AN14731

S32K14x to MCX E24x Migration Guidelines

Rev. 1.1 — 20 November 2025 **Application note**

Document information

Information	Content
Keywords	AN14731, MCX E24, S32K14x, migration, FRDM-MCXE247
Abstract	This application note outlines the key differences between the S32K14 and MCX E24 microcontroller families, with a focus on software-related aspects.

1 Introduction

This application note outlines the key differences between the S32K14 and MCX E24 microcontroller families, with a focus on software-related aspects. It provides essential guidance and considerations for migrating software applications from the S32K14 platform to the MCX E24 family.

2 Hardware migration

The migration from the S32K14x to the MCX E247 device is fully pin-to-pin compatible, ensuring a seamless transition from a hardware perspective.

Key hardware migration highlights:

- Pinout compatibility: The MCX E247 shares the package and pin configuration as the S32K1 family, enabling direct replacement without PCB layout changes.
- No schematic modifications: Existing schematics can be reused 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.

Table 1 provides a detailed mapping between MCX E24x and S32K14x devices for pin-to-pin compatibility.

Table 1. Pin-to-pin compatibility between S32K14x and MCX E24x devices

MCX P/N	S32K mapping
MCXE245VLF	FS32K144HAT0MLFR
	FS32K144HAT0MLFT
	FS32K144HAT0VLFR
	FS32K144HAT0VLFT
	FS32K144HBT0VLFR
	FS32K144HBT0VLFT
	FS32K144HFT0MLFR
	FS32K144HFT0VLFR
	FS32K144HFT0VLFT
	FS32K144HNT0MLFR
	FS32K144UAT0VLFT
	FS32K144UIT0VLFR
	FS32K144UIT0VLFT
MCXE245VLH	FS32K144HAT0CLHT
	FS32K144HAT0MLHR
	FS32K144HAT0MLHT
	FS32K144HAT0VLHR
	FS32K144HAT0VLHT
	FS32K144HBT0MLHR
	FS32K144HBT0VLHR
	FS32K144HFT0MLHR

Table 1. Pin-to-pin compatibility between S32K14x and MCX E24x devices*...continued*

MCX P/N	S32K mapping
	FS32K144HFT0MLHT
	FS32K144HFT0VLHR
	FS32K144HFT0VLHT
	FS32K144HNT0MLHR
	FS32K144HNT0VLHR
	FS32K144HPT0MLHR
	FS32K144HVT0VLHR
	FS32K144HVT0VLHT
	FS32K144MAT0VLHR
	FS32K144MFT0MLHR
	FS32K144MFT0VLHR
	FS32K144MNT0CLHT
	FS32K144MNT0MLHR
	FS32K144MNT0MLHT
	FS32K144MNT0VLHR
	FS32K144UAT0VLHR
	FS32K144UAT0VLHT
	FS32K144UFT0VLHR
	FS32K144UFT0VLHT
	FS32K144UIT0VLHR
	FS32K144UIT0VLHT
	FS32K144ULT0VLHT
MCXE245VLL	FS32K144HAT0CLLT
	FS32K144HAT0MLLR
	FS32K144HAT0MLLT
	FS32K144HAT0VLLR
	FS32K144HAT0VLLT
	FS32K144HFT0CLLT
	FS32K144HFT0MLLR
	FS32K144HFT0MLLT
	FS32K144HFT0VLLR
	FS32K144HFT0VLLT
	FS32K144HNT0CLLR
	FS32K144HNT0CLLT
	FS32K144HNT0VLLT
	FS32K144MFT0CLLT

Table 1. Pin-to-pin compatibility between S32K14x and MCX E24x devices*...continued*

MCX P/N	S32K mapping
	FS32K144MFT0VLLT
	FS32K144MNT0CLLT
	FS32K144MNT0MLLR
	FS32K144MNT0MLLT
	FS32K144MNT0VLLR
	FS32K144MNT0VLLT
	FS32K144UAT0VLLR
	FS32K144UAT0VLLT
	FS32K144UFT0VLLR
	FS32K144UFT0VLLT
	FS32K144ULT0VLLT
MCXE246VLH	FS32K146HAT0MLHR
	FS32K146HAT0MLHT
	FS32K146HAT0VLHR
	FS32K146HAT0VLHT
	FS32K146HBT0VLHR
	FS32K146HFT0MLHR
	FS32K146HFT0MLHT
	FS32K146HFT0VLHR
	FS32K146HFT0VLHT
	FS32K146HVT0VLHT
	FS32K146UAT0VLHR
	FS32K146UAT0VLHT
MCXE246VLL	FS32K146HAT0MLLR
	FS32K146HAT0MLLT
	FS32K146HAT0VLLR
	FS32K146HAT0VLLT
	FS32K146HFT0MLLR
	FS32K146HFT0MLLT
	FS32K146HFT0VLLR
	FS32K146HFT0VLLT
	FS32K146HNT0VLLT
	FS32K146UAT0VLLR
	FS32K146UAT0VLLT
	FS32K146UIT0VLLT
	FS32K146ULT0VLLT

Table 1. Pin-to-pin compatibility between S32K14x and MCX E24x devices*...continued*

MCX P/N	S32K mapping
MCXE246VLQ	FS32K146HAT0MLQR
	FS32K146HAT0MLQT
	FS32K146HAT0VLQR
	FS32K146HAT0VLQT
	FS32K146HFT0MLQR
	FS32K146HFT0MLQT
	FS32K146HFT0VLQR
	FS32K146HFT0VLQT
	FS32K146HNT0CLQR
	FS32K146HNT0CLQT
	FS32K146UAT0VLQR
	FS32K146UAT0VLQT
MCXE247VLL	FS32K148HAT0MLLR
	FS32K148HAT0MLLT
	FS32K148HAT0VLLR
	FS32K148HAT0VLLT
	FS32K148HFT0MLLR
	FS32K148HFT0MLLT
	FS32K148HFT0VLLR
	FS32K148HFT0VLLT
	FS32K148UAT0VLLR
	FS32K148UAT0VLLT
	FS32K148UBT0VLLR
	FS32K148UJT0VLLR
	FS32K148UJT0VLLT
	FS32K148UNT0VLLR
MCXE247VLQ	FS32K148HAT0MLQR
	FS32K148HAT0MLQT
	FS32K148HAT0VLQR
	FS32K148HAT0VLQT
	FS32K148HET0CLQR
	FS32K148HET0CLQT
	FS32K148HET0MLQR
	FS32K148HET0MLQT
	FS32K148HFT0MLQR
	FS32K148HFT0MLQT

Table 1. Pin-to-pin compatibility between S32K14x and MCX E24x devices*...continued*

MCX P/N	S32K mapping
	FS32K148HFT0VLQR
	FS32K148HFT0VLQT
	FS32K148HST0MLQT
	FS32K148UAT0VLQR
	FS32K148UAT0VLQT
	FS32K148UET0VLQT
	FS32K148UGT0VLQT
	FS32K148UIT0VLQT
	FS32K148UJT0VLQR
	FS32K148UJT0VLQT

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 S32K14 family typically relies on an Automotive Open System Architecture (AUTOSAR)-compliant Real-Time Drivers (RTD) framework, designed for automotive safety and modularity. In contrast, the MCUXpresso SDK supports the MCX E24 family, which targets industrial applications and offers a more flexible and lightweight driver model.

4 Software architecture comparison

The 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 (MCAL, complex drivers, and service layer) and are tightly integrated 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.

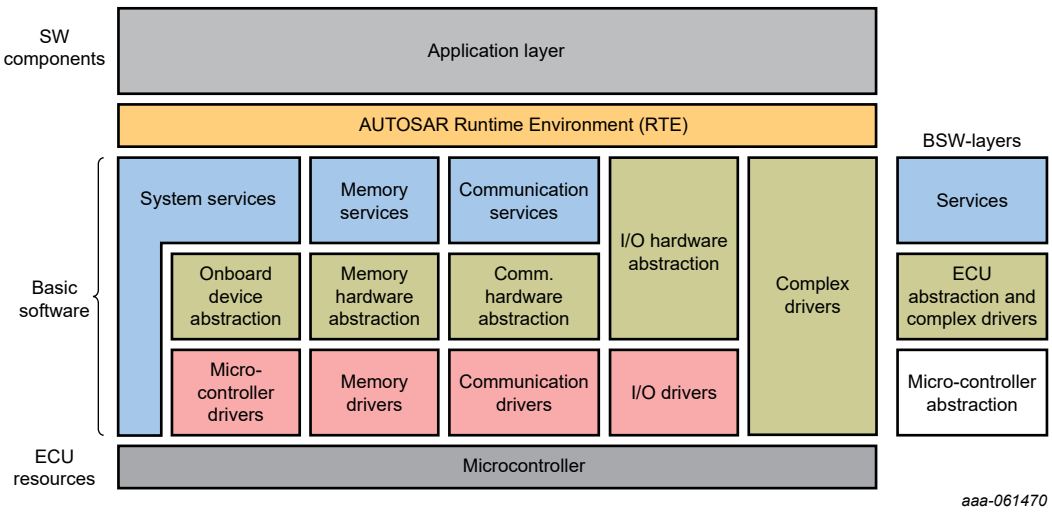

Figure 1. AUTOSAR architecture

In contrast, the MCUXpresso SDK follows a more lightweight, monolithic approach. It provides peripheral drivers as standalone C modules, typically without a separation between configuration and implementation layers. This design 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. It is also faster to adapt for industrial and general-purpose embedded applications where safety certification or standardized software layering is not mandatory.

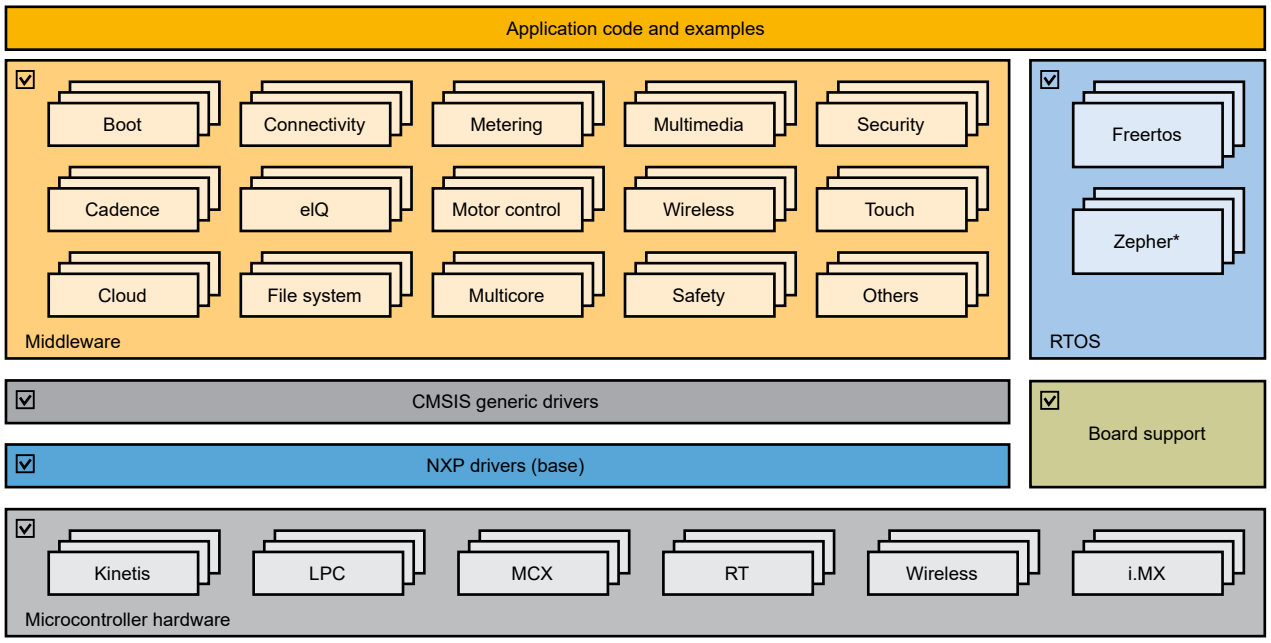

Figure 2. MCUXpresso architecture

Table 2. SDKs standards

Aspect	AUTOSAR RTD	MCUXpresso SDK
Architecture	Three layers (SWC, 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	Really high (MCAL, ECU Abstraction)	Low to moderate (faster execution, better real-time performance)
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

Table 3 compares the security features offered by the AUTOSAR RTD framework and the MCUXpresso SDK, highlighting differences in modules, key management, encryption, diagnostics, and secure boot capabilities.

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)	Not integrated, depends on MCU and application
Authentication and encryption	Standardized APIs (CryptoIf + CryIf backends)	To be implemented via mbedTLS or NXP SDK (CSEc, PSA crypto)
Security diagnosis	Diagnostic Event Manager (DEM)	Not present, to be implemented manually
Secure boot/firmware update	Support possible via a secure bootloader	Secure bootloader must be implemented with boot ROM or flashloader

5.1 Secure installation and manufacturing support

One of the key enhancements brought by the MCUXpresso SDK ecosystem for the MCX series, including the MCX E24x, is the support for a secure manufacturing process out-of-the-box. The SDK provides a preflashed secure flashloader on the device, enabling firmware provisioning and updates over UART or SPI interfaces. This feature allows OEMs to implement secure factory programming using standard communication ports without relying on a debug interface.

In contrast, the S32K14x platform requires a secured production environment, as firmware installation must occur over the debug port (for example, SWD/JTAG). It also does not include an equivalent preinstalled secure flashloader. This constraint limits flexibility and increases complexity for secure line programming.

This change significantly simplifies the secure manufacturing flow and reduces the need for debug access in production.

6 AUTOSAR RTD API -> MCUXpresso API

This section outlines differences between the RTD and MCUXpresso SDK APIs for each IP of the MCX E24x family.

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.

The following examples illustrate how ADC reading is implemented in both AUTOSAR RTD and MCUXpresso SDK, highlighting the differences in their approach.

Example 1: Typical reading RTD

```
Adc_StartGroupConversion(ADC_GROUP_0);
while (Adc_GetGroupStatus(ADC_GROUP_0) != ADC_STREAM_COMPLETED) {}
```

```
Adc_ReadGroup(ADC_GROUP_0, &amp;amp;dataBuffer);
```

Example 2: Typical reading MCUXpresso

```
adc12_channel_config_t channelConfig = {
    .channelNumber = 0U,
    .enableInterruptOnConversionCompleted = false
};
ADC12_DoAutoCalibration(ADC12);  // Optional
ADC12_SetChannelConfig(ADC12, 0U, &channelConfig);
while (!ADC12_GetChannelStatusFlags(ADC12, 0U)) {}
uint16_t result = ADC12_GetChannelConversionValue(ADC12, 0U);
```

Table 4 summarizes the equivalent API functions between AUTOSAR RTD and MCUXpresso SDK for ADC operations:

Table 4. ADC APIs comparison

AUTOSAR	MCUXpresso
Adc_GroupType	channelNumber in adc12_channel_config_t
Adc_Init (Adc_ConfigType*)	ADC12_Init (ADC_Type*, adc12_config_t*)
Adc_ReadGroup (Adc_GroupType, Adc_ValueGroup Type*)	ADC12_GetChannelConversionValue (ADC_Type*, channel Group)
Adc_StartGroupConversion (Adc_GroupType)	ADC12_SetChannelConfig (ADC_Type*, channelGroup, adc12_channel_config_t*)

6.2 Comparator (CMP)

This section addresses the migration of the CMP IP from AUTOSAR RTD to the MCUXpresso SDK. The CMP module provides analog comparator functionality essential for signal monitoring and event triggering in embedded applications. 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.

The following examples illustrate how CMP initialization and interrupt enabling are implemented in both AUTOSAR RTD and MCUXpresso SDK, highlighting the differences in their approach.

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

```
acmp_config_t config;
acmp_channel_config_t channel_config ;
ACMP_GetDefaultConfig(&config);
ACMP_Init(ACMP0, &config);
ACMP_SetChannelConfig(ACMP0, &channel_config);
ACMP_EnableInterrupts(ACMP0, kACMP_OutputRisingInterruptEnable |
 kACMP_OutputFallingInterruptEnable);
EnableIRQ(ACMP0_IRQn);
ACMP_Enable(ACMP0, true);
```

Table 5 summarizes the equivalent API functions between AUTOSAR RTD and MCUXpresso SDK for CMP operations:

Table 5. CMP APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Cmp_Ip_Init (instance, Cmp_Ip_ConfigType*)	ACMP_Init (CMP_Type*, acmp_config_t*)
Config	Cmp_Ip_ConfigType structure (generated by .arxml)	ACMP_SetChannelConfig (CMP_Type*, acmp_channel_config_t*)
Irq	Cmp_Ip_EnableInterrupt (instance)	ACMP_EnableInterrupts (CMP_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. Also, error handling such as timeouts and NACKs must be managed explicitly in the MCUXpresso environment.

The following examples illustrate how data transmission using polling is implemented in both AUTOSAR RTD and MCUXpresso SDK, highlighting the differences in their approach.

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 equivalent API functions between AUTOSAR RTD and MCUXpresso SDK for LPI2C operations:

Table 6. LPI2C APIs comparison

Functions	AUTOSAR	MCUXpresso
Controller initialization	Lpi2c_Ip_MasterInit (Channel, I2c_HwUnitConfigType*) through I2c_Init (I2c_ConfigType*)	LPI2C_MasterInit (LPI2C_Type*, lpi2c_master_config_t*, srcClkHz) + LPI2C_MasterGetDefaultConfig (lpi2c_master_config_t_Type*) 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*, I2c_HwUnitConfigType*)	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*, srcClkHz) + LPI2C_SlaveGetDefaultConfig (lpi2c_slave_config_t*)

Table 6. LPI2C APIs comparison*...continued*

Functions	AUTOSAR	MCUXpresso
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.

The following examples illustrate how synchronous transmission and interrupt-based reception are implemented in both AUTOSAR RTD and MCUXpresso SDK, highlighting the differences in their approach.

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 equivalent API functions between AUTOSAR RTD and MCUXpresso SDK for LPUART operations:

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)

Table 7. LPUART APIs comparison*...continued*

Functions	AUTOSAR	MCUXpresso
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.

The following examples illustrate how full-duplex asynchronous transmission is implemented in both AUTOSAR RTD and MCUXpresso SDK, highlighting the differences in their approach.

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 equivalent API functions between AUTOSAR RTD and MCUXpresso SDK for LPSPI operations:

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)

Table 8. LPSPI APIs comparison*...continued*

Functions	AUTOSAR	MCUXpresso
Transfer mode configuration	Lpspi_Ip_UpdateTransferMode (instance, Lpspi_Ip_ModeType)	No equivalent function and mode is linked to the transmit function used

6.6 General-Purpose Input/Output (GPIO)

In AUTOSAR RTD, GPIO control is typically split between the Port module (for pin configuration) and the Dio module (for reading/writing pin states). However, in some RTD setups like on S32K1, Dio can be replaced by IP-level drivers, such as `Gpio_Dio_Ip_WritePin()`, which directly manage pin states without going through the full AUTOSAR abstraction.

In contrast, the MCUXpresso SDK uses a unified GPIO driver (for example, `fsl_gpio`) where configuration and I/O are handled more directly, with less abstraction. This approach leads to simpler code but reduced modularity compared to the AUTOSAR approach.

Table 9 summarize the equivalent API functions between AUTOSAR RTD and MCUXpresso SDK for GPIO operations:

Table 9. GPIO APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Port_Ci_Port_Ip_Init (uint32_t, Port_Ci_Port_Ip_PinSettingsConfig)	GPIO_PinInit (GPIO_Type*, uint32_t, gpio_pin_config_t*)
Write	Gpio_Dio_Ip_WritePin (GPIO_Type*, Gpio_Dio_Ip_PinsChannelType, Gpio_Dio_Ip_PinsLevelType)	GPIO_PinWrite (GPIO_Type*, uint32_t, uint8_t)
Read	Gpio_Dio_Ip_ReadPin (GPIO_Type*, Gpio_Dio_Ip_PinsChannelType)	GPIO_PinRead (GPIO_Type*, uint32_t)

6.7 FlexTimer Module (FTM)

The FTM is a multifunction peripheral commonly used in NXP microcontrollers for applications such as timers, input capture, signal measurement, PWM, and more.

In the AUTOSAR RTD environment, access to the FTM is encapsulated in abstract software modules, such as PWM, ICU, or OCU, depending on the usage mode. These modules rely on low-level IP drivers, such as `Ftm_<module_name>_Ip`, which manage the actual hardware configuration. This architecture strictly adheres to the layers defined by AUTOSAR (MCAL -> BSW -> SWC), and interaction with the hardware is primarily configured via tools, such as EB tresos. The goal is to ensure portability, functional safety, and compliance with industry standards.

Conversely, in the MCUXpresso SDK, the FTM module is driven directly via the fsl_ftm driver, which provides a functional interface in C covering all the functionalities: counter modes, captures, comparisons, interrupts, synchronization, and so on. The developers manually configure the registers and functions according to their needs, with direct, flexible, and nonstandardized access, well suited to industrial, or real-time embedded systems without AUTOSAR constraints.

Table 10 summarizes the equivalent API functions between AUTOSAR RTD and MCUXpresso SDK for FTM operations:

Table 10. FTM APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	No equivalent function. Initialization is done by abstracting modules (PWM, ICU…)	FTM_Init (FTM_Type*, ftm_config_t*) + FTM_Get DefaultConfig (ftm_config_t*)
PWM Initialization	Ftm_Pwm_Ip_Init (instance, Ftm_Pwm_Ip_User CfgType*)	FTM_SetupPwm (FTM_Type*, ftm_chnl_pwm_ signal_param_t*, nbChan, ftm_pwd_mode_t, pwm FreqHz, srcClkHz)
PWM Period	Ftm_Pwm_Ip_UpdatePwmPeriod (instance, NewVal, isSoftTrig)	FTM_SetTimerPeriod (FTM_Type*, ticks) -> be careful when using it, call FTM_SetupPwm() after calling this function
PWM Channel	Ftm_Pwm_Ip_UpdatePwmChannel (instance, channel, FirstEdge, SecondEdge, isSoftTrig)	FTM_UpdatePwmDutycycle (FTM_Type*, ftm_ chnl_t, ftm_pwm_mode_t, NewDutyCyclePercent) FTM_SetSoftwareTrigger (FTM_Type*, isEnable)
OCU Initialization	Ftm_Ocu_Ip_Init (Ftm_Ocu_Ip_ModuleConfig Type*)	FTM_SetupOutputCompare (FTM_Type*, ftm_ chnl_t, ftm_output_compare_mode_t, compareVal)
OCU Start	Ftm_Ocu_Ip_StartChannel (instance, ChNum)	FTM_StartTimer (FTM_Type*, ftm_clock_source_t)
OCU Irq	Ftm_Ocu_Ip_EnableNotification (instance, ChNum)	FTM_EnableInterrupts (FTM_Type*, mask) + EnableIRQ (IRQn_Type)
GPT functions	Ftm_Gpt_Ip_Init (instance, Ftm_Gpt_Ip_Instance ConfigType*) Ftm_Gpt_Ip_InitChannel (instance, Ftm_Gpt_Ip_ ChannelConfigType*) Ftm_Gpt_Ip_StartTimer(instance) Ftm_Gpt_Ip_EnableChannelInterrupt (instance, channel)	Unlike the AUTOSAR RTD where GPT functionalities are implemented on top of FTM, the MCUXpresso SDK does not provide a GPT abstraction layer based on FTM.
ICU Initialization	Ftm_Icu_Ip_Init (instance, Ftm_Icu_Ip_Config Type*)	FTM_SetupInputCapture (FTM_Type*, ftm_chnl_t, ftm_input_capture_edge_t, filterVal)
ICU Irq	Ftm_Icu_Ip_EnableNotification (instance, hw Channel)	FTM_EnableInterrupts (FTM_Type*, mask) + EnableIRQ (IRQn_Type)
QDEC Initialization	Ftm_Qdec_Ip_Init (instance, Ftm_Qdec_Ip_Config Type*) + Ftm_Qdec_Ip_SetMode (instance, Ftm_ ModeType)	FTM_SetupQuadDecode (FTM_Type*, ftm_ phase_params_t*, ftm_phase_params_t*, ftm_ quad_decode_mode_t)

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.

The following examples illustrate how LIN frame transfer is implemented using LPUART in both AUTOSAR RTD and MCUXpresso SDK, highlighting the differences in their approach.

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 Flexible Input/Output (FLEXIO)

The FLEXIO module is a highly configurable peripheral that enables flexible and efficient implementation of various serial communication protocols, such as I2C, SPI, and UART. By using programmable shifters, timers, and pins, FLEXIO allows the emulation of these interfaces even when dedicated hardware modules are unavailable or when more 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 via FLEXIO:

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

6.9.1 Inter-Integrated Circuit (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.

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

The following examples illustrate how I2C controller read operations using FLEXIO are implemented in both AUTOSAR RTD and MCUXpresso SDK, highlighting the differences in their approach.

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 Serial Peripheral Interface (SPI)

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

The following examples illustrate how SPI controller read operations using FLEXIO are implemented in both AUTOSAR RTD and MCUXpresso SDK, highlighting the differences in their approach.

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 Universal Asynchronous Receiver/Transmitter (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.

The following examples illustrate how UART controller read operations using FLEXIO are implemented in both AUTOSAR RTD and MCUXpresso SDK, highlighting the differences in their approach.

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 equivalent API functions between AUTOSAR RTD and MCUXpresso SDK for FLEXIO operations:

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*, I2c_HwUnitConfigType*) 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*)
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 summarizes the equivalent API functions between AUTOSAR RTD and MCUXpresso SDK for watchdog operations:

Table 12. Watchdog APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Wdog_Ip_Init (instance, Wdog_Ip_ConfigType*)	WDOG32_Init (WDOG_Type*, wdog32_config_t*)
Refresh counter	Wdog_Ip_Service (instance)	WDOG32_Refresh (WDOG_Type*)

6.11 Low-Power Interrupt Timer (LPIT)

In the MCUXpresso SDK, LPIT is configured using a unified API regardless of the use case, whereas in the RTD, LPIT is split into different drivers such as `Lpit_Gpt_Ip` and `Lpit_Icu_Ip` depending on whether it's used as a general-purpose timer or for input capture functionality.

Table 13 tables summarize the equivalent API functions between AUTOSAR RTD and MCUXpresso SDK for LPIT operations:

Table 13. LPIT APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Lpit_Gpt_Ip_Init (instance, Lpit_Gpt_Ip_Instance ConfigType*) / Lpit_Icu_Ip_Init (instance, Lpit_Icu_Ip_Config Type*)	LPIT_Init (LPIT_Type*, lpit_config_t*)
Configure channel	Lpit_Gpt_Ip_InitChannel (instance, Lpit_Gpt_Ip_ ChannelConfigType*)	LPIT_SetupChannel (LPIT_Type*, lpit_chnl_t, lpit_ chnl_params_t*)
Configure Irq	Lpit_Gpt_Ip_EnableChInterrupt (instance, hw Channel) / Lpit_Icu_Ip_EnableInterrupt (instance, hwChannel)	LPIT_EnableInterrupts (LPIT_Type*, mask) + LPIT_SetTimerPeriod (LPIT_Type*, lpit_chnl_t, ticks)
Start	Lpit_Gpt_Ip_StartTimer() / Lpit_Icu_Ip_StartTimestamp (instance, hwChannel, uint32_t*, buffSize, notifyInterval)	LPIT_StartTimer (LPIT_Type*, lpit_chnl_t)

6.12 Trigger Multiplexer (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 summarizes the available TRGMUX API functions in MCUXpresso SDK:

Table 14. TRGMUX APIs comparison

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

Table 14. TRGMUX APIs comparison*...continued*

Functions	MCUXpresso
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 MCUXpresso SDK provides a set of dedicated APIs for configuring and using EDMA transfers, the RTD abstracts these operations, and DMA usage is often implicitly managed within peripheral drivers like LPUART, LPI2C, or LPSPI through configuration structures.

Also, 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 via the generator tools and not exposed through standalone APIs.

Table 15 summarizes the available EDMA API functions in MCUXpresso SDK:

Table 15. EDMA APIs comparison

Functions	MCUXpresso
Initialization	EDMA_Init (DMA_Type*, edma_config_t*)
Handle initialization	EDMA_CreateHandle (edma_handle_t*, DMA_Type*, channel)
Configuration	EDMA_SetTransferConfig (DMA_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 (DMA_Type*, channel, mask)
Submit transfer request	EDMA_SubmitTransfer (edma_handle_t*, edma_transfer_config_t*)
Start transfer	EDMA_StartTransfer (edma_handle_t*)
Status	EDMA_GetChannelStatusFlags (DMA_Type*, channel)

6.14 Flexible Controller Area Network (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 summarizes the equivalent API functions between AUTOSAR RTD and MCUXpresso SDK for FlexCAN operations:

Table 16. FLEXCAN APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Can_43_FLEXCAN_Init (Can_43_FLEXCAN_ConfigType*)	FLEXCAN_Init (Can_Type*, flexcan_config_t*, srcClkHz)
Configure controller mode	Can_43_FLEXCAN_SetControllerMode (controller, Can_ControllerStateType)	FLEXCAN_EnterFreezeMode (CAN_Type*)/ FLEXCAN_ExitFreezeMode (CAN_Type*)

Table 16. FLEXCAN APIs comparison*...continued*

Functions	AUTOSAR	MCUXpresso
Asynchronous writing	Can_43_FLEXCAN_Write (Can_HwHandleType, Can_PduType*)	FLEXCAN_TransferSendNonBlocking (CAN_ Type*, flexcan_handle_t*, flexcan_mb_transfer_t*)

6.15 Programmable Delay Block (PDB)

In the RTD, the PDB module is used in coordination with the ADC via the `Pdb_Adc_Ip_*` APIs. It is tightly coupled with ADC triggering use cases and abstracts much of the low-level configuration.

In contrast, the MCUXpresso SDK offers a more direct and hardware-centric API under the *fsl_pdb.h* driver. It exposes low-level configuration options through `PDB_Init()` and related functions, providing more flexibility but requiring manual setup for trigger sources, delays, and pretriggers.

Table 17 summarizes the equivalent API functions between AUTOSAR RTD and MCUXpresso SDK for PDB operations:

Table 17. PDB APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Pdb_Adc_Ip_Init (instance, Pdb_Adc_Ip_Config Type*)	PDB_GetDefaultConfig (pdb_config_t*) + PDB_Init (PDB_Type*, pdb_config_t*)
Trigger ADC conversion (Software)	Pdb_Adc_Ip_SwTrigger (instance)	PDB_DoSoftwareTrigger (PDB_Type*)

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 summarize the equivalent API functions between AUTOSAR RTD and MCUXpresso SDK for ENET operations:

Table 18. ENET APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Eth_43_ENET_Init (Eth_43_ENET_ConfigType*)	ENET_GetDefaultConfig (enet_config_t*) + ENET_Init (ENET_Type*, enet_handle_t*, enet_ config_t*, enet_buffer_config_t*, uint8_t*, SrcClk Hz)
Set ENET mode	Eth_43_ENET_SetControllerMode (CtrlIdx, Eth_ ModeType)	no equivalent function (integrated in the enet_ config_t structure)
Get MAC address	Eth_43_ENET_GetPhysAddr (CtrlIdx, uint8_t*)	ENET_GetMacAddr (ENET_Type*, uint8_t*)
Asynchronous writing	Eth_43_ENET_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_ENET_TxConfirmation(CtrlIdx)	ENET_TransmitIRQHandler (ENET_Type*, enet_ handle_t*, uint32_t)

Table 18. ENET APIs comparison*...continued*

Functions	AUTOSAR	MCUXpresso
Asynchronous reading	Eth_43_ENET_Receive(CtrlIdx, FifoIdx, Eth_Rx StatusType*)	ENET_ActiveRead (ENET_Type*) + ENET_Read Frame (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, PTP support can be abstracted and managed through dedicated service layers or integration with the time synchronization managers. In contrast, the MCUXpresso SDK exposes lower-level control, and available functionality can differ significantly depending on the hardware platform.

For the MCX E247 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.

Key points for migration:

- AUTOSAR RTD implementations often abstract PTP support or require manual timestamp handling via low-level driver extensions.
- In MCUXpresso, PTP is natively supported through functions such as:
  - `ENET_Ptp1588Configure` (`ENET_Type*`, `enet_handle_t*`, `enet_ptp_config_t*`)
  - `ENET_Ptp1588GetTimer` (`ENET_Type*`, `enet_handle_t*`, `enet_ptp_time_t*`)
- Hardware timestamping is enabled and configured via 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.

6.17 Cyclic Redundancy Check (CRC)

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

In AUTOSAR RTD, the CRC module is highly configurable and 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 via 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.

The following examples illustrate how CRC calculations are implemented using RTD and MCUXpresso SDK APIs.

Example 19: CRC8 calculation 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);
```

6.18 EdgeLock Accelerator (CSEC)

The Cryptographic Services Engine Controller (CSEC) is a hardware security module. It provides key cryptographic capabilities, including AES-128 encryption, CMAC generation, secure key storage, and secure boot support.

In the AUTOSAR RTD environment, cryptographic operations are implemented using CSEC-specific APIs, which offer hardware-accelerated security features essential for automotive and safety-critical applications.

Since MCX E24x is equipped with a hardware CSEC module, it is possible to use the native CSEC API provided by the MCUXpresso SDK. However, the preferred method for accessing cryptographic services is through the NXP Crypto Library (CryptoLib), which offers higher abstraction, better integration, and enhanced portability across NXP platforms.

This section outlines how to migrate typical CSEC use cases — such as MAC generation, AES encryption, and random number generation — to their equivalents in the MCUXpresso SDK, detailing API-level changes and any functional differences.

Table 19. CSEC APIs comparison

Functions	AUTOSAR	MCUXpresso
Initialization	Csec_Ip_Init (Csec_Ip_StateType*)	CSEC_DRV_Init (csec_state_t*)
AES Encryption/ Decryption in ECB Mode	Csec_Ip_EncryptEcb (Csec_Ip_ReqType*, Csec_Ip_KeyIdType, uint8_t*, length, uint8_t*) / Csec_Ip_DecryptEcb (Csec_Ip_ReqType*, Csec_Ip_KeyIdType, uint8_t*, length, uint8_t*)	CSEC_DRV_EncryptECB (csec_key_id_t, uint8_t*, length, uint8_t*, timeout) / CSEC_DRV_DecryptECB (csec_key_id_t, uint8_t*, length, uint8_t*, timeout)
AES Encryption/ Decryption in CBC Mode	Csec_Ip_EncryptCbc (Csec_Ip_ReqType*, Csec_Ip_KeyIdType, uint8_t*, length, uint8_t*, uint8_t*) / Csec_Ip_DecryptCbc (Csec_Ip_ReqType*, Csec_Ip_KeyIdType, uint8_t*, length, uint8_t*, uint8_t*)	CSEC_DRV_EncryptCBC (csec_key_id_t, uint8_t*, length, uint8_t*, uint8_t*, timeout) / CSEC_DRV_DecryptCBC (csec_key_id_t, uint8_t*, length, uint8_t*, uint8_t*, timeout)
Calculating a Message Authentication Code (MAC)	Csec_Ip_GenerateMac (Csec_Ip_ReqType*, Csec_Ip_KeyIdType, uint8_t*, length, uint8_t*)	CSEC_DRV_GenerateMAC (csec_key_id_t, uint8_t*, msgLen, uint8_t*, timeout)
Refresh RNG seed	Csec_Ip_ExtendSeed (uint8_t*)	CSEC_DRV_ExtendSeed (uint8_t*)

7 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 replaces the legacy AMMCLib, which was previously used in automotive

applications (for example, with S32K14x). Developers migrating to MCX E24x must transition to RTCESL to benefit from improved integration and support.

- Lightweight IP (LwIP): compact TCP/IP stack optimized for embedded systems; supports IPv4/IPv6, sockets, and various network protocols; integrated with ENET driver of the MCUXpresso SDK for Ethernet connectivity.
- FreeMASTER: real-time data visualization and control tool, enabling runtime monitoring, variable tuning, and oscilloscope-style data logging via UART, CAN or USB.
- MbedTLS 3.x: modern, lightweight cryptographic library providing TLS/SSL protocols and a wide range of cryptographic primitives (AES, SHA, RSA, and so on), suitable for embedded secure communication.
- PSA crypto driver: Platform security architecture-compliant interface layer offering a more flexible alternative to traditional HIS SHE-based security mechanisms in AUTOSAR, enabling secure key management and hardware-accelerated cryptographic operations on supported MCUs.

7.1 Availability in MCUXpresso SDK

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

8 RTOS migration considerations

This section provides information regarding the migration of RTOS, particularly between OsIf and FreeRTOS/ Zephyr.

8.1 From OsIf (RTD) to FreeRTOS (MCUXpresso)

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.

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

8.1.1 Core differences

Table 20 compares core functional differences between AUTOSAR RTD with OsIf and MCUXpresso SDK with FreeRTOS, highlighting variations in API usage and abstraction levels.

Table 20. OsIf and FreeRTOS comparison

Feature	AUTOSAR RTD + OsIf	MCUXpresso SDK + FreeRTOS
Delays	OsIf_GetCounter (OsIf_CounterType) + Os If_MicroToTicks (micros, OsIf_CounterType)	vTaskDelay (TickType_t)

Table 20. OsIf and FreeRTOS comparison...*continued*

Feature	AUTOSAR RTD + OsIf	MCUXpresso SDK + FreeRTOS
	+ OsIf_GetElapsed (uint32_t*, OsIf_Counter Type)	
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

8.1.2 Additional considerations

In addition to the core differences, the following aspects must be reviewed during migration:

- 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. To improve robustness, always handle these return values.

8.2 From OsIf (RTD) to Zephyr operating system

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 more features targeting IoT and modern embedded use cases.

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

Following are the migration considerations when moving from OsIf (AUTOSAR RTD) to Zephyr operating system:

- 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.

8.2.1 Core differences

Table 21 highlights core API differences between AUTOSAR RTD (OsIf) and Zephyr OS.

Table 21. OsIf and Zephyr comparison

Feature	AUTOSAR RTD + OsIf	MCUXpresso SDK + Zephyr
Delays	OsIf_GetCounter (OsIf_CounterType) + Os If_MicroToTicks (micros, OsIf_CounterType) + OsIf_GetElapsed (uint32_t*, OsIf_Counter Type)	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

8.3 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`.

9 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, 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.

This chapter covers both secure and nonsecure firmware flashing tools, supported IDEs and debuggers, firmware formats, and the bootloader infrastructure commonly used with NXP MCX microcontrollers. This chapter also outlines typical 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 are developing in a BareMetal environment, or working with RTOSs like FreeRTOS or Zephyr, this chapter helps ensure a smooth and efficient transition.

9.1 Software accessibility

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

9.1.1 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: To generate a preconfigured SDK, select target devices, middleware components, and example projects.
- 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.

9.1.2 GitHub repositories

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

- MCUXpresso SDK: mcuxsdk-manifests - contains core drivers, middleware, and utilities.
- Board Support Packages (BSPs): Includes device-specific configuration and startup code.
- Example projects: Reference implementations showcasing peripheral usage and RTOS integration.
- Issue tracking and contributions: Developers can report issues, request features, and contribute fixes via pull requests.

You can fetch this repo using west CLI:

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

9.1.3 Documentation and support

For documentation and developer support, consider the following resources:

- 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.

9.2 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.

9.2.1 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.

9.2.2 Patch workflow in MCUXpresso SDK-based projects

Compared to the AUTOSAR RTD environment (often centrally maintained and updated via 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 …).
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.*

9.3 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 22 is an overview of key support and knowledge-sharing components available when working with NXP MCX-class devices and the MCUXpresso SDK.

Table 22. Key support and knowledge-sharing components

Feature	AUTOSAR RTD	MCUXpresso SDK
Official Support Contracts	✓ Yes	✗ No (only community or via NXP support portal)
Structured Knowledge Base	✓ Yes (Centralized)	✗ No (Distributed: forums, GitHub, documentation)
Training Programs	✓ Yes (Vendor-defined)	✓ Yes (Public and on-demand)

Table 22. Key support and knowledge-sharing components*...continued*

Feature	AUTOSAR RTD	MCUXpresso SDK
Migration Assistance	✓ Yes (using 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.*

9.4 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 23 is an overview of commonly used IDEs and their compatibility with the MCUXpresso SDK:

Table 23. 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.

Following are the IDE migration considerations:

- 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 via the MCUXpresso SDK Builder or manually integrated.

9.5 Flash and debug tools

Flashing firmware to the MCX series microcontrollers, such as the MCX E247, can be achieved using several tools and workflows depending on the development stage (prototyping vs production), bootloader configuration (secure vs nonsecure), and toolchain preferences.

This section summarizes the available options and their use cases.

9.5.1 Flashing using debug probe

Table 24 lists commonly supported debug probes and their associated tools for flashing and debugging in MCUXpresso SDK projects.

Table 24. 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.

Following are the typical use cases:

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

9.6 Secure installer for unsecure manufacturing lines

The diagram illustrates the difference between S32K1 and MCX E24 flash layouts in the context of OEM factory provisioning using the `Flash_loader`. On S32K1, devices are delivered with blank flash, whereas MCX E24x devices come preloaded with a one-time `Flash_loader` and `Flash_loader_loader` image in flash memory (starting at address `0x0000_0000`), as provided by NXP SCE. Upon boot, the loader is copied from flash to RAM and executed from there. This loader interacts with a host PC (via UART or SPI) using tools like `blhost`, allowing flash programming and CSEc key provisioning. The setup also requires a boot hold from the factory for testing and IFR locking. The `Flash_loader` execution is prioritized to ensure proper initialization and provisioning before any application runs.

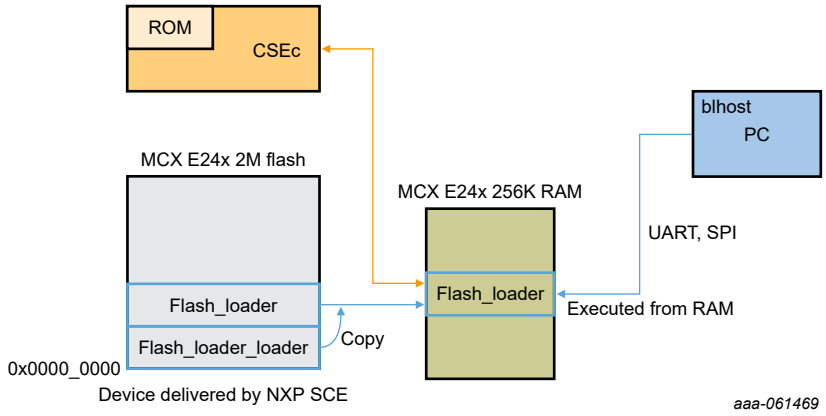

Figure 3. Flashloader

9.7 MCUBoot support

On the MCX E247 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.

9.7.1 Key features

The following are the key features provided by MCUboot on the MCX E247 platform:

- 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 iif there is a boot failure (not default behaviour, 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)

9.7.2 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 (via imgtool or the West toolchain of the Zephyr).
- 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.

10 Licensing

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

10.1 Licensed software components

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

Table 25. License migrations

Component/Tool	Vendor	License type	License required	Migration required
AUTOSAR runtime drivers	NXP	Commercial (Flexera)	✓ Yes	✓ Yes
Cryptography stack (CSEC)	Third party	Flexera Node Locked	✓ Yes	✓ Yes
Debugger (for example, Lauterbach)	Lauterbach	Dongle	✓ Yes	✗ No

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

10.2 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.

10.3 MCUXpresso SDK licensing

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

Table 26. 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
Cryptography stack (CSEC)	Permissive (BSD-style)

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.*

11 Acronyms

Table 27 lists the acronyms used in this document.

Table 27. Acronyms

Acronym	Description
ADC	Analog-to-Digital Converter
AUTOSAR	Automotive Open System Architecture
CAN	Controller Area Network
CRC	Cyclic Redundancy Check
CryptoLib	Crypto Library
CSEC	Cryptographic Services Engine Controller
DEM	Diagnostic Event Manager
EDMA	Enhanced Direct Memory Access
ENET	Ethernet

Table 27. Acronyms...*continued*

Acronym	Description
FlexCAN	Flexible Controller Area Network
FLEXIO	Flexible Input/Output
FTM	FlexTimer Module
GPIO	General-Purpose Input/Output
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
LIN	Local Interconnect Network
LPI2C	Low-Power Inter-Integrated Circuit
LPIT	Low-Power Interrupt Timer
LPSPI	Low-Power Serial Peripheral Interface
LPUART	Low-Power Universal Asynchronous Receiver/Transmitter
LwIP	Lightweight IP
OsIf	Operating System Interface
PDB	Programmable Delay Block
PTP	Precision Time Protocol
RTCESL	Real time Control Embedded Software Library
RTOS	Real-Time Operating System
RTD	Real-Time Drivers
SPI	Serial Peripheral Interface
TRGMUX	Trigger Multiplexer
UART	Universal Asynchronous Receiver/Transmitter

12 Note about the source code in the document

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

Copyright 2025 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials must be provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

13 Revision history

Table 28 summarizes the revisions to this document.

Table 28. Revision history

Document ID	Release date	Description
AN14731 v.1.1	20 November 2025	Made the following updates: • Added a table titled, 'Pin-to-Pin compatibility between S32K14x and MCX E24x devices' in the section Section 2 • Added Acronyms table • Made editorial changes
AN14731 v.1.0	10 July 2025	Initial public release

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents