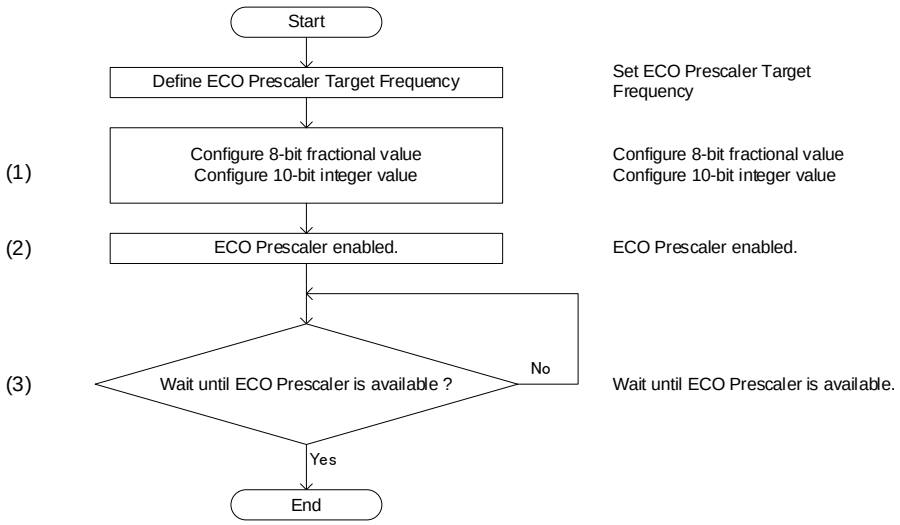
5.9 Setting ECO_Prescaler

5.9.1 Operation Overview

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

Figure 15 shows the steps to enable ECO_Prescaler. below. For details on ECO_Prescaler, see Architecture TRM and Registers TRM.

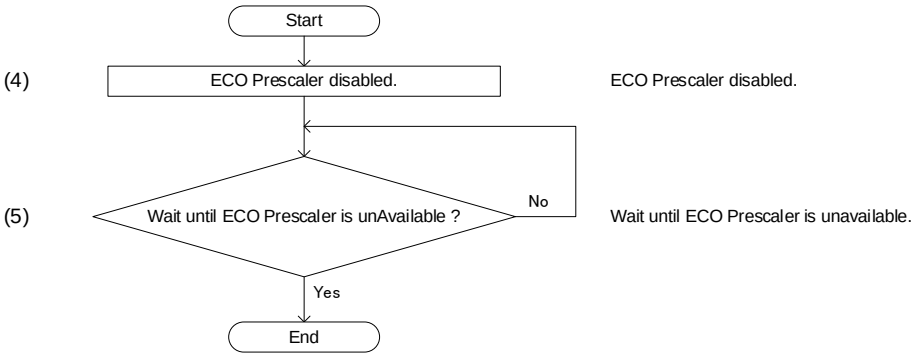
Figure 15. Enabling ECO_Prescaler



Note: Do not change the ECO_FRAC_DIV and ECO_INT_DIV settings when ECO_DIV_ENABLE = 1.

Figure 16 shows the steps to disable ECO_Prescalar. For details on ECO_Prescaler, see Architecture TRM and Registers TRM.

Figure 16. Disabling ECO_Prescaler



5.9.2 Use case

- Input Clock Frequency: 16 MHz
- ECO Prescaler Target Frequency: 1.234567 MHz

5.9.3 Configuration

Table 14 lists the parameters and Table 15 lists the functions of the configuration part of in SDL for ECO Prescaler settings.

Table 14. 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 15. List of ECO Prescaler Settings Functions

Functions	Description	Value
AllClockConfiguration()	Clock Configuration	-
Cy_SysClk_SetEcoPrescale(Inclk, Targetclk)	Set ECO Frequency and Target Frequency	Inclk = PATH_SOURCE_CLOCK_FREQ, Targetclk = ECO_PRESCALER_TARGET_FREQ
Cy_SysClk_EcoPrescaleEnable(Timeout value)	Set 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 prescaler status	-
Cy_SysLib_DelayUs(Wait Time)	Delays by the specified number of microseconds	Wait Time = 1u (1us)

5.9.4 Sample Code for Initial Configuration of ECO Prescaler settings

There is a sample code as shown Code 28 to Code 34.

Code 28. General Configuration of ECO Prescaler Settings

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

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


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

Define ECO Prescaler Target Frequency

Define ECO Clock Frequency

Define TIMEOUT Variable

ECO Prescaler setting. See Code 29.

Code 29. 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);
        CY_ASSERT(ecoPreStatus == CY_SYSCLK_SUCCESS);

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

   return;
}
```

ECO Prescaler setting. See Code 30.

ECO Prescaler enable. See Code 32.

Code 30. 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)
                                   );

    return CY_SYSCLK_SUCCESS;
}
```

Configure ECO Prescaler. See Code 31 .

Code 31. 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;
    tempRegEcoPrescale.stcField.u10ECO_INT_DIV = divInt;
    tempRegEcoPrescale.stcField.u8ECO_FRAC_DIV = divFract;
    SRSS->unCLK_ECO_PRESCALE.u32Register       = tempRegEcoPrescale.u32Register;

    return;
}
```

(1) Configure ECO Prescaler.

Code 32. 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;

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

        Cy_SysLib_DelayUs(1u);

        timeoutus--;
    }

    return CY_SYSCLK_SUCCESS;
}
```

(2) Enable ECO Prescaler

(3) Wait until ECO Prescaler is available. See Code 33

Code 33. 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 prescaler status.

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

Code 34. 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;

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

        Cy_SysLib_DelayUs(1u);

        timeoutus--;
    }

    return CY_SYSCLK_SUCCESS;
}
```

(4) Disable ECO Prescaler.

(5) Wait until ECO Prescaler is unavailable. See Code 33

6 Supplementary Information

6.1 Input Clocks in Peripheral Functions

Table 16 to Table 20 list the clock input to each peripheral function. For detailed values of PCLK, see the Peripheral Clocks section in the Datasheet.

Table 16. Clock Input to TCPWM

Peripheral Function	Operation Clock	Channel Clock
TCPWM (16-bit)	CLK_GR3 (Group 3)	PCLK (PCLK_TCPWM0_CLOCKSx, x=0-62)
TCPWM (16-bit) (Motor Control)		PCLK (PCLK_TCPWM0_CLOCKSy, y=256-267)
TCPWM (32-bit)		PCLK (PCLK_TCPWM0_CLOCKSz, z=512-515)

Table 17. 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)
		Ch2: PCLK (PCLK_CANFD0_CLOCK_CANFD2)
CAN FD1		Ch0: PCLK (PCLK_CANFD1_CLOCK_CANFD0)
		Ch1: PCLK (PCLK_CANFD1_CLOCK_CANFD1)
		Ch2: PCLK (PCLK_CANFD1_CLOCK_CANFD2)

Table 18. 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)
		Ch2: PCLK (PCLK_LIN_CLOCK_CH_EN2)
		Ch3: PCLK (PCLK_LIN_CLOCK_CH_EN3)
		Ch4: PCLK (PCLK_LIN_CLOCK_CH_EN4)
		Ch5: PCLK (PCLK_LIN_CLOCK_CH_EN5)
		Ch6: PCLK (PCLK_LIN_CLOCK_CH_EN6)
		Ch7: PCLK (PCLK_LIN_CLOCK_CH_EN7)

Table 19. 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)
SCB6		PCLK (PCLK_SCB6_CLOCK)
SCB7		PCLK (PCLK_SCB7_CLOCK)

Table 20. Clock Input to SAR ADC

Peripheral Function	Operation Clock	Unit Clock
SAR ADC	CLK_GR9 (Group 9)	Unit0: PCLK (PCLK_PASS_CLOCK_SAR0)
		Unit1: PCLK (PCLK_PASS_CLOCK_SAR1)
		Unit2: PCLK (PCLK_PASS_CLOCK_SAR2)

6.2 Use Case of Clock Calibration Counter Function

6.2.1 How to Use the Clock Calibration Counter

6.2.1.1 Operation Overview

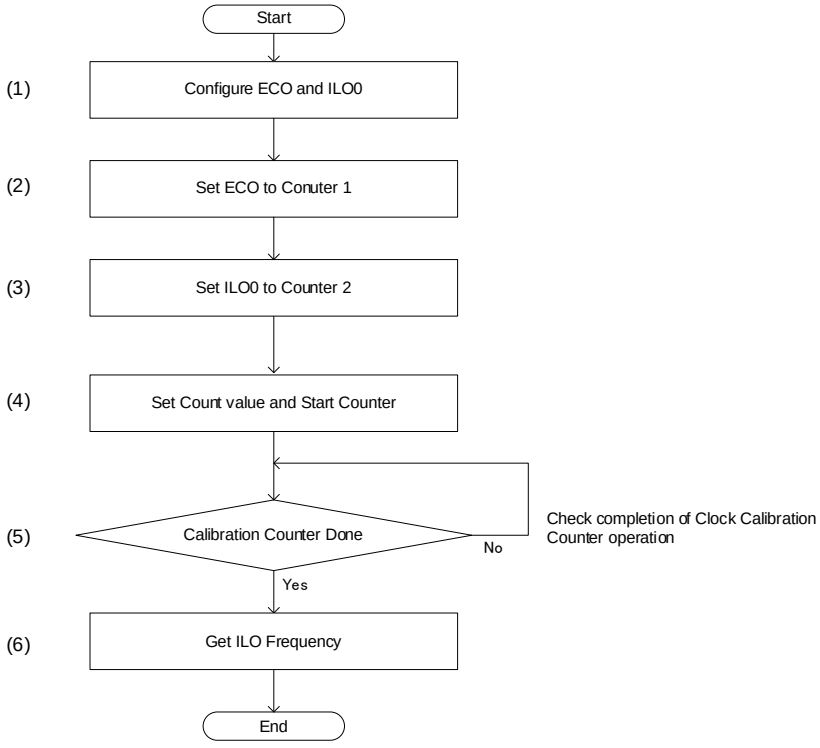
The Clock Calibration Counter has two counters that can be used to compare the frequency of two clock sources. All clock sources are available as a source for these two clocks.

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

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

Figure 17 shows an example of the Clock Calibration Counter function when ILO0 and ECO are used. ILO0 and ECO must be enabled. See Setting ILO0/ILO1 and Setting ECO for ECO and ILO0 configuration.

Figure 17. Example of Clock Calibration Counter with ILO0 and ECO



6.2.1.2 Use case

- Measurement Clock: ILO0 Clock Frequency 32.768 kHz
- Reference Clock: ECO Clock Frequency 16 MHz
- Reference Clock Count Value: 40000ul

6.2.1.3 Configuration

Table 21 lists the parameters and Table 22 lists the functions of the configuration part of in SDL for Clock Calibration Counter with ILO0 and ECO settings.

Table 21. List of Clock Calibration Counter with ILO0 and ECO Settings Parameters

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

Table 22. List of Clock Calibration Counter with ILO0 and ECO Settings Functions

Functions	Description	Value
`GetILOClockFreq()`	Get ILO 0 Frequency	-
`Cy_SysClk_StartClkMeasurementCounters(clk1, count1,clk2)`	Set and Start calibration Clk1: Reference clock Count1: Measurement period Clk2: measurement clock	[Set the counter] clk1 = CY_SYSCLK_MEAS_CLK_ECO = 0x101ul count1 = counter1 clk2 = clockMeasuredInfo[].name
`Cy_SysClk_ClkMeasurementCountersDone()`	Check if the Counter Measurement is done	-
`Cy_SysClk_ClkMeasurementCountersGetFreq (MesauredFreq, refClkFreq)`	Get measurement clock frequency MesauredFreq: Stored measurement clock frequency refClkFreq: Reference clock frequency	MesauredFreq = clockMeasuredInfo[].measuredFreq refClkFreq = CLK_FREQ_ECO

6.2.1.4 Sample Code for Initial Configuration of Clock Calibration Counter with ILO0 and ECO Settings

There is a sample code as shown Code 35.

Code 35. General Configuration of Clock Calibration Counter with ILO0 and ECO Settings

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

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

int32_t ILOFreq;

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

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

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

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

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

Define CY_SYSCLK_DIV_ROUND function

Define measurement clock (ILO0)

(1) ECO and ILO0 setting. See Setting ECO and Setting ILO0/1.

Get Clock Frequency. See Code 36.

Code 36. 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))
    {
       while(1);
    }

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

    while(Cy_SysClk_ClkMeasurementCountersDone() == false);

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

:

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

Check ECO status

Start Clock Measurement Counter. See Code 37.

Check the Counter Measurement is done. See Code 38

Get ILO frequency. See Code 39.

Code 37. 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;
:
    SRSS->unCLK_OUTPUT_SLOW.stcField.u4SLOW_SEL1 = (uint32_t)clock2;
    SRSS->unCLK_OUTPUT_FAST.stcField.u4FAST_SEL1 =  7ul; /*slow_sel1 output*/;
:
    rtnval = CY_SYSCLK_SUCCESS;

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

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

(2) Setting the reference clock (ECO)

(3) Setting the measurement clock (ILO0)

(4) Set Count value and Start Counter

Code 38. 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 completion of Clock Calibration Counter Operation

Code 39. 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;

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

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

    return(CY_SYSCLK_SUCCESS);
}
```

Get ILO 0 Count value

(6) Get ILO 0 Frequency

6.2.2 ILO0 Calibration Using Clock Calibration Counter Function

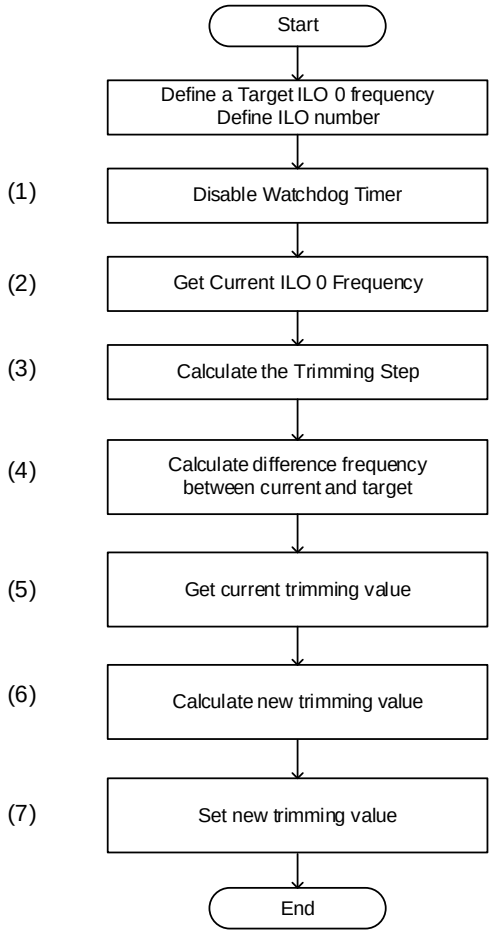
6.2.2.1 Operation Overview

The ILO frequency is determined during manufacturing; however, the ILO frequency can be updated on the field to change according to the voltage and temperature conditions.

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.ENABLE. For the specification of the WDT_CTL register, see the Watchdog Timer section of the Traveo II Architecture TRM.

Figure 18 shows an example flow of ILO0 calibration using Clock Calibration Counter and the CLK_TRIM_ILOx_CTL register.

Figure 18. ILO0 Calibration



6.2.2.2 Configuration

Table 23 lists the parameters and Table 24 lists the functions of the configuration part of in SDL for ILO0 Calibration Using Clock Calibration Counter settings.

Table 23. 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 ILO_0 setting parameter	0ul
ILO_1	Define ILO_1 setting parameter	1ul
ILONo	Define measurement clock	ILO_0
iloFreq	Current ILO 0 frequency stored	-

Table 24. List of ILO0 Calibration Using Clock Calibration Counter Settings Functions

Functions	Description	Value
`Cy_WDT_Disable ()`	WDT disable	-
`Cy_WDT_Unlock()`	Unlocks the Watchdog Timer	-
`GetILOClockFreq()`	Get current ILO 0 frequency.	-

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

6.2.2.3 Sample Code for Initial Configuration of ILO0 Calibration Using Clock Calibration Counter settings

There is a sample code as shown Code 40.

Code 40. General Configuration of ILO 0 Calibration

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

#define CY_SYSCLK_ILO_TARGET_FREQ  32768ul

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

int32_t iloFreq;

int main(void)
{

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

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

}
```

Define CY_SYSCLK_DIV_ROUND function

Define Target ILO 0 frequency

Define ILO 0 nmber

(1) Watchdog Timer disable.

(2) Get Current ILO 0 frequency. See Code 36

Watchdog timer unlock

Trimming the ILO 0. See Code 41

Code 41. 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);

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

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

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

(3) Calculate Trimming Step

(4) Calculate Diff between current and target

Check if diff is greater than trimming step.

(5) Read current trimming value

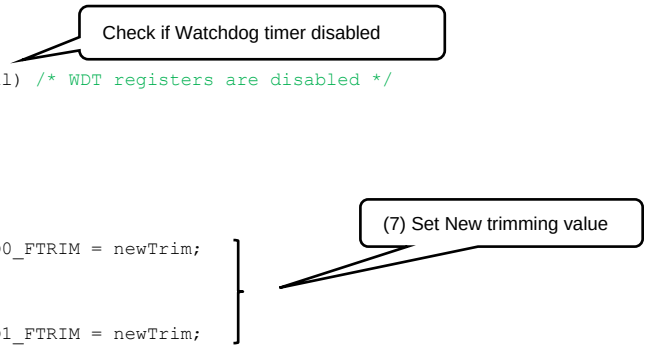
Check if current frequency is smaller than target frequency.

(6) Calculate new trim value.

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



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



7 Glossary

Terms	Description
FPU	Floating Point Unit
RTC	Real Time Clock
IMO	Internal Main Oscillator
ILO	Internal Low-speed Oscillators
ECO	External Crystal Oscillator
WCO	Watch Crystal Oscillator
EXT_CLK	External Clock
PLL	Phase Locked Loop
FLL	Frequency Locked Loop
CLK_HF	High Frequency Clock. The CLK_HF derive both CLK_FAST and CLK_SLOW. CLK_HF, CLK_FAST, and CLK_SLOW are synchronous to each other.
CLK_FAST	Fast Clock. The CLK_FAST is used for the CM4 and CPUSS Fast Infrastructure.
CLK_SLOW	Slow Clock. The CLK_FAST is used for the CM4 and CPUSS Slow Infrastructure.
CLK_PERI	Peripheral Clock. The CLK_PERI is the clock source for CLK_SLOW, CLK_GR, and peripheral clock divider.
CLK_GR	Group Clock. The CLK_GR is the clock input to peripheral functions.
Peripheral Clock Divider	Peripheral clock divider derives a clock to use of each peripheral function.
MCWDT	Multi-Counter Watchdog Timer. See Watchdog Timer chapter of Traveo II Architecture TRM for details.
TCPWM	Timer, Counter, and Pulse Width Modulator. See the Timer, Counter, and PWM chapter of Traveo II Architecture TRM for details.
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 II Architecture TRM for details.
LIN	Local Interconnect Network. See the Local Interconnect Network (LIN) chapter in Traveo II Architecture TRM for details.
SCB	Serial Communications Block. See the Serial Communications Block (SCB) chapter in Traveo II Architecture TRM for details.
SAR ADC	Successive Approximation Register Analog-to-Digital Converter. See the SAR ADC chapter in Traveo II Architecture TRM for details.
Clock Calibration Counter	Clock Calibration Counter has a function to calibrate the clock using two clocks.

8 Related Documents

- Technical Reference Manuals
  - Traveo II Automotive Body Controller Entry Family Architecture Technical Reference Manual (Contact Technical Support)
  - Traveo II Automotive Body Controller Entry Registers Technical Reference Manual (Contact Technical Support)
- Datasheet
  - Traveo II Device Datasheet (Contact Technical Support)
- User Guide
  - Setting ECO Parameters in Traveo II User Guide (Contact Technical Support)

9 Other References

A Sample Driver Library (SDL) including startup as sample software to access various peripherals is provided. 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 because it does not qualify to automotive standards. The code snippets in this application note are part of the SDL. Contact Technical Support to obtain the SDL.

Document History

Document Title: AN220208 – Clock Configuration Setup in Traveo II Body Entry Family

Document Number: 002-20208

Revision	ECN	Submission Date	Description of Change
**	6100869	04/17/2019	New application note.
*A	7036349	12/03/2020	Added flowchart and example codes

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community | Code Examples | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
An Infineon Technologies Company
198 Champion Court
San Jose, CA 95134-1709