

Clock configuration setup in TRAVEO™ T2G family CYT2C series

About this document

Scope and purpose

AN229513 describes how to set up clock sources in TRAVEO™ T2G family CYT2C series MCUs, and provides examples including configuring PLL/FLL and calibrating the ILO.

Table of contents

About this document	1
Table of contents	1
1 Introduction	3
2 TRAVEO™ T2G clock system	4
2.1 Clock resources	4
2.2 Clock system functions	4
2.2.1 Clock selection and multiplier	4
2.2.2 High-frequency clock (CLK_HF0)	7
2.2.3 Peripheral clock divider (CLK_PERI)	8
2.2.4 High-frequency clocks (CLK_HF1 to CL_HF5)	9
2.3 Basic clock system settings	9
3 Configuring the clock resources	10
3.1 ECO	10
3.1.1 Use case	10
3.1.2 Configuration	11
3.1.3 Sample code for ECO initial configuration	12
3.2 WCO	18
3.2.1 Configuration	18
3.2.2 Sample code for WCO initial configuration	19
3.3 IMO	20
3.4 ILO0/ILO1	20
3.5 LPECO	20
3.5.1 Use case	21
3.5.2 Sample code for LPECO initial configuration	21
4 Configuration of FLL and PLL	24
4.1 FLL	24
4.1.1 Use case	24
4.1.2 Configuration	25
4.1.3 Sample code for the initial FLL configuration	26
4.2 PLL	29
4.2.1 Use case	31
4.2.2 Configuration	31
4.2.3 Sample code for PLL initial configuration	36
5 Configuring the internal clocks	44
5.1 CLK_PATHx	44

Table of contents

5.2	CLK_HF _x	45
5.3	CLK_LF.....	46
5.4	CLK_FAST.....	46
5.5	CLK_PERI.....	46
5.6	CLK_SLOW.....	46
5.7	CLK_GR.....	47
5.8	PCLK.....	47
5.8.1	PCLK configuration example.....	48
5.8.1.1	Use case.....	48
5.8.1.2	Configuration.....	48
5.8.2	Sample code for PCLK initial configuration (TCPWM timer example).....	49
5.9	ECO_Prescaler.....	51
5.9.1	Use case.....	52
5.9.2	Configuration.....	52
5.9.3	Sample code for ECO prescaler initial configuration.....	52
5.10	LPECO_Prescaler.....	55
5.10.1	Use case.....	55
5.10.2	Configuration.....	56
5.10.3	Sample code for the initial configuration of LPECO prescaler settings.....	56
6	Supplementary information.....	59
6.1	Input clocks in peripheral functions.....	59
6.2	Clock calibration counter.....	61
6.2.1.1	Use case.....	62
6.2.1.2	Configuration.....	62
6.2.1.3	Sample code for initial configuration of clock calibration counter with ILO0 and ECO.....	63
6.3	CSV diagram, monitored clock, and reference clock.....	65
6.3.1	ILO0 calibration using clock calibration counter.....	66
7	Glossary.....	68
8	References.....	69
9	Other references.....	70
	Revision history.....	71

Introduction

1 Introduction

TRAVEO™ T2G family CYT2C series MCUs, targeted at automotive systems such as instrument clusters, are 32-bit automotive microcontrollers based on the Arm® Cortex®-M4 processor with FPU (single 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.

TRAVEO™ T2G clock system supports high-speed and low-speed clocks using both internal and external clock sources, including an internal real-time clock (RTC). TRAVEO™ T2G supports phase-locked loop (PLL) and frequency-locked loop (FLL) to generate clocks that operate the internal circuit at a high speed.

TRAVEO™ T2G also can monitor the clock operation and measure the clock difference with a known clock.

See the “Clocking system” chapter in the [architecture technical reference manual \(TRM\)](#). In this document, TRAVEO™ T2G family MCUs refer to CYT2C series.

TRAVEO™ T2G clock system

2 TRAVEO™ T2G clock system

The clock system in this series of MCUs is 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 cores and other peripheral functions. However, there are some exceptions where the RTC can connect directly to a clock resource.

See the TRAVEO™ T2G [architecture TRM](#) and [datasheet](#) for more details.

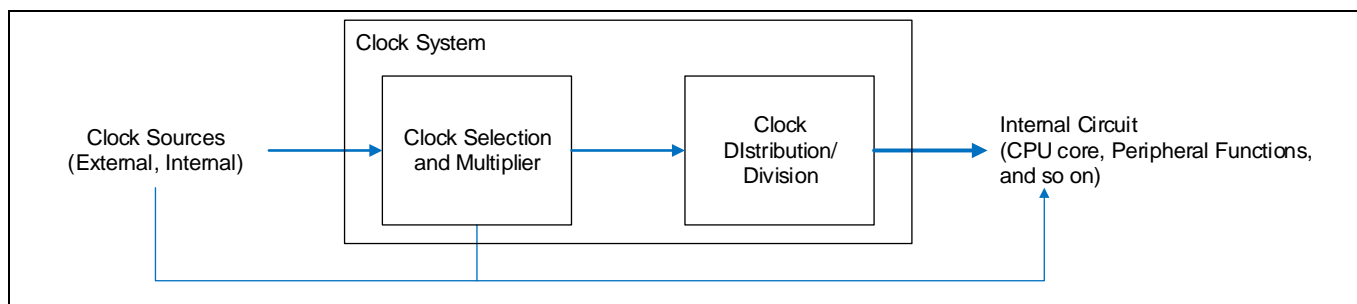


Figure 1 Clock system overview

2.1 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): Built-in clock at 8 MHz (typ.).
 - Internal low-speed oscillator 0 (ILO0): Built-in clock at 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 (0.25 MHz to 100 MHz) can be sourced from a signal on a designated I/O pin. This clock can be used as the source clock for PLL or FLL, or can be used directly by high-frequency clocks.
 - Low-power external crystal oscillator (LPECO): This clock uses an external oscillator. Input frequency range is between 4 MHz and 8 MHz. LPECO can be regarded as an ECO operating in low-power mode.

2.2 Clock system functions

2.2.1 Clock selection and multiplier

This block generates root frequency clocks (CLK_HF0 to CLK_HF6) from clock sources. This block can select one of the supported clock resources, FLL, or PLL to generate the required high-speed clocks. These MCUs support two types of PLLs: PLL without spread spectrum clock generation (SSCG) and fractional operation (PLL200#x), and PLL with SSCG and fractional operation (PLL400#x).

TRAVEO™ T2G clock system

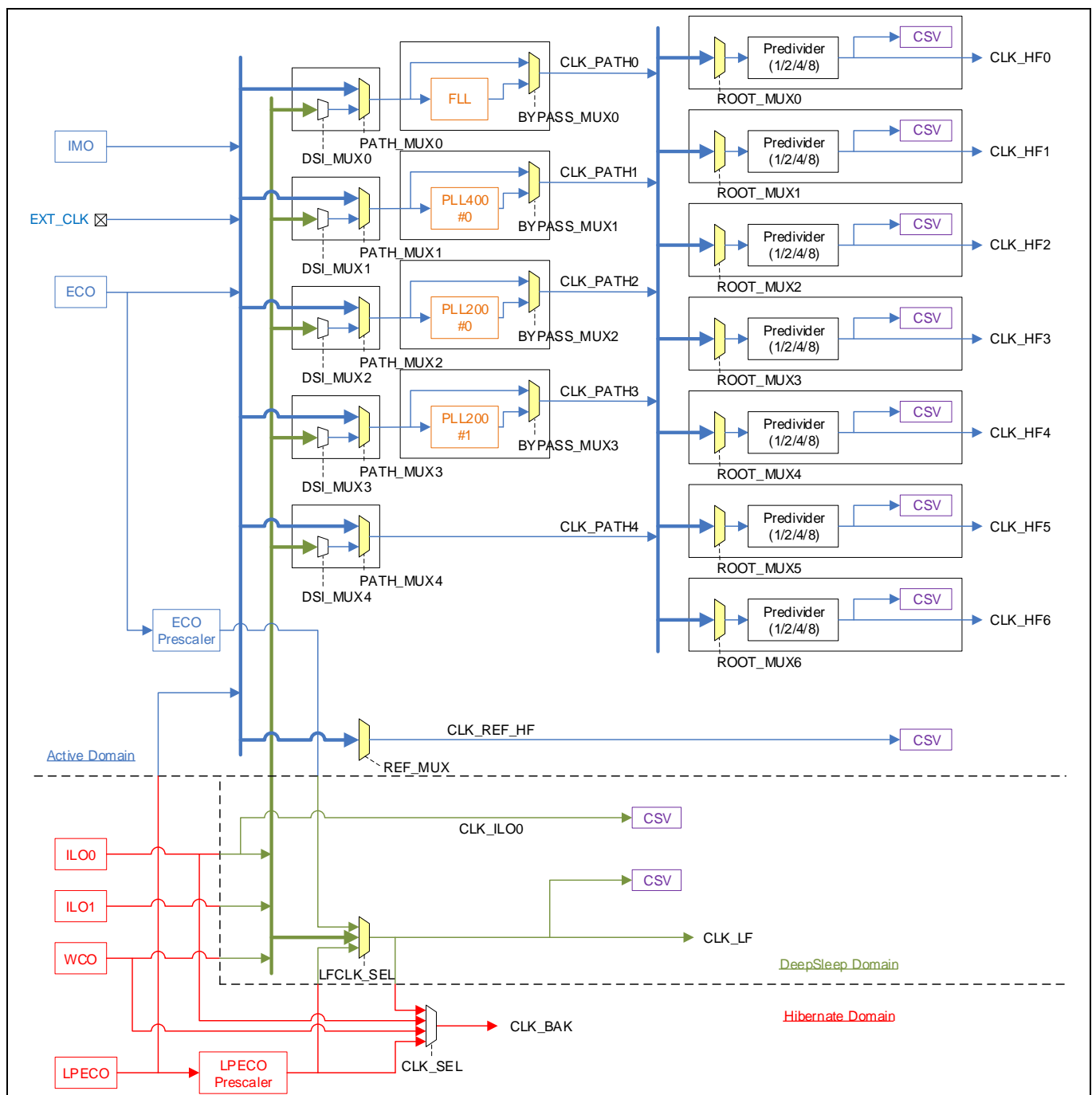


Figure 2 Clock selection and multiplier block

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

TRAVEO™ T2G clock system

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. Two types of PLL: <ul style="list-style-type: none">• PLL200 without SSCG and fractional operation• PLL400 with SSCG and fractional operation
BYPASS_MUX	Selects the clock to be output to CLK_PATH. In 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 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 CSV of CLK_HF
LFCLK_SEL	Selects the CLK_LF clock source
CLK_LF	MCWDT source clock
CLK_SEL	Selects the clock to be input to RTC
CLK_BAK	Mainly input to RTC
CSV	Clock supervision to monitor clock operation

TRAVEO™ T2G clock system

CLK_SLOW	Clock input to CM0+, CPUSS of slow infrastructure and SMIF
CLK_GR	Clock input to peripheral functions. CLK_GR is grouped by the clock gater. CLK_GR has six groups.
PCLK	Peripheral clock used in peripheral functions. PCLK can be configured each channel of IPs independently and select one divider to generate PCLK.
Divider	Divider has a function to divide each clock. It can be configured from 1 division to 256 divisions.

2.2.3 Peripheral clock divider (CLK_PERI)

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

This peripheral clock divider has many clock dividers to generate 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.

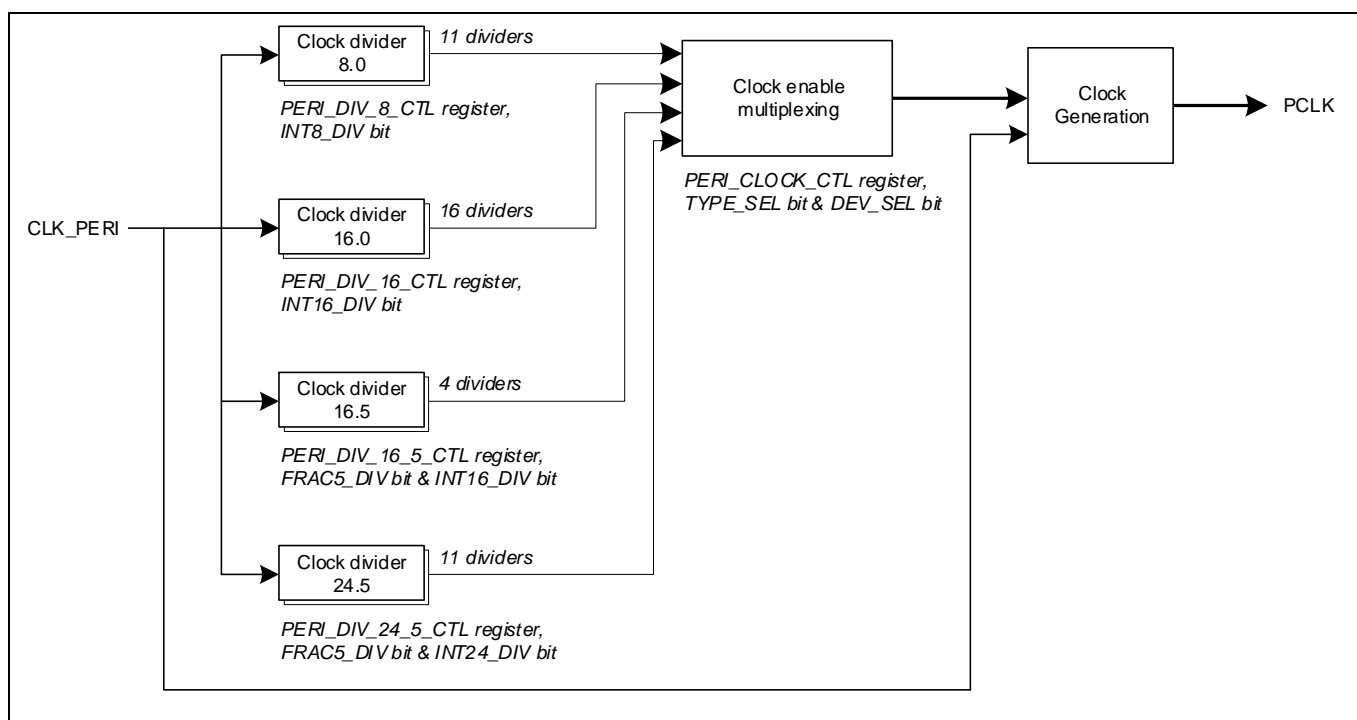


Figure 4 Peripheral clock divider #0

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 CLK_PERI based on clock divider

TRAVEO™ T2G clock system

2.2.4 High-frequency clocks (CLK_HF1 to CL_HF5)

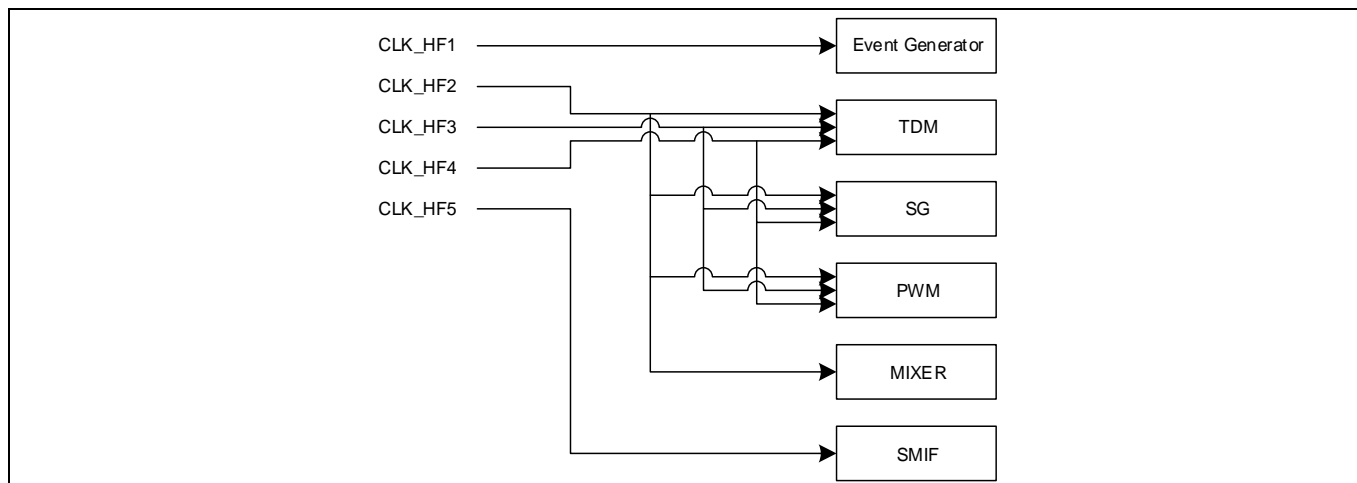


Figure 5 CLK_HFx (x= 1 to 5)

CLK_HF6 is dedicated to CSV. See the [architecture TRM](#) for the CSV description.

2.3 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 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.

Configuring the clock resources

3 Configuring the clock resources

See the TRAVEO™ T2G [architecture TRM](#), [registers TRM](#), and [datasheet](#) for details.

3.1 ECO

By default, the ECO is disabled and need to be enabled for usage. Also, trimming is necessary to use the ECO. **Figure 6** shows how to configure the clock for enabling the ECO.

To disable the ECO, write '0' to the ECO_EN bit of the CLK_ECO_CONFIG register.

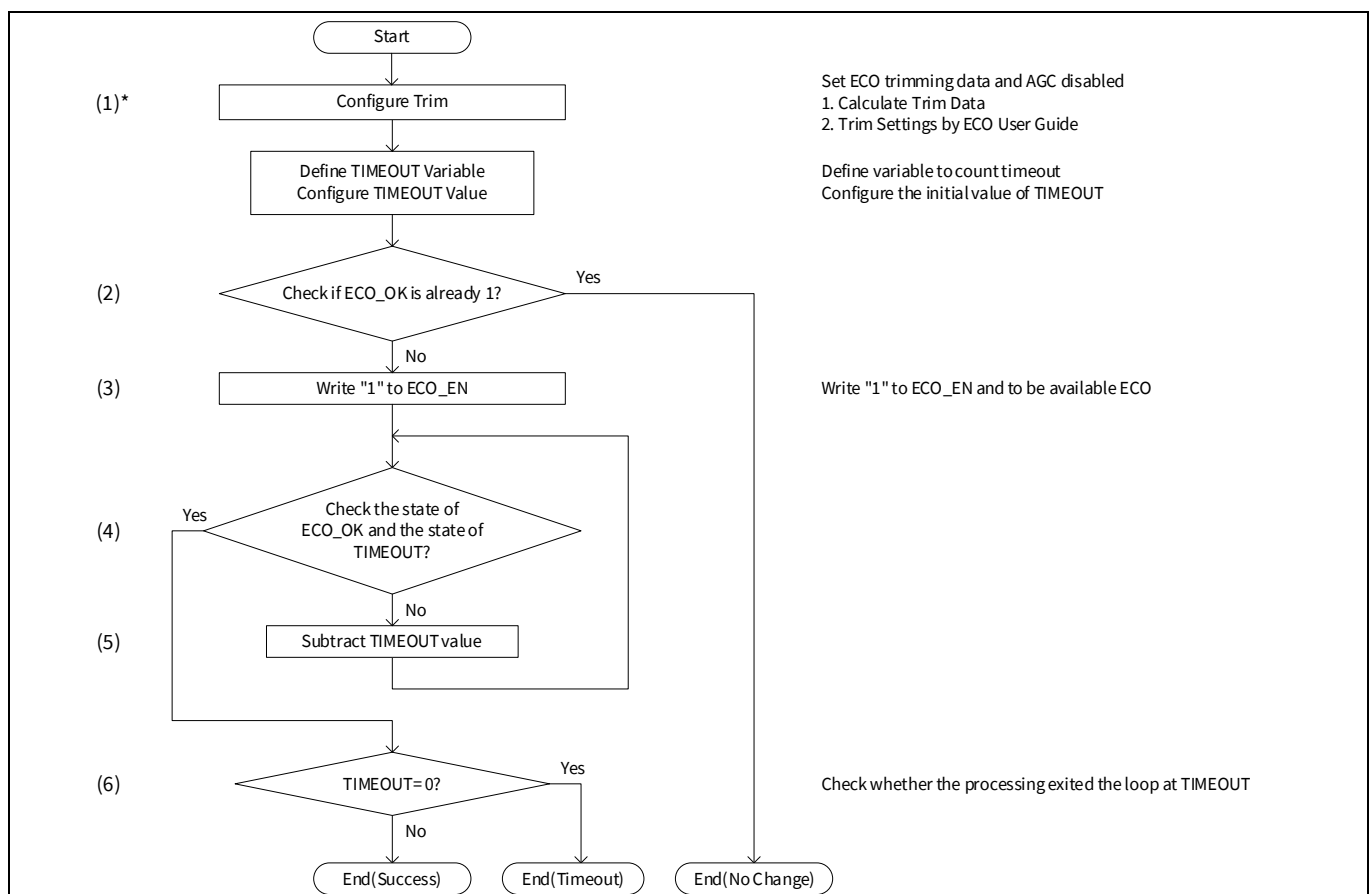


Figure 6 ECO configuration

3.1.1 Use case

- Oscillator to use: Crystal unit
- Fundamental frequency: 16 MHz
- Maximum drive level: 300.0 μ W
- 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.

Configuring the clock resources

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 ECO trim parameters

Parameters	Description	Value
CLK_ECO_CONFIG2.WDTRIM	Watchdog trim Calculated from Setting the ECO Parameters in the TRAVEO™ T2G user guide	7ul
CLK_ECO_CONFIG2.ATRIM	Amplitude trim Calculated from Setting the ECO Parameters in the TRAVEO™ T2G user guide	0ul
CLK_ECO_CONFIG2.FTRIM	Filter trim of 3 rd harmonic oscillation Calculated from Setting the ECO Parameters in the TRAVEO™ T2G user guide	3ul
CLK_ECO_CONFIG2.RTRIM	Feedback resistor trim Calculated from Setting the ECO Parameters in the TRAVEO™ T2G user guide	3ul
CLK_ECO_CONFIG2.GTRIM	Startup time of the gain trim Calculated from Setting the ECO Parameters in the TRAVEO™ T2G user guide	3ul
CLK_ECO_CONFIG.AGC_EN	Automatic gain control (AGC) disabled Calculated from Setting the ECO Parameters in the 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_200M_0_PATH_NO	PLL number for PLL_200M_0	2ul
PLL_200M_1_PATH_NO	PLL number for PLL_200M_1	3ul
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 ECO trim setting functions

Functions	Description	Value
Cy_WDT_Disable()	Disable watchdog timer	–
Cy_SysClk_FllDisable Sequence(Wait Cycle)	Disable FLL	Wait cycle = WAIT_FOR_STABILIZATION
Cy_SysClk_Pll400M Disable(PLL Number)	Disable PLL400M_0	PLL number = PLL_400M_0_PATH_NO

Configuring the clock resources

Functions	Description	Value
Cy_SysClk_PllDisable (PLL Number)	Disable PLL200M_0	PLL number = PLL_200M_0_PATH_NO
	Disable 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)	Delays by the specified number of microseconds	Wait time = 1u (1us)

3.1.3 Sample code for ECO initial configuration

See [Code Listing 1](#) for the 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 [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 Listing 1 General ECO configuration

```

:
/** Wait time definition */
#define WAIT_FOR_STABILIZATION (10000ul)
:
#define CLK_FREQ_ECO          (16000000ul)
:
#define PLL_400M_0_PATH_NO    (1ul)
#define PLL_200M_0_PATH_NO    (2ul)
#define PLL_200M_1_PATH_NO    (3ul)
:
#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 */

```

Define TIMEOUT Variable.

Define oscillator parameters to use for software calculation.

Define PLL number.

Configuring the clock resources

Code Listing 1 General ECO configuration

```

Cy_WDT_Disable();
:
/* Disable Fll */
CY_ASSERT(Cy_SysClk_FllDisableSequence(WAIT_FOR_STABILIZATION) == CY_SYSClk_SUCCESS);

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

/* Enable interrupt */
__enable_irq();

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

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

Watchdog Timer disable.

Disable FLL

Disable PLL

Trim and ECO setting. See [Code Listing 2](#).

Code Listing 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 Listing 4](#).

(1)-2. Trim settings according to the ECO User Guide

ECO Enable. See [Code Listing 3](#).

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

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

Code Listing 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);
    }
}
    
```

(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.

Configuring the clock resources

Code Listing 3 Cy_SysClk_EcoEnable() function

```

rtnval = ((timeoutus == 0ul) ? CY_SYSClk_TIMEOUT : CY_SYSClk_SUCCESS);
return rtnval;
}

```

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

Code Listing 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 calculated values */
    uint32_t atrim, agcen, wdtrim, gtrim, rtrim, ftrim;
    atrim = Cy_SysClk_SelectEcoAtrim(maxAmplitude);
    if(atriim == 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(atriim);

    /* 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;
}

```

Trim Calculation by software

Get Atrim Value. See [Code Listing 5](#).

Get AGC enable setting. See [Code Listing 6](#).

Get Wdtrim Value. See [Code Listing 7](#).

Get Gtrim Value. See [Code Listing 8](#).

Get Rtrim Value. See [Code Listing 9](#).

Get Ftrim Value. See [Code Listing 10](#).

Configuring the clock resources

Code Listing 4 Cy_SysClk_EcoConfigureWithMinRneg() function

```
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);
}
```

Code Listing 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.

Configuring the clock resources

Code Listing 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 Listing 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 Listing 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);
    }
}

```

Get Gtrim value.

Configuring the clock resources

Code Listing 8 Cy_SysClk_SelectEcoGtrim() function

```

{
    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);
}
}
    
```

Code Listing 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 Listing 10 Cy_SysClk_SelectEcoFtrim() function

```

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

Get Ftrim value.

Configuring the clock resources

3.2 WCO

By default, the WCO is disabled and need to be enabled for usage. **Figure 7** shows how to configure registers for enabling the WCO. To disable the WCO, write '0' to the WCO_EN bit of the BACKUP_CLT register.

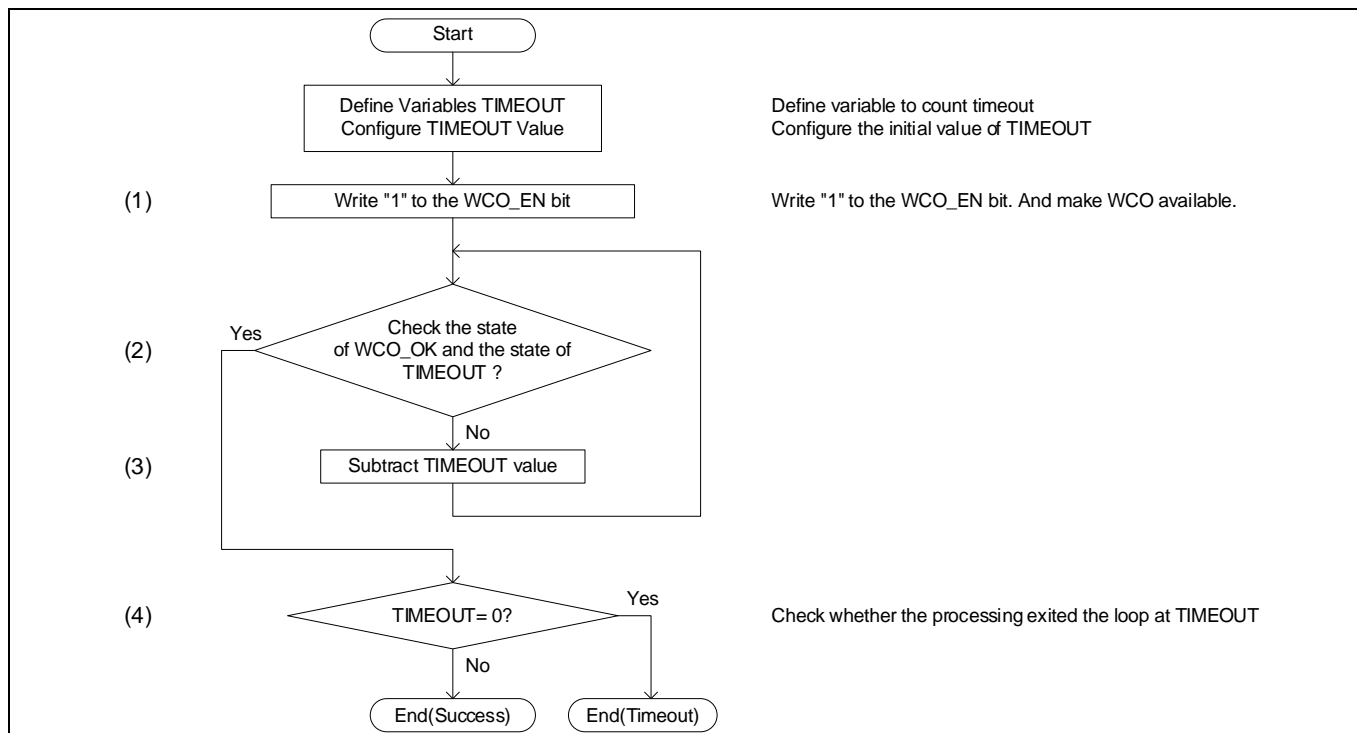


Figure 7 Enabling the WCO

3.2.1 Configuration

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

Table 3 WCO setting parameters

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

Table 4 WCO setting functions

Functions	Description	Value
Cy_WDT_Disable()	Disable watchdog timer	–
Cy_SysClk_FllDisable Sequence(Wait Cycle)	Disable FLL	Wait cycle = WAIT_FOR_STABILIZATION
Cy_SysClk_Pll400M Disable(PLL Number)	Disable PLL400M_0	PLL number = PLL_400M_0_PATH_NO

Configuring the clock resources

Functions	Description	Value
Cy_SysClk_PllDisable (PLL Number)	Disable PLL200M_0	PLL number = PLL_200M_0_PATH_NO
	Disable PLL200M_1	PLL number = PLL_200M_1_PATH_NO
AllClockConfigurationw()	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)

3.2.2 Sample code for WCO initial configuration

See [Code Listing 11](#) to [Code Listing 13](#) for the sample code.

Code Listing 11 General WCO configuration

```

:
: /** Wait time definition */
: #define WAIT_FOR_STABILIZATION (10000ul)
:
: #define PLL_400M_0_PATH_NO (1ul)
: #define PLL_200M_0_PATH_NO (2ul)
: #define PLL_200M_1_PATH_NO (3ul)
:
: static void AllClockConfiguration(void);
:
: int main(void)
: {
:     /* disable watchdog timer */
:     Cy_WDT_Disable();
:
:     /* Disable Fll */
:     CY_ASSERT(Cy_SysClk_FllDisableSequence(WAIT_FOR_STABILIZATION) == CY_SYSClk_SUCCESS);
:
:     /* Disable Pll */
:     CY_ASSERT(Cy_SysClk_Pll400MDisable(PLL_400M_0_PATH_NO) == CY_SYSClk_SUCCESS);
:     CY_ASSERT(Cy_SysClk_PllDisable(PLL_200M_0_PATH_NO) == CY_SYSClk_SUCCESS);
:     CY_ASSERT(Cy_SysClk_PllDisable(PLL_200M_1_PATH_NO) == CY_SYSClk_SUCCESS);
:
:     /* Enable interrupt */
:     __enable_irq();
:
:     /* Set Clock Configuring registers */
:     AllClockConfiguration();
:
:     /* Please 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 Listing 12](#).

Code Listing 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 Listing 13](#).

Configuring the clock resources

Code Listing 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(1ul);
    }
    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.

(3) Subtract TIMEOUT value.

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

Wait for 1 us.

3.3 IMO

By default, IMO is enabled so that all functions work properly. IMO will automatically be disabled during DeepSleep, Hibernate, and XRES modes; therefore, no explicit configuration for IMO is required.

3.4 ILO0/ILO1

ILO0 is enabled by default.

Note that ILO0 is used as the operating clock of the watchdog timer (WDT). Therefore, if ILO0 is disabled, it is necessary to disable the WDT. To disable 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.

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

3.5 LPECO

LPECO is disabled by default. LPECO cannot be used unless it is enabled. **Figure 8** shows how to configure registers for enabling LPECO. To disable the LPECO, write '0' to the LPECO_EN bit of BACKUP_LPECO_CTL register.

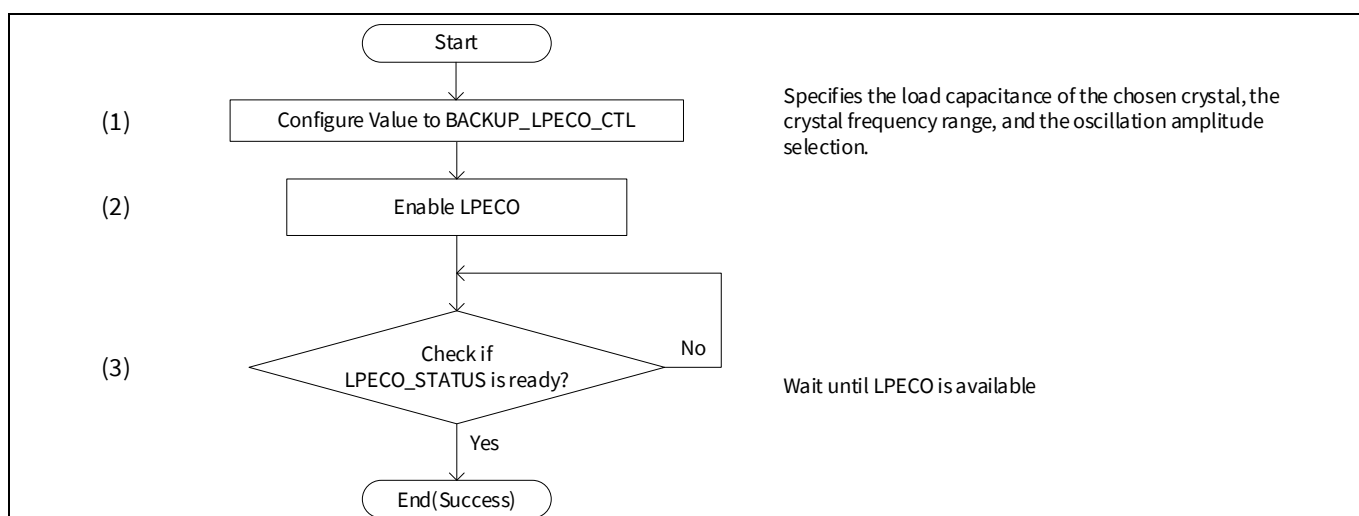


Figure 8 LPECO configuration

Configuring the clock resources

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. **Table 6** lists the functions of the configuration part of in SDL for LPECO settings.

Table 5 LPECO setting 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 (maximum)	0ul

Table 6 LPECO setting 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_Read()	Return the status of LPECO stabilization	–

3.5.2 Sample code for LPECO initial configuration

See **Code Listing 14** to **Code Listing 20** for the sample code.

Code Listing 14 General LPECO configuration

```

:
/** Wait time definition */
#define WAIT_FOR_STABILIZATION (10000ul)
:
#define CLK_FREQ_LPECO (8000000ul)
:
/** LPECO parameters */
#define LPECO_CRANGE (CY_SYSCLK_BAK_LPECO_LCAP_5TO10PF)
#define LPECO_FRANGE (CY_SYSCLK_BAK_LPECO_FREQ_6TO8MHZ)
#define LPECO_AMPSEL (CY_SYSCLK_BAK_LPECO_AMP_MAX_1P35V)
:
static void AllClockConfiguration(void);
:
int main(void)
{

```

Define TIMEOUT Variable.

Define oscillator parameters to use for software calculation.

Configuring the clock resources

Code Listing 14 General LPECO configuration

```
/* 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(;;);
}
```

Code Listing 15 AllClockConfiguration () function

```
static void AllClockConfiguration(void)
{
:
#ifdef LPECO_ENABLE
/***** LPECO setting *****/
{
    Cy_SysClk_ClkBak_LPECO_SetLoadCap(LPECO_CRANGE);
    Cy_SysClk_ClkBak_LPECO_SetFrequency(LPECO_FRANGE);
    Cy_SysClk_ClkBak_LPECO_SetAmplitude(LPECO_AMPSEL);
}

    Cy_SysClk_ClkBak_LPECO_Enable();
    while(Cy_SysClk_ClkBak_LPECO_Ready() == false);

:
#endif
:
    return;
}
```

(1) Configure Value to BACKUP_LPECO_CTL. See [Code Listing 16](#), [Code Listing 17](#) and [Code Listing 18](#).

(2) Enable LPECO. See [Code Listing 19](#).

(3) Wait until LPECO is available. See [Code Listing 20](#).

Code Listing 16 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;
}
```

Code Listing 17 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;
}
```

Code Listing 18 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;
}
```

Configuring the clock resources

Code Listing 19 Cy_SysClk_ClkBak_LPECO_Enable () function

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

Code Listing 20 Cy_SysClk_ClkBak_LPECO_Ready () function

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

Configuration of FLL and PLL

4 Configuration of FLL and PLL

See the TRAVEO™ T2G [architecture TRM](#), [registers TRM](#) and [datasheet](#) for more details.

4.1 FLL

You must configure the FLL before using it. FLL has a current-controlled oscillator (CCO); the output frequency of this CCO is controlled by adjusting the trim of the CCO.

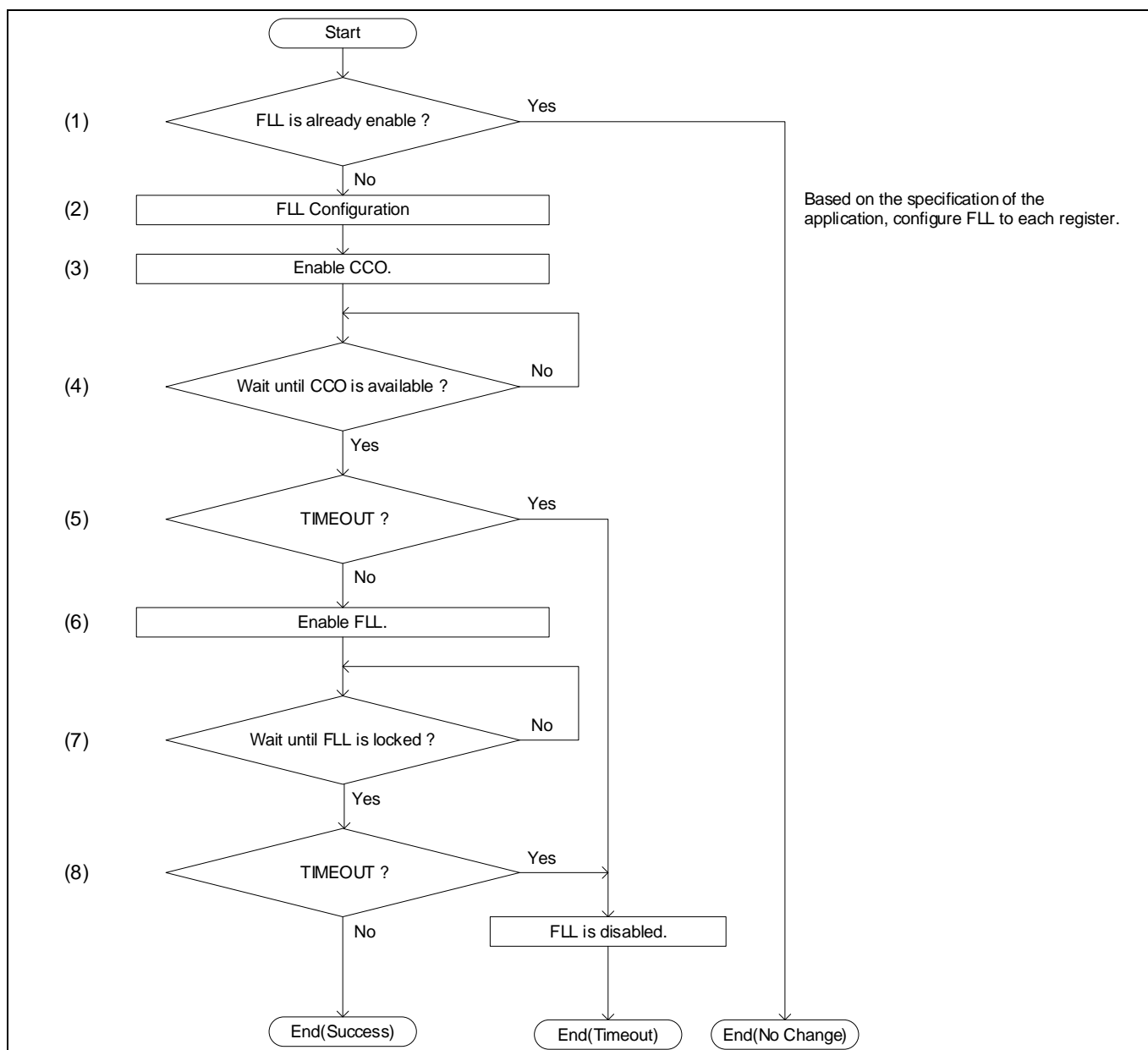


Figure 9 Configuring the FLL

CLK_FLL_CONFIGx is used to configure the FLL output frequency and CCO operating conditions.

4.1.1 Use case

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

Configuration of FLL and PLL

4.1.2 Configuration

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

Table 7 FLL 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	<p>FLL output mode</p> <p>CY_SYSCLK_FLLPLL_OUTPUT_AUTO: Automatic using lock indicator.</p> <p>CY_SYSCLK_FLLPLL_OUTPUT_LOCKED_OR_NOTHING: Similar to AUTO, except that the clock is gated off when unlocked.</p> <p>CY_SYSCLK_FLLPLL_OUTPUT_INPUT: Select the FLL reference input (bypass mode)</p> <p>CY_SYSCLK_FLLPLL_OUTPUT_OUTPUT: Select the FLL output. Ignores the lock indicator.</p> <p>See SRSS_CLK_FLL_CONFIG3 in the registers TRM for details.</p>	0ul

Table 8 FLL setting functions

Functions	Description	Value
AllClockConfiguration()	Clock configuration	-
Cy_SysClk_FllConfigureStandard(inputFreq, outputFreq, outputMode)	<p>inputFreq: Input frequency</p> <p>outputFreq: Output frequency</p> <p>outputMode: FLL output mode</p>	<p>inputFreq = PATH_SOURCE_CLOCK_FREQ,</p> <p>outputFreq = FLL_TARGET_FREQ,</p> <p>outputMode = CY_SYSCLK_FLLPLL_OUTPUT_AUTO</p>
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)

Configuration of FLL and PLL

4.1.3 Sample code for the initial FLL configuration

See [Code Listing 21](#) to [Code Listing 25](#) for the sample code.

Code Listing 21 General FLL configuration

```

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

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

```

Define TIMEOUT Variable.

Define FLL Target Frequency.

Define FLL Input Frequency.

Define FLL number.

FLL setting. See [Code Listing 22](#).

Code Listing 22 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 Listing 23](#).

FLL Enable. See [Code Listing 25](#).

Code Listing 23 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 */
    }
}

```

(1) Check if FLL is already enabled.

Check the FLL output range.

Check if FLL frequency ratio

Configuration of FLL and PLL

Code Listing 23 Cy_SysClk_FllConfigureStandard() function

```

SRSS->unCLK_FLL_CONFIG3.stcField.u2BYPASS_SEL = (uint32_t)outputMode;
return(CY_SYSCLK_SUCCESS);
}

cy_stc_fll_manual_config_t config = { 0ul };

config.outputMode = outputMode;

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

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

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

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

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

/* 6. Compute the lock tolerance.
    Recommendation: ROUNDUP((refDiv / fref) * ccoFreq * 3 * CCO_Trim_Step) + 2 */
config.updateTolerance = CY_SYSCLK_DIV_ROUNDUP(config.fllMult, 100ul /* Reciprocal number of Ratio */);
config.lockTolerance = config.updateTolerance + 20ul /*Threshold*/;
// 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);
}

```

FLL parameter calculation

Set FLL registers. See [Code Listing 24](#).

Configuration of FLL and PLL

Code Listing 24 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_SYSCLOCK_SUCCESS;

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

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

        /* update CLK_FLL_CONFIG2 register with 2 parameters */
        un_CLK_FLL_CONFIG2_t tempConfig2;
        tempConfig2.u32Register = SRSS->unCLK_FLL_CONFIG2.u32Register;
        tempConfig2.stcField.u13FLL_REF_DIV = config->refDiv;
        tempConfig2.stcField.u8LOCK_TOL = config->lockTolerance;
        tempConfig2.stcField.u8UPDATE_TOL = config->updateTolerance;
        SRSS->unCLK_FLL_CONFIG2.u32Register = tempConfig2.u32Register;

        /* update CLK_FLL_CONFIG3 register with 4 parameters */
        un_CLK_FLL_CONFIG3_t tempConfig3;
        tempConfig3.u32Register = SRSS->unCLK_FLL_CONFIG3.u32Register;
        tempConfig3.stcField.u4FLL_LF_IGAIN = config->igain;
        tempConfig3.stcField.u4FLL_LF_PGAIN = config->pgain;
        tempConfig3.stcField.u13SETTLING_COUNT = config->settleCount;
        tempConfig3.stcField.u2BYPASS_SEL = (uint32_t)(config->outputMode);
        SRSS->unCLK_FLL_CONFIG3.u32Register = tempConfig3.u32Register;

        /* update CLK_FLL_CONFIG4 register with 1 parameter; preserve other bits */
        un_CLK_FLL_CONFIG4_t tempConfig4;
        tempConfig4.u32Register = SRSS->unCLK_FLL_CONFIG4.u32Register;
        tempConfig4.stcField.u3CCO_RANGE = (uint32_t)(config->ccoRange);
        tempConfig4.stcField.u9CCO_FREQ = (uint32_t)(config->ccoFreq);
        tempConfig4.stcField.u1CCO_HW_UPDATE_DIS = (uint32_t)(config->ccoHwUpdateDisable);
        SRSS->unCLK_FLL_CONFIG4.u32Register = tempConfig4.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.

Configuration of FLL and PLL

Code Listing 25 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 occur.

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 PLL

You must configure the PLL before using it.

Configuration of FLL and PLL

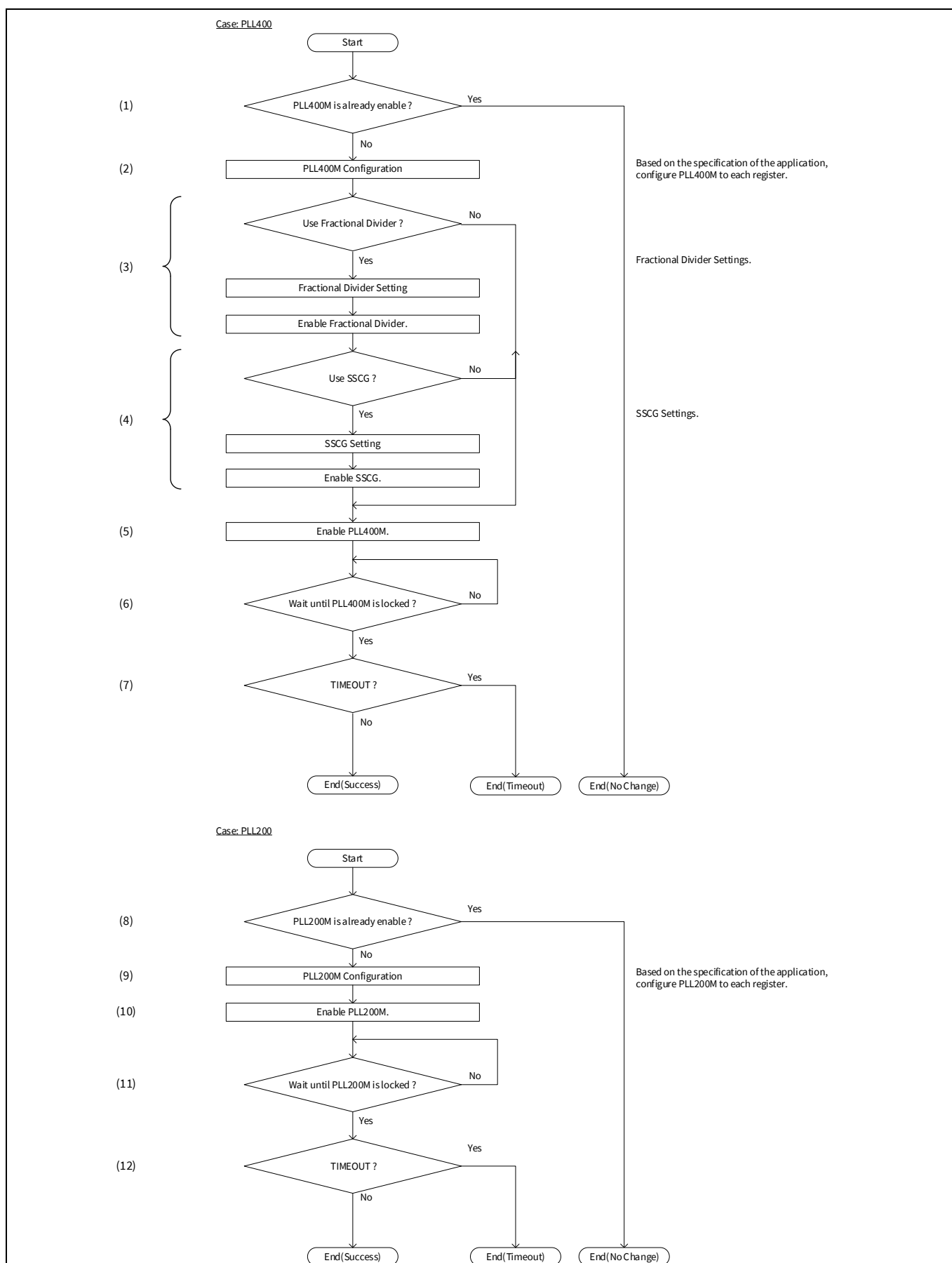


Figure 10 Configuring the PLL

Configuration of FLL and PLL

4.2.1 Use case

- Input clock frequency: 16.000 MHz
- Output clock frequency:
196.608 MHz (PLL400 #0)
160.000 MHz (PLL200 #0)
80.000 MHz (PLL200 #1)
- Fractional divider:
Enable
- SSCG:
Disable
- SSCG dithering:
Disable
- LF mode: 200 MHz to 400 MHz (PLL200)

4.2.2 Configuration

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

Table 9 PLL 400 parameters

Parameters	Description	Value
PLL400_0_TARGET_FREQ	PLL400 #0 target frequency	196.608 MHz (196608000ul)
WAIT_FOR_STABILIZATION	Waiting for stabilization	10000ul
PLL_400M_0_PATH_NO	PLL400 #0 number	1u
CLK_FREQ_ECO	ECO clock frequency	16000000ul (16 MHz)
PATH_SOURCE_CLOCK_FREQ	PATH 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 FLL output. Ignores the lock indicator. See SRSS_CLK_FLL_CONFIG3 in registers TRM for details.	0ul
pllConfig.inputFreq	Input PLL frequency	PATH_SOURCE_CLOCK_FREQ
pllConfig.outputFreq	Output PLL frequency (PLL400 #0)	PLL400_0_TARGET_FREQ
pllConfig.outputMode	Output mode 0: CY_SYSCLK_FLLPLL_OUTPUT_AUTO	CY_SYSCLK_FLLPLL_OUTPUT_AUTO

Configuration of FLL and PLL

Parameters	Description	Value
	1: CY_SYSClk_FLLPLL_OUTPUT_LOCKED_OR_NOHING 2: CY_SYSClk_FLLPLL_OUTPUT_INPUT 3: CY_SYSClk_FLLPLL_OUTPUT_OUTPUT	
pllConfig.fracEn	Enable the fractional divider (PLL400 #0)	true
pllConfig.fracDitherEn	Enable the dithering operation (PLL400 #0)	true
pllConfig.sscgEn	Enable SSCG (PLL400 #0)	false
pllConfig.sscgDitherEn	Enable the SSCG dithering operation (PLL400 #0)	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 HFCLK source. ... 16: divide by 16. Suitable for direct usage as 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_NOHING 2: PLL_REF 3: PLL_OUT	config->outputMode (Calculated value)

Configuration of FLL and PLL

Table 10 PLL 400 setting functions

Functions	Description	Value
AllClockConfiguration()	Clock configuration	–
Cy_SysClk_Pll400MConfigure(PLL Number, PLL Configure)	Set the PLL path number and configure the PLL (PLL400 #0)	PLL number = PLL_400M_0_PATH_NO, PLL configure = g_pll400_0_Config
Cy_SysClk_Pll400MEnable(PLL Number, Timeout value)	Set the PLL path number and monitor the PLL configuration (PLL400 #0)	PLL number = PLL_400M_0_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 = PLL_400M_0_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
Cy_SysClk_PllCalucDividers()	Calculate the appropriate divider settings according to the PLL input/output frequency	
Cy_SysClk_Pll400MEnable(PLL Number, Timeout value)	Set the PLL path number and monitor then PLL configuration (PLL400 #0)	PLL number = PLL_400M_0_PATH_NO, Timeout value = WAIT_FOR_STABILIZATION

Table 11 PLL 200 setting 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)
WAIT_FOR_STABILIZATION	Waiting for stabilization	10000ul
PLL_200M_0_PATH_NO	PLL200 #0 number	2u
PLL_200M_1_PATH_NO	PLL200 #1 number	3u
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)

Configuration of FLL and PLL

Parameters	Description	Value
pllConfig.outputMode	Output mode: 0: CY_SYCLK_FLLPLL_OUTPUT_AUTO 1: CY_SYCLK_FLLPLL_OUTPUT_LOCKED_OR_NOHING 2: CY_SYCLK_FLLPLL_OUTPUT_INPUT 3: CY_SYCLK_FLLPLL_OUTPUT_OUTPUT	CY_SYCLK_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)
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_NOHING 2: PLL_REF 3: PLL_OUT	config->outputMode (Calculated value)
manualConfig.fracDiv	Fractional divider value	config->fracDiv (Calculated value)

Table 12 PLL 200 setting 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 = PLL_200M_0_PATH_NO, PLL configure = g_pll200_0_Config
	Set the PLL path number and configure the PLL (PLL200 #1)	PLL number = PLL_200M_1_PATH_NO, PLL configure = g_pll200_1_Config
Cy_SysLib_DelayUs (Wait Time)	Delays by the specified number of microseconds	Wait time = 1u (1us)

Configuration of FLL and PLL

Functions	Description	Value
Cy_SysClk_PllManualConfigure(PLL Number, PLL Manual Configure)	Set the PLL path number and manually configure the PLL (PLL200 #0)	PLL number = PLL_200M_0_PATH_NO, PLL manual configure = manualConfig
	Set the PLL path number and manually configure the PLL (PLL200 #1)	PLL number = PLL_200M_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_PllCalucDividers(InputFreq, OutputFreq, PLLlimit, FracBitNum, RefDiv, OutputDiv, FeedBackFracDiv)	Calculates appropriate divider settings according to the PLL input/output frequency	InputFreq = PATH_SOURCE_CLOCK_FREQ OutputFreq = 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 = PLL_200M_0_PATH_NO, Timeout value = WAIT_FOR_STABILIZATION
	Set the PLL path number and monitor the PLL configuration (PLL200 #1)	PLL number = PLL_200M_1_PATH_NO, Timeout value = WAIT_FOR_STABILIZATION

Configuration of FLL and PLL

4.2.3 Sample code for PLL initial configuration

Code Listing 26 to **Code Listing 32** show the example code for PLL400 #0. **Code Listing 33** to **Code Listing 39** show the example code for PLL200 #0.

Code Listing 26 General configuration of PLL 400 #0

```

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

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

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

```

PLL Target frequency.

Define TIMEOUT variable.

Define PLL number.

PLL400 #0 Configuration.

PLL400 #0 setting. See [Code Listing 27](#).

Code Listing 27 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);

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

```

PLL400 Configuration. See [Code Listing 28](#).

PLL400 Enable. See [Code Listing 32](#).

Configuration of FLL and PLL

Code Listing 28 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);
    if(status != CY_SYSCLK_SUCCESS)
    {
        return(status);
    }

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

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

    if(status != CY_SYSCLK_SUCCESS)
    {
        return(status);
    }

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

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

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

Check if the valid clock path and PLL400 Number. See [Code Listing 30](#).

(1) Check if PLL400 is already enabled.

PLL400 Manual Configure. See [Code Listing 29](#).

Code Listing 29 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);
    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);
    }
}
```

Getting PLL400 PATH Number. See [Code Listing 30](#).

Configuration of FLL and PLL

Code Listing 29 Cy_SysClk_Pll400MManualConfigure() function

```

}

un_CLK_PLL400M_CONFIG_t tempClkPLL400MConfigReg;
tempClkPLL400MConfigReg.u32Register = SRSS->CLK_PLL400M[p11No].unCONFIG.u32Register;
if (tempClkPLL400MConfigReg.stcField.u1ENABLE != 0ul) /* 1 = enabled */
{
    return(CY_SYSClk_INVALID_STATE);
}

/* no errors */
/* If output mode is bypass (input routed directly to output), then done.
The output frequency equals the input frequency regardless of the frequency parameters. */
if (config->outputMode != CY_SYSClk_FLLPLL_OUTPUT_INPUT)
{
    tempClkPLL400MConfigReg.stcField.u8FEEDBACK_DIV = (uint32_t) config->feedbackDiv;
    tempClkPLL400MConfigReg.stcField.u5REFERENCE_DIV = (uint32_t) config->referenceDiv;
    tempClkPLL400MConfigReg.stcField.u5OUTPUT_DIV = (uint32_t) config->outputDiv;
}
tempClkPLL400MConfigReg.stcField.u2BYPASS_SEL = (uint32_t) config->outputMode;
SRSS->CLK_PLL400M[p11No].unCONFIG.u32Register = tempClkPLL400MConfigReg.u32Register;

un_CLK_PLL400M_CONFIG2_t tempClkPLL400MConfig2Reg;
tempClkPLL400MConfig2Reg.u32Register = SRSS->CLK_PLL400M[p11No].unCONFIG2.u32Register;
tempClkPLL400MConfig2Reg.stcField.u24FRAC_DIV = config->fracDiv;
tempClkPLL400MConfig2Reg.stcField.u3FRAC_DITHER_EN = config->fracDitherEn;
tempClkPLL400MConfig2Reg.stcField.u1FRAC_EN = config->fracEn;
SRSS->CLK_PLL400M[p11No].unCONFIG2.u32Register = tempClkPLL400MConfig2Reg.u32Register;

un_CLK_PLL400M_CONFIG3_t tempClkPLL400MConfig3Reg;
tempClkPLL400MConfig3Reg.u32Register = SRSS->CLK_PLL400M[p11No].unCONFIG3.u32Register;
tempClkPLL400MConfig3Reg.stcField.u10SSCG_DEPTH = (uint32_t) config->sscgDepth;
tempClkPLL400MConfig3Reg.stcField.u3SSCG_RATE = (uint32_t) config->sscgRate;
tempClkPLL400MConfig3Reg.stcField.u1SSCG_DITHER_EN = (uint32_t) config->sscgDitherEn;
tempClkPLL400MConfig3Reg.stcField.u1SSCG_EN = (uint32_t) config->sscgEn;
SRSS->CLK_PLL400M[p11No].unCONFIG3.u32Register = tempClkPLL400MConfig3Reg.u32Register;

return (CY_SYSClk_SUCCESS);
}
    
```

(2) PLL400 Configuration

(3) Fractional Divider Settings

(4) SSCG Settings

Code Listing 30 Cy_SysClk_GetPll400MNo() function

```

__STATIC_INLINE cy_en_sysclk_status_t Cy_SysClk_GetPll400MNo(uint32_t pathNo, uint32_t* p11No)
{
    /* check for error */
    if ((pathNo <= 0ul) || (pathNo > SRSS_NUM_PLL400M))
    {
        /* invalid clock path number */
        return(CY_SYSClk_BAD_PARAM);
    }

    *p11No = pathNo - 1ul;
    return(CY_SYSClk_SUCCESS);
}
    
```

Code Listing 31 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 = 0xFFFFFFFFFFFFFFFFFull;

    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);
    }
}
    
```

Configuration of FLL and PLL

Code Listing 31 Cy_SysClk_PllCalucDividers() function

```

}

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 & ((1ul << (uint64_t)fracBitNum) -
1ul));
                }
            }
            *refDiv = i_refDiv;
            *outputDiv = i_outDiv;
            errorMin = error;
            if(errorMin == 0ul){break;}
        }
    }
    if(errorMin == 0ul){break;}
}

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

Configuration of FLL and PLL

Code Listing 32 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.ulENABLE = 1ul;

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

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

        return (status);
    }
}

```

Code Listing 33 General configuration of PLL 200 #0

```
#define PLL200_0_TARGET_FREQ      (160000000ul) }
#define PLL200_1_TARGET_FREQ      (100000000ul) }

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

:
#define PLL_200M_0_PATH_NO      (2ul)
#define PLL_200M_1_PATH_NO      (3ul)
#define BYPASSED_PATH_NO        (4ul) }

:
/** Parameters for Clock Configuration */
cy_stc_pll_config_t g_pll200_0_Config =
{
    .inputFreq  = PATH_SOURCE_CLOCK_FREQ,      // ECO: 16MHz
    .outputFreq = PLL200_0_TARGET_FREQ,        // target PLL output
    .lfMode     = false,                       // VCO frequency is [200MHz, 400MHz]
    .outputMode = CY_SYSCLK_FLLPLL_OUTPUT_AUTO,
};

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

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

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

Code Listing 34 AllClockConfiguration() function

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

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

    return;
}
```

PLL200 Configuration. See [Code Listing 35](#).

PLL200 Enable. See [Code Listing 39](#).

Configuration of FLL and PLL

Code Listing 35 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_SYSCCLK_SUCCESS)
    {
        return(status);
    }

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

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

    if(status != CY_SYSCCLK_SUCCESS)
    {
        return(status);
    }

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

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

(8) Check if PLL200 is already enabled.

PLL200 Calculating Dividers Settings. See [Code Listing 38](#).

PLL200 Manual Configuring Settings. See [Code Listing 36](#).

Code Listing 36 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_SYSCCLK_SUCCESS)
    {
        return(status);
    }

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

    if((config->referenceDiv < MIN_REF_DIV) || (MAX_REF_DIV < config->referenceDiv))
    {
        return(CY_SYSCCLK_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_SYSCCLK_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_SYSCCLK_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. */
}
```

Configuration of FLL and PLL

Code Listing 36 Cy_SysClk_PllManualConfigure() function

```

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[p11No].u32Register = tempClkPLLConfigReg.u32Register;

return (CY_SYSCLK_SUCCESS);
}
    
```

(9) PLL200 Configuration

Code Listing 37 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);
}
    
```

Code Listing 38 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;
        }
    }
}
    
```

Configuration of FLL and PLL

Code Listing 38 Cy_SysClk_PllCalucDividers() function

```

/* 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 & ((1ul << (uint64_t)fracBitNum) -
1ul));
            }
        }
        *refDiv = i_refDiv;
        *outputDiv = i_outDiv;
        errorMin = error;
        if(errorMin == 0ul){break;}
    }
    if(errorMin == 0ul){break;}

    if(errorMin == 0xFFFFFFFFFFFFFFFFFull)
    {
        return (CY_SYSClk_BAD_PARAM);
    }
    else
    {
        return (CY_SYSClk_SUCCESS);
    }
}

```

Code Listing 39 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;

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

        status = ((timeoutus == 0ul) ? CY_SYSClk_TIMEOUT : CY_SYSClk_SUCCESS);
        return (status);
    }
}

```

(10) Enable PLL200

(11) Wait until PLL200 is locked.

(12) Check Timeout.

Wait for 1 us.

Configuring the internal clocks

5 Configuring the internal clocks

See the TRAVEO™ T2G [architecture TRM](#), [registers TRM](#), and [datasheet](#) for more details.

5.1 CLK_PATHx

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

Table 13 Relationship between FLL/PLLs and PATHx

FLL/PLLs	CLK_PATHx
FLL	CLK_PATH0
PLL400#0	CLK_PATH1
PLL200#0	CLK_PATH2
PLL200#1	CLK_PATH3
Directly (FLL and PLL cannot be selected)	CLK_PATH4

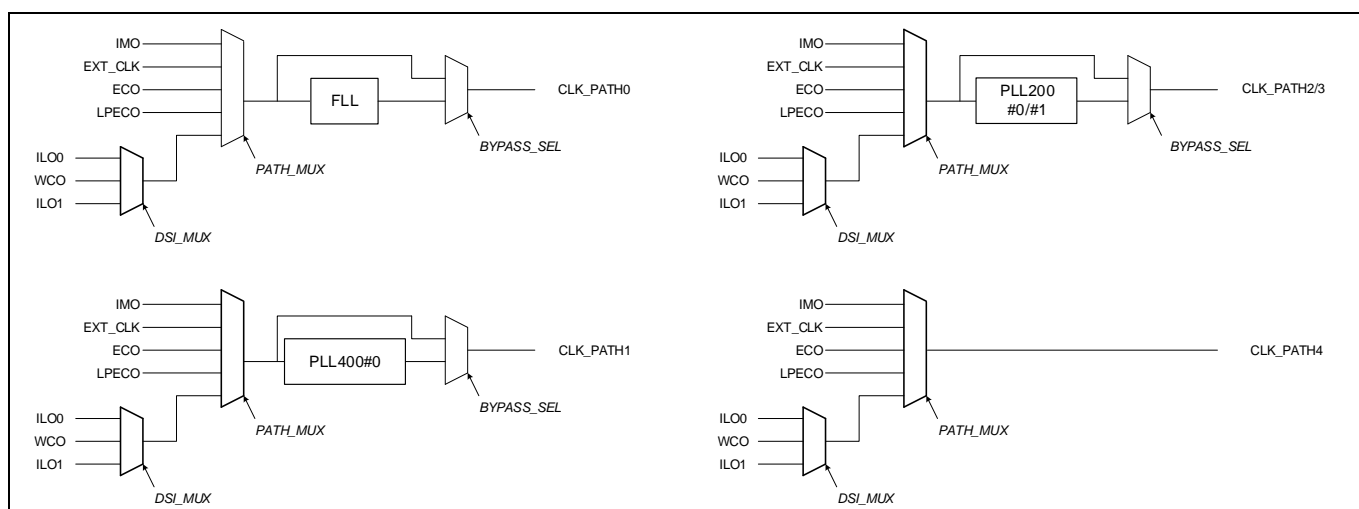


Figure 11 How CLK_PATH is generated

To configure CLK_PATHx, you should configure DSI_MUX and PATH_MUX. BYPASS_MUX is also required for CLK_PATHx. [Table 14](#) shows the registers necessary for configuring CLK_PATHx.

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

Configuring the internal clocks

Register name	Bit name	Value	Selected clock and item
CLK_FLL_CONFIG3	BYPASS_SEL[29:28]	20	ILO1
		Other	Reserved. Do not use.
		0 (Default)	AUTO ¹
		1	LOCKED_OR_NOTHING ²
		2	FLL_REF (bypass mode) ³
CLK_PLL_CONFIG	BYPASS_SEL[29:28]	3	FLL_OUT ³
		0 (Default)	AUTO ¹
		1	LOCKED_OR_NOTHING ²
		2	PLL_REF (bypass mode) ³
		3	PLL_OUT ³

5.2 CLK_HF_x

CLK_HF_x (x=0 to 6) can be selected from CLK_PATH_y (y= 0 to 4). A predivider is available to divide the selected CLK_PATH_x. CLK_HF0 is always enabled because it is the source clock for the CPU cores. It is possible to disable CLK_HF_x.

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

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

Figure 12 shows the details of ROOT_MUX and the predivider.

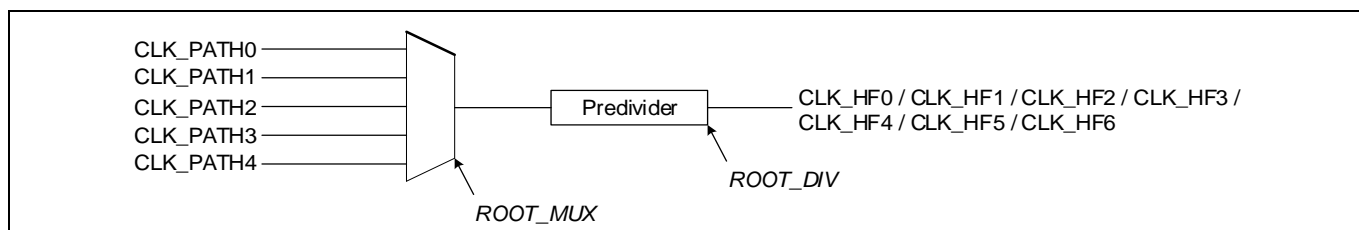


Figure 12 ROOT_MUX and predivider

Table 15 Configuring CLK_HF_x

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
		Other	Reserved. Do not use.

1 Switching automatically according to locked state.

2 The clock is gated off when unlocked.

3 In this mode, lock state is ignored.

Configuring the internal clocks

Register name	Bit name	Value	Selected item
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 CLK_LF

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

Figure 13 shows the details of LFCLK_SEL that CLK_LF is configured.

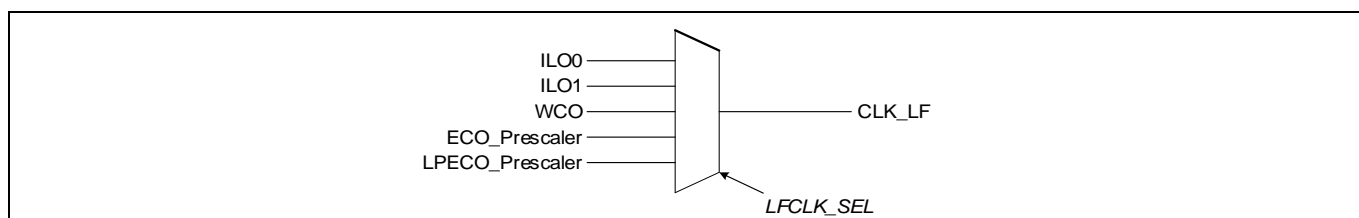


Figure 13 LFCLK_SEL

Table 16 Configuring 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 CLK_FAST

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

5.5 CLK_PERI

CLK_PERI is the clock input to the peripheral clock divider and CLK_GR. 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 a value (x= 0 to 255) divided by the PERI_INT_DIV bit of the CM0_CLOCK_CTL register.

5.6 CLK_SLOW

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

Configuring the internal clocks

5.7 CLK_GR

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

5.8 PCLK

PCLK is a clock that activates each peripheral function. Peripheral clock dividers have a function to divide CLK_PERI and generate a clock for each peripheral function. See the "Peripheral clocks" section in the [datasheet](#).

Figure 14 shows the steps to configure peripheral clock dividers. See the [architecture TRM](#) for more details.

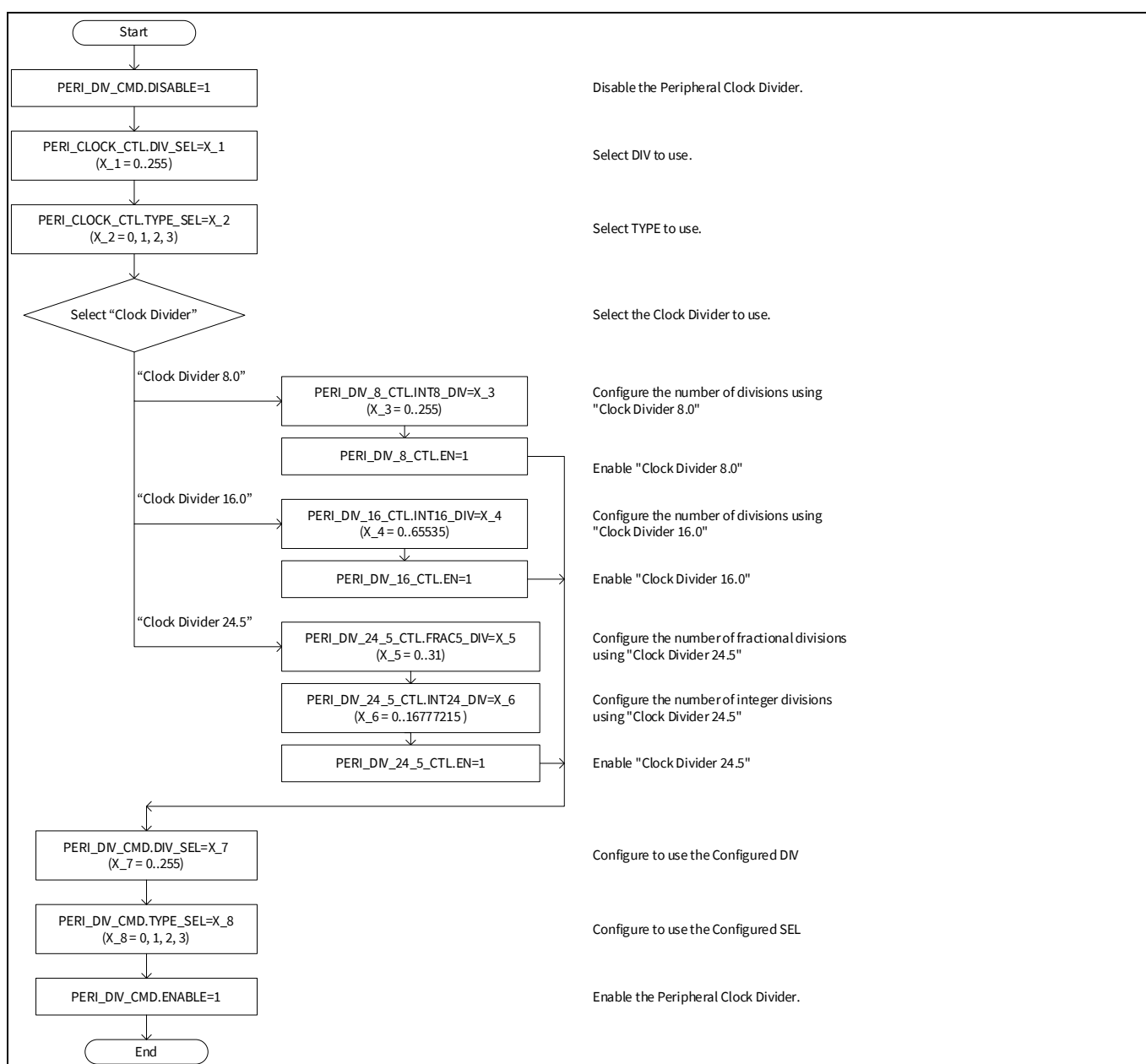


Figure 14 Generating PCLK

Configuring the internal clocks

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 PCLK configuration example

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)

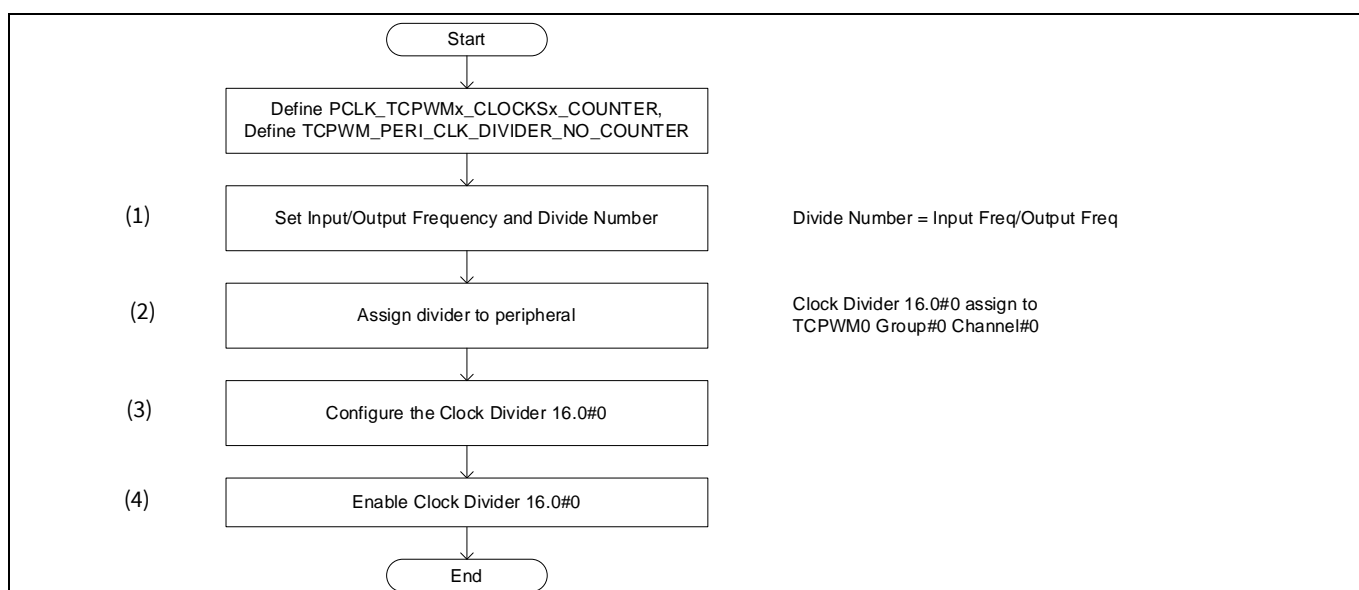


Figure 15 PCLK configuration example

5.8.1.2 Configuration

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

Table 17 PCLK (TCPWM timer example) 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	1ul

Configuring the internal clocks

Parameters	Description	Value
	CY_SYSCLK_DIV_24_5_BIT = 3u, 24.5-bit fractional divider	
periFreq	Peripheral clock frequency	80000000ul (80 MHz)
targetFreq	Target clock frequency	2000000ul (2 MHz)
divNum	Divide number	periFreq/targetFreq

Table 18 PCLK (example of the TCPWM timer) setting 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 the peripheral divider	dividerType, = CY_SYSCLK_DIV_16_BIT dividerNum = TCPWM_PERI_CLK_DIVIDER_NO_COUNTER dividerValue = divNum-1ul
Cy_SysClk_PeriphEnableDivider(dividerType, dividerNum)	Enable the peripheral divider	dividerType, = CY_SYSCLK_DIV_16_BIT dividerNum = TCPWM_PERI_CLK_DIVIDER_NO_COUNTER

5.8.2 Sample code for PCLK initial configuration (TCPWM timer example)

See [Code Listing 40](#) to [Code Listing 43](#) for the sample code.

Code Listing 40 General PCLK configuration (TCPWM timer example)

```

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

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

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

    CY_ASSERT((periFreq % targetFreq) == 0ul); // inaccurate target clock
    Cy_SysClk_PeriphAssignDivider(PCLK_TCPWMx_CLOCKSx_COUNTER, CY_SYSCLK_DIV_16_BIT,
    TCPWM_PERI_CLK_DIVIDER_NO_COUNTER);
    /* Sets the 16-bit divider */
    Cy_SysClk_PeriphSetDivider(CY_SYSCLK_DIV_16_BIT, TCPWM_PERI_CLK_DIVIDER_NO_COUNTER, (divNum-1ul));
    Cy_SysClk_PeriphEnableDivider(CY_SYSCLK_DIV_16_BIT, TCPWM_PERI_CLK_DIVIDER_NO_COUNTER);

    for(;;);
}

```

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 Listing 41](#).

Peripheral Divider Enable setting. See
[Code Listing 43](#).

Peripheral Divider setting. See [Code Listing 42](#).

Configuring the internal clocks

Code Listing 41 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 Listing 42 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 Listing 43 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.

Configuring the internal clocks

5.9 ECO_Prescaler

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

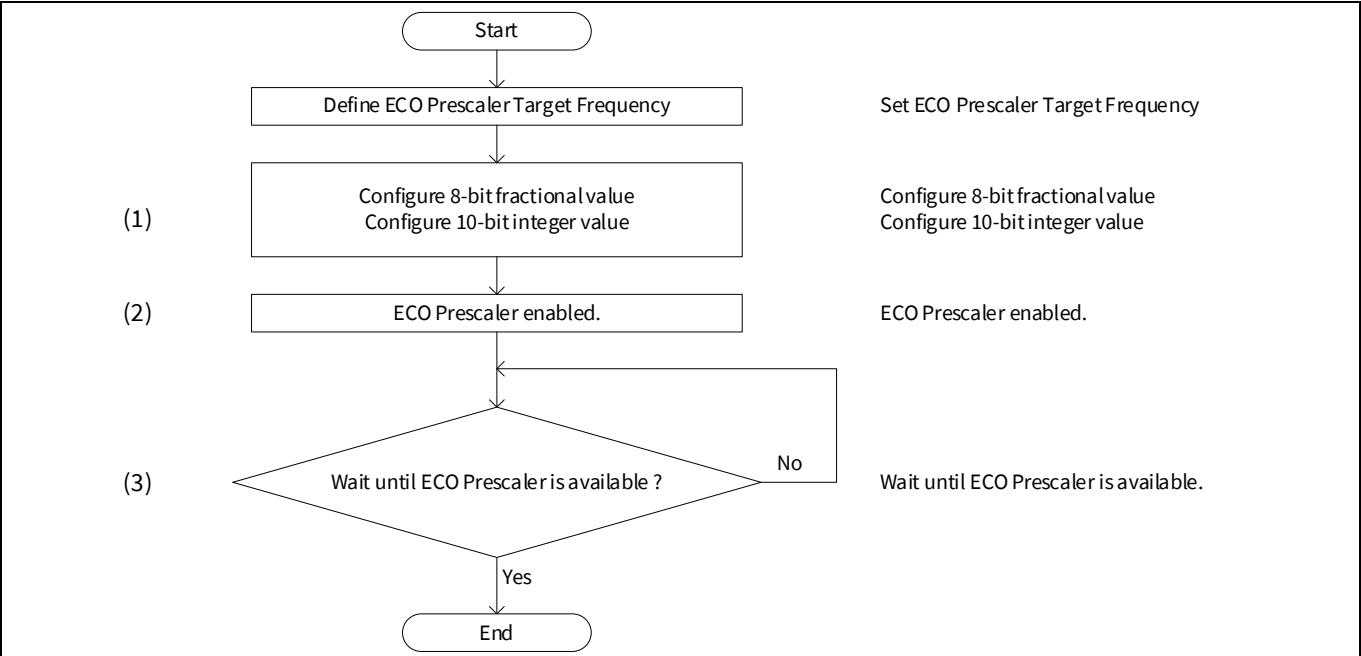


Figure 16 Enabling ECO_Prescaler

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

Figure 17 shows the flow to disable ECO_Prescaler. For details on ECO_Prescaler, see the [architecture TRM](#).

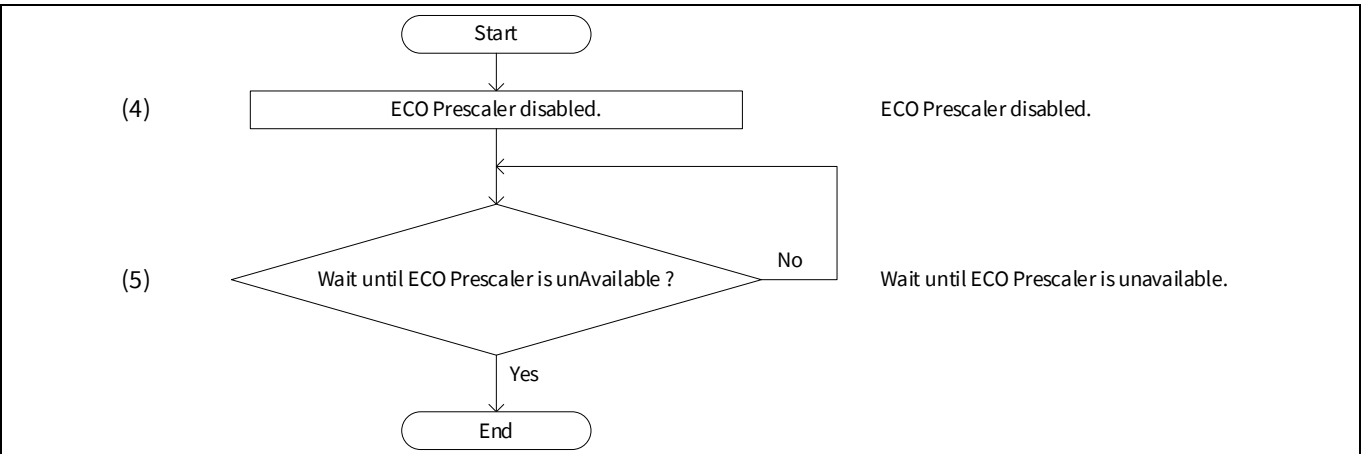


Figure 17 Disabling ECO_Prescaler

Configuring the internal clocks

5.9.1 Use case

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

5.9.2 Configuration

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

Table 19 ECO prescaler 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 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, divFrac)	divInt: 10-bit integer value allows for ECO frequencies divFrac: 8-bit fractional value	–
Cy_SysClk_GetEcoPrescaleStatus	Check the prescaler status	–

5.9.3 Sample code for ECO prescaler initial configuration

See **Code Listing 44** to **Code Listing 50** for the sample code.

Code Listing 44 General ECO prescaler configuration

```

:
#define ECO_PRESCALER_TARGET_FREQ (1234567ul)
/** Wait time definition */
#define WAIT_FOR_STABILIZATION (10000ul)
:
#define CLK_FREQ_ECO (16000000ul)
:
#define PATH_SOURCE_CLOCK_FREQ CLK_FREQ_ECO
:

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 Waiting for stabilization

Define ECO Clock Frequency

ECO Prescaler setting. See [Code Listing 45](#).

Configuring the internal clocks

Code Listing 45 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 Listing 46](#).

ECO Prescaler enable. See [Code Listing 48](#).

Code Listing 46 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 Listing 47](#).

Code Listing 47 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.u10ECO_INT_DIV = divInt;
:  tempRegEcoPrescale.stcField.u8ECO_FRAC_DIV = divFract;
:  SRSS->unCLK_ECO_PRESCALE.u32Register = tempRegEcoPrescale.u32Register;
:
:  return;
:
: }
```

(1) Configure ECO Prescaler.

Configuring the internal clocks

Code Listing 48 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.

Code Listing 49 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.

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

Code Listing 50 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.

Configuring the internal clocks

5.10 LPECO_Prescaler

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

Figure 18 shows the steps to enable LPECO_Prescaler. For details on LPECO_Prescaler, see the [architecture TRM](#).

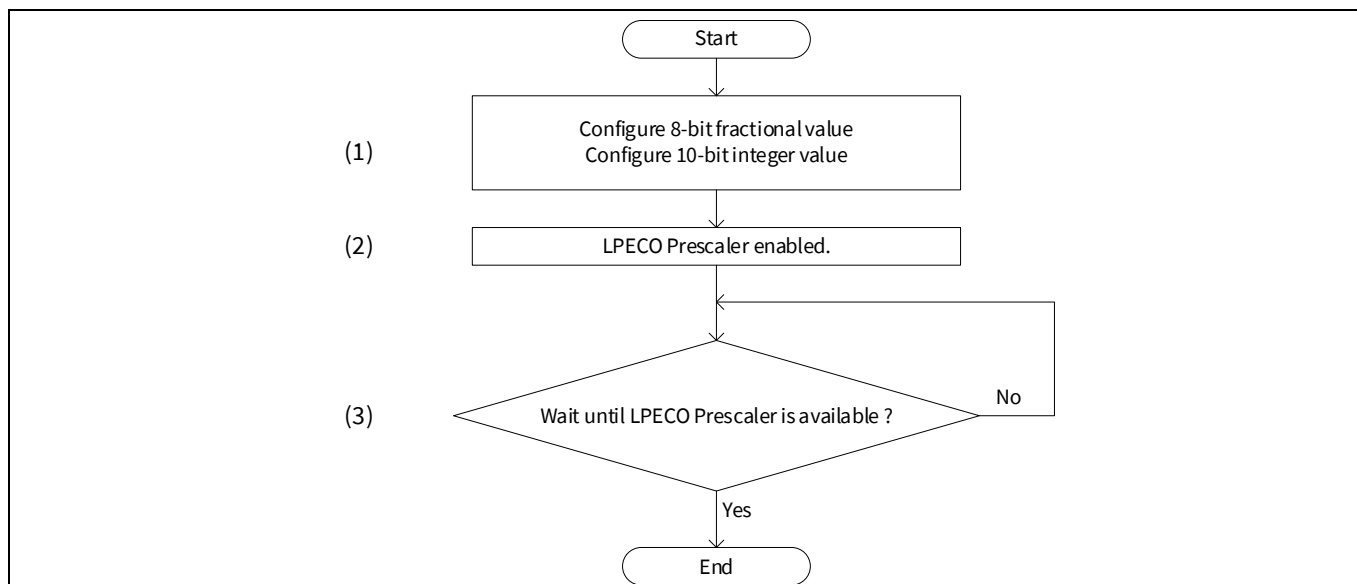


Figure 18 Enabling the LPECO_Prescaler

Note: Do not change the LPECO_FRAC_DIV and LPECO_INT_DIV settings when LPECO_DIV_ENABLE = 1.

Figure 19 shows the steps to disable the LPECO_Prescaler. For details on the LPECO_Prescaler, see the [architecture TRM](#).

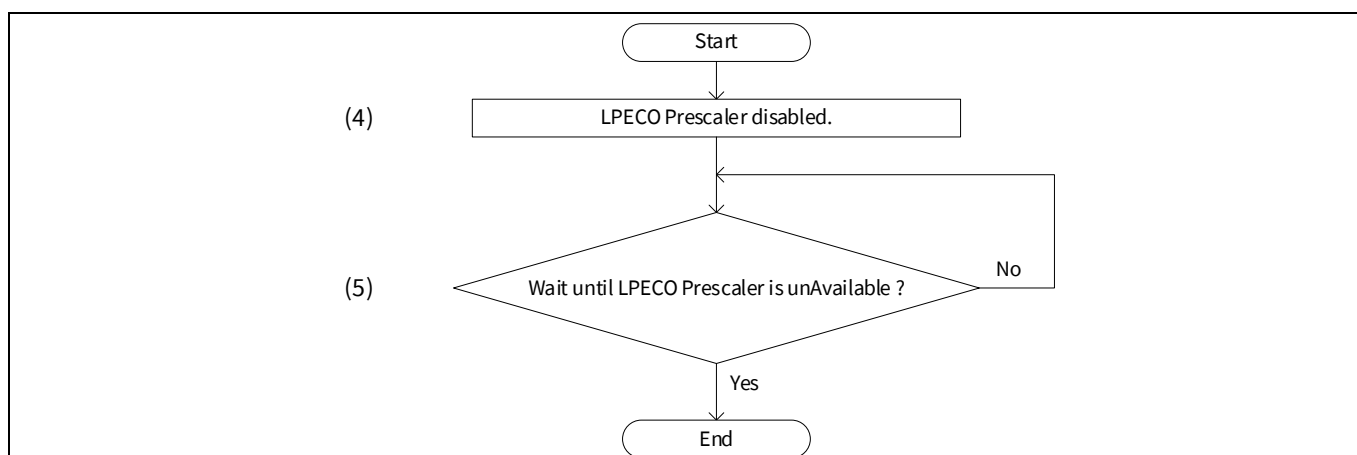


Figure 19 Disabling the LPECO_Prescaler

5.10.1 Use case

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

Configuring the internal clocks

5.10.2 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.	–

5.10.3 Sample code for the initial configuration of LPECO prescaler settings

Code Listing 51 to **Code Listing 55** show the sample code.

Code Listing 51 General configuration of LPECO prescaler settings

```

:
#define LPECO_PRESCALER_TARGET_FREQ (1234567ul)
:
/** Wait time definition */
#define WAIT_FOR_STABILIZATION (10000ul)
:
#define CLK_FREQ_LPECO (8000000ul)
:
#define PATH_SOURCE_CLOCK_FREQ CLK_FREQ_LPECO
:

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

```

Define the LPECO prescaler target frequency.

Define the TIMEOUT variable.

Define the LPECO clock frequency.

LPECO prescaler setting. See [Code Listing 52](#).

Configuring the internal clocks

Code Listing 52 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);

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

LPECO prescaler setting. See [Code Listing 53](#).

LPECO prescaler enable. See [Code Listing 54](#).

Code Listing 53 Cy_SysClk_ClkBak_LPECO_SetPrescale () 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 << 8ul);
    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 < 0x100ul) && (fixedPointDivNum64 > 0x40000ul))
    {
        return CY_SYSCLK_BAD_PARAM;
    }

    fixedPointDivNum = (uint32_t)fixedPointDivNum64;

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

    return CY_SYSCLK_SUCCESS;
}
```

Configure the LPECO prescaler. See [Code Listing 54](#).

Code Listing 54 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;
        BACKUP->unLPECO_PRESCALE.stcField.u10LPECO_INT_DIV = intDiv;
    }
}
```

(1) Configure the 8-bit fractional value and 10-bit integer value.

Code Listing 55 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;

    // Wait for eco prescaler to get enabled
    while(CY_SYSCLK_ECO_PRESCALE_ENABLE != Cy_SysClk_ClkBak_LPECO_PrescalerOkay())
    {
        if(0ul == timeoutus)
        {
            // Timeout
        }
    }
}
```

(2) Enable the LPECO prescaler.

(3) Wait until the LPECO prescaler is available.

Configuring the internal clocks

Code Listing 55 Cy_SysClk_ClkBak_LPECO_PrescaleEnable() function

```
        return CY_SYSCLK_TIMEOUT;
    }

    Cy_SysLib_DelayUs(1u);

    timeoutus--;
}

return CY_SYSCLK_SUCCESS;
}
```

Code Listing 56 Cy_SysClk_ClkBak_LPECO_PrescalerOkay() function

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

Check the prescaler status.

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

Code Listing 57 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.ulLPECO_EN = 0ul;

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

        Cy_SysLib_DelayUs(1u);

        timeoutus--;
    }

    return CY_SYSCLK_SUCCESS;
}
```

(4) Disable the LPECO prescaler.

(5) Wait until the LPECO prescaler is unavailable.

Supplementary information

6 Supplementary information

6.1 Input clocks in peripheral functions

Table 23 to **Table 34** list the clock inputs to each peripheral function. For detailed values of PCLK, see the “Peripheral clocks” section of the [datasheet](#).

Table 23 Clock input to the TCPWM

Peripheral function	Operation clock	Channel clock
TCPWM	CLK_GR3 (Group 3)	PCLK (PCLK_TCPWM0_CLOCKSx, x= 0 to 33)
		PCLK (PCLK_TCPWM0_CLOCKSy, y= 256 to 267)
		PCLK (PCLK_TCPWM0_CLOCKSz, z=512-527)

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)
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)

Supplementary information

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_HF0	CLK_GR4	CLK_HF5

Table 30 Clock input to LCD

Peripheral function	AHB interface clock	High-frequency clock, CLK_PERI	Low-frequency clock, CLK_LF
LCD	CLK_GR3	PCLK_LCD0_CLOCK	CLK_LF

Table 31 Clock input to TDM

Peripheral function	CLK_IF_SRSS			CLK_GRy
TDM	CLK_HF2	CLK_HF3	CLK_HF4	CLK_GR8

Table 32 Clock input to SG

Peripheral function	CLK_IF_SRSS			CLK_GRy
SG	CLK_HF2	CLK_HF3	CLK_HF4	CLK_GR8

Table 33 Clock input to PWM

Peripheral function	CLK_IF_SRSS			CLK_GRy
PWM	CLK_HF2	CLK_HF3	CLK_HF4	CLK_GR8

Table 34 Clock input to MIXER

Peripheral function	CLK_IF_SRSS			CLK_GRy
MIXER	CLK_HF2	CLK_HF3	CLK_HF4	CLK_GR8

Supplementary information

6.2 Clock calibration counter

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 more details, see the [architecture TRM](#).

Do the following to calibrate clock sources using the clock calibration counter:

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

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

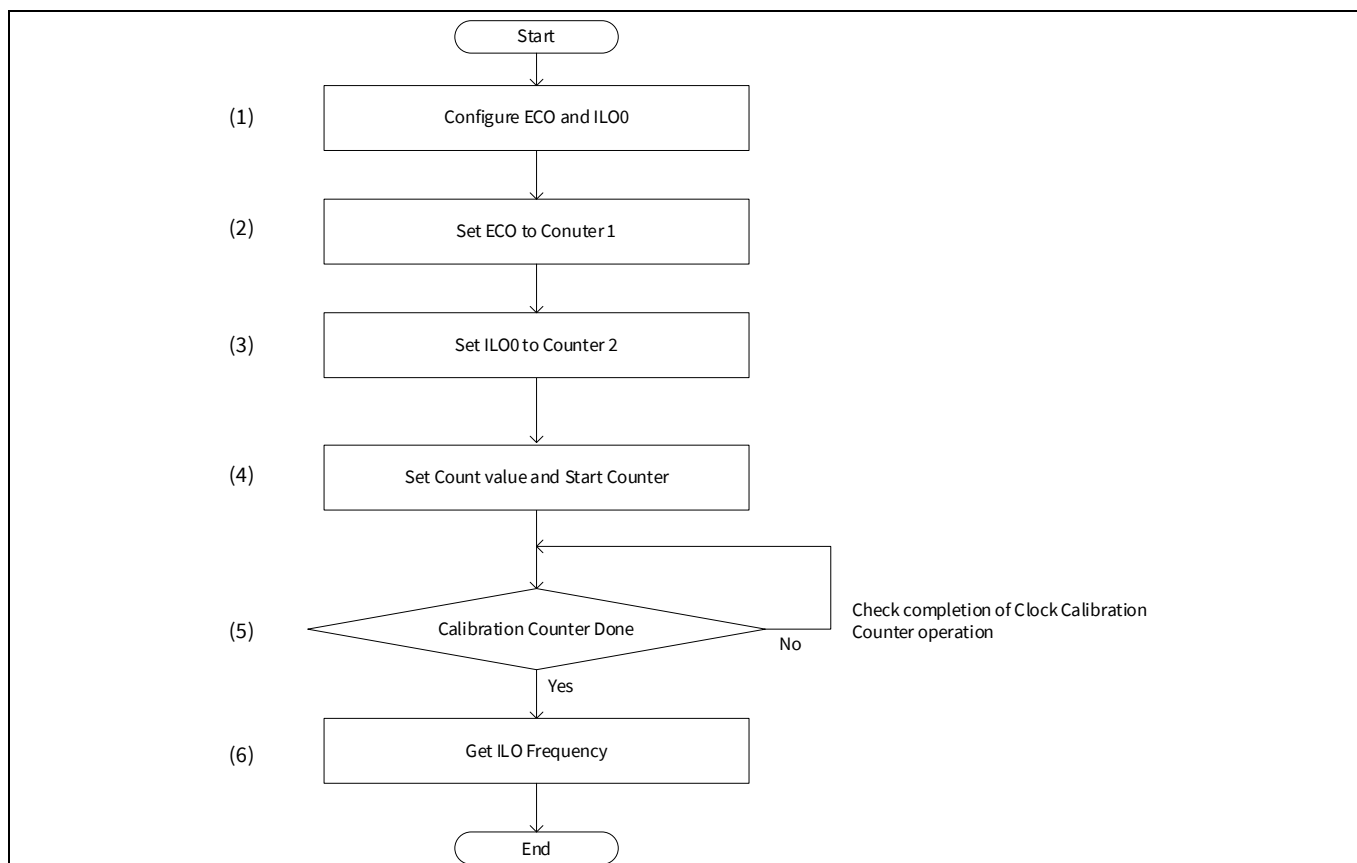


Figure 20 Example: Clock calibration counter with ILO 0 and ECO

Supplementary information

6.2.1.1 Use case

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

6.2.1.2 Configuration

Table 35 lists the parameters and **Table 36** lists the functions of the configuration part of in SDL for clock calibration counter with ILO0 and ECO settings.

Table 35 Clock calibration counter with ILO0 and ECO 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 36 Clock calibration counter with ILO0 and ECO setting functions

Functions	Description	Value
GetILOClockFreq()	Get the 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 the measurement clock frequency MesauredFreq: Stored measurement clock frequency refClkFreq: Reference clock frequency	MesauredFreq = clockMeasuredInfo[].measuredFreq refClkFreq = CLK_FREQ_ECO

Supplementary information

6.2.1.3 Sample code for initial configuration of clock calibration counter with ILO0 and ECO

See [Code Listing 58](#) to [Code Listing 62](#) for the sample code.

Code Listing 58 General configuration of clock calibration counter with ILO0 and ECO

```
#define CY_SYSClk_DIV_ROUND(a, b) (((a) + ((b) / 2u11)) / (b))
#define ILO_0 0u1
#define ILO_1 1u1
#define ILONo ILO_0
#define CLK_FREQ_ECO (16000000u1)

int32_t ILOFreq;

stc_clock_measure clockMeasuredInfo[] =
{
    #if(ILONo == ILO_0)
        { .name = CY_SYSClk_MEAS_CLK_ILO0, .measuredFreq= 0u1},
    #else
        { .name = CY_SYSClk_MEAS_CLK_ILO1, .measuredFreq= 0u1},
    #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 [3.1 ECO](#) and [3.4 ILO0/ILO1](#).

Get Clock Frequency. See [Code Listing 59](#).

Code Listing 59 GetILOClockFreq() function

```
uint32_t GetILOClockFreq(void)
{
    uint32_t counter1 = 40000u1;

    if((SRSS->unCLK_ECO_STATUS.stcField.u1ECO_OK == 0u1) || (SRSS->unCLK_ECO_STATUS.stcField.u1ECO_READY == 0u1))
    {
        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 Listing 60](#).

Check if the Counter Measurement is done. See [Code Listing 61](#).

Get ILO frequency. See [Code Listing 62](#).

Code Listing 60 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;
```

Supplementary information

Code Listing 60 Cy_SysClk_StartClkMeasurementCounters() function

```

:
:   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 Listing 61 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 Listing 62 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.

Supplementary information

6.3 CSV diagram, monitored clock, and reference clock

Figure 21 shows the clock diagram with the monitored clock and the reference clock of CSV. **Table 37** shows the relationship between the monitored clock and the reference clock.

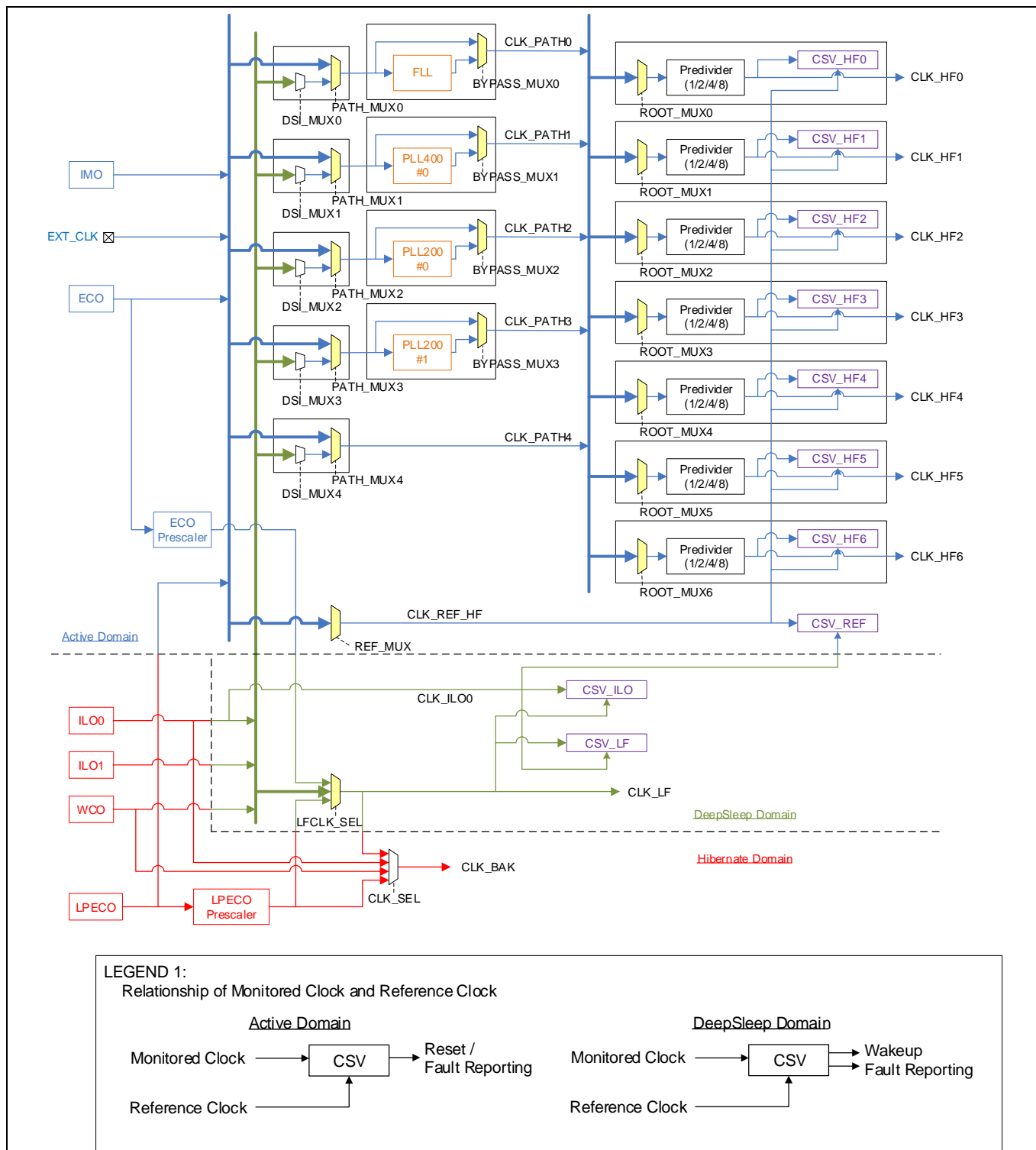


Figure 21 CSV diagram

Supplementary information

Table 37 Monitored clock and reference clock

CSV components	Monitor clock	Reference clock	Note
CSV_HF0	CLK_HF0	CLK_REF_HF	CLK_REF_HF is selected CLK_IMO, EXT_CLK, CLK_ECO or CLK_LPECO.
CSV_HF1	CLK_HF1	CLK_REF_HF	CLK_REF_HF is selected CLK_IMO, EXT_CLK, CLK_ECO or CLK_LPECO.
CSV_HF2	CLK_HF2	CLK_REF_HF	CLK_REF_HF is selected CLK_IMO, EXT_CLK, CLK_ECO or CLK_LPECO.
CSV_HF3	CLK_HF3	CLK_REF_HF	CLK_REF_HF is selected CLK_IMO, EXT_CLK, CLK_ECO or CLK_LPECO.
CSV_HF4	CLK_HF4	CLK_REF_HF	CLK_REF_HF is selected CLK_IMO, EXT_CLK, CLK_ECO or CLK_LPECO.
CSV_HF5	CLK_HF5	CLK_REF_HF	CLK_REF_HF is selected CLK_IMO, EXT_CLK, CLK_ECO or CLK_LPECO.
CSV_HF6	CLK_HF6	CLK_REF_HF	CLK_REF_HF is selected 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 WCO, ILO0, ILO1, ECO_Prescaler or LPECO_Prescaler.
CSV_LF	CLK_LF	ILO0(CLK_ILO0)	-

6.3.1 ILO0 calibration using clock calibration counter

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](#), and [registers TRM](#).

Figure 22 shows an example flow of ILO0 calibration using clock calibration counter and the CLK_TRIM_ILOx_CTL register.

Supplementary information

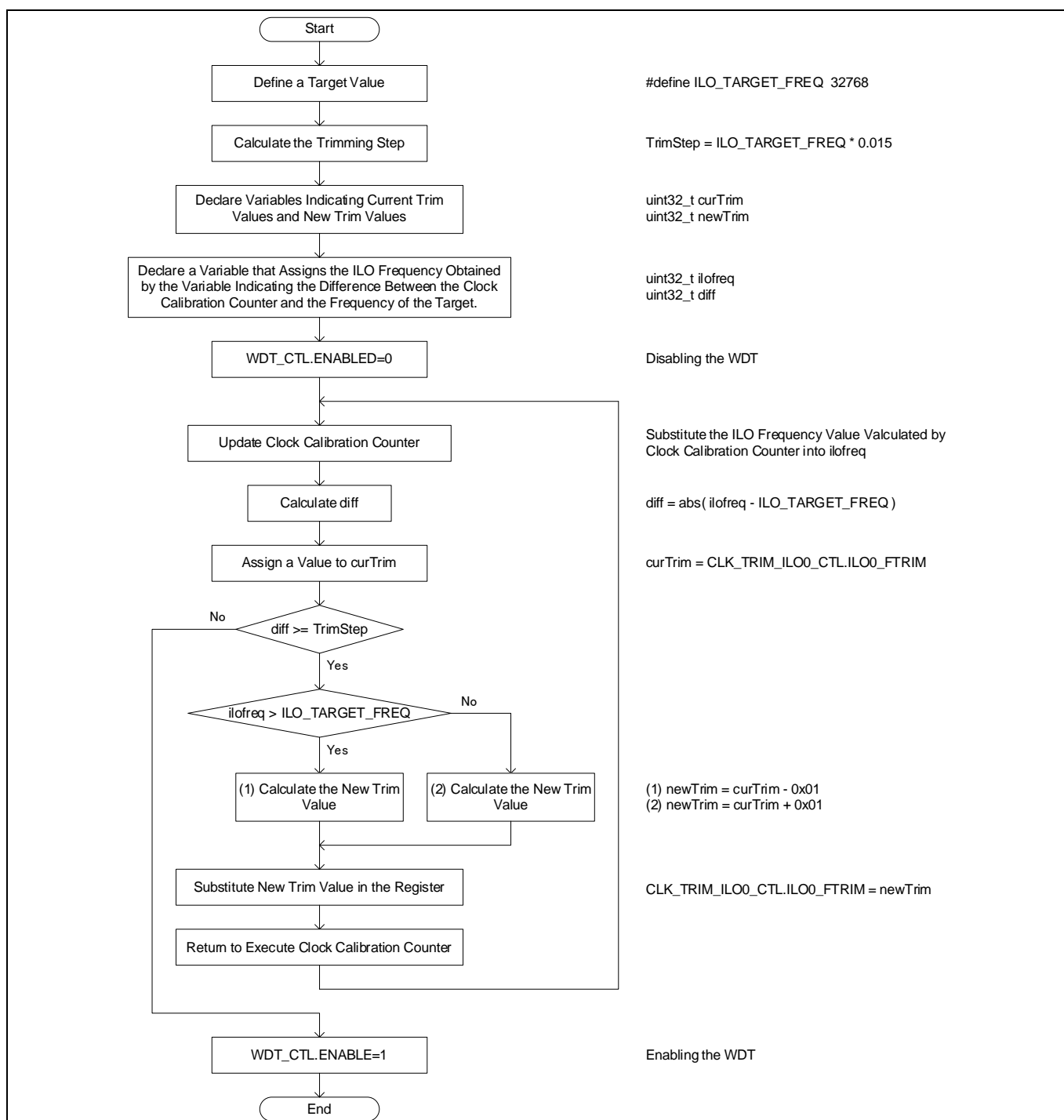


Figure 22 ILO0 calibration

Note: Calculation of each variable must be appropriately dealt with so that decimal places do not occur.

Glossary

7 Glossary

Terms	Description (See architecture TRM chapter)
CAN FD	CAN FD = CAN with Flexible Data rate; CAN = Controller Area Network. (See "CAN FD Controller").
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_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 CM0+ 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.
Clock calibration counter	clock calibration counter has a function to calibrate clock using two clocks.
CSV	clock supervision.
CXPI	clock extension peripheral interface. (See "Clock Extension Peripheral Interface (CXPI)")
ECO	external crystal oscillator
EXT_CLK	external clock
FLL	frequency-locked loop
FPU	floating-point unit
IMO	internal main oscillator
ILO	internal low-speed oscillator
LCD	Liquid Crystal Display. (See "LCD Controller")
LIN	Local Interconnect Network. (See "Local Interconnect Network (LIN)")
LPECO	low-power external crystal oscillator
MIXER	mixer combine multiple PCM source streams into a single PCM destination stream. (See "Sound subsystem")
Peripheral clock divider	peripheral clock divider derive a clock to use of each peripheral function.
PLL200	phase-locked loop. This PLL is not implemented with SSCG and fractional operation.
PLL400	phase-locked loop. This PLL is implemented with SSCG and fractional operation.
PWM	pulse width modulated. (See "Sound subsystem")
SAR ADC	successive approximation register analog-to-digital converter. (See "SAR ADC")
SCB	serial communications block. (See "Serial communications block (SCB)").
SG	sound generator. (See "Sound subsystem")
SMIF	serial memory interface. (See "Serial memory interface")
TCPWM	timer, counter, and pulse width modulator. (See "Timer, counter, and PWM")
TDM	time division multiplexed. (See "Sound subsystem")
WCO	watch crystal oscillator

References

8 References

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

[1] Device datasheets:

- CYT2CL datasheet 32-bit Arm® Cortex® -M4F microcontroller TRAVEO™ T2G family (Doc No. 002-32508)

[2] Technical reference manuals:

- Cluster entry family
 - TRAVEO™ T2G automotive cluster entry family architecture technical reference manual (TRM) (Doc No. 002-33175)
 - TRAVEO™ T2G automotive cluster entry registers technical reference manual (TRM) for CYT2CL (Doc No. 002-33404)

[3] User Guide:

- Setting the ECO Parameters in TRAVEO™ T2G User Guide (Contact [Technical Support](#))

Other references

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 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

Revision history

Document version	Date of release	Description of changes
**	2020-06-10	New application note.
*A	2021-02-24	Migrated to Infineon template.
*B	2021-11-04	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”.

Trademarks

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

Edition 2021-11-04

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2021 Infineon Technologies AG.

All Rights Reserved.

Do you have a question about this document?

Go to www.cypress.com/support

Document reference

002-29513 Rev. *B

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.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (www.infineon.com).

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.