AN1492: Clock Manager Migration Guide

This application note provides guidelines and considerations for migrating projects that were created using the previous clock management architecture in the Gecko SDK to the new Clock Manager module in the Simplicity SDK.

KEY POINTS

- Oscillator and clock tree configuration using the Clock Manager module
- CMU emlib API to Clock Manager API replacements
- Step-by-step guide on how to migrate a Series 2 Gecko SDK project to a Simplicity SDK project that uses the Clock Manager module

1. Device Compatibility

This application note supports multiple device families.

EFR32 Wireless Gecko Series 2 consists of:

- EFR32BG21
- EFR32MG21
- EFR32BG22
- EFR32FG22
- EFR32MG22
- EFR32FG23
- EFR32ZG23
- EFR32SG23
- EFR32BG24
- EFR32MG24
- EFR32FG25
- EFR32BG27
- EFR32MG27
- EFR32FG28
- EFR32ZG28
- EFR32SG28

EFM32 Series 2 consists of:

- EFM32PG22
- EFM32PG23
- EFM32PG28

2. Introduction

The Clock Manager module is a new software module introduced in the Simplicity SDK (SiSDK). The Clock Manager module uses Clock Manager APIs and consists of two software components:

- Clock Manager component: handles initialization and configuration of the oscillators and clock tree
- Clock Manager Runtime component: handles runtime processes such as clock configuration, calibration, tuning, and more

In the Gecko SDK (GSDK), clock initialization and configuration is handled by some of the Device Initialization software components. The relevant components consist of:

- Clock initialization component: `sl_device_init_clocks`
- HFRCO initialization component: `sl_device_init_hfrco`
- HFXO initialization component: `sl_device_init_hfxo`
- LFRCO initialization component: `sl_device_init_lfrco`
- LFXO initialization component: `sl_device_init_lfxo`
- RFF PLL initialization component: `sl_device_init_rffpll`
- USB PLL initialization component: `sl_device_init_usbpll`

The clock initialization component automatically configures the clock tree depending on which Device Initialization components are installed. The remaining components configure and initialize the respective oscillators and PLLs. The clocks can further be configured during runtime using the CMU emlib APIs.

When migrating a Series 2 project from the GSDK to the SiSDK:

- The Clock Manager component is not automatically installed. The clock related Device Initialization components will remain installed. These components are mutually exclusive, meaning they cannot be installed at the same time. Users can manually install the Clock Manager component via the Project Configurator in the *slcp file. Installing the Clock Manager component will automatically uninstall the clock related Device Initialization components.
- The Clock Manager Runtime component is typically automatically installed as a dependency for other standard project software components. Although the Clock Manager Runtime component is intended to be a replacement for the CMU emlib APIs, Clock Manager APIs can be used alongside CMU emlib APIs. User calls to CMU emlib APIs are not automatically replaced with the Clock Manager APIs when updating the SDK.

This allows users to migrate their Series 2 projects from the GSDK to the SiSDK with minimal changes. Users have the option to manually migrate their project to only use the Clock Manager module. Full adoption of the Clock Manager module is recommended as future devices will not support CMU emlib APIs nor the clock related Device Initialization components.

3. Initialization

The GSDK uses multiple Device Initialization components and configuration files to configure and initialize each oscillator. The Clock Manager module consolidates the oscillator configuration and initialization into a single Clock Manager component. Oscillators are configured via the Project Configurator or the CMSIS annotated configuration file, sl_clock_manager_oscillator_config.h.

Table 3.1 Initialization Substitutions on page 4 contains the CMU emlib initialization functions that can be called by the Device Initialization components. The Clock Manager component combines the initialization into a single Clock Manager API.

Table 3.1. Initialization Substitutions

CMU emlib API	Clock Manager API

`void CMU_LFXOInit(`
`  const CMU_LFXOInit_TypeDef *lfxoInit)`
 	
`void CMU_HFXOInit(`
`  const CMU_HFXOInit_TypeDef *hfxoInit)`
 	
`bool CMU_DPLLLock(`
`  const CMU_DPLLInit_TypeDef *init)`
 	
`void CMU_RFFPLLInit(`
`  const CMU_RFFPLL_Init_TypeDef *pllInit)`
 	
`void CMU_USBPLLInit(`
`  const CMU_USBPLL_Init_TypeDef *pllInit)`
	`sl_status_t sl_clock_manager_init(void)`

4. Configuration

The clock initialization component, `sl_device_init_clocks`, is a Device Initialization component that generates a non-user configurable clock tree during initialization. The Clock Manager component allows for user customization of the device's clock tree during initialization. The clock tree can be configured via the Project Configurator or the CMSIS annotated configuration file, sl_clock_manager_tree_config.h.

There are CMU emlib API functions for setting and getting individual clock sources and dividers, but there are no equivalent Clock Manager API functions to read or modify the clock tree during runtime. Instead, the clock source or divider is defined by macros in the configuration file. Table 4.1 Table 2 on page 5 shows the CMU emlib APIs for configuring the clock tree and examples of the corresponding Clock Manager macros, specifically for the PCLK divider and SYSCLK source.

Table 4.1. Configuration Substitutions Example

CMU emlib API	Clock Manager Configuration
`CMU_ClkDiv_TypeDef CMU_ClockDivGet( CMU_Clock_TypeDef clock)`	`SL_CLOCK_MANAGER_PCLK_DIVIDER`
`void CMU_ClockDivSet( CMU_Clock_TypeDef clock, CMU_ClkDiv_TypeDef div)`	`#define SL_CLOCK_MANAGER_PCLK_DIVIDER divider`
`void CMU_ClockSelectGet( CMU_Clock_TypeDef clock)`	`SL_CLOCK_MANAGER_SYSCLK_SOURCE`
`void CMU_ClockSelectSet( CMU_Clock_TypeDef clock, CMU_Select_TypeDef ref)`	`#define SL_CLOCK_MANAGER_SYSCLK_SOURCE source`

Table 4.2 Additional Configuration Examples on page 5 shows the remaining CMU emlib APIs that do not have a corresponding Clock Manager API function but instead have a corresponding macro configuration.

Table 4.2. Additional Configuration Examples

CMU emlib API	Clock Manager Configuration
`CMU_HFRCODPLLFreq_TypeDef CMU_HFRCODPLLBandGet(void)`	`SL_CLOCK_MANAGER_HFRCO_Band`
`void CMU_HFRCODPLLBandSet( CMU_HFRCODPLLFreq_TypeDef freq)`	`#define SL_CLOCK_MANAGER_HFRCO_BAND freq`
`void CMU_LFXOPrecisionSet( uint16_t precision)`	`#define SL_CLOCK_MANAGER_LFXO_PRECISION precision`
`uint16_t CMU_LFXOPrecisionGet(void)`	`SL_CLOCK_MANAGER_LFXO_PRECISION`
`void CMU_HFXOPrecisionSet(uint16_t precision)`	`#define SL_CLOCK_MANAGER_HFXO_PRECISION precision`
`uint16_t CMU_HFXOPrecisionGet(void)`	`SL_CLOCK_MANAGER_HFXO_PRECISION`
`sl_status_t CMU_HFXOCTuneSet(uint32_t ctune)`	`#define SL_CLOCK_MANAGER_HFXO_CTUNE ctune`
`void CMU_LFRCOSetPrecision( CMU_Precision_TypeDef precision)`	`#define SL_CLOCK_MANAGER_LFRCO_PRECISION precision`
`void CMU_PCNTClockExternalSet( unsigned int instance, true)`	`#define SL_CLOCK_MANAGER_PCNT0CLK_SOURCE CMU_PCNT0CLKCTRL_CLKSEL_PCNTS0`

CMU emlib API	Clock Manager Configuration
`void CMU_PCNTClockExternalSet( unsigned int instance, false)`	`#define SL_CLOCK_MANAGER_PCNT0CLK_SOURCE CMU_PCNT0CLKCTRL_CLKSEL_EM23GRPACLK`
`CMU_HFRCOEM23Freq_TypeDef CMU_HFRCOEM23BandGet(void)`	`SL_CLOCK_MANAGER_HFRCOEM23_BAND`
`void CMU_HFRCOEM23BandSet( CMU_HFRCOEM23Freq_TypeDef freq)`	`#define SL_CLOCK_MANAGERHFRCOEM23_BAND freq`
`void CMU_HFXOStartCrystalSharingLeader( const CMU_BUFOUTLeaderInit_TypeDef *bufoutInit, GPIO_Port_TypeDef port, unsigned int pin)`	`SL_CLOCK_MANAGER_HFXO_CRYSTAL_SHARING_LEADER_EN`
`void CMU_HFXOCrystalSharingFollowerInit( CMU_PRS_Status_Output_Select_TypeDef prsStatusSelectOutput, unsigned int prsAsyncCh, GPIO_Port_TypeDef port, unsigned int pin)`	`SL_CLOCK_MANAGER_HFXO_CRYSTAL_SHARING_FOLLOWER_EN`

5. Runtime

Table 5.1 Runtime Substitutions on page 7 contains CMU emlib runtime APIs and their equivalent Clock Manager APIs. Some CMU emlib APIs require calls to multiple Clock Manager APIs to maintain similar functionality. Note that function arguments and return values may be different.

Table 5.1. Runtime Substitutions

CMU emlib API	Clock Manager API	Notable Changes

`uint32_t CMU_Calibrate(`
`  uint32_t cycles,`
`  CMU_Select_TypeDef ref)`
	`sl_status_t`
	`sl_clock_manager_configure_rco_calibration(`
	`  uint32_t cycles,`
	`  sl_clock_manager_clock_calibration_t down_counter_selection,`
	`  sl_clock_manager_clock_calibration_t up_counter_selection,`
	`  bool continuous_calibration)`
	
	`sl_status_t`
	`sl_clock_manager_start_rco_calibration(void)`
	
	`sl_status_t`
	`sl_clock_manager_get_rco_calibration_count(`
	`  uint32_t *count)`
		CMU_Calibrate calibrates an oscillator using the number of HCLK cycles and a reference clock and returns the number of ticks of the reference clock. Three Clock Manager APIs are required to replicate the behavior of *CMU_Calibrate*. The Clock Manager API *configure_rco_calibration* takes in the number of clock cycles, down and up counter selection, and a Boolean type that can enable continuous calibration. Next, *start_rco_calibration* starts the RCO calibration. Lastly, *get_rco_calibration_count* retrieves the calibration count value and updates the *count* variable via call-by-reference. All three of the Clock Manager APIs return the status.

CMU emlib API	Clock Manager API	Notable Changes

`void CMU_CalibrateConfig(`
`  uint32_t downCycles,`
`  CMU_Select_TypeDef downSel,`
`  CMU_Select_TypeDef upSel)`
 	
`void CMU_CalibrateCont(bool enable)`
	`sl_status_t`
	`sl_clock_manager_configure_rco_calibration(`
	`  uint32_t cycles,`
	`  sl_clock_manager_clock_calibration_t down_counter_selection,`
	`  sl_clock_manager_clock_calibration_t up_counter_selection,`
	`  bool continuous_calibration)`
		CMU_CalibrateConfig configures the clock calibration by specifying the number of clock cycles and selecting a down and up counter. *CMU_CalibrateCont* enables continuous calibration according to the Boolean argument. The Clock Manager API combines the two CMU APIs and takes in the number of clock cycles, down and up counter selection, and a Boolean type that can enable continuous calibration. Lastly, it returns the status.

`uint32_t CMU_CalibrateCountGet(void)`	`void`
	`sl_clock_manager_wait_rco_calibration(`
	`  void)`
	
	`sl_status_t`
	`sl_clock_manager_get_rco_calibration_count(`
	`  uint32_t *count)`
		CMU_CalibrateCountGet returns the calibration count of the UPSEL clock. Two Clock Manager APIs are required to replicate the behavior of *CalibrateCountGet*. The first function waits for RCO calibration to complete and the second function updates the *count* variable via call-by-reference and returns the status.

`void CMU_ClkOutPinConfig(`
`  uint32_t clkNo,`
`  CMU_Select_TypeDef sel,`
`  CMU_ClkDiv_TypeDef clkDiv,`
`  GPIO_Port_TypeDef port,`
`  unsigned int pin)`
	`sl_status_t`
	`sl_clock_manager_set_gpio_clock_output(`
	`  sl_clock_manager_export_clock_source_t export_clock_source,`
	
	`sl_clock_manager_export_clock_output_select_t output_select,`
	`  uint16_t hfexp_divider,`
	`  uint32_t port,`
	`  uint32_t pin)`
		CMU_ClkOutPinConfig configures the pin by assigning it the clock output number, clock divider, and the port and pin number. The Clock Manager API configures the pin by assigning the clock source, clock output number, divider value, and the port and pin number. The clock sources and output numbers are changed to Clock Manager types and the status is returned.

CMU emlib API	Clock Manager API	Notable Changes

`void CMU_ClockEnable(`
`  CMU_Clock_TypeDef clock,`
`  bool enable)`
	`sl_status_t`
	`sl_clock_manager_enable_bus_clock(`
	`  sl_bus_clock_t module)`
	
	`sl_status_t`
	`sl_clock_manager_disable_bus_clock(`
	`  sl_bus_clock_t module)`
		CMU_ClockEnable selects a clock and enables it using a Boolean type. The Clock Manager separates the CMU API into two functions. The Clock Manager APIs argument is a bus clock pointer, and it returns the status.

`uint32_t CMU_ClockFreqGet(`
`  CMU_Clock_TypeDef clock)`
	`sl_status_t`
	`sl_clock_manager_get_oscillator_frequency(`
	`  sl_oscillator_t oscillator,`
	`  uint32_t *frequency)`
	
	`sl_status_t`
	`sl_clock_manager_get_clock_branch_frequency(`
	`  sl_clock_branch_t clock_branch,`
	`  uint32_t *frequency)`
		CMU_ClockFreqGet retrieves the frequency of a specified clock and returns its value. The Clock Manager API separates the CMU API into two functions, which retrieves either the oscillator or the clock branch frequency. The frequency variable is then updated via call-by-reference and the status is returned.

`uint16_t CMU_HF_ClockPrecisionGet(`
`  CMU_Clock_TypeDef clock)`
 	
`uint16_t CMU_LF_ClockPrecisionGet(`
`  CMU_CLock_TypeDef clock)`
	`sl_status_t`
	`sl_clock_manager_get_clock_branch_precision(`
	`  sl_clock_branch_t clock_branch,`
	`  uint16_t *precision)`
		The CMU API retrieves the precision of either an LF or HF clock and returns its value. The Clock Manager API retrieves the precision of a selected clock branch. The precision variable is updated via call-by-reference and the status is returned.

CMU emlib API	Clock Manager API	Notable Changes

`uint32_t CMU_OscillatorTuningGet(`
`  CMU_Osc_TypeDef osc)`
	`sl_status_t`
	`sl_clock_manager_get_rc_oscillator_calibration(`
	`  sl_oscillator_t oscillator,`
	`  uint32_t *val)`
	
	`sl_status_t`
	`sl_clock_manager_get_hfxo_calibration(`
	`  uint32_t *val)`
	
	`sl_status_t`
	`sl_clock_manager_get_lfxo_calibration(`
	`  uint32_t *val)`
		CMU_OscillatorTuningGet retrieves the tuning frequency of the specified oscillator and returns its value. The Clock Manager API separates the CMU API into three functions.The Clock Manager retrieves the tuning frequency of the specified oscillator. The tuning frequency variable is updated via call-by-reference and the status is returned. If the oscillator is the HFXO or the LFXO, the Clock Manager API *hfxo_calibration* or *lfxo_calibration* should be used respectively.

`void CMU_OscillatorTuningSet(`
`  CMU_Osc_TypeDef osc,`
`  uint32_t val)`
	`sl_status_t`
	`sl_clock_manager_set_rc_oscillator_calibration(`
	`  sl_oscillator_t oscillator,`
	`  uint32_t val)`
	
	`sl_status_t`
	`sl_clock_manager_set_hfxo_calibration(`
	`  uint32_t val)`
	
	`sl_status_t`
	`sl_clock_manager_set_lfxo_calibration(`
	`  uint32_t val)`
		CMU_OscillatorTuningSet sets the tuning frequency of an oscillator according to the argument, *val*. The Clock Manager API separates the CMU API into three functions. The Clock Manager sets the tuning frequency of the oscillator according to the argument, *val*. The status is then returned.
		If the oscillator is the HFXO or the LFXO, the Clock Manager API *set_hfxo_calibration* or *set_lfxo_calibration* should be used respectively.

`void CMU_CalibrateStart(void)`	`void`
	`sl_clock_manager_start_rco_calibration(`
	`  void)`
		-

`void CMU_CalibrateStop(void)`	`void`
	`sl_clock_manager_stop_rco_calibration(`
	`  void)`
		-

`uint32_t CMU_HFXOCTuneGet(void)`	`sl_status_t`
	`slx_clock_manager_hfxo_get_ctune(`
	`  uint32_t *ctune)`
		CMU_HFXOTuneGet returns the crystal tuning capacitance of the HFXO. The Clock Manager API updates the *ctune* variable via call-by-reference and returns the status.

CMU emlib API	Clock Manager API	Notable Changes
`sl_status_t CMU_HFXOCTuneSet( uint32_t ctune)`	`sl_status_t slx_clock_manager_hfxo_set_ctune( uint32_t ctune)`	*CMU_HFXOCTuneSet* sets the crystal tuning capacitance of the HFXO according to the argument, *ctune*, and returns the status. The Clock Manager API sets the crystal tuning capacitance of the HFXO according to the argument, *ctune*, and returns the status.
`void CMU_HFXOCoreBiasCurrentCalibrate( void)`	`sl_status_t slx_clock_manager_hfxo_calibrate_ctune( uint32_t ctune)`	The CMU API recalibrates the HFXO's core bias current. A CTUNE value can be passed in the argument *ctune* to change the value of CTUNE before launching a core bias current calibration. If the current *ctune* value is given, only a core bias optimization will be performed and CTUNE will remain unchanged.

Table 5.2 CMU emlib APIs with No Substitutions on page 11 contains the CMU emlib APIs that have no equivalent Clock Manager API.

Table 5.2. CMU emlib APIs with No Substitutions

CMU API

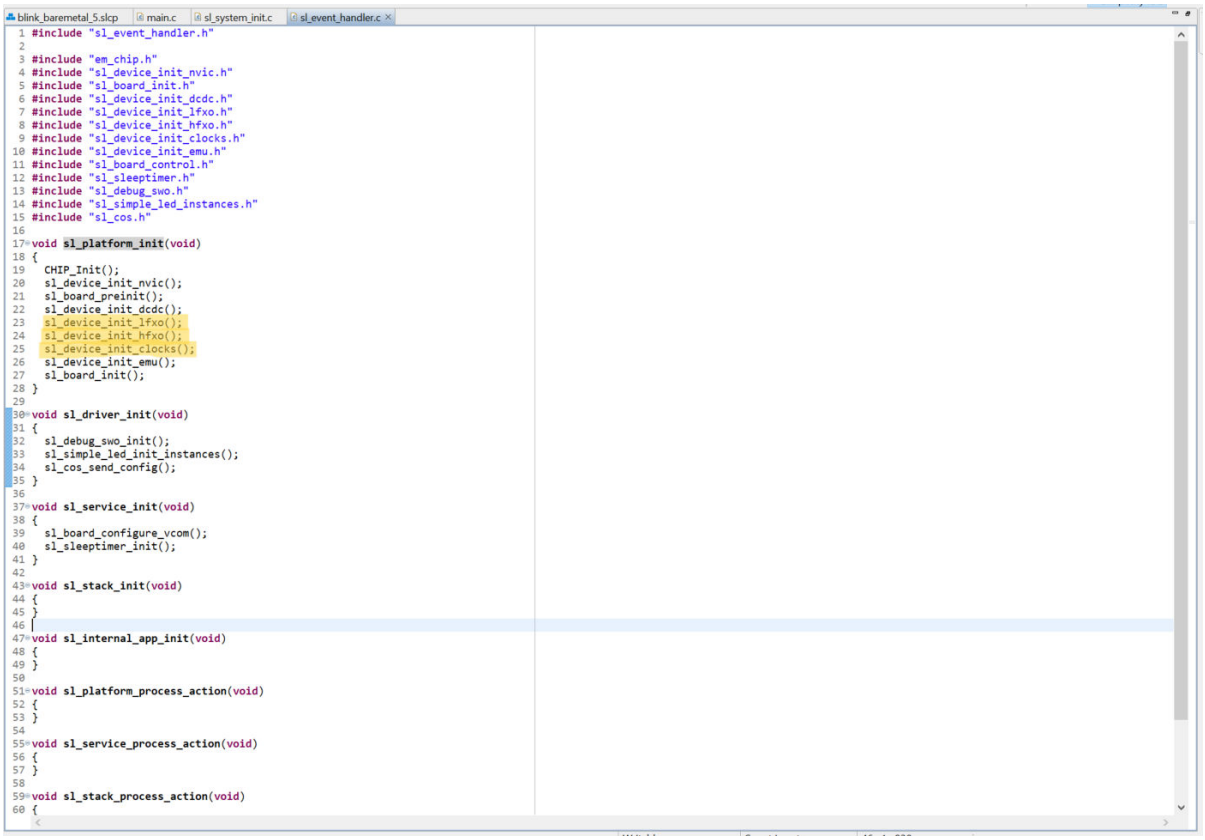
`CMU_DPLLLock`
`CMU_WaitUSBPLLLock`
`CMU_WaitRFFPLLLock`
`CMU_DPLLUnlock`
`CMU_WdogLock`
`CMU_WdogUnlock`
`CMU_HFXOCTuneDeltaSet`
`CMU_HFXOCTuneDeltaGet`
`CMU_UpdateWaitStates`
`CMU_IntClear`
`CMU_IntEnable`
`CMU_IntGet`
`CMU_IntGetEnabled`
`CMU_IntSet`
`CMU_Lock`
`CMU_Unlock`

6. Migration Guide

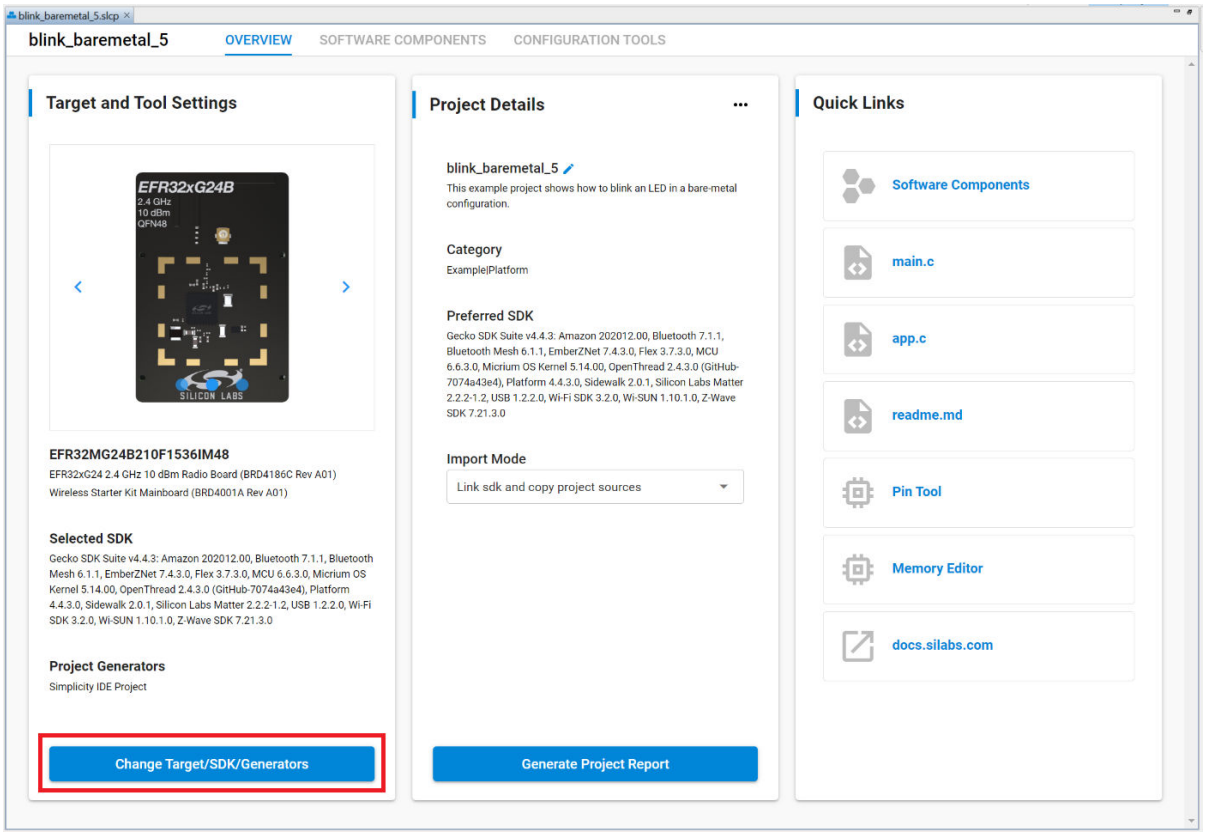
This section provides a step-by-step guide on how to migrate an existing project that uses the GSDK to using the SiSDK and the Clock Manager module.

6.1 Installation

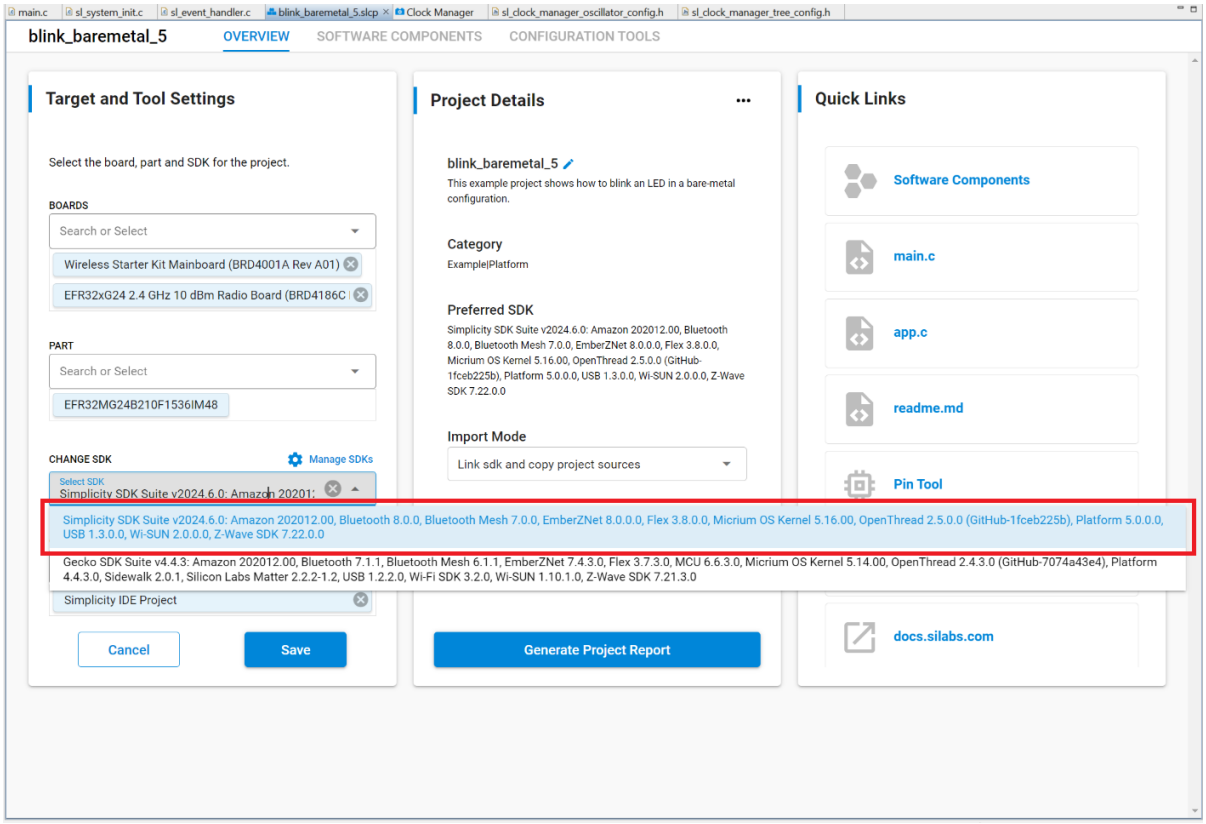
1. In the Gecko SDK, the installed Device Initialization software components can be found in the Project Configurator. The Device Initialization components are located inside the `sl_platform_init` function which is embedded within the `sl_system_init` function inside the main.c file. `sl_device_init_lfxo`, `sl_device_init_hfxo`, and `sl_device_init_clocks` are installed in this example project.



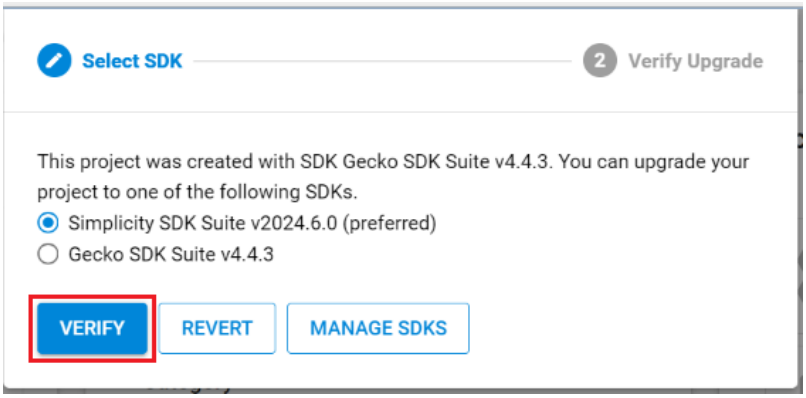
2. To update the SDK, open the *.slcp file and press Change Target/SDK/Generators.



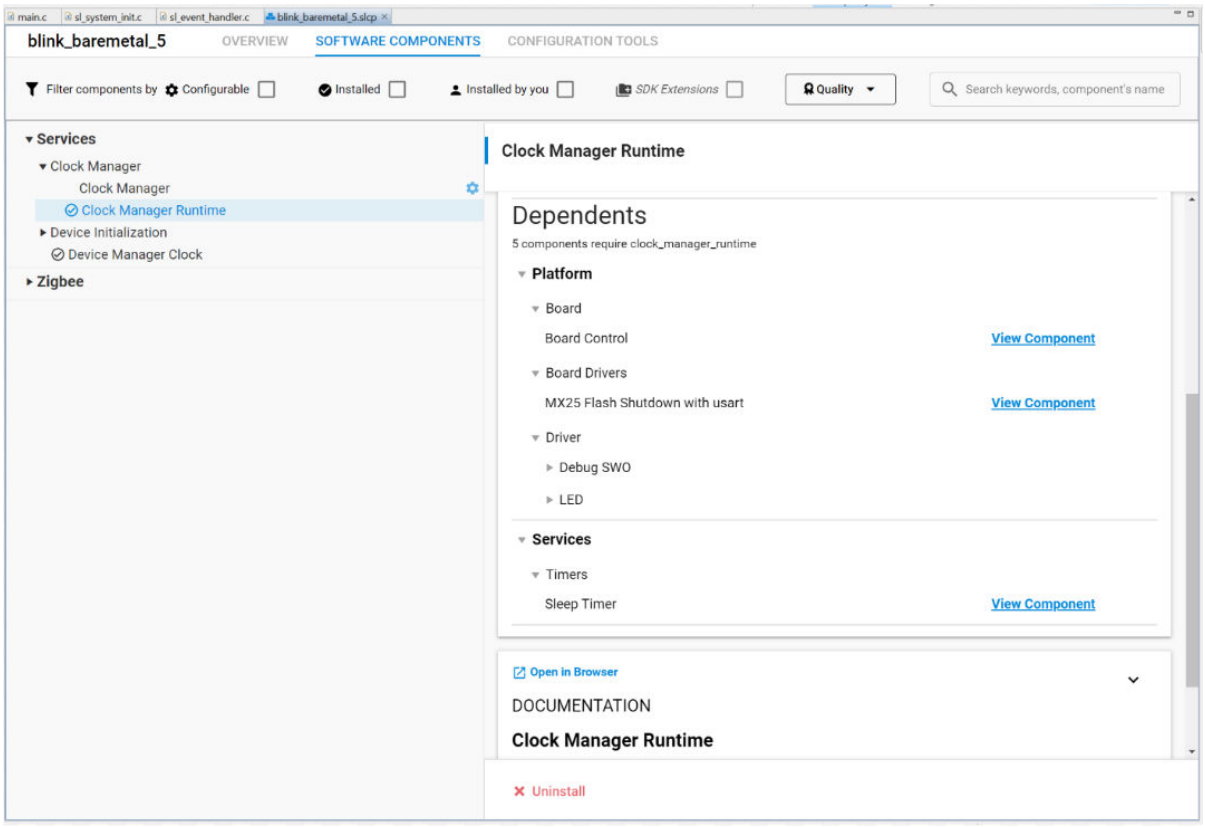
3. Change the SDK to the Simplicity SDK. In this installation example, the SDK is upgraded from the Gecko SDK Suite v4.4.3 to the Simplicity SDK Suite v2024.6.0.



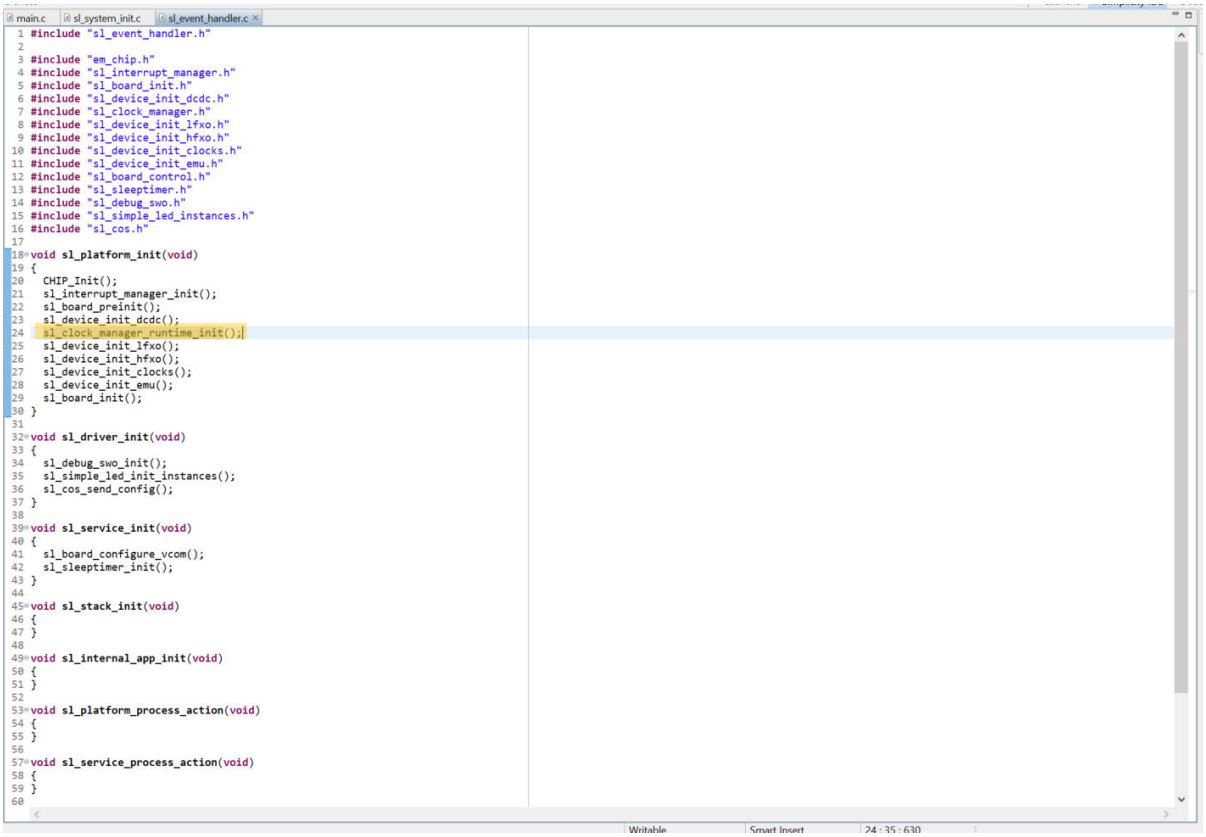
4. Press Verify.



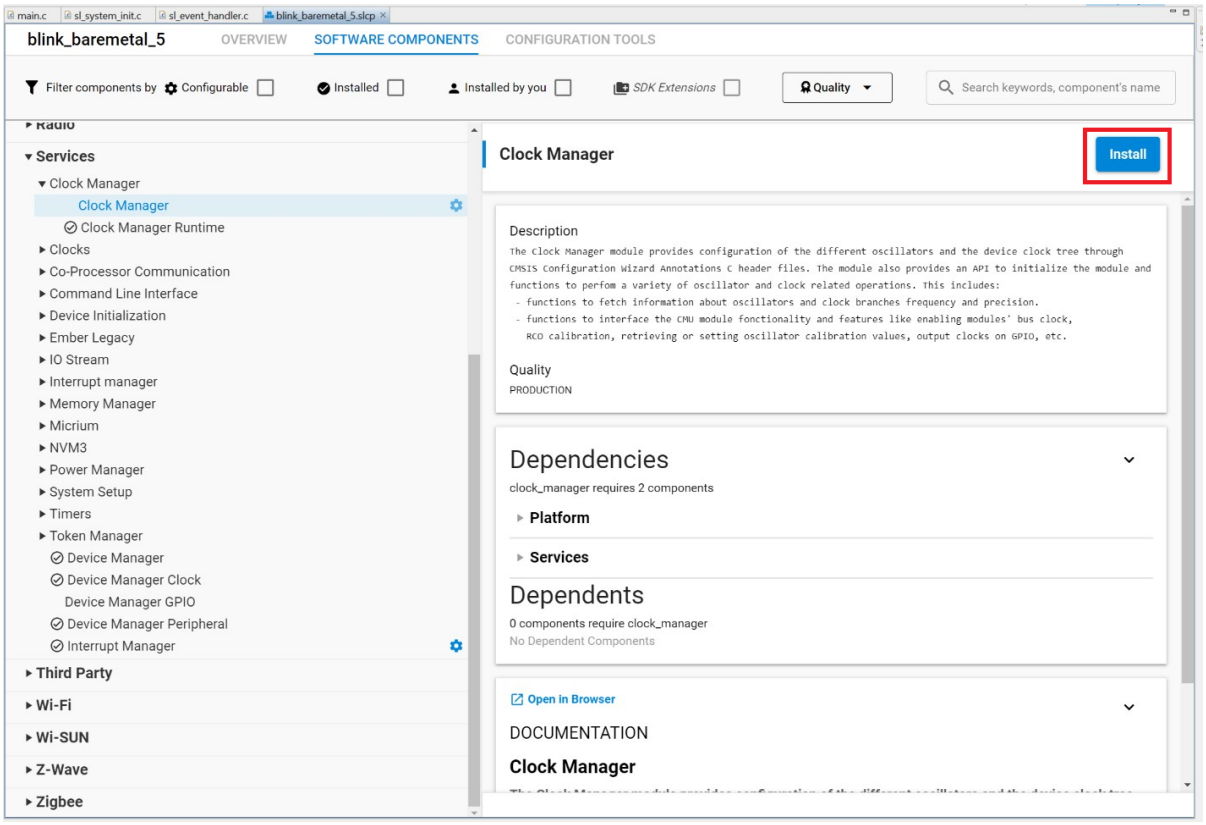
5. After the SDK update, the Clock Manager Runtime component should be automatically installed if there are dependencies in the project. The Clock Manager Runtime component and its dependencies can be found in the *.slcp, under the Services section.



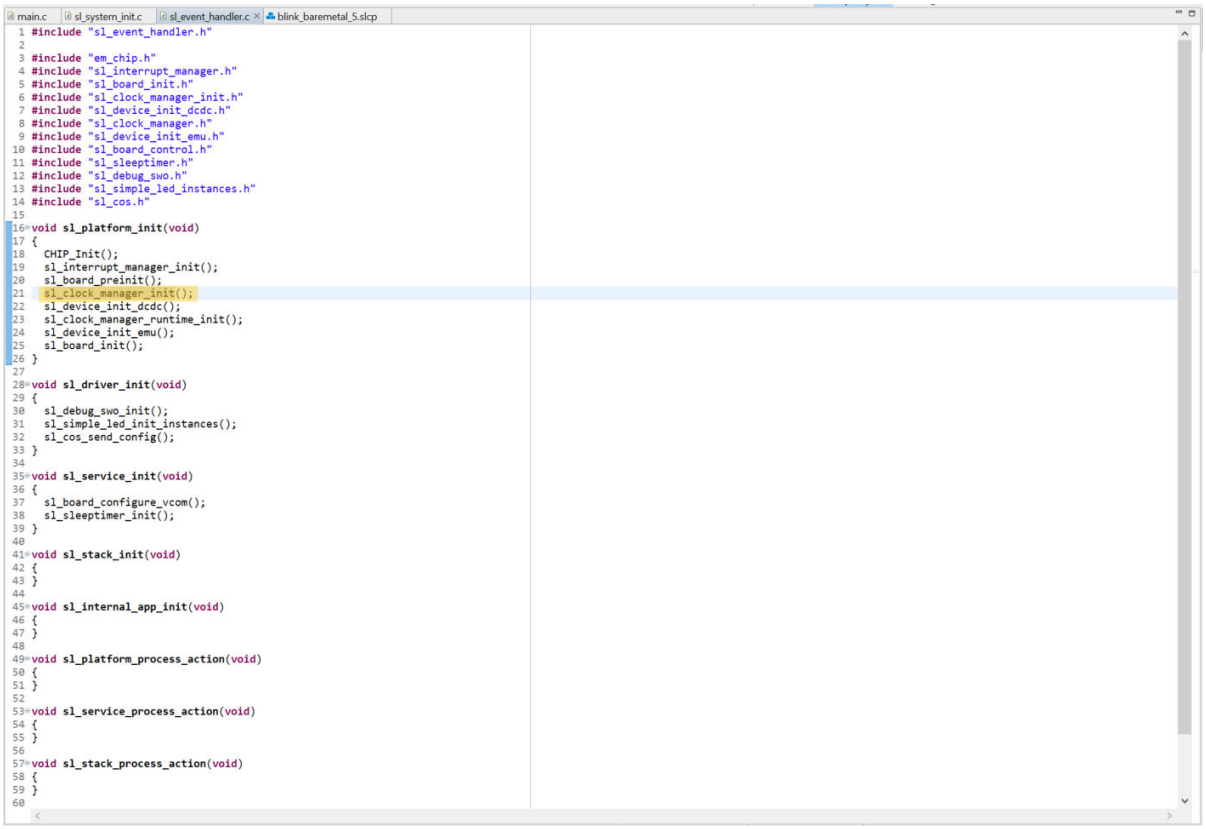
6. After the Clock Manager Runtime component has been installed, the `sl_platform_init` function will be updated as follows.



7. Users have the option to install the Clock Manager component.

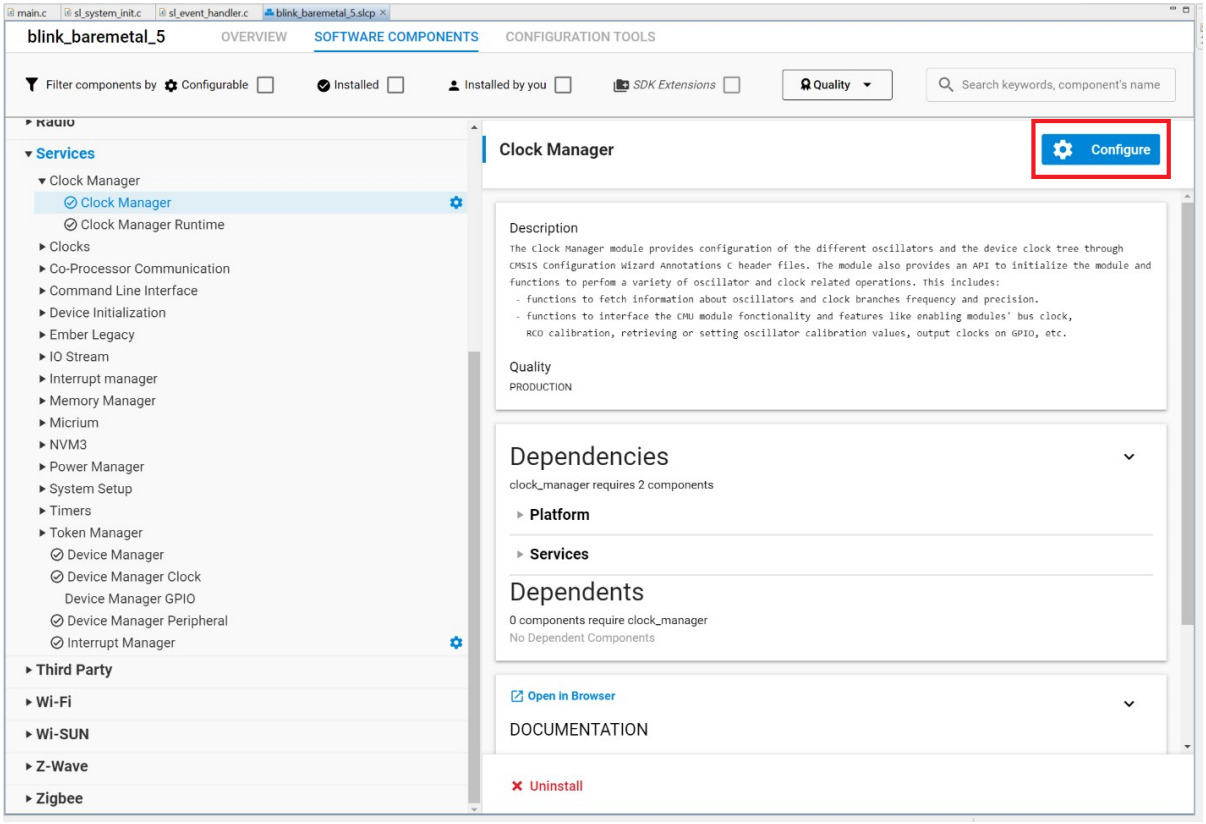


8. After installing the Clock Manager component, `sl_platform_init` now calls the `sl_clock_manager_init` function and no longer calls the Device Initialization components.

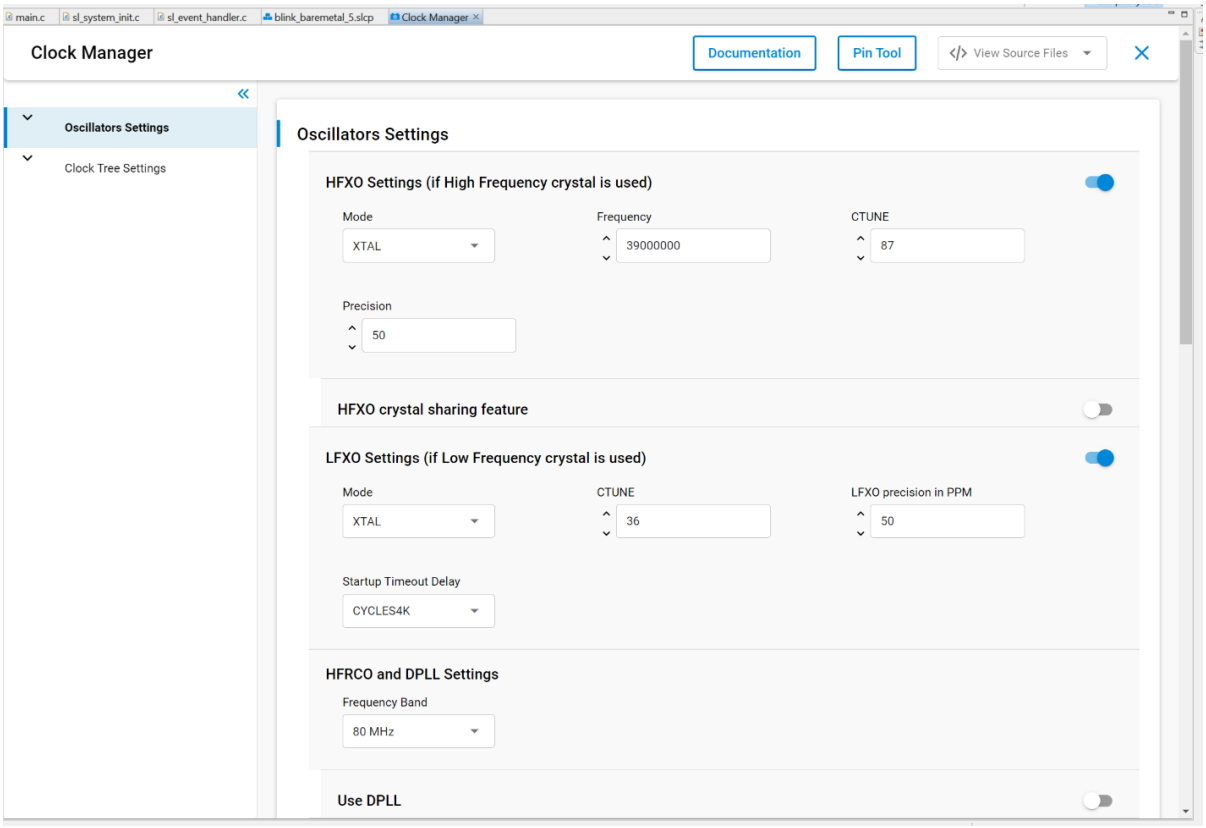


6.2 Configuration

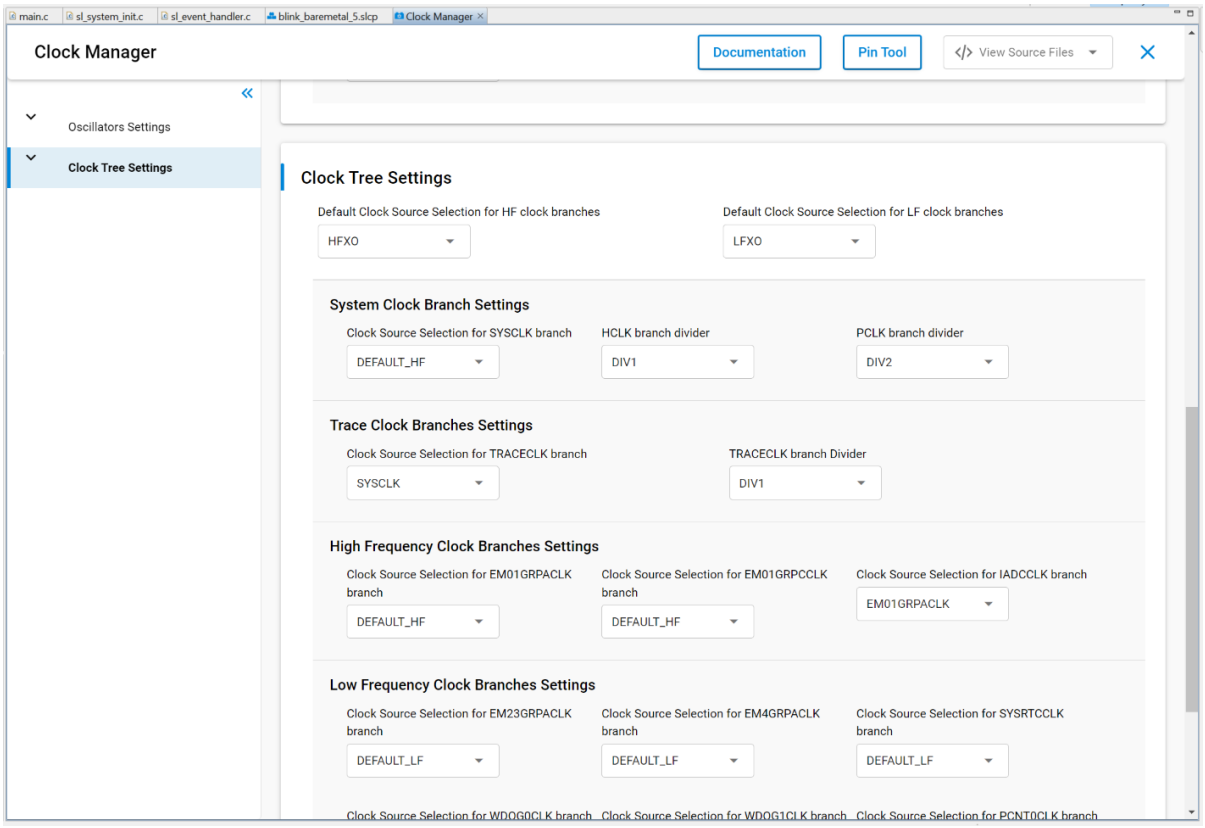
1. After installation, the Clock Manager component can be configured in the *.slcp. Press Configure to open the oscillator and clock tree settings.



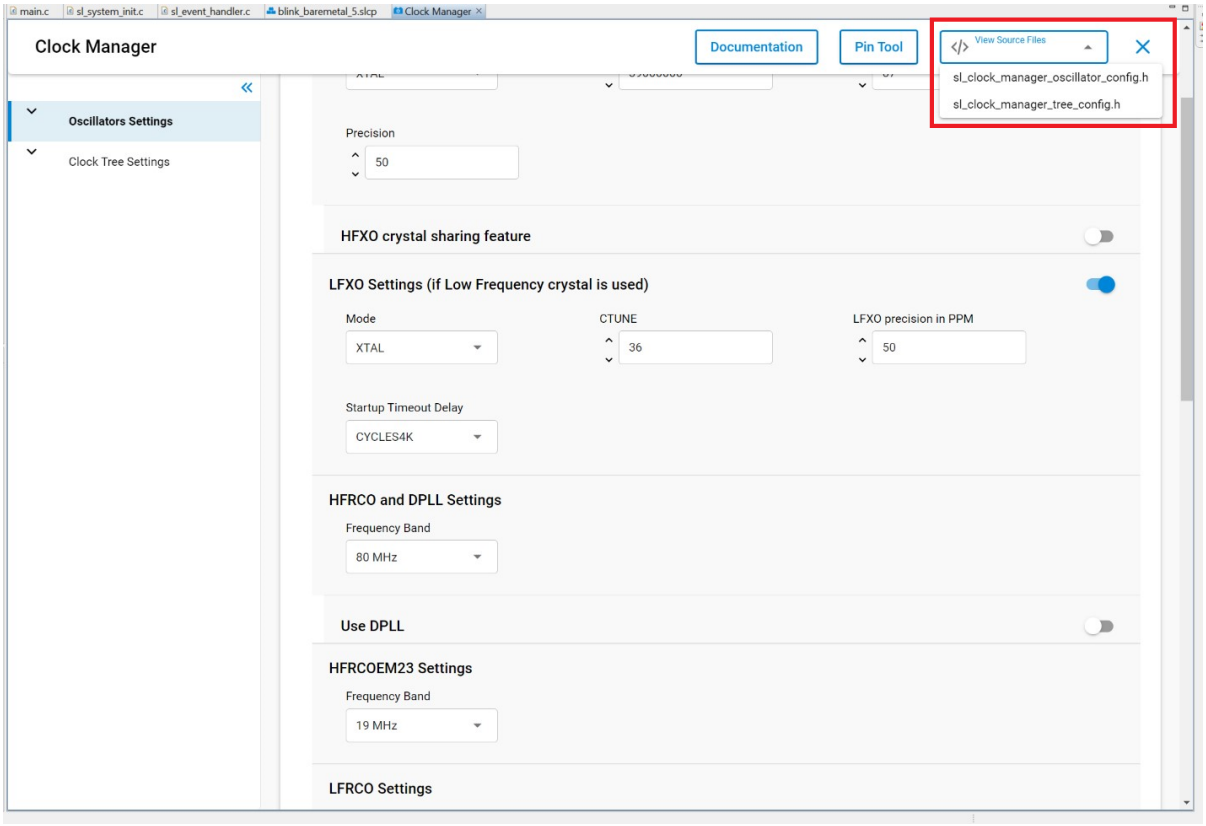
2. Both the oscillators and the clock tree settings are configurable in the *.slcp.



The clock sources and dividers are configurable in the clock tree settings.



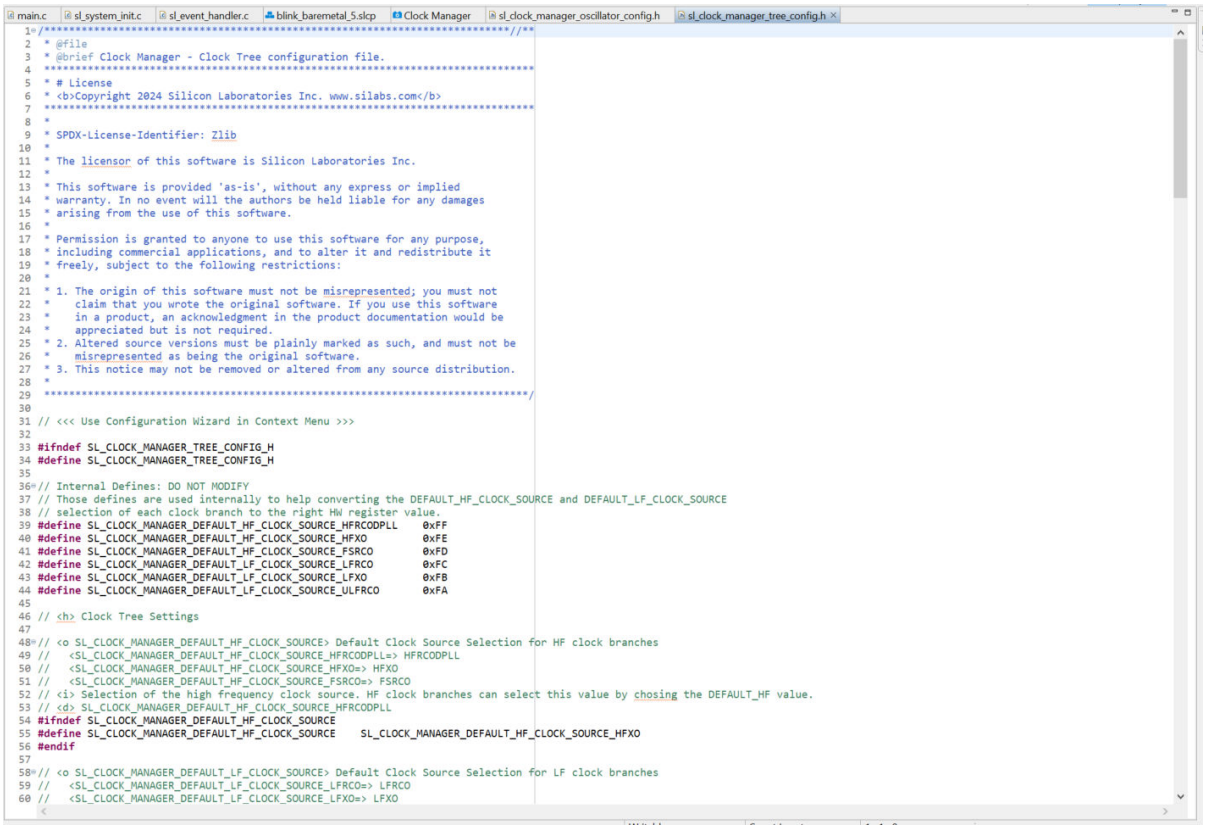
3. The CMSIS annotated configuration files can be accessed in the *.slcp by pressing View Source Files.



4. The configuration macros can be viewed and modified in the configuration files. Oscillator configuration macros can be found in sl_clock_manager_config.h.

```
/***************************************************************************//**
 * @file
 * @brief Clock Manager - Oscillators configuration file.
 *******************************************************************************
 * # License
 * <b>Copyright 2024 Silicon Laboratories Inc. www.silabs.com</b>
 *******************************************************************************
 *
 * SPDX-License-Identifier: Zlib
 *
 * The licensor of this software is Silicon Laboratories Inc.
 *
 * This software is provided 'as-is', without any express or implied
 * warranty. In no event will the authors be held liable for any damages
 * arising from the use of this software.
 *
 * Permission is granted to anyone to use this software for any purpose,
 * including commercial applications, and to alter it and redistribute it
 * freely, subject to the following restrictions:
 *
 * 1. The origin of this software must not be misrepresented; you must not
 *    claim that you wrote the original software. If you use this software
 *    in a product, an acknowledgment in the product documentation would be
 *    appreciated but is not required.
 * 2. Altered source versions must be plainly marked as such, and must not be
 *    misrepresented as being the original software.
 * 3. This notice may not be removed or altered from any source distribution.
 *
 ******************************************************************************/

// <<< Use Configuration Wizard in Context Menu >>>

#ifndef SL_CLOCK_MANAGER_OSCILLATOR_CONFIG_H
#define SL_CLOCK_MANAGER_OSCILLATOR_CONFIG_H

// <h> Oscillators Settings

// <e SL_CLOCK_MANAGER_HFXO_EN> HFXO Settings (if High Frequency crystal is used)
// <i> Enable to configure HFXO
#ifndef SL_CLOCK_MANAGER_HFXO_EN
#define SL_CLOCK_MANAGER_HFXO_EN    1
#endif

// <o SL_CLOCK_MANAGER_HFXO_MODE> Mode
// <i>
// <HFXO_CFG_MODE_XTAL=> XTAL
// <HFXO_CFG_MODE_EXTCLK=> EXTCLK
// <HFXO_CFG_MODE_EXTCLKPKDET=> EXTCLKPKDET
// <d> HFXO_CFG_MODE_XTAL
#ifndef SL_CLOCK_MANAGER_HFXO_MODE
#define SL_CLOCK_MANAGER_HFXO_MODE    HFXO_CFG_MODE_XTAL
#endif

// <o SL_CLOCK_MANAGER_HFXO_FREQ> Frequency <38000000-40000000>
// <d> 39000000
#ifndef SL_CLOCK_MANAGER_HFXO_FREQ
#define SL_CLOCK_MANAGER_HFXO_FREQ    39000000
#endif

// <o SL_CLOCK_MANAGER_HFXO_CTUNE> CTUNE <0-255>
```

Clock tree configuration macros can be found in sl_clock_manager_tree_config.h.



```
/***************************************************************************//**
 * @file
 * @brief Clock Manager - Clock Tree configuration file.
 *******************************************************************************
 * # License
 * <b>Copyright 2024 Silicon Laboratories Inc. www.silabs.com</b>
 *******************************************************************************
 *
 * SPDX-License-Identifier: Zlib
 *
 * The licensor of this software is Silicon Laboratories Inc.
 *
 * This software is provided 'as-is', without any express or implied
 * warranty. In no event will the authors be held liable for any damages
 * arising from the use of this software.
 *
 * Permission is granted to anyone to use this software for any purpose,
 * including commercial applications, and to alter it and redistribute it
 * freely, subject to the following restrictions:
 *
 * 1. The origin of this software must not be misrepresented; you must not
 *    claim that you wrote the original software. If you use this software
 *    in a product, an acknowledgment in the product documentation would be
 *    appreciated but is not required.
 * 2. Altered source versions must be plainly marked as such, and must not be
 *    misrepresented as being the original software.
 * 3. This notice may not be removed or altered from any source distribution.
 *
 ******************************************************************************/

// <<< Use Configuration Wizard in Context Menu >>>

#ifndef SL_CLOCK_MANAGER_TREE_CONFIG_H
#define SL_CLOCK_MANAGER_TREE_CONFIG_H

// Internal Defines: DO NOT MODIFY
// Those defines are used internally to help converting the DEFAULT_HF_CLOCK_SOURCE and DEFAULT_LF_CLOCK_SOURCE
// selection of each clock branch to the right HW register value.
#define SL_CLOCK_MANAGER_DEFAULT_HF_CLOCK_SOURCE_HFRCODPLL    0xFF
#define SL_CLOCK_MANAGER_DEFAULT_HF_CLOCK_SOURCE_HFXO         0xFE
#define SL_CLOCK_MANAGER_DEFAULT_HF_CLOCK_SOURCE_FSRCO        0xFD
#define SL_CLOCK_MANAGER_DEFAULT_LF_CLOCK_SOURCE_LFRCO        0xFC
#define SL_CLOCK_MANAGER_DEFAULT_LF_CLOCK_SOURCE_LFXO         0xFB
#define SL_CLOCK_MANAGER_DEFAULT_LF_CLOCK_SOURCE_ULFRCO       0xFA

// <h> Clock Tree Settings

// <o SL_CLOCK_MANAGER_DEFAULT_HF_CLOCK_SOURCE> Default Clock Source Selection for HF clock branches
//   <SL_CLOCK_MANAGER_DEFAULT_HF_CLOCK_SOURCE_HFRCODPLL=> HFRCODPLL
//   <SL_CLOCK_MANAGER_DEFAULT_HF_CLOCK_SOURCE_HFXO=> HFXO
//   <SL_CLOCK_MANAGER_DEFAULT_HF_CLOCK_SOURCE_FSRCO=> FSRCO
// <i> Selection of the high frequency clock source. HF clock branches can select this value by chosing the DEFAULT_HF value.
// <d> SL_CLOCK_MANAGER_DEFAULT_HF_CLOCK_SOURCE_HFRCODPLL
#ifndef SL_CLOCK_MANAGER_DEFAULT_HF_CLOCK_SOURCE
#define SL_CLOCK_MANAGER_DEFAULT_HF_CLOCK_SOURCE    SL_CLOCK_MANAGER_DEFAULT_HF_CLOCK_SOURCE_HFXO
#endif

// <o SL_CLOCK_MANAGER_DEFAULT_LF_CLOCK_SOURCE> Default Clock Source Selection for LF clock branches
//   <SL_CLOCK_MANAGER_DEFAULT_LF_CLOCK_SOURCE_LFRCO=> LFRCO
//   <SL_CLOCK_MANAGER_DEFAULT_LF_CLOCK_SOURCE_LFXO=> LFXO
```

7. Revision History

Revision 0.1

August, 2024

- Initial revision.

Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

www.silabs.com