AN14792

S32K31x to MCX E31x Migration Guidelines

Rev. 1.1 — 7 November 2025 Application note

Document information

Information	Content
Keywords	AN14792, S32K31x, MCX E31x
Abstract	This application note outlines the key differences between the S32K31 and MCX E31 microcontroller families, with a focus on software-related aspects. It provides essential guidance and considerations for migrating software applications from the S32K31 platform to the MCX E31 family.

1 Introduction

This application note outlines the key differences between the S32K31 and MCX E31 microcontroller families, with a focus on software-related aspects. It provides essential guidance and considerations for migrating software applications from the S32K31 platform to the MCX E31 family.

NXP proposes 2 portfolios of microcontrollers with similar hardware configuration: S32K31 automotive microcontrollers focus on automotive applications requiring Automotive-grade components and certification. MCX E31 focuses on industrial applications with industrial mission profiles and certification.

While S32K31 also serves industrial applications, MCX E31 provides a complete and more coherent environment with the rest of MCX portfolio for industrial applications.

2 Hardware migration

The migration from the S32K31x to the MCX E31x device is fully pin-to-pin compatible, ensuring a seamless transition from a hardware perspective.

The key hardware migration highlights are:

- **Pinout compatibility:** The MCX E31x shares the package and pin configuration as the S32K31 family, enabling direct replacement without PCB layout changes.
- **No schematic modifications:** Existing schematics support reuse without alteration. All required power domains, clock sources, and I/O configurations are preserved.
- **Identical package handling conditions:** Mechanical and thermal characteristics, including moisture sensitivity level (MSL), remain unchanged.

The following table provides a pin-to-pin compatibility mapping between the S32K31x device variants and their corresponding MCX E31x counterparts.

Table 1. Pin-to-pin compatibility between S32K31x and MCX E31x devices

MCX P/N	S32K mapping
MCXE315MLF	S32K310NHT0MLFST
	S32K310NHT0VLFST
	S32K310NHT0VLFSR
	S32K310NHT0MLFSR
MCXE315MPA	S32K310NHT0MPAST
	S32K310NHT0MPAIT
	S32K310NHT0VPAST
	S32K310NHT0VPASR
	S32K310NHT0MPASR
	S32K310NHT0MPAIR
MCXE316MLF	S32K311NHT0MLFST
	S32K311NHT0VLFST
	S32K311NHT0VLFSR
	S32K311NHT0MLFSR
MCXE316MPA	S32K311NHT0MPAST
	S32K311NHT0MPAIT

Table 1. Pin-to-pin compatibility between S32K31x and MCX E31x devices*...continued*

MCX P/N	S32K mapping
	S32K311NHT0VPAST
	S32K311NHT0VPASR
	S32K311NHT0MPASR
	S32K311NHT0MPAIR
MCXE317MPA	S32K312NHT0MPAST
	S32K312NHT0MPAIT
	S32K312NHT0VPAST
	S32K312NHT0VPASR
	S32K312NHT0MPASR
	S32K312NHT0MPAIR
MCXE317MPB	S32K312NHT0MPBST
	S32K312NHT0VPBST
	S32K312NHT0VPBSR
	S32K312NHT0MPBSR
MCXE31BMPB	S32K314EHT1MPBIT
	S32K314EHT1VPBST
	S32K314EHT1MPBST
	S32K314EHT1VPBSR
	S32K314EHT1MPBSR
	S32K314NHT1VPBST
	S32K314NHT1VPBSR
	S32K314NHT1MPBST
	S32K314NHT1MPBSR
	S32K314EHT1MPBIR

3 Software ecosystem differences

While both microcontroller families share similar hardware capabilities in terms of CPU core architecture and peripheral sets, their software ecosystems differ significantly. The S32K31 family relies on an AUTOSAR-compliant Real-Time Drivers (RTD) framework, designed for automotive safety and modularity. The MCUXpresso SDK supports the MCX E31 family, targeting industrial applications and offering a more flexible and lightweight driver model.

4 Software architecture comparison

The Automotive Open System Architecture (AUTOSAR) RTD framework is based on a layered architecture aligned with the AUTOSAR standard. It provides strict separation between hardware abstraction, operating system services, and application logic. Drivers are structured in multiple layers—Microcontroller Abstraction Layer (MCAL), complex drivers, and service layer and integrate tightly with configuration tools and XML-based code generation. This architecture ensures modularity, standard compliance (for example, ISO 26262), and support for scalable, safety-critical automotive applications.

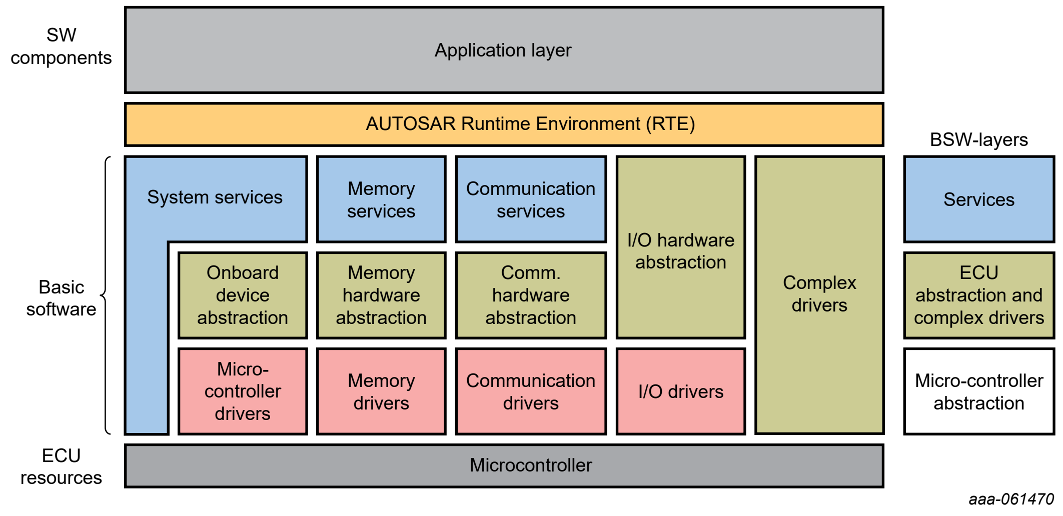

Figure 1. AUTOSAR architecture

In contrast, the MCUXpresso Software Development Kit (SDK) follows a more lightweight, monolithic approach. It provides peripheral drivers as standalone C modules, without a separation between configuration and implementation layers. This has several practical implications:

- **Improved real-time performance:** Direct access to peripheral drivers minimizes latency, which is critical for time-sensitive applications (for example, motor control, PTP timestamping).
- **Lower memory footprint:** The absence of layered abstractions reduces RAM/Flash usage.

Application development is more direct, with code examples, API reference manuals, and optional middleware components (such as FreeRTOS, USB, or lwIP). While less rigid, this architecture is more flexible and faster to adapt for industrial and general-purpose embedded applications where safety certification or standardized software layering aren't mandatory.

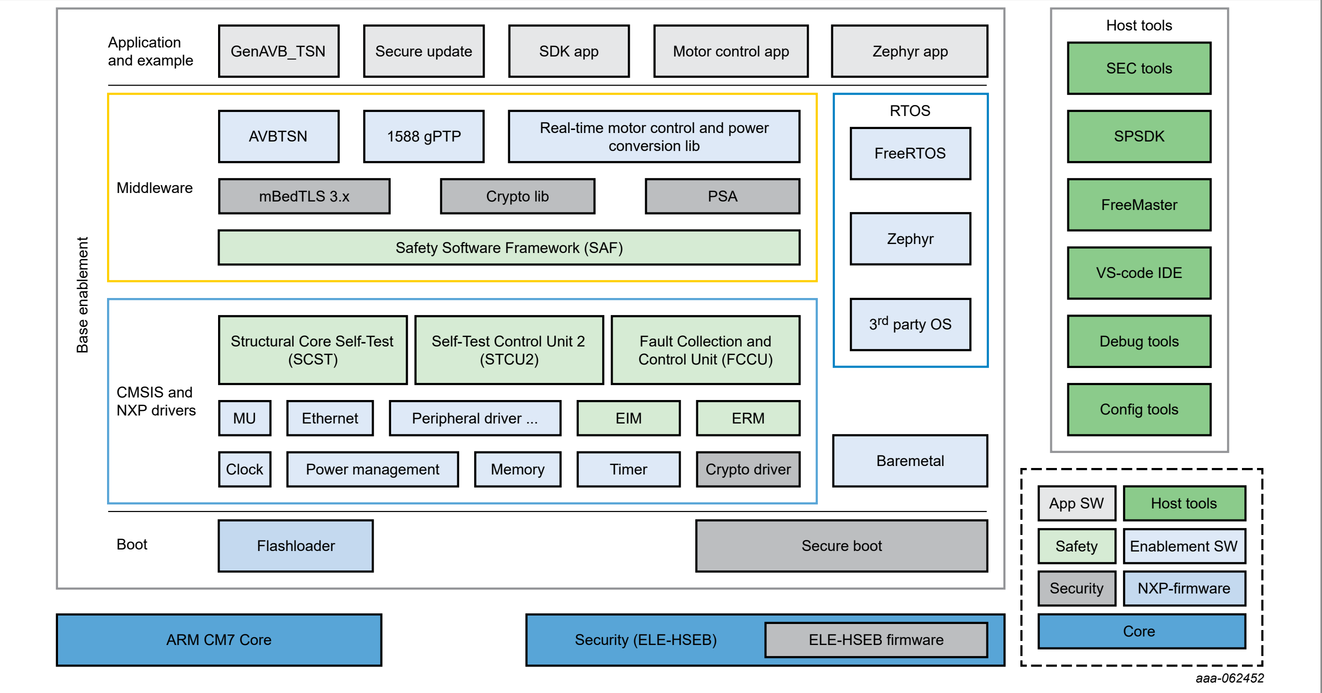

Figure 2. MCUXpresso architecture

Table 2. SDKs standard

Aspect	AUTOSAR RTD	MCUXpresso SDK
Architecture	Three layers (SWC, basic software (BSW), RTE)	Three layers (App, middleware, drivers)
Code generation	Arxml + tools (Tresos …)	Manual coding/config tools
Goal	ISO 26262, automotive, and safety	Flexibility, general onboard, industrial
Hardware abstraction	High (MCAL, ECU Abstraction)	Low to moderate (faster execution, better real-time performance)
Real-time operating system (RTOS) integration	Integrated operating system (OSEK, AUTOSAR OK)	Optional (FreeRTOS, Zephyr)
Documentation	Standard, sometimes heavy	Code + Doxygen + examples
Rigidity/flexibility	Rigid but structured	Flexible but less standard

5 Security

This section describes the detailed matrix between RTD AUTOSAR and MCUXpresso SDK across various aspects.

Table 3. Security features comparison

Aspect	RTD AUTOSAR	MCUXpresso SDK
Security module	CSM, CryptoIf, secure flash manager	Crypto support possible
Key management	Integrated with key management (KMU, HSE-B)	Managed externally through SPSDK, SEC tools, and optional EdgeLock 2GO integration

Table 3. Security features comparison*...continued*

Aspect	RTD AUTOSAR	MCUXpresso SDK
Authentication and encryption	Standardized APIs (CryptoIf + CryIf backends)	To be implemented through mbedTLS or NXP SDK (HSE-B, PSA crypto)
Security diagnosis	Diagnostic event manager (DEM)	Not present, to be implemented manually
Secure boot/firmware update	Support is possible through a secure bootloader	Secure bootloader must be implemented with boot ROM or flashloader
Secure manufacturing	Developer-unfriendly requires secured production environment	Support with pre-flashed flashloader (see Section 5.1)

Note: *HSE-B related content will be included in a future version of this migration guide.*

5.1 Secure installation and manufacturing support

One key enhancement from the MCUXpresso SDK ecosystem for the MCX series, including MCX E31x, is out-of-the-box support for secure manufacturing processes. The SDK provides a pre-flashed secure flashloader on the device, enabling firmware provisioning and updates over the UART interface. This allows OEMs to implement secure factory programming using a standard communication port without relying on a debug interface.

The S32K31x platform delivers the HSE-B firmware as an encrypted binary that flashes alongside the application image. A built-in secure bootloader automatically installs the HSE-B firmware using keys provisioned during NXP production, so no separate secured debug-port provisioning environment is required. The MCX E31x workflow is simpler because its secure bootloader is pre-installed and ready to accept encrypted firmware over a standard UART interface. No special image packaging, or key handling is required on the factory floor. In contrast, the S32K31x process, while also secure, mandates combining the encrypted HSE-B firmware with the application image ahead of time. Both must be flashed together, adding an extra preparation step before production programming.

6 AUTOSAR RTD API > MCUXpresso API

This section outlines the API-level mapping between AUTOSAR RTD and MCUXpresso SDK for easier migration and comparison.

6.1 Analog-to-Digital Converter (ADC)

This section covers the migration of the ADC IP from AUTOSAR RTD to the MCUXpresso SDK. The ADC module enables precise analog signal conversion critical for sensor data acquisition and control systems. While both platforms provide ADC functionalities, their APIs, configuration approaches, and driver implementations differ. This chapter outlines key differences, maps equivalent functions, and offers best practices to ensure a seamless migration that preserves accuracy and performance.

Example 1: Typical Reading RTD

```
Adc_StartGroupConversion(ADC_GROUP_0);
while (Adc_GetGroupStatus(ADC_GROUP_0) != ADC_STREAM_COMPLETED) {}
Adc_ReadGroup(ADC_GROUP_0, &dataBuffer);
```

Example 2: Typical Reading MCUXpresso

```
Adc_conv_result_t result;
adc_chain_config_t chainConfig;
adc_calibration_config_t calibrationConfig;
ADC_SetConvChainConfig(DEMO_ADC_BASE, &chainConfig);
ADC_DoCalibration(DEMO_ADC_BASE, &calibrationConfig);  // Optional
ADC_StartConvChain(DEMO_ADC_BASE, kADC_NormalConvScanMode);
ADC_GetChannelConvResult(DEMO_ADC_BASE, &result, DEMO_ADC_CHANNEL0);
```

Table 4 summarizes the comparison of ADC APIs.

Table 4. ADC APIs comparison

AUTOSAR	MCUXpresso
Adc_GroupType	channelNumber in adc_channel_config_t
Adc_Init (Adc_ConfigType*)	ADC_Init (ADC_Type*, adc_config_t*)
Adc_ReadGroup (Adc_GroupType, Adc_ValueGroup Type*)	ADC_GetChannelConvResult (ADC_Type*, adc_conv_result_t*, channelIdx)
Adc_StartGroupConversion (Adc_GroupType)	ADC_SetConvChainConfig (ADC_Type*, adc_chain_config_t*) + ADC_StartConvChain (ADC_Type*, adc_conv_mode_t)

6.2 Comparator (CMP)

This section addresses the CMP IP migration from AUTOSAR RTD to the MCUXpresso SDK, where the comparator functionality is provided by low power comparator (LPCMP) module. The LPCMP is a lightweight, low-power analog comparator designed for signal monitoring and threshold detection in embedded systems. Although both platforms support comparator features, their APIs, configuration models, and integration methods differ. This chapter highlights these differences, maps corresponding functions, and provides guidance to ensure a smooth migration while maintaining signal integrity and system responsiveness.

Example 3: Init RTD

```
#define STD_ON CMP_IP_USED
Cmp_Ip_Init(instance, Cmp_Ip_ConfigType*) ;
Cmp_Ip_EnableInterrupt(instance);
```

Example 4: Init MCUXpresso

```
lpcmp_config_t mLpcmpConfigStruct;
lpcmp_dac_config_t mLpcmpDacConfigStruct;
LPCMP_GetDefaultConfig(&mLpcmpConfigStruct);
LPCMP_Init(DEMO_LPCMP_BASE, &mLpcmpConfigStruct);
LPCMP_SetDACConfig(DEMO_LPCMP_BASE, &mLpcmpDacConfigStruct);
LPCMP_SetInputChannels(DEMO_LPCMP_BASE, DEMO_LPCMP_USER_CHANNEL,
 DEMO_LPCMP_DAC_CHANNEL);
EnableIRQ(DEMO_LPCMP_IRQ_ID);
LPCMP_EnableInterrupts(DEMO_LPCMP_BASE, kLPCMP_OutputRisingInterruptEnable |
 kLPCMP_OutputFallingInterruptEnable);
```

Table 5 summarizes the comparison of CMP APIs.

Table 5. CMP APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Cmp_Ip_Init (instance, Cmp_Ip_ConfigType*)	LPCMP_Init (LPCMP_Type*, lpcmp_config_t*)
Config	Cmp_Ip_ConfigType structure (generated by arxml)	LPCMP_SetInputChannels (LPCMP_Type*, positive Channel, negativeChannel)
Irq	Cmp_Ip_EnableInterrupt (instance)	LPCMP_EnableInterrupts (LPCMP_Type*, mask)

6.3 Low Power Inter-Integrated Circuit (LPI2C)

This section covers the migration of the LPI2C IP from AUTOSAR RTD to the MCUXpresso SDK. The LPI2C module supports low-power I2C communication tailored for efficient embedded systems. While AUTOSAR provides a standardized asynchronous API with built-in callbacks. The MCUXpresso SDK offers greater flexibility by allowing users to select polling, interrupt, or DMA transfer methods. Additionally, error handling such as timeouts and NACKs must be managed explicitly in the MCUXpresso environment.

Example 5: Send data (polling) RTD

```
I2c_RequestType Req[2] = {
{0x32, FALSE, FALSE, FALSE, 8U, I2C_SEND_DATE, txBuffer},
{0x32, FALSE, FALSE, FALSE, 8U, I2C_RECEIVE_DATE, rxBuffer},
} // txBuffer -> Data to be transmitted, rxBuffer-> Buffer to receive Data
I2c_AsyncTransmit(0U, &Req[0]);
```

Example 6: Send data (polling) MCUXpresso

```
uint8_t data[] = { 0x01, 0x02 };
LPI2C_MasterStart(LPI2C0, slave_addr_7bits, kLPI2C_Write);
LPI2C_MasterSend(LPI2C0, &deviceAddress, 1);
LPI2C_MasterSend(LPI2C0, data, sizeof(data));
LPI2C_MasterStop(LPI2C0);
```

Table 6 summarizes the comparison of LPI2C APIs.

Table 6. LPI2C APIs comparison

Functions	AUTOSAR	MCUXpresso
Controller initialization	Lpi2c_Ip_MasterInit (instance, Lpi2c_Ip_MasterConfigType*) through I2c_Init (I2c_ConfigType*)	LPI2C_MasterInit (LPI2C_Type*, lpi2c_master_config_t*, srcClk Hz) + LPI2C_MasterGetDefaultConfig (lpi2c_master_config_t*) LPI2C_MasterStart (LPI2C_Type*, address, lpi2c_direction_t)
Synchronous writing	I2c_SyncTransmit (Channel, I2c_RequestType*)	LPI2C_MasterTransferBlocking (LPI2C_Type*, lpi2c_master_transfer_t*)
Asynchronous writing	I2c_AsyncTransmit (Channel, I2c_RequestType*)	LPI2C_MasterTransferNonBlocking (LPI2C_Type*, lpi2c_master_handle_t*, lpi2c_master_transfer_t*)
Target initialization	Lpi2c_Ip_SlaveInit (Instance, Lpi2c_Ip_SlaveConfigType*) through I2c_Init (I2c_ConfigType*)	LPI2C_SlaveInit (LPI2C_Type*, lpi2c_slave_config_t*, srcClk Hz) + LPI2C_SlaveGetDefaultConfig (lpi2c_slave_config_t*)
Controller/ target status	I2c_Ipw_GetStatus (Channel, I2c_Hw UnitConfigType*)	LPI2C_SlaveGetStatusFlags (LPI2C_Type*) LPI2C_MasterGetStatusFlags (LPI2C_Type*)

6.4 Low Power Universal Asynchronous Receiver/Transmitter (LPUART)

This section covers the migration of the LPUART IP from AUTOSAR RTD to the MCUXpresso SDK. The LPUART module provides low-power UART communication optimized for energy-efficient embedded applications. While both platforms support LPUART functionality, there are differences in APIs, configuration methods, and driver behavior. This chapter highlights these differences, offers a function mapping, and provides recommendations to facilitate a seamless migration without compromising communication reliability.

Example 7: Synchronous transmission RTD

```
Lpuart_Uart_Ip_StatusType status;
const uint8 txData[] = "Hello UART\n";
status = Lpuart_Uart_Ip_SyncSend(UART_INSTANCE, txData, sizeof(txData));
if (status != LPUART_UART_IP_STATUS_SUCCESS) {
    // Manage error
}
// Reception with IRQ:
void LPUART0_RxTx_IRQHandler(void)
{
    Lpuart_Uart_Ip_IrqHandler(UART_INSTANCE); // Call low level handler
    …
}
uint8 rxBuffer;
Lpuart_Uart_Ip_AsyncReceive(LPUART_INSTANCE, &rxBuffer, length);
```

Example 8: Synchronous transmission MCUXpresso

```
const char *msg = "Hello UART\n";
LPUART_WriteBlocking(LPUART0, (const uint8_t *)msg, strlen(msg));
 // Reception with IRQ:
LPUART_TransferCreateHandle(LPUART0, &g_lpuartHandle, LPUART_UserCallback,
 NULL);
LPUART_TransferReceiveNonBlocking(LPUART0, &g_lpuartHandle, &receiveXfer, NULL)
```

Table 7 summarizes the comparison of LPUART APIs.

Table 7. LPUART APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Lpuart_Uart_Ip_Init (instance, Lpuart_Uart_Ip_UserConfigType*)	LPUART_Init (LPUART_Type*, lpuart_config_t*, srcClkHz) + LPUART_GetDefaultConfig (lpuart_config_t*)
Synchronous writing	Lpuart_Uart_Ip_SyncSend (instance, uint8_t*, uint32_t, uint32_t)	LPUART_WriteBlocking (LPUART_Type*, uint8_t*, size_t)
Synchronous reading	Lpuart_Uart_Ip_SyncReceive (instance, uint8_t*, uint32_t, uint32_t)	LPUART_ReadBlocking (LPUART_Type*, uint8_t*, size_t)
Asynchronous writing	Lpuart_Uart_Ip_AsyncSend (instance, uint8_t*, uint32_t)	LPUART_TransferSendNonBlocking (LPUART_Type*, lpuart_handle_t*, lpuart_transfer_t*)
Asynchronous reading	Lpuart_Uart_Ip_AsyncReceive (instance, uint8_t*, uint32_t)	LPUART_TransferReceiveNonBlocking (LPUART_Type*, lpuart_handle_t*, lpuart_transfer_t*, size_t*)

6.5 Low Power Serial Peripheral Interface (LPSPI)

This section addresses the migration of the LPSPI IP from AUTOSAR RTD to the MCUXpresso SDK. The LPSPI module delivers low-power SPI communication suitable for modern embedded systems requiring energy efficiency. Despite similar core capabilities, API structures and configuration details differ between the two platforms. This chapter presents key changes, maps API functions, and suggests best practices to ensure a smooth and effective migration, preserving SPI communication performance.

Example 9: Full duplex asynchronous transmission RTD

```
Lpspi_Ip_UpdateTransferMode(LPSPI0, LPSPI_IP_INTERRUPT);
Lpspi_Ip_AsyncTransmit(LPSPI0, TxMasterBuffer, RxSlaveBuffer, NUMBER_OF_BYTES,
 callback);
```

Example 10: Full duplex asynchronous transmission MCUXpresso

```
lpspi_transfer_t transfer;
LPSPI_MasterTransferCreateHandle(LPSPI0_MASTER_BASE, &g_lpspiHandle,
 LPSPI_UserCallback, NULL);
LPSPI_MasterTransferNonBlocking(LPSPI0_MASTER_BASE, &g_lpspiHandle, &transfer);
```

Table 8 summarizes the comparison of LPSPI APIs.

Table 8. LPSPI APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Lpspi_Ip_Init (Lpspi_Ip_ConfigType*)	LPSPI_MasterInit (LPSPI_Type*, lpspi_master_config_t*, srcClkHz) + LPSPI_MasterGetDefaultConfig (lpspi_master_config_t*)/LPSPI_SlaveInit (LPSPI_Type*, lpspi_slave_config_t*) + LPSPI_SlaveGetDefaultConfig (lpspi_slave_config_t*)
Synchronous writing	Lpspi_Ip_SyncTransmit (Lpspi_Ip_ExternalDeviceType*, uint8_t*, uint8_t*, uint16_t, uint32_t) -> Full duplex Lpspi_Ip_SyncTransmitHalfDuplex (Lpspi_Ip_ExternalDeviceType*, uint8_t*, uint16_t, Lpspi_Ip_HalfDuplexType, uint32_t) -> half-duplex	LPSPI_MasterTransferBlocking (LPSPI_Type*, lpspi_transfer_t*)
Asynchronous writing	Lpspi_Ip_AsyncTransmit (Lpspi_Ip_ExternalDeviceType*, uint8_t*, uint8_t*, uint16_t, Lpspi_Ip_CallbackType) -> Full duplex Lpspi_Ip_AsyncTransmitHalfDuplex (Lpspi_Ip_ExternalDeviceType*, uint8_t*, uint16_t, Lpspi_Ip_HalfDuplexType, Lpspi_Ip_CallbackType) -> half-duplex	LPSPI_MasterTransferNonBlocking (LPSPI_Type*, lpspi_master_handle_t*, lpspi_transfer_t*) + LPSPI_MasterTransferCreateHandle (LPSPI_Type*, lpspi_master_handle_t*, lpspi_master_transfer_callback_t, userData)
Transfer mode configuration	Lpspi_Ip_UpdateTransferMode (instance, Lpspi_Ip_ModeType)	No equivalent function mode is linked to the transmit function used

6.6 System Integration Unit Lite 2 (SIUL2)

In AUTOSAR RTD, GPIO functionality is split between the port module (for pin configuration) and the dio module (for reading/writing pin states). However, in some RTD setups like on S32K3, IP-level drivers such as Siul2_Dio_Ip_WritePin() can replace Dio and directly manage pin states without going through the full AUTOSAR abstraction.

Similarly, the MCUXpresso SDK does not provide a generic fsl_gpio driver, as the MCU relies on the SIUL2 peripheral for all GPIO operations. The SDK offers a direct API (fsl_siul2) to configure pin and perform I/O through functions like SIUL2_PinInit(), SIUL2_PortPinWrite(), and SIUL2_PortPinRead(). This low-level approach provides tight control over hardware and simplifies the software stack, but comes with less architectural abstraction compared to AUTOSAR-based designs.

Table 9. GPIO APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Port_Ci_Port_Ip_Init (uint32_t, Port_Ci_Port_Ip_PinSettingsConfig)	SIUL2_PinInit (siul2_pin_settings_t*)
Write	Siul2_Dio_Ip_WritePin (Siul2_Dio_Ip_GpioType*, Siul2_Dio_Ip_PinsChannelType, Siul2_Dio_Ip_PinsLevelType)	SIUL2_PortPinWrite (SIUL2_Type*, siul2_port_num_t, uint32_t, uint8_t)
Read	Siul2_Dio_Ip_ReadPin (Siul2_Dio_Ip_GpioType *, Siul2_Dio_Ip_PinsChannelType)	SIUL2_PortPinRead (SIUL2_Type*, siul2_port_num_t, uint8_t)

6.7 Enhanced Modular Input Output System (eMIOS)

The eMIOS is a flexible and powerful timing peripheral available in many Automotive-grade NXP microcontrollers. It supports multiple timing and signal generation functions such as pulse width modulation (PWM), input capture, output compare, and signal measurement. This functionality makes it suitable for complex automotive control applications including motor control, sensor interfacing, and actuator triggering.

In the AUTOSAR RTD environment, eMIOS functionality is accessed through abstracted software layers like PWM, Input Capture Unit (ICU), and Output Compare Unit (OCU). These layers rely on IP-level drivers such as Emios_<module>_Ip, and configuration is typically handled through tools like EB tresos, in compliance with the AUTOSAR layered architecture. This approach emphasizes functional safety, portability, and standardization across projects.

MCUXpresso SDK includes a fsl_emios driver, which provides direct access to the eMIOS peripheral. This driver offers a C-based API for configuring and using the eMIOS channels in various modes, such as PWM generation, input capture, and output compare. It allows developers to manage the peripheral flexibly without relying on AUTOSAR abstractions or external configuration tools.

This chapter highlights the migration of typical eMIOS use cases from AUTOSAR RTD to the MCUXpresso SDK, comparing configuration flows, driver usage, and functional consideration.

Table 10. eMIOS APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Emios_Mcl_Ip_Init (instance, Emios_Mcl_Ip_ConfigType*)	EMIOS_Init (EMIOS_Type*, emios_config_t*) + EMIOS_GetDefaultConfig (emios_config_t*)
PWM initialization	Emios_Pwm_Ip_InitChannel (instance, Emios_Pwm_Ip_ChannelConfigType*)	EMIOS_ConfigPWM (EMIOS_Type*, emios_uc_pwm_config_t*, uint8_t)
PWM period	Emios_Pwm_Ip_SetPeriod (instance, channel, Emios_Pwm_Ip_PeriodType)	No equivalent function
PWM channel	Emios_Pwm_Ip_SetDutyCycle (instance, channel, Emios_Pwm_Ip_DutyType)	EMIOS_UpdatePWM (EMIOS_Type*, emios_uc_pwm_config_t*, channel)
OCU initialization	Emios_Ocu_Ip_Init (Emios_Ocu_Ip_ConfigType*)	EMIOS_ConfigOutputCompare (EMIOS_Type*, emios_uc_oc_config_t*, channel)
OCU start	Emios_Ocu_Ip_StartChannel (instance, ChNum)	No equivalent function, it is done through EMIOS_ConfigOutputCompare()
OCU Irq	Emios_Ocu_Ip_EnableNotification (instance, ChNum)	EMIOS_EnableUCInterrupt (EMIOS_Type*, channel)
GPT functions	Emios_Gpt_Ip_InitChannel (instance, Emios_Gpt_Ip_ChannelConfigType*) Emios_Gpt_Ip_EnableChannelInterrupt (instance, channel)	No GPT functions

Table 10. eMIOS APIs comparison*...continued*

Functions	AUTOSAR	MCUXpresso
	Emios_Gpt_Ip_StartTimer (instance) Emios_Gpt_Ip_StartTimer (instance, channel, compareVal)	
ICU initialization	Emios_Icu_Ip_Init (instance, eMios_Icu_Ip_Config Type*)	EMIOS_ConfigInputCapture (EMIOS_Type*, emios_uc_ic_config_t*, channel)
ICU Irq	Emios_Icu_Ip_EnableNotification (instance, hw Channel)	EMIOS_EnableUCInterrupt (EMIOS_Type*, channel)

6.8 Local Interconnect Network (LIN)

Only basic LIN support is available through the LPUART peripheral in the MCUXpresso SDK, such as break detection and generation. A full LIN protocol stack is not provided.

Example 11: LPUART LIN Frame Transfer RTD

```
Lin_43_LPUART_FLEXIO_Init(NULL_PTR);
Lin_43_LPUART_FLEXIO_WakeupInternal(Channel_Index);
Lin_43_LPUART_FLEXIO_SendFrame(Channel_Index, &linFrame);
T_LinStatus = Lin_43_LPUART_FLEXIO_GetStatus(Channel_Index, &linSduPtr);
```

Example 12: LPUART LIN Frame Transfer MCUXpresso

```
lin_user_config_t linUserConfigMaster;
LIN_GetMasterDefaultConfig(&linUserConfigMaster);
LIN_Init(MASTER_INSTANCE, linUserConfig, linCurrentState, clockSource);
LIN_InstallCallback(MASTER_INSTANCE, CallbackHandler);
LIN_SetResponse(MASTER_INSTANCE, rxBuff, rxSize, timeout);
```

6.9 FLEXIO

The FLEXIO module is a highly configurable peripheral that enables flexible and efficient implementation of various serial communication protocols, such as Inter-Integrated Circuit (I2C), Serial Peripheral Interface (SPI), and Universal Asynchronous Receiver/Transmitter (UART). By using programmable shifters, timers, and pins, FLEXIO allows the emulation of these interfaces. It is useful when dedicated hardware modules are unavailable or when extra instances are required.

This flexibility makes FLEXIO a versatile solution for expanding communication capabilities on MCUs where pin or peripheral resources are limited.

The following subsections detail the API migration and usage for each supported protocol implemented through FLEXIO:

- I2C - Emulated I2C communication using FLEXIO
- SPI - SPI protocol implemented through FLEXIO
- UART - UART communication emulated by FLEXIO

6.9.1 I2C

The FLEXIO I2C module emulates the standard I2C protocol using the flexible hardware resources of the FLEXIO peripheral. It enables software-configurable I2C communication without requiring a dedicated I2C hardware block, offering flexibility in pin assignment and bus configuration.

FLEXIO peripheral supports only I2C controller mode in both the MCUXpresso SDK and the AUTOSAR RTD.

Example 13: FLEXIO I2C controller read RTD

```
#define STD_ON I2C_FLEXIO_USED
I2c_RequestType pRequest[1] = {0x32, FALSE, FALSE, FALSE, 8U, I2C_RECEIVE_DATA,
 rxBufferMaster};
Flexio_I2c_Ip_MasterInit(FLEXIO_INSTANCE, channel, &flexioMasterConfig);
Flexio_I2c_Ip_SetMasterCallback(FLEXIO_INSTANCE, channel,
 &flexioMasterCallback);
I2c_AsyncTransmit(channel, &pRequest[0]);
```

Example 14: FLEXIO I2C controller read MCUXpresso

```
flexio_i2c_master_config_t masterConfig;
FLEXIO_I2C_MasterGetDefaultConfig(&masterConfig);
FLEXIO_I2C_MasterInit(FLEXIO_I2C_Type*, &masterConfig, FLEXIO_CLOCK_FREQUENCY);
FLEXIO_I2C_MasterTransferCreateHandle(FLEXIO_I2C_Type*, &g_m_handle,
 flexio_i2c_master_callback, NULL);
masterXfer.direction = kFLEXIO_I2C_Read;
   …
FLEXIO_I2C_MasterTransferNonBlocking(FLEXIO_I2C_Type*, &g_m_handle,
 &master_Xfer);
```

6.9.2 SPI

The FLEXIO SPI module provides SPI communication functionality through programmable shifters and timers within the FLEXIO peripheral. It allows for flexible SPI implementations on devices lacking multiple dedicated SPI modules or requiring additional SPI instances.

Example 15: FLEXIO SPI controller read RTD

```
#define SPI_IPW_SPI_FLEXIO_ENABLE STD_ON
Flexio_Spi_Ip_Init(MASTER_PHY);
Flexio_Spi_Ip_SyncTransmit(MASTER_DEV, TxSlaveBuffer, RxSlaveBuffer,
 NUMBER_OF_BYTES, TIMEOUT); // Could also be an async transfer
… Reading of RxSlaveBuffer …
```

Example 16: FLEXIO SPI controller read MCUXpresso

```
flexio_spi_transfer_t xfer = {
    .txData = txBuf,
    .rxData = rxBuf,
    .dataSize = N,
    .flags = kFLEXIO_SPI_csActiveLow,
};
FLEXIO_SPI_MasterInit(&spiDev, &userConfig, FLEXIO_CLK_FREQ);
FLEXIO_SPI_MasterTransferBlocking(FLEXIO_SPI_BASEADDR, &xfer);
… Reading of xfer.rxData field …
```

6.9.3 UART

The FLEXIO UART module enables UART serial communication using the programmable features of FLEXIO. It supports customizable baud rates, data framing, and pin configurations, making it suitable for applications needing UART interfaces without dedicated hardware UART modules.

Example 17: FLEXIO UART controller read RTD

```
uint8_t txData[TX_BUFFER_SIZE] = "HELLO\n";
uint8_t rxData[TX_BUFFER_SIZE] = {0};
Flexio_Uart_Ip_Init(INSTANCE, &config);
Flexio_Uart_Ip_SyncSend(INSTANCE, txData, TX_BUFFER_SIZE, 1000u);
Flexio_Uart_Ip_SyncReceive(INSTANCE, rxData, TX_BUFFER_SIZE, 1000u);
```

Example 18: FLEXIO UART controller read MCUXpresso

```
uint8_t txData[BUFFER_SIZE] = "HELLO\n";
uint8_t rxData[BUFFER_SIZE] = {0};
FLEXIO_UART_Init(&uartDev, &config, FLEXIO_CLK_FREQ);
FLEXIO_UART_WriteBlocking(&uartDev, txData, BUFFER_SIZE);
FLEXIO_UART_ReadBlocking(&uartDev, rxData, BUFFER_SIZE);
```

Table 11 summarizes the comparison of FLEXIO APIs.

Table 11. **FLEXIO APIs comparison**

Functions	AUTOSAR	MCUXpresso
I2C controller initialization	Flexio_I2c_Ip_MasterInit (instance, channel, Flexio_I2c_Ip_MasterConfigType*)	FLEXIO_I2C_MasterInit (FLEXIO_I2C_Type*, flexio_i2c_master_config_t*, srcClkHz)
I2C asynchronous writing	I2c_AsyncTransmit (Channel, I2c_requestType*) with STD_ON=I2C_FLEXIO_USED	FLEXIO_I2C_MasterTransferNonBlocking (FLEXIO_I2C_Type*, flexio_i2c_master_handle_t*, flexio_i2c_master_transfer_t*)
I2C synchronous writing	I2c_SyncTransmit (Channel, I2c_RequestType*) with STD_ON=I2C_FLEXIO_USED	FLEXIO_I2C_MasterTransferBlocking (FLEXIO_I2C_Type*, flexio_i2c_master_transfer_t*)
I2C controller status	I2c_Ipw_GetStatus (Channel, I2c_HwUnitConfigType*) with STD_ON=I2C_FLEXIO_USED	FLEXIO_I2C_MasterGetStatusFlags (FLEXIO_I2C_Type*)
SPI controller/target initialization	Spi_Init (Lpspi_Ip_ConfigType*) with SPI_IPW_SPI_FLEXIO_ENABLE = STD_ON	FLEXIO_SPI_MasterInit (FLEXIO_SPI_Type*, flexio_spi_master_config_t*, srcClkHz)/ FLEXIO_SPI_SlaveInit (FLEXIO_SPI_Type*, flexio_spi_slave_config_t*)
SPI controller asynchronous writing	Flexio_Spi_Ip_AsyncTransmit (Flexio_Spi_Ip_ExternalDeviceType*, uint8_t*, uint8_t*, uint16_t, Flexio_Spi_Ip_CallbackType)	FLEXIO_SPI_MasterTransferNonBlocking (FLEXIO_SPI_Type*, flexio_spi_master_handle_t*, flexio_spi_transfer_t*) + FLEXIO_SPI_MasterTransferCreateHandle (FLEXIO_SPI_Type*, flexio_spi_master_handle_t*, flexio_spi_master_transfer_callback_t, userData)
SPI controller synchronous writing	Flexio_Spi_Ip_SyncTransmit (Flexio_Spi_Ip_ExternalDeviceType*, uint8_t*, uint8_t*, uint16_t, uint32_t)	FLEXIO_SPI_MasterTransferBlocking (FLEXIO_SPI_Type*, flexio_spi_transfer_t*)
SPI target asynchronous writing	Not supported for a FLEXIO SPI device configured as a target.	FLEXIO_SPI_SlaveTransferNonBlocking (FLEXIO_SPI_Type*, flexio_spi_slave_handle_t*, flexio_spi_transfer_t*) + FLEXIO_SPI_SlaveTransferCreateHandle (FLEXIO_SPI_Type*, flexio_spi_slave_handle_t*, flexio_spi_slave_transfer_callback_t, userData)
SPI controller/target status	Flexio_Spi_Ip_GetStatus (instance)	FLEXIO_SPI_GetStatusFlags (FLEXIO_SPI_Type*)

Table 11. FLEXIO APIs comparison...*continued*

Functions	AUTOSAR	MCUXpresso
UART initialization	Flexio_Uart_Ip_Init (channel, Flexio_Uart_Ip_UserConfigType*)	FLEXIO_UART_Init (FLEXIO_UART_Type*, flexio_uart_config_t*, srcClkHz)
UART asynchronous writing	Flexio_Uart_Ip_AsyncSend (channel, uint8_t*, TxSize)	FLEXIO_UART_TransferSendNonBlocking (FLEXIO_UART_Type*, flexio_uart_handle_t*, flexio_uart_transfer_t*)
UART asynchronous reading	Flexio_Uart_Ip_AsyncReceive (channel, uint8_t*, RxSize)	FLEXIO_UART_TransferReceiveNonBlocking (FLEXIO_UART_Type*, flexio_uart_handle_t*, flexio_uart_transfer_t*, size_t*)
UART synchronous writing	Flexio_Uart_Ip_SyncSend (channel, uint8_t*, TxSize, Timeout)	FLEXIO_UART_WriteBlocking (FLEXIO_UART_Type*, uint8_t*, txSize)
UART synchronous reading	Flexio_Uart_Ip_SyncReceive (channel, uint8_t*, RxSize, Timeout)	FLEXIO_UART_ReadBlocking (FLEXIO_UART_Type*, uint8_t*, rxSize)
UART status (asynchronous only)	Flexio_Uart_Ip_GetStatus (channel, uint32_t*)	FLEXIO_UART_GetStatusFlags (FLEXIO_UART_Type*)

6.10 Watchdog

This section covers the migration of the watchdog IP from AUTOSAR RTD to the MCUXpresso SDK. The watchdog module plays a crucial role in system reliability by monitoring software execution and triggering resets on failure conditions. While both platforms provide watchdog functionality, their APIs and configuration approaches differ. This chapter highlights key differences, mapping of functions, and considerations to ensure a smooth transition and maintain system safety during migration.

Table 12. Watchdog APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Wdg_ChannelInit (Wdg_Ipw_InstanceType, Wdg_ConfigType*)	WDOG32_Init (WDOG_Type*, wdog32_config_t*)
Refresh counter	Wdg_ChannelService (Wdg_Ipw_InstanceType)	WDOG32_Refresh (WDOG_Type*)

6.11 General-Purpose Timer (GPT)

In the AUTOSAR RTD framework, the GPT module serves as a high-level abstraction over multiple underlying timer IPs, primarily Periodic Interrupt Timer (PIT), and System Timer Module (STM). Depending on the configuration, a GPT channel may map to either hardware block, with each backend initialized by its respective IP driver, such as Pit_Ip_Init() or Stm_Ip_Init(). This approach allows for flexible channel allocation and consistent GPT API usage across hardware.

In contrast, the MCUXpresso SDK does not include a unified GPT abstraction. Instead, applications directly interact with the PIT and STM modules through low-level drivers, such as fsl_pit.h and fsl_stm.h. While this reduces software abstraction and configuration complexity, it shifts the responsibility of timer selection and management to the application developer.

Table 13. GPT APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Pit_Ip_Init (instance, Pit_Ip_InstanceConfigType*)/ Stm_Ip_Init (instance, Stm_Ip_InstanceConfigType*)	PIT_Init (LPIT_Type*, pit_config_t*)/ STM_Init (STM_Type*, stm_config_t*)

Table 13. GPT APIs comparison*...continued*

Functions	AUTOSAR	MCUXpresso
Configure the channel	Pit_Ip_InitChannel (Gpt_Ipw_HwChannelConfig Type*)/ Stm_Ip_InitChannel (instance, Stm_Ip_Channel ConfigType*)	No equivalent for both modules
Configure Irq	Pit_Ip_EnableChannelInterrupt (instance, hw Channel)/ For an STM enabled channel the interrupts are always activated	PIT_EnableInterrupts (PIT_Type*, pit_chnl_t, mask) + PIT_SetTimerPeriod (PIT_Type*, pit_ chnl_t, uint32_t)
Start	Pit_Ip_StartChannel (instance, channel, countVal)/ Stm_Ip_StartCounting (instance, channel, compareVal)	LPIT_StartTimer (LPIT_Type*, lpit_chnl_t)/ STM_StartTimer (STM_Type*)

6.12 TRGMUX

In AUTOSAR RTD, there is generally no dedicated TRGMUX driver exposed. Trigger routing is configured through integration tools or within specific IP configuration, unlike the MCUXpresso SDK where TRGMUX is handled explicitly.

Table 14. TRGMUX MCUXpresso APIs

Functions	MCUXpresso
Configure trigger	TRGMUX_SetTriggerSource (TRGMUX_Type*, index, trgmux_trigger_input_t, triggerSrc)
Lock configuration	TRGMUX_LockRegister (TRGMUX_Type*, index)

6.13 Enhanced Direct Memory Access (EDMA)

There is no direct equivalent of the EDMA module in the AUTOSAR RTD SDK. While the MCUXpresso SDK provides a set of dedicated APIs for configuring and using EDMA transfers, the RTD abstracts these operations. DMA usage is often implicitly managed within peripheral drivers like LPUART, LPI2C, or LPSPI through configuration structures.

Additionally, the MCUXpresso SDK includes dedicated driver files, such as lpi2c_edma.c/h, lpspi_edma.c/h, and lpuart_edma.c/h, which offer fine-grained control over DMA transfers for these specific peripherals. This separation is not present in the RTD, where the DMA behavior is typically configured through the generator tools and not exposed through standalone APIs.

Table 15. EDMA MCUXpresso APIs

Functions	MCUXpresso
Initialization	EDMA_Init (EDMA_Type*, edma_config_t*)
Handle initialization	EDMA_CreateHandle (edma_handle_t*, EDMA_Type*, channel)
Configuration	EDMA_SetTransferConfig (EDMA_Type*, channel, edma_transfer_config_t*, edma_tcd_t*) + EDMA_ PrepareTransferConfig (edma_transfer_config_t*, srcAddr, srcWidth, srcOffset, destAddr, destWidth, destOffset, bytesEachRequest, transferBytes)
Irq	EDMA_EnableChannelInterrupts (EDMA_Type*, channel, mask)
Submit transfer request	EDMA_SubmitTransfer(edma_handle_t*, edma_transfer_config_t*)

Table 15. EDMA MCUXpresso APIs*...continued*

Functions	MCUXpresso
Start transfer	EDMA_StartTransfer (edma_handle_t*)
Status	EDMA_GetChannelStatusFlags (EDMA_Type*, channel)

6.14 FLEXCAN

This section addresses the migration of the FLEXCAN IP from AUTOSAR RTD to the MCUXpresso SDK. FLEXCAN is a widely used Controller Area Network (CAN) peripheral that facilitates reliable communication in automotive and industrial applications. Although both environments support FLEXCAN, differences exist in API design, configuration options, and driver features. This chapter outlines the key changes, provides a function mapping, and discusses best practices to ensure seamless migration while preserving CAN functionality and performance.

Table 16. FLEXCAN APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Can_43_FLEXCAN_Init (Can_43_FLEXCAN_ ConfigType*)	FLEXCAN_Init (Can_Type*, flexcan_config_t*, src ClkHz)
Configure controller mode	Can_43_FLEXCAN_SetControllerMode (controller, Can_ControllerStateType)	FLEXCAN_EnterFreezeMode (CAN_Type*)/ FLEXCAN_ExitFreezeMode (CAN_Type*)
Asynchronous writing	Can_43_FLEXCAN_Write (Can_HwHandleType, Can_PduType*)	FLEXCAN_TransferSendNonBlocking(CAN_Type *, flexcan_handle_t*, flexcan_mb_transfer_t*)

6.15 Logic Control Unit (LCU)

In the RTD, the LCU is not exposed through a dedicated API. If used at all, it is typically configured indirectly through other modules, such as PWM or timers, or through low-level register settings managed through tools like EB Tresos. As a result, direct software control of the LCU is limited or unavailable in standard RTD-based applications.

In contrast, the MCUXpresso SDK provides an explicit driver for the LCU through the fsl_lcu.h interface. This API allows direct configuration of logical input sources, channel behavior, and output conditions using functions like LCU_Init() and LCU_MuxSelect(). This functionality enables more transparent and programmable use of the LCU for implementing logic-based signal routing or control schemes.

Table 17. LCU MCUXpresso APIs

Functions	MCUXpresso
Initialization	LCU_Init (LCU_Type*)
Input configuration	LCU_MuxSelect (LCU_Type*, lcu_inputs_t, lcu_muxcel_source_t)
Output initialization	LCU_OutputInit (LCU_Type*, lcu_outputs_t, lcu_output_config_t*)
Set interrupt	LCU_SetLutInterrupt (LCU_Type*, mask, isEnable)

6.16 Ethernet (ENET)

This section focuses on migrating the ENET IP from AUTOSAR RTD to the MCUXpresso SDK. The ENET module provides Ethernet communication capabilities essential for modern connected embedded systems. Although both frameworks support Ethernet functionalities, their APIs, driver architectures, and configuration mechanisms differ. This chapter details the key differences, maps corresponding functions, and offers guidance to ensure a smooth transition, preserving network performance and reliability throughout the migration process.

Table 18. ENET APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Eth_43_GMAC_Init (Eth_43_GMAC_ConfigType*)	ENET_GetDefaultConfig (enet_config_t*) + ENET_Init (ENET_Type*, enet_handle_t*, enet_config_t*, enet_buffer_config_t*, uint8_t*, srcClkHz)
Set ENET mode	Eth_43_GMAC_SetControllerMode (CtrlIdx, Eth_ModeType)	No equivalent function (integrated in the enet_config_t structure)
Get MAC address	Eth_43_GMAC_GetPhysAddr (CtrlIdx, uint8_t*)	ENET_GetMacAddr (ENET_Type*, uint8_t*)
Asynchronous writing	Eth_43_GMAC_Transmit (CtrlIdx, Eth_BufIdxType, Eth_FrameType, isTxConfirmation, LenByte, uint8_t*)	ENET_SendFrame (ENET_Type*, enet_handle_t*, uint8_t*, uint32_t, uint8_t, isTs, ctx)
Validate transmit	Eth_43_GMAC_TxConfirmation (CtrlIdx)	ENET_TransmitIRQHandler (ENET_Type*, enet_handle_t*, uint32_t)
Asynchronous reading	Eth_43_GMAC_Receive (CtrlIdx, FifoIdx, Eth_RxStatusType*)	ENET_ActiveRead (ENET_Type*) + ENET_ReadFrame (ENET_Type*, enet_handle_t*, uint8_t*, length, uint8_t, uint32_t*)

6.16.1 Precision Time Protocol (PTP) 1588

The MCUXpresso SDK supports IEEE 1588 PTP on supported devices, enabling hardware timestamping for Ethernet frames. This feature is useful in time-sensitive applications requiring synchronization across a network, such as industrial automation or automotive Ethernet.

Migration considerations:

In the AUTOSAR RTD implementation, dedicated service layers or time synchronization managers can abstract and manage PTP support. In contrast, the MCUXpresso SDK exposes lower-level control, and available functionality can differ significantly depending on the hardware platform.

For the MCX E31x and other MCX-class devices, PTP support is provided through the ENET driver. However, this driver offers a limited subset of the PTP API compared to more feature-rich platforms like i.MX RT.

The following are the key points for migration:

- AUTOSAR RTD implementations often abstract PTP support or require manual timestamp handling through low-level driver extensions.
- MCUXpresso supports PTP natively through functions such as:
  - ENET_Ptp1588Configure(ENET_Type*, enet_handle_t*, enet_ptp_config_t*)
  - ENET_Ptp1588GetTimer(ENET_Type*, uint64_t*, uint32_t*)
- Hardware timestamping is enabled and configured through the enet_ptp_config_t structure and ENET_Init(ENET_Type*, enet_handle_t*, enet_config_t*, enet_buffer_config_t*, uint8_t*, srcClkHz).

Using these native SDK functions allows precise synchronization with minimal CPU overhead and avoids reliance on external timing mechanisms.

6.16.2 Time-Sensitive Networking (TSN)

In the MCUXpresso SDK, MCX E31x include a variant of the standard Ethernet driver named enet_qos. This driver extends the base ENET functionality with features intended for Quality of Service (QoS) and TSN use cases.

Table 19. ENET QoS APIs

Functions	MCUXpresso
Initialization	ENET_QOS_GetDefaultConfig (enet_qos_config_t*) + ENET_QOS_Init (ENET_QOS_Type*, enet_qos_config_t*, uint8_t*, uint8_t, uint32_t)
Get MAC address	ENET_QOS_GetMacAddr (ENET_QOS_Type*, uint8_t*, uint8_t)
Asynchronous writing	ENET_QOS_SendFrame (ENET_QOS_Type*, enet_qos_handle_t*, uint8_t*, uint32_t, uint8_t, bool, void*, enet_qos_tx_offload_t)
Validate transmit	ENET_QOS_CommonIRQHandler (ENET_QOS_Type*, enet_qos_handle_t*,
Asynchronous reading	ENET_QOS_GetRxFrameSize (ENET_QOS_Type*, enet_qos_handle_t*, uint32_t*, uint8_t) + ENET_QOS_ReadFrame (ENET_QOS_Type*, enet_qos_handle_t*, uint8_t*, uint32_t, uint8_t, enet_qos_ptp_time_t*)
PTP Get timer	ENET_QOS_Ptp1588GetTimer (ENET_QOS_Type*, uint64_t*, uint32_t*)

The enet_qos drivers expose key functions required for standard Ethernet operation, with added support for time-sensitive features. While PTP support is already partially available through functions like ENET_QOS_Ptp1588GetTimer, the driver is designed to integrate with advanced TSN use cases, such as frame scheduling, timestamping, and traffic prioritization. This TSN readiness allows the MCX E31x platform to support protocols like IEEE 802.1AS (gPTP), 802.1Qbv (time-aware shaping), and 802.1Qbu (frame preemption), positioning it for real-time and deterministic Ethernet communication.

6.17 Cyclic Redundancy Check (CRC)

The CRC peripheral is used to compute checksums for data integrity verification. This functionality is useful in applications such as communication protocols, flash data validation, and bootloader security.

In AUTOSAR RTD, the CRC module is highly configurable. It supports multiple logic channels, calculation methods (software, lookup tables, or hardware), and protocol types (for example, CRC-8, CRC-16, CRC-32). It abstracts CRC operations through standardized APIs with configuration driven by external tools.

In contrast, the MCUXpresso SDK adopts a simpler approach, offering direct access to a hardware CRC peripheral through low-level drivers. Configuration is performed programmatically using structures like crc_config_t, and only one CRC instance is typically available, with fewer protocols supported depending on the MCU.

Example 19: CRC8 Calculation RTD

The following codeblock demonstrates the CRC8 calculation for RTD:

```
Crc_Init(NULL_PTR;
CrcResult = Crc_SetChannelCalculate(CRC_LOGIC_CHANNEL_1, &CRC_data[0],
 CRC_DATA_SIZE, 0U, TRUE);
```

Example 20: CRC16 Calculation MCUXpresso

```
crc_config_t config;
CRC_GetDefaultConfig(&config);
config.polynomial = 0x1021U; // CRC-CCITT
CRC_Init(CRC0, &config);
```

```
CRC_WriteData(CRC0, dataBuffer, length);
uint32_t crc = CRC_Get16bitResult(CRC0);
```

7 Safety libraries

The MCX E31x platform introduces a dedicated set of software components to support the development of safety-critical applications in accordance with IEC 61508. While the S32K family is aligned with ISO 26262 for automotive functional safety, the MCX E31x targets broader industrial use cases with safety support structured around the MCX E31x Safety Software Framework (SAF).

Several key safety software elements are provided by NXP:

- **Structural core self-test (SCST CM7):** A standalone library for detecting permanent hardware faults in the application core. It executes deterministic instruction sequences and compares the results to validate core integrity.
  Note: *This component is delivered separately and is not part of the MCX E31x SAF.*
- **MCX E31x Safety Software Framework (SAF):** The SAF includes high-level components to validate and monitor safety configurations throughout the system life cycle. The sBoot component checks safety settings at boot time, while SquareCheck (sCheck) provides runtime validation of safety-critical configurations and supports “check-the-checker” patterns. These tests help detect latent faults and ensure that safety mechanisms remain operational during execution.

Together, these components include pre-certified software libraries for IEC 61508 SIL compliance, enabling developers to build safety-critical applications on MCX E31x with reduced certification effort. Migration from an ISO 26262-compliant platform like S32K3 can require adapting to different documentation, test coverage expectations, and certification artifacts. Developers must consult the MCX E31x SAF documentation and safety manuals to align their safety strategy with the expectations of the new platform.

8 Middleware libraries

The MCUXpresso SDK offers a rich set of middleware libraries that simplify the development of advanced control and signal processing applications.

- **Real time Control Embedded Software Library (RTCESL):** Optimized for real-time applications like motor control and digital filters. RTCESL is the industrial equivalent of the AMMCLib automotive library. The motor control subsystem on MCX E31x includes blocks, such as TRGMUX, eMIOS, BCTU, LCU, and LPCMP. However, their complexity is mostly abstracted when using RTCESL, allowing for a simplified and high-level development experience.
- **Lightweight IP (lwIP):** Compact TCP/IP stack optimized for embedded systems; supports IPv4/IPv6, sockets, and various network protocols; integrated with MCUXpresso SDK’s ENET driver for Ethernet connectivity.
- **Time-Sensitive Networking (TSN):** Enabling real-time and deterministic communication over standard networks. A dedicated TSN stack is available, supporting protocols such as 802.1AS (gPTP), 802.1Qbv (Time-Aware Shaper), and 802.1Qbu (Frame Preemption). This functionality makes MCX E31x suitable for industrial Ethernet applications requiring precise synchronization and guaranteed latency.
- **FreeMASTER:** Real-time data visualization and control tool, enabling runtime monitoring, variable tuning, and oscilloscope-style data logging through UART, CAN or USB.

8.1 Real-Time Control Embedded Software Library (RTCESL)

The RTCESL is a collection of optimized and modular signal processing and control functions, designed for real-time motor control applications on NXP microcontrollers. It enables developers to implement precise and efficient control loops, such as Field Oriented Control (FOC), sensorless estimators, and current regulation, using pre-validated building blocks.

RTCESL is composed of several specialized sub-libraries, each targeting a functional domain. MLIB for low-level math operations, GFLIB for general control utilities, GDFLIB for digital filtering, GMCLIB for motor-specific transformations, and AMCLIB for advanced control structures like observers and flux weakening. These libraries support both fixed-point and floating-point formats and offer performance-optimized implementations through inline C and assembly variants.

This section provides an overview of each sub-library and highlights representative APIs to guide developers in selecting the appropriate building blocks for their application. A deeper focus is given to GDFLIB and GMCLIB, which play a central role in signal conditioning and motor vector control.

- **Math Library (MLIB):** Provides basic arithmetic operations optimized for fixed-point math, such as saturated addition, multiplication, and shift operations.
  Example 21: MLIB example APIs
  ```
  MLIB_AddSat_F16(frac16_t, frac16_t);
  MLIB_Mul_F16(frac16_t, frac16_t);
  ```
- **Advanced Motor Control Library (AMCLIB):** Contains motor control-specific algorithms such as flux weakening, BEMF observers, and tracking observers, mostly in Q15 format.
  Example 22: AMCLIB example APIs
  ```
  AMCLIB_TrackObsrv_F16(frac16_t, AMCLIB_TRACK_OBSRV_T_F32*);
  AMCLIB_PMSMBemfObsrvDQInit_F16(AMCLIB_BEMF_OBSRV_DQ_T_A32*);
  AMCLIB_CtrlFluxWkng_F16(frac16_t, frac16_t, frac16_t,
   AMCLIB_CTRL_FLUX_WKNG_T_A32*);
  ```
- **General Function Library (GFLIB):** Offers generic mathematical functions like sine/cosine, ramp generators, and limiters used in control applications.
  Example 23: GFLIB example APIs
  ```
  GFLIB_SinCos_F16(frac16_t, GMCLIB_2COOR_SINCOS_T_F16*);
  GFLIB_Ramp_F16(frac16_t, GFLIB_RAMP_T_F16*);
  GFLIB_Limit_F16(frac16_t, frac16_t, frac16_t);
  ```

For a full API reference and implementation details, refer to Section 14 to access the official user guides for each library:

- MLIB User's Guide
- AMCLIB User's Guide
- GFLIB User's Guide

8.1.1 General Motor Control Library (GMCLIB)

The GMCLIB provides essential mathematical transformations used in vector control algorithms, especially in FOC of AC motors. It includes implementations of Clarke and Park transforms, inverse transforms, and coordinate rotations that are fundamental to operating in rotating reference frames. The functions are highly optimized and available in fixed-point and floating-point formats, with variants for C inline and assembly-level performance.

These blocks are used at the core of the control loop. They convert three-phase quantities into two-axis orthogonal representations (αβ or DQ frames), enabling independent control of torque and flux components. GMCLIB functions are reusable across different motor types and can be integrated flexibly in custom control strategies.

Table 20. GMCLIB APIs

Feature	RTCESL API
Clarke transformation/ Inverse clarke transformation (float32, fast C inline)	GMCLIB_Clark_FLT(GMCLIB_3COOR_T_FLT*, GMCLIB_2COOR_ALBE_T_FLT*)/ GMCLIB_ClarkInv_FLT(GMCLIB_2COOR_ALBE_T_FLT*, GMCLIB_3COOR_T_FLT*)
Park transformation/ Inverse park transformation (float32, fast C inline)	GMCLIB_Park_FLT(GMCLIB_2COOR_ALBE_T_FLT*, GMCLIB_2COOR_SINCOS_T_FLT*, GMCLIB_2COOR_DQ_T_FLT*)/GMCLIB_ParkInv_FLT(GMCLIB_2COOR_DQ_T_FLT*, GMCLIB_2COOR_SINCOS_T_FLT*, GMCLIB_2COOR_ALBE_T_FLT*)

For detailed information on API functions, data types, and usage guidelines, refer to the Section 14 to access GMCLIB User's Guide, which provides comprehensive documentation.

8.1.2 General Digital Filter Library (GDFLIB)

The GDFLIB provides signal filtering and conditioning functions required in motor control, sensing, and real-time regulation applications. It includes implementations of Infinite Impulse Response (IIR) filters, moving average filters, and controllers (such as PID and PI) all designed for deterministic real-time execution on embedded targets.

These functions are used to reduce noise on measurement inputs (for example, current or speed feedback), shape control signals, or implement robust feedback loops. GDFLIB functions are available in fixed-point formats and optimized for Cortex-M performance, making them suitable for high-frequency control loops with limited computational overhead.

Table 21. GDFLIB APIs

Feature	RTCESL API
Initialize exponential filter with initial value	GDFLIB_FilterExpInit_FLT(float_t, GDFLIB_FILTER_EXP_T_FLT*)
Apply exponential filter to input value	GDFLIB_FilterExp_FLT(float_t, GDFLIB_FILTER_EXP_T_FLT*)
Initialize first-order IIR filter	GDFLIB_FilterIIR1Init_FLT(GDFLIB_FILTER_IIR1_T_FLT*)
Apply first-order IIR filter to input	GDFLIB_FilterIIR1_FLT(float_t, GDFLIB_FILTER_IIR1_T_FLT*)

For a complete list of available filter functions and data types, refer to the Section 14 to access GDFLIB User’s Guide, which provides exhaustive API details.

8.2 FreeMASTER

This section highlights the core functions required to enable FreeMASTER communication and runtime signal capture on MCX E31x platforms.

FreeMASTER is a real-time data visualization and control tool provided by NXP. It is designed to assist developers in debugging, tuning, and monitoring embedded applications without halting the system. It enables non-intrusive access to variables, oscilloscope signal monitoring, remote procedure calls, and communication with custom host applications through serial or CAN interfaces.

FreeMASTER integrates seamlessly with the MCUXpresso SDK and supports various middleware components such as motor control libraries (RTCESL). Its key runtime components include a polling engine, recorder for high-frequency signal tracking, and optional interrupt-based communication for performance-sensitive use cases. Developers can quickly instrument their application by calling a minimal set of FreeMASTER APIs.

Table 22. FreeMASTER APIs

Feature	FreeMASTER API
Initialization	FMSTR_Init()
Handle communication and processing	FMSTR_Poll()
Optional ISR handler for non-blocking serial communication	FMSTR_SerialIsr()
Recorder initialization, enabling runtime data logging/Record one sample	FMSTR_RecorderCreate(FMSTR_INDEX, FMSTR_REC_BUFF)/FMSTR_Recorder(FMSTR_INDEX)

For full details on FreeMASTER's capabilities and APIs, such as initialization, polling, interrupt handling, and recorder functionality, refer to the Section 14 to access FreeMASTER Serial Communication driver – User Guide.

8.3 Availability in MCUXpresso SDK

The MCUXpresso SDK includes a set of libraries designed to support high-performance control applications. These libraries are:

- Optimized for Cortex-M cores with support for fixed-point and floating-point math.
- Supplied with example applications, making it easier to prototype and validate control systems (for example, PMSM FOC).
- Released as precompiled libraries with header files and optional CMSIS-DSP dependencies.
- Compatible with BareMetal and FreeRTOS environments.

This functionality gives developers access to high-performance control algorithms without building them from scratch or integrating external IP.

9 RTOS migration considerations

This section describes considerations for Real Time Operating System (RTOS) migration.

9.1 From OsIf (RTD) to FreeRTOS (MCUXpresso)

This section provides guidance on how to transition these services and identifies key differences and considerations.

The AUTOSAR RTD stack abstracts the operating system through a layer called the Operating System Interface (OsIf). OsIf provides basic RTOS-like services, such as delay functions, semaphores, and interrupt management. It is typically implemented over a lightweight internal RTOS or scheduler, or mapped to a third-party RTOS depending on the integration.

In contrast, the MCUXpresso SDK relies on direct integration with FreeRTOS, a widely adopted, open source real-time operating system. Migrating from OsIf to FreeRTOS requires mapping RTD abstractions to native FreeRTOS mechanisms.

Additionally, for applications requiring certified safety RTOS support, the MCUXpresso SDK provides a simple integration path to Green Hills SafeRTOS, offering a commercially supported alternative that aligns with IEC 61508 SIL requirements.

9.1.1 Core differences

Table 23 describes the comparison matrix between OsIf and FreeRTOS among various features:

Table 23. OsIf and FreeRTOS comparison

Feature	AUTOSAR RTD + OsIf	MCUXpresso SDK + FreeRTOS
Delays	OsIf_GetCounter (OsIf_CounterType) + OsIf_MicroToTicks (micros, OsIf_CounterType) + OsIf_GetElapsed (uint32_t*, OsIf_CounterType)	vTaskDelay (TickType_t)
Semaphores	OsIf_Software_Semaphore_Lock (uint32_t*, LockVal) OsIf_Software_Semaphore_Unlock (uint32_t*, LockVal)	xSemaphoreTake (SemaphoreHandle_t, TickType_t) xSemaphoreGive (SemaphoreHandle_t)

Interrupt management	OsIf_Interrupts_SuspendAllInterrupts()
	OsIf_Interrupts_ResumeAllInterrupts()
		taskENTER_CRITICAL()
		taskEXIT_CRITICAL()

Scheduler	Abstracted	vTaskStartScheduler()
Task creation	Managed outside OsIf	xTaskCreate (TaskFunction_t, char*, configSTACK_DEPTH_TYPE, void*, UBaseType_t, TaskHandle_t*)
Timebase	Configurable operating system tick	configTICK_RATE_HZ

9.1.2 Additional considerations

The following considerations are important when working with FreeRTOS in place of AUTOSAR RTD:

- **Startup and initialization:** In AUTOSAR RTD, operating system startup is typically implicit or layered. With FreeRTOS, you must explicitly call vTaskStartScheduler() after creating your tasks.
- **Priority handling:** FreeRTOS uses numeric priorities with configMAX_PRIORITIES granularity. Review task and interrupt priorities carefully to match RTD system behavior.
- **Error handling:** Unlike OsIf, FreeRTOS functions return explicit error codes or status values. Always handle these return values to improve robustness.

9.2 From OsIf (RTD) to Zephyr operating system

This section provides guidance on how to transition these services and identifies key differences and considerations.

The Zephyr operating system is an open source, scalable Real-Time Operating System (RTOS) designed for resource-constrained embedded devices. Like FreeRTOS, it provides essential services such as task scheduling, synchronization primitives, and device drivers, but with additional features targeting IoT and modern embedded use cases.

When transitioning from OsIf (AUTOSAR RTD) to the Zephyr operating system, consider the following factors into account:

- **RTOS integration:** Whereas OsIf provides a lightweight abstraction layer primarily to interface with the underlying AUTOSAR OS or BareMetal environment, Zephyr is a full RTOS offering standardized APIs, device drivers, and middleware.
- **API changes:** Application code, which uses interrupt management or task services of OsIf, needs adaptation to APIs of Zephyr (k_thread_*, k_mutex, irq_*), which provide richer functionality but differ in semantics and structure.
- **Configuration and build system:** Zephyr uses a Device Tree and Kconfig-based configuration system, which requires a shift from the static configuration styles often used in AUTOSAR RTD projects.
- **Multicore and SMP support:** Zephyr supports SMP on multiple cores, whereas AUTOSAR RTD OsIf typically assumes single-core or tightly coupled cores, impacting concurrency and synchronization strategies.

- **Community and ecosystem:** Zephyr has an active open source community and supports a wide range of hardware platforms, easing the integration of new drivers and protocols compared to proprietary AUTOSAR stacks.

In summary, migrating from OsIf to Zephyr involves moving from a minimal operating system abstraction to a feature-rich RTOS platform, enabling more complex and connected embedded applications but requiring adjustments in API usage, configuration, and system design.

9.2.1 Core differences

Table 24 describes the comparison matrix between OsIf and FreeRTOS among various features:

Table 24. OsIf and Zephyr comparison

Feature	AUTOSAR RTD + OsIf	MCUXpresso SDK + Zephyr
Delays	OsIf_GetCounter (OsIf_CounterType) + OsIf_MicroToTicks (micros, OsIf_CounterType) + OsIf_GetElapsed (uint32_t*, OsIf_CounterType)	k_sleep (k_timeout_t)/k_busy_wait (usec_to_wait)
Semaphores	OsIf_Software_Semaphore_Lock (uint32_t*, LockVal) OsIf_Software_Semaphore_Unlock (uint32_t*, LockVal)	k_sem_take (k_sem*, k_timeout_t) k_sem_give (k_sem*)
Interrupt management	OsIf_Interrupts_SuspendAllInterrupts () OsIf_Interrupts_ResumeAllInterrupts ()	irq_lock () irq_unlock (key)
Scheduler	Abstracted	Automatically started; cooperative or preemptive
Task creation	Managed outside OsIf	k_thread_create (k_thread*, k_thread_stack_t*, stack_size, k_thread_entry_t, void*, void*, void*, prio, options, k_timeout_t)
Timebase	Configurable operating system tick	CONFIG_SYS_CLOCK_TICKS_PER_SEC

9.3 OsIf internals: Zephyr and FreeRTOS integration

The OsIf layer in the AUTOSAR RTD stack is designed to abstract the underlying RTOS implementation. This design allows the same driver and middleware code to operate over different system-level platforms, such as:

- FreeRTOS
- Zephyr RTOS
- AUTOSAROS
- BareMetal fallback (no RTOS, minimalistic critical section emulation)

Depending on the project configuration and target board, the OsIf implementation conditionally compiles different logic using preprocessor flags like USING_OS_FREERTOS, USING_OS_ZEPHYR, or USING_OS_AUTOSAROS.

10 Tooling overview

As part of the migration from the AUTOSAR RTD ecosystem to the MCUXpresso SDK, developers must also adapt to a new set of development tools. They must also adjust to different build environments and flashing procedures. This chapter provides an overview of the tooling landscape surrounding MCUXpresso and highlights key differences from the traditional RTD-based workflow. It also covers both secure and non-secure firmware flashing tools, supported IDEs and debuggers, firmware formats, and the bootloader infrastructure commonly used with NXP MCX microcontrollers. Additionally, it outlines the workflows and automation strategies for development, testing, and deployment.

The goal is to equip you with a clear understanding of:

- How to set up and use the development environment effectively
- Which tools to use for flashing and updating firmware (depending on whether secure boot is enabled)
- Differences in firmware file formats and signing processes
- Best practices for working with bootloaders such as MCUboot
- How to script or automate the build and deployment pipeline

Whether you're developing in a BareMetal environment, or working with RTOSes like FreeRTOS or Zephyr, this chapter helps ensure a smooth and efficient transition.

10.1 Compiler

The choice of compiler is an important aspect when considering toolchain qualification for safety-related applications.

In the AUTOSAR RTD environment, development typically relies on GCC-based toolchains (for example, GNU Arm Embedded, Green Hills, or others). It can include ISO 26262/IEC 61508 compliance options, such as MISRA compliance, specific warnings, and code quality checks. However, these toolchains are not safety-certified by default and require additional qualification effort.

In contrast, the MCUXpresso ecosystem targeting the MCX E31x series supports and integrates the IAR Embedded Workbench compiler, which is available in a safety-certified version (certified by TÜV SÜD for ISO 26262 and IEC 61508 up to SIL 3/ASIL D). This reduces the qualification burden for safety projects and enables smoother certification paths in industrial applications.

The availability of a pre-certified compiler makes MCUXpresso a more convenient choice for applications requiring functional safety compliance without extensive toolchain qualification effort.

10.2 Software accessibility

To facilitate software development and migration, NXP provides various tools and resources designed to simplify access to SDKs, examples, and documentation.

10.2.1 SDK builder

The NXP SDK Builder is an online web tool that allows developers to create custom MCUXpresso SDK packages tailored to their specific hardware and software needs.

- **Customization:** Select target devices, middleware components, and example projects for the generation of a pre-configured SDK.
- **Ease of use:** Generates an archive including all necessary drivers, middleware, and sample code, reducing setup time.
- **Regular updates:** Always offers the latest SDK versions and patches, ensuring up-to-date software stacks.

10.2.2 GitHub repositories

NXP maintains an official repositories on GitHub providing the latest SDK source code, examples, and bug fixes:

- **MCUXpresso SDK:** See mcuxsdk-manifests - contains core drivers, middleware, and utilities.
- **Board Support Packages (BSPs):** Includes device-specific configuration and startup code.

- **Example projects:** Reference implementations showcase peripheral usage and RTOS integration.
- **Issue tracking and contributions:** Developers can report issues, request features, and contribute fixes through pull requests.

You can fetch this repo using West CLI:

west init -m https://github.com/nxp-mcuxpresso/mcuxsdk-manifests .

west update

Documentation and support:

- MCUXpresso SDK package of NXP includes detailed API documentation accessible offline.
- Online resources such as the NXP Community Forums and the MCUXpresso IDE provide extensive support for developers.

10.3 Software maintenance

Maintaining the security and integrity of embedded software is a critical aspect of product lifecycle management. When migrating from an AUTOSAR RTD-based stack to the MCUXpresso SDK, it is important to understand how software updates — particularly security patches — are managed in this new ecosystem.

10.3.1 Security patch management in MCUXpresso SDK

NXP does not currently provide a centralized Common Vulnerabilities and Exposures (CVE) tracking system for MCUXpresso SDK components. However, updates are published regularly through:

- **MCUXpresso SDK Builder portal (mcuxpresso.nxp.com):** Updated SDK packages include driver bug fixes, cryptographic improvements, and vulnerability mitigations when applicable.
- **NXP GitHub Repositories (for example, mcux-sdk, mcuboot, mbedtls):** Used for open source contributions, patches, and issue tracking.

Developers are responsible for:

- Regularly checking for new SDK versions and changelogs.
- Manually integrating patches into their project repositories.
- Reviewing updated middleware (for example, mbedTLS, USB stacks) for upstream security fixes.

Patch workflow in MCUXpresso SDK-based projects

Compared to the AUTOSAR RTD environment (often centrally maintained and updated through vendor release schedules), MCUXpresso-based projects require a manual integration strategy, such as:

1. Monitor SDK release notes (NXP portal or GitHub).
2. Identify impacted modules (for example, crypto, net, and so on).
3. Perform regression testing after patch integration.
4. Validate compatibility with bootloader, OTA, or provisioning flow.

Note: *If using a secure bootloader like MCUBoot, make sure to re-sign firmware images after applying critical updates.*

10.4 Software support and ecosystem resources

When migrating from a comprehensive AUTOSAR-based environment, where software support is centralized and tightly managed, it is important to understand the differences in structure and tooling provided by the MCUXpresso SDK ecosystem.

Unlike AUTOSAR-based environment, which can include structured support contracts, official training tracks, and vendor-specific integration layers, MCUXpresso relies on a more community-driven and open-access model. Table 21 is an overview of key support and knowledge-sharing components available when working with NXP MCX-class devices and the MCUXpresso SDK.

Table 25. Key support and knowledge sharing components

Feature	AUTOSAR RTD	MCUXpresso SDK
Official support contracts	✓ Yes	✗ No (only community or through NXP support portal)
Structured knowledge base	✓ Yes (centralized)	✗No (distributed: forums, GitHub, documentation)
Training programs	✓ Yes (vendor-defined)	✓ Yes (public and on-demand)
Migration assistance	✓ Yes (through integrator)	✓ Yes (community/self-driven)
Code ownership	✗No (distributed: abstracted APIs)	✓ Yes (full access to low-level code)

Note: *There is no equivalent to AUTOSAR-specific advanced training tracks, but MCUXpresso learning resources are beginner-friendly and ideal for self-paced onboarding.*

11 Integrated Development Environments (IDEs)

When migrating from an AUTOSAR RTD development environment to MCUXpresso SDK, developers must adapt to a different ecosystem of supported IDEs. Table 22 is an overview of commonly used IDEs and their compatibility with the MCUXpresso SDK:

Table 26. IDEs

IDE	Vendor	MCUXpresso SDK support	Notes
MCUXpresso IDE	NXP	✓ Yes	Official IDE from NXP. Fully integrated with MCUXpresso SDK. Supports SDK import, debug tools, project wizards. Based on Eclipse.
IAR embedded workbench	IAR systems	✓ Yes	Supported through dedicated project templates. Requires SDK export.
S32 design studio	NXP	✗No	Only supports S32 SDK and AUTOSAR RTD. Not compatible with MCUXpresso SDK.
VS code	Microsoft	✓ Yes	Community-driven support through CMake or custom build systems.

The following are the considerations for IDE migration:

- MCUXpresso IDE is recommended for initial development and debugging due to tight integration with the SDK, board support packages, and flashing tools.
- Existing RTD-based workflows in S32DS must be ported manually; no direct project import is available.
- When using IDEs like IAR, SDK components must be exported through the MCUXpresso SDK builder or manually integrated.

11.1 Flash and debug tools

This section summarizes the available options and their use cases.

Flashing firmware to the MCX series microcontrollers, such as the MCX E31, can be achieved using several tools and workflows depending on the development stage (prototyping vs production), bootloader configuration (secure vs non-secure), and toolchain preferences.

11.1.1 Flashing through debug probes

Table 27 describes the commonly supported probes.

Table 27. Commonly supported probes

Probe	Support	Notes
J-Link	MCUXpresso IDE, J-Link commander, OpenOCD	Popular and robust; supports scriptable flashing and debugging.
CMSIS-DAP	MCUXpresso IDE, pyOCD, OpenOCD	Open standard; available on some dev kits.

Typical use cases:

- Application development and debug
- BareMetal flashing without a bootloader
- Erasing/reprogramming internal flash during development

11.2 Secure installer for unsecure manufacturing lines

The flashloader is a key software component in the HSE-B boot process of the MCX E31x platform. After reset, the Secure Boot and Flashloader (SBAF) is the first code executed on the HSE-B secure core. It performs essential system initialization, parses the Image Vector Table (IVT), runs the secure boot process, and ultimately triggers the startup of the application core. As part of this flow, the SBAF is responsible for decrypting, authenticating, and programming the HSE-B firmware into code flash.

To support this, the flashloader image is temporarily loaded and executed from RAM. It communicates with a host PC tool (for example, blhost) through UART or SPI and uses the Message Unit (MU) to interact with the HSE-B. During provisioning, it facilitates the transfer of the HSE-B firmware image, which includes memory configuration and cryptographic services. Once validated, the firmware is written in plaintext to the final HSE-B firmware region in flash, completing the secure provisioning process.

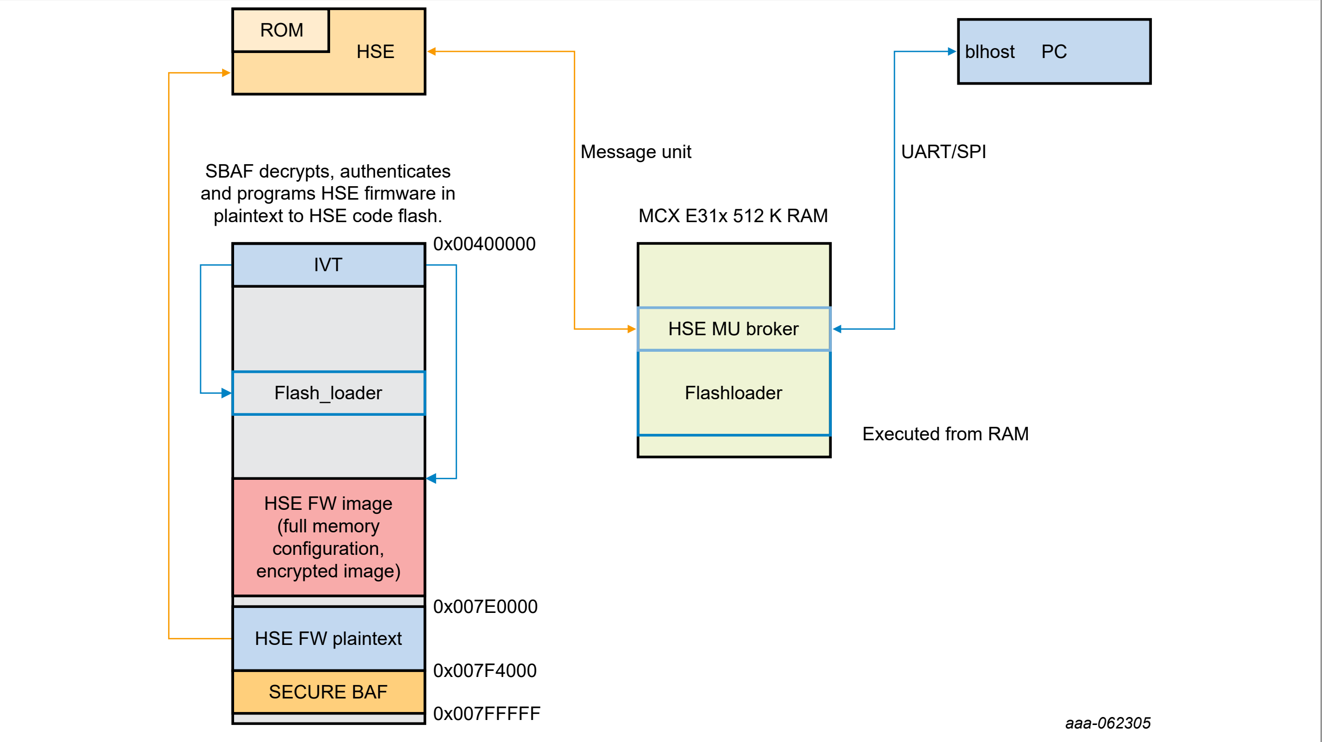

Figure 3. MCX E31x flashloader

11.3 MCUBoot support

On the MCX E31 platform, the MCUBoot open source bootloader is provided as a secondary bootloader to enable secure firmware boot and update capabilities. It is a widely adopted, vendor-agnostic solution that can work with Zephyr, FreeRTOS, or BareMetal MCUXpresso applications.

11.3.1 Key features

Below are the key features of MCUBoot support:

- Authenticated firmware execution
  - Only digitally signed images are allowed to run, preventing unauthorized or tampered code.
- Cryptographic algorithms
  - Signing: ECDSA, ED25519, RSA
  - Integrity: SHA-256, SHA-512
  - Optional encryption: AES
  - Crypto libraries: Mbed TLS or TinyCrypt
- Firmware updates
  - UART, USB, or Over-the-Air (OTA) supported
  - Primary/secondary slots for safe image swapping
  - Rollback mechanism if there is boot failure (not default behavior, configurable)
  - Supports direct or swapping update strategies
- Partition Management
  - Uses predefined flash areas for application images and metadata:
    - primary_slot (active image)
    - secondary_slot (update candidate)

– scratch (temporary swap area, optional)

11.3.2 Migration considerations

When migrating from an AUTOSAR RTD setup, (which often uses raw .hex or .elf files flashed-directly) to MCUXpresso with MCUBoot enabled:

- **Image format:** The application binary must be converted into a MCUBoot-compatible format (with header, signature, and metadata). Use imgtool or integration from the Zephyr/MCUXpresso build system.
- **Signing process:** Establish a signing mechanism using a private key (through imgtool or Zephyr's West toolchain).
- **Bootloader configuration:** Make sure that linker files match the expected layout of MCUBoot (for example, vector table offset).
- **Tools:** Flashing can now involve MCUBoot-specific tools (blhost) instead of traditional flashing utilities.

12 Licensing

This section summarizes the licensing implications when migrating from the previous SDK (AUTOSAR RTD) to the new development environment (MCUXpresso SDK).

12.1 Licensed software components

The original AUTOSAR-based platform relies on licensed software components, some of which required activation through Flexera or equivalent license management systems.

Table 28. License migrations

Component/Tool	Vendor	License type	License required	Migration required
AUTOSAR runtime drivers	NXP	Commercial (Flexera)	✓ Yes	Yes
HSE-B firmware	NXP	Proprietary Binary (EULA)	✓ Yes	✓ Yes
Debugger (for example, Lauterbach)|	Lauterbach	Dongle	✓ Yes	✗ No

Note: *Any license tied to a specific machine (node-locked) or managed through a license server must be reviewed for transfer or deactivation.*

12.2 Flexera licensing

Flexera was used for license activation of several tools in the AUTOSAR SDK. Upon migration, the license keys must be:

- Deactivated in the previous development environment
- Reallocated or reactivated on the new setup

Refer to the Flexera documentation or support contact of your vendor for guidance on license migration procedures.

12.3 MCUXpresso SDK licensing

The MCUXpresso SDK is primarily distributed under open source and permissive licenses:

Table 29. MCUXpresso licenses

Component	License type
Core SDK drivers	BSD-3-Clause
Middleware (FreeRTOS, and so on)	MIT/BSD
CMSIS and Arm components	Apache 2.0/BSD
HSE-B firmware	EULA

A key improvement in the MCUXpresso SDK environment is the integrated delivery of security-related software, such as cryptographic libraries or secure boot components.

These licenses generally allow:

- Use in commercial applications
- Redistribution of binaries without royalty
- Modifications, provided license texts are preserved

Note: *You are responsible for reviewing license obligations before integrating third-party or modified open source code into production software.*

13 Acronyms

This section lists acronyms used in this document.

Table 30. Acronyms

Acronym	Definition
ADC	Analog-to-Digital Converter
AMCLIB	Advanced Motor Control Library
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basic Software
BSPs	Board Support Packages
CAN	Controller Area Network
CMSIS	Cortex Microcontroller Software Interface Standard
CMP	Comparator
CRC	Cyclic Redundancy Check
CVE	Common Vulnerabilities and Exposures
DEM	Diagnostic Event Manager
DMA	Direct Memory Access
EDMA	Enhanced Direct Memory Access
ENET	Ethernet
eMIOS	Enhanced Modular Input Output System
EULA	End User License Agreement
FMSTR	FreeMASTER
FOC	Field Oriented Control

Table 30. Acronyms*...continued*

Acronym	Definition
FreeRTOS	Free Real-Time Operating System
GDFLIB	General Digital Filter Library
GFLIB	General Function Library
GMCLIB	General Motor Control Library
GPT	General-Purpose Timer
gPTP	Generalized Precision Time Protocol
HSE-B	Hardware Security Engine - B Variant
ICU	Input Capture Unit
IDE	Integrated development environment
IIR	Infinite Impulse Response
IRQ	Interrupt Request
ISO	International Organization for Standardization
ISR	Interrupt Service Routine
IVT	Image Vector Table
KMU	Key Management Unit
LIN	Local Interconnect Network
LPI2C	Low Power Inter-Integrated Circuit
LPSPI	Low Power Serial Peripheral Interface
LPUART	Low Power Universal Asynchronous Receiver/Transmitter
MAC	Media Access Control
MCAL	Microcontroller Abstraction Layer
MCU	Microcontroller Unit
MCUboot	Microcontroller Bootloader
MISRA	Motor Industry Software Reliability Association
MSL	Moisture sensitivity level
MU	Message unit
OCU	Output Compare Unit
OTA	Over-the-Air
PDU	Protocol Data Unit
PIT	Periodic Interrupt Timer
PSA	Platform Security Architecture
PTP	Precision Time Protocol
PWM	Pulse Width Modulation
QoS	Quality of Service
RTCESL	Real-Time Control Embedded Software Library
RTD	Real-Time Drivers

Table 30. Acronyms*...continued*

Acronym	Definition
RTE	Runtime Environment
SAF	Safety Software Framework
SBAF	Secure Boot and Flashloader
SCST	Structural Core Self-Test
SDK	Software Development Kit
SIUL2	System Integration Unit Lite 2
SPI	Serial Peripheral Interface
SPSDK	Secure Provisioning SDK
STM	System Timer Module
SWC	Software Component
TCD	Transfer Control Descriptor
TRGMUX	Trigger Multiplexer
TSN	Time-Sensitive Networking
UART	Universal Asynchronous Receiver/Transmitter
Zephyr	Zephyr Real-Time Operating System

14 References

Table 31 lists the references used to supplement this document.

Table 31. Related Documentation

Document	Link
MLIB User's Guide (document CM7FMLIBUG)	CM7FMLIBUG
AMCLIB User's Guide (document CM7FAMCLIBUG)	CM7FAMCLIBUG
GFLIB User's Guide (document CM7FGFLIBUG)	CM7FGFLIBUG
GMCLIB User's Guide (document CM7FGMCLIBUG)	CM7FGMCLIBUG
GDFLIB User’s Guide (document CM7FGDFLIBUG)	CM7FGDFLIBUG
FreeMASTER Serial Communication Driver – User Guide (document FMSTRSCIDRVUG)	FMSTRSCIDRVUG

15 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

16 Revision history

Table 32 summarizes revisions to this document.

Table 32. Revision history

Document ID	Release date	Description
AN14792 v.1.1	07 November 2025	• Added: Table 1 "Pin-to-pin compatibility between S32K31x and MCX E31x devices" • Made some editorial changes
AN14792 v.1.0	20 August 2025	Initial public release

Legal information

Tables

Figures

Contents

Date of release: 7 November 2025
Document identifier: AN14792