NXP Semiconductors
Application Notes

Document Number: AN13435
Rev. 0, 10/2021

SDK/MCAL to Real Time Drivers

Migration Guide SDK/MCAL to Real Time Drivers

by: NXP Semiconductors

1. Introduction

Real Time Drivers (RTD) are offered as production qualified software abstraction of complex hardware features to be used in AUTOSAR and Non-AUTOSAR applications.

The RTD offer standardized API (AUTOSAR and AUTOSAR extensions) available across products and dedicated hardware specific interfaces, with ISO26262 compliance for API.

The RTD provide multiple software features as extensions to AUTOSAR standard (to expose specialized hardware features) and provide full coverage of hardware features and hardware peripherals. RTD have integrated driver examples with default configurations. The examples are offered for both CT and EB Tresos configuration tools. This brings the approaches from SDK and MCAL. The RTD are composed from AUTOSAR MCAL drivers and a set of complex device drivers.

Mapping of the RTD on the peripherals is shown in the platform specific appendix.

The RTD integrate use cases from AUTOSAR and Non-AUTOSAR environment, therefore from customer perspective existing functionalities from SDK and MCAL experience is maintained. Each extension from the standard package can be enabled or disabled.

Contents

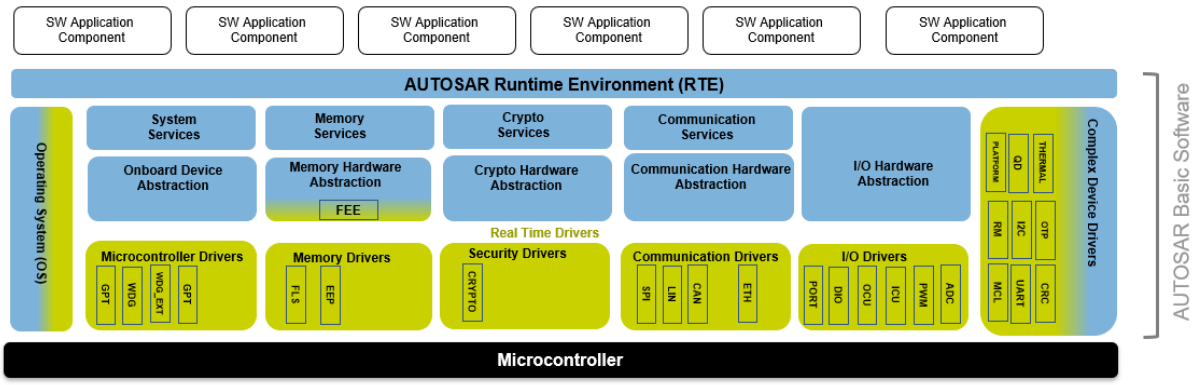

Figure 1. **Real Time Drivers overview in AUTOSAR Stack**

From MCAL customer perspective, the RTD extend the standard functionalities as defined by the standard with full coverage of hardware functionalities. In AUTOSAR applications the standardized interface is available and it is recommended to be used in order to maintain portability across applications.

From SDK customer perspective, the RTD extend the standardized functionalities, along with adding multicore support, running in user mode support, memory mapping of code and data to specific memory sections. For Non-AUTOSAR applications, both interfaces (standardized and IP) are available and can be used. It depends on the application type and project constraints.

The RTD can be configured with any AUTOSAR configurator and NXP S32Design Studio. From the configuration point of view, the NXP S32 Design Studio offers support of configuring both the driver and peripheral layer independent of the driver. The usage of standardized interface and peripheral layer interface is exclusive on the same hardware unit.

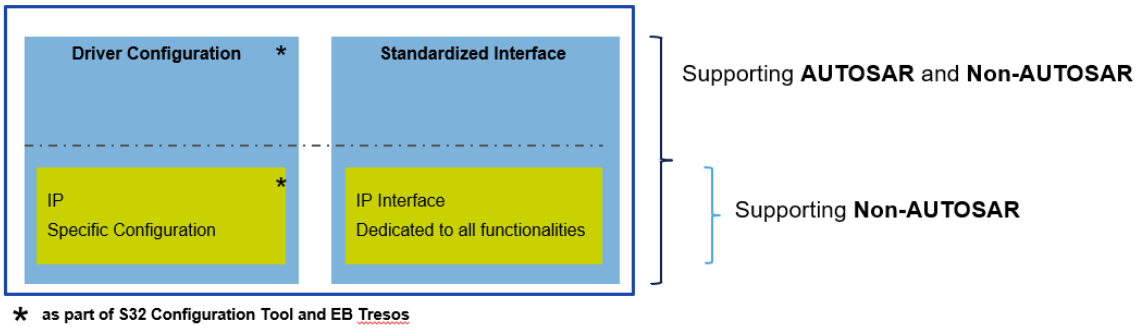

Figure 2. **Overview of driver interface and configuration**

2. MCAL migration guide to RTD

The following features should be considered to migrate an application from MCAL to RTD.

2.1. AUTOSAR version and configuration impact

The standard AUTOSAR MCAL modules that are part of the RTD are implemented following the AUTOSAR 4.4 requirements. Therefore the interface and the configuration for those drivers are compliant with the standard.

The same configuration tool is supported. The default configuration files provided for the modules can be used for configuring the drivers in AUTOSAR projects. No change is needed for already supported tools.

The old MCAL configuration can be imported into an RTD project. Any parameter which has been updated or added will not be updated automatically by the tool importer and it needs to be updated accordingly in the project configuration stage.

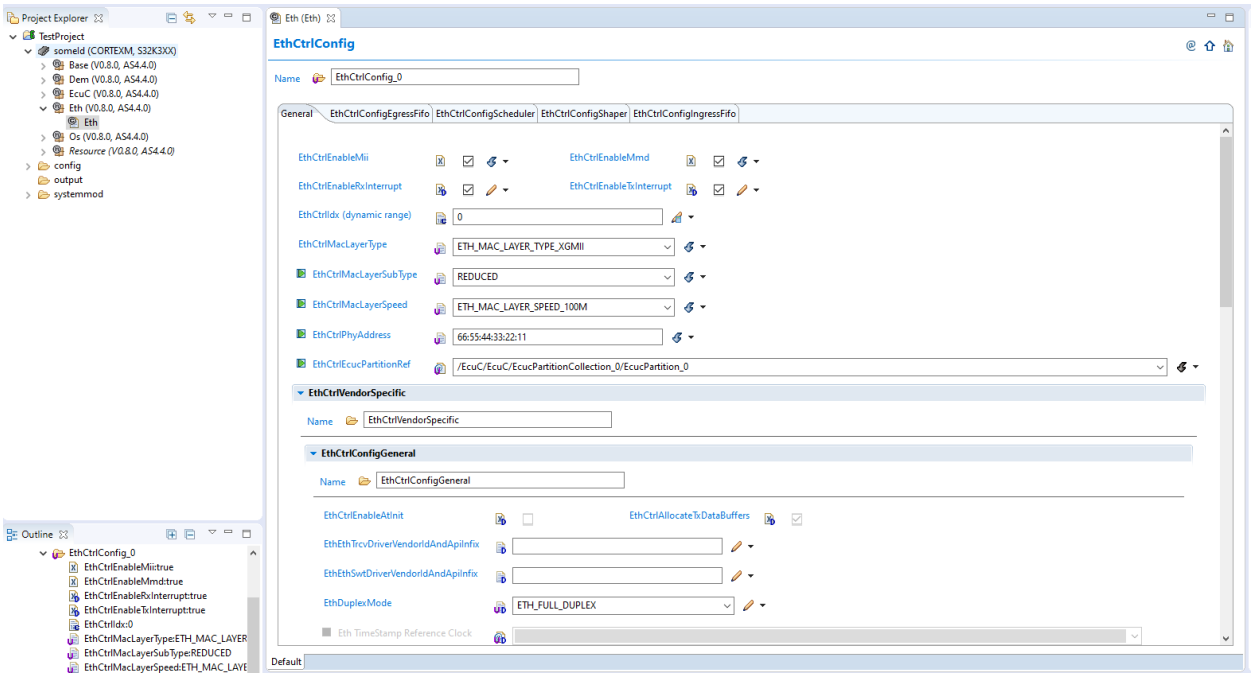

Figure 3. **Driver configuration in EB Tresos**

Additionally, the drivers have integrated support for the S32 Design Studio Configuration Tool. It allows configuring the entire driver (i.e, its configuration in Tresos) and configuration independent of the peripheral interface. For example configuring the AUTOSAR CAN driver (AUTOSAR interface), or only the Non-AUTOSAR FlexCan module (peripheral interface).

Details for each specific platform migration is available in the platform specific appendix.

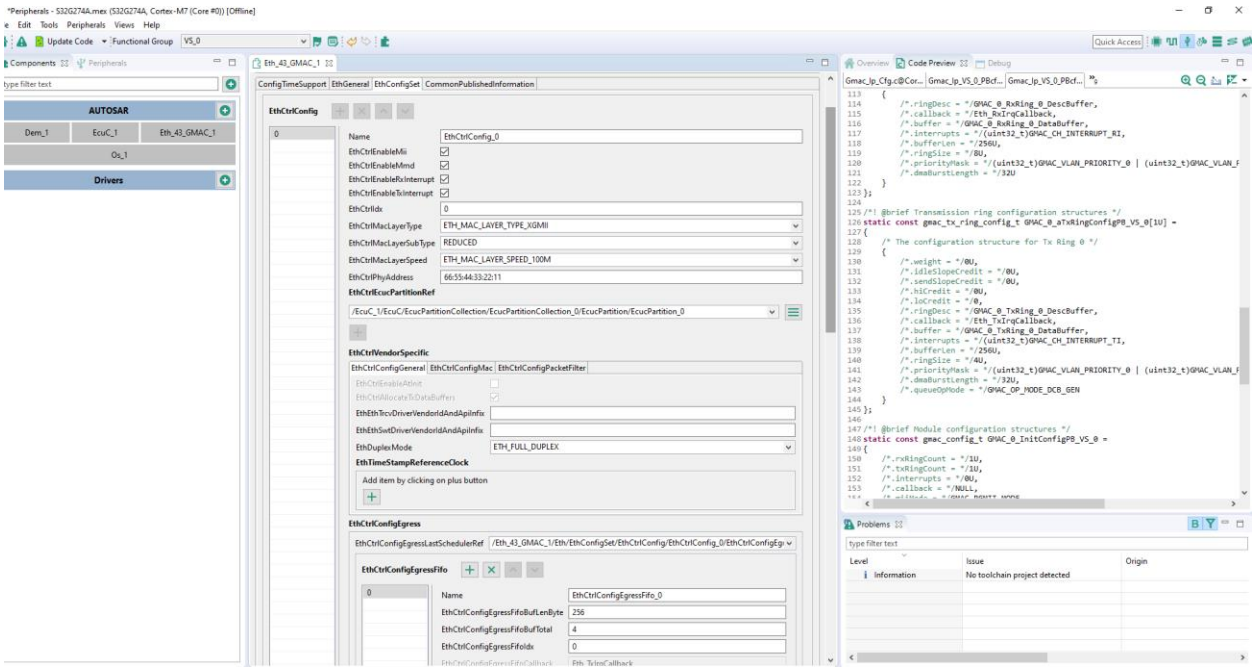

Figure 4. **Driver configuration in S32DS**

2.2. Functionality impact

The RTD provide services to access an extended set of hardware features. From AUTOSAR perspective, the RTD come with additional functional extensions and new CDDs to address most of the hardware features on top of already standardized AUTOSAR functionalities.

2.3. Standard functionalities impact

Migrating a project that utilize all the standard functionalities described by AUTOSAR is seamless. In RTD the AUTOSAR extensions are also supported as described in previous MCAL releases.

In addition, to support previous MCAL use-cases the standard drivers have a new APIs to extend the hardware functionality coverage. The graphic interface is updated to offer the possibility to configuring those features.

2.4. CDD functionalities impact

New CDDs are added to enable the hardware peripherals that were not covered by the previous MCAL releases (i.e.; UART, Quadrature, RM, Platform).

A new feature is available for bare-metal applications through a Platform CDD. It allows configuring and handling the interrupts for the application. The usage of this CDD is optional and there is the option to implement the interrupt management in any other way decided for the application.

Previous functionalities that were available only on MCL are now migrated to RM (Resource Manager CDD) and MCL to cover better the hardware platform features and split the functionalities. The MCL supports enablement for DMA and Caches. The RM will supports enablement for XRDC, SEMA42, MPU and crossbar related (if applicable) configuration.

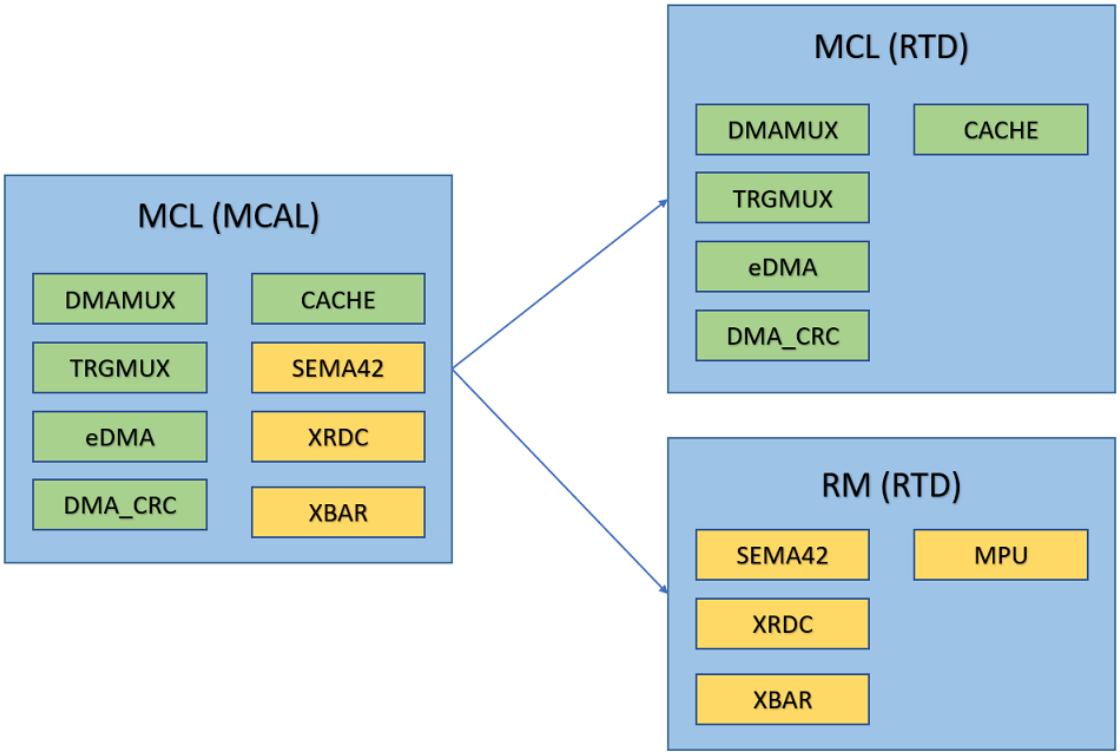

Figure 5. **MCAL to RTD**

2.5. File structure impact

2.5.1. Plugin structure

For MCAL user perspective there is limited impact in the RTD plugin structure. There is a similar file and folder structure like on the previous MCAL release. Additional files and folders are required to be included (i.e.; “headers” folder in Base plugin). The details of each file dependency are specified in the driver IM.

The configuration data files are now split following a more granular approach to ensure the possibility of using the stand-alone peripheral drivers. From a functional point of view, all the data that is needed in an AUTOSAR application will be exported through the HLD files, so nothing changes in the application flow.

All modules require configuration and generation of configuration files prior to their usage. Default configurations files are to be provided to serve as a starting point.

Table 1. **MCAL and RTD differences**

MCAL	RTD	Comments
<Mdl>_Cfg.h	<Mdl>_Cfg.h <Mdl>_Ipw_Cfg.h <Ip>_Cfg.h	Contains precompile parameters used in the driver, usually defines and constants, extern declarations and data types.
<Mdl>_Cfg.c	<Mdl>_Cfg.c <Mdl>_Ipw_Cfg.c <Ip>_Cfg.c	Static configuration structures containing only variables that are not variant aware, configured and generated only once. This file alone does not contain the whole structure needed by *<Mdl>_Init* function to configure the driver. Based on the number of variants configured in the EcuC, there can be more than one configuration structure for one module even for PreCompile variant.
<Mdl>_PBcfg_<Variant>.c	<Mdl>_PBcfg_<Variant>.c <Mdl>_Ipw_PBcfg_<Variant>.c <Ip>_PBcfg_<Variant>.c	There is one file for each variant. The name of the file contains the name of the variant, as defined in the EcuC. This file contains the configuration structure used by the driver that have variant aware members. Each file contains the configuration parameters for its corresponding variant. All parameters and/or structures that are not variant aware and were generated once in the <Mdl>_Cfg.c file are referenced in the structures from <Mdl>_PBcfg_<Variant>.c files if needed. The configuration structures are used in all variants.
<Mdl>_PBcfg_<Variant>.h	<Mdl>_PBcfg_<Variant>.h <Mdl>_Ipw_PBcfg_<Variant>.h <Ip>_PBcfg_<Variant>.h	It was created to export the extern declaration of each configuration structure, to be used when calling *<Mdl>_Init* in the application. There is one file for each variant. The name of the file contains the name of the variant, as defined in the EcuC.

2.6. Exclusive areas

Thread-safety will be ensured by means of Exclusive Areas, with no change in the Exclusive Area name, which will be kept as in the previous MCAL releases.

2.7. Timeout handling

The OsIf module is added to RTD to allow a more flexible approach for applications with regards to OS integration and also more options for users when configuring timeouts. For example, using an OS or a hardware timer for precise timing or loop counting for avoiding any additional resource usage.

The OsIf module needs to be configured within the Base component:

- The type of OS used
- The types of counters/timers to enable
- References to the OS counters or the MCU clock, by case

It is possible to select the type of timeout (precise timing in microseconds or loop counting) in each driver.

From migration perspective, timeout handling is backward compatible, as loop counting is a configuration valid option in the drivers.

2.8. Compiler abstraction

The compiler abstraction has been simplified by removing the memory mapping related to AUTOSAR specific macros (i.e.; FUNC, VAR). From a migration perspective, no impact is expected as the memory mapping is supported through means of MemMap functionality because the supported compilers do not require such abstraction macros and has the benefit of improving code clarity and compliance with code parser tools.

2.9. Migration steps

The sub sections shows the steps to be followed in order to migrate an MCAL project to a RTD project.

2.9.1. Driver configuration

Due to changes in the hardware peripherals, migrating the configuration needs to be done using the following steps.

1. All the general configuration can be imported from an older project by using the import features of the configuration tools chosen. This will import all the configuration fields which are not changed (all the AUTOSAR specific parameters, most of the high-level configuration parameters, etc.).
2. Platform specific updates need to be performed in the configuration tool for all the platform specific parameters which cannot be directly imported by the tool. The provided Recommended

configurations mitigate the migration effort by providing a default configuration as a starting point.

3. The RTD updated/new parameters need to be re-configured. The provided recommended configurations mitigate the migration effort by providing a default configuration as a starting point.

2.9.2. Driver build

1. File structure update, to include the new sources, where ever required.
2. Mapping the changes for rearchitected drivers (i.e.; MCL, RM, MCU. More details in their specific document section) configuration, functionality and symbol names.

2.9.3. Functionality updates

1. The RTD maintains the previous MCAL provided functionality and builds on top of that with support for other hardware peripherals.

3. SDK migration guide to RTD

The RTD offer the functionalities that were available in SDK, extending the PAL with AUTOSAR implementation and support for use-cases addressing both AUTOSAR and Non-AUTOSAR applications. The RTD abstract all the hardware functionalities, as supported in SDK.

The RTD architecture enables decoupling the peripheral layer in order to be used standalone, by providing the peripheral interfaces.

Migrating an application from SDK to RTD implies several changes, as imposed by the RTD architecture.

3.1. Configuration tool impact

From the SDK perspective, the S32 Configuration Tool is used to configure the drivers. In the S32 Configuration Tool both HL and IP interfaces of the driver can be configured to maintain the functionality already provided in SDK.

Specific details are provided in the platform specific appendix.

3.2. Driver configuration changes

The configuration for RTD are designed to address both ASR and non-ASR customers, hence the configuration code can be generated using both S32CT and EB Tresos. The driver configuration files follow the same layered architecture, split between HL and IP layers, with the internal IPW glue-layer in between.

NOTE

The main difference between SDK and RTD configuration scheme is the usage of dynamic(configurable) precompile configuration parameters. It allows the application to disable parts of driver code that are not intended for use (RTD driver sources include the configuration headers) which results in more flexibility and less code size on the application side.

3.2.1. Configuration classes and variant support

The RTD configuration follows the ASR configuration concept that applies on all layers. In order to support multiple configurations that are selectable at different binding times, variant and configuration classes support was added. Consequently, the configuration code is generated in multiple files, corresponding to the variants defined in the project.

Variant support translates into the possibility of generating multiple configuration structures, selectable at runtime. Variant support main use-case is to allow runtime reconfiguration of an ECU (different mode support – startup vs runtime vs shutdown driver behavior, same code with different behaviors depending on external input – left door vs right door car integration, etc.)

Configuration class support translates into the possibility of generating multiple configuration structures, selectable at pre-compile time, link time, post-build time. Each driver has support for a predefined one, several, or all configuration classes. Figure 6 depicts how the configuration files are organized.

All the configuration structures are generated as constants, to avoid their spurious corruption, and allocated in the memory space of the drivers, with support for independent memory mapping (for example: map the configuration data into a slow/read-only memory, map the driver code into a fast memory).

NOTE

Support for initializing a driver without the need to pass a configuration pointer to the initialization function has been replaced by the PRE_COMPILE mode configuration support (as in the AUTOSAR methodology).

No default configuration is stored as internal driver global variables, instead, a (single/unique) PRE_COMPILE configuration can be generated, which is referenced directly by the driver using internal mechanisms. This has the benefit of supporting the same use cases plus offering support for tailoring this configuration with help of the GUI configurator, support for memory mapping relocation, reduced memory consumption and consistent approach for all configuration modes.

All modules require configuration and generation of configuration files prior to their usage. Default configurations files are to be provided to serve as a starting point. These need to be processed by the configuration tool before module initialization can be performed.

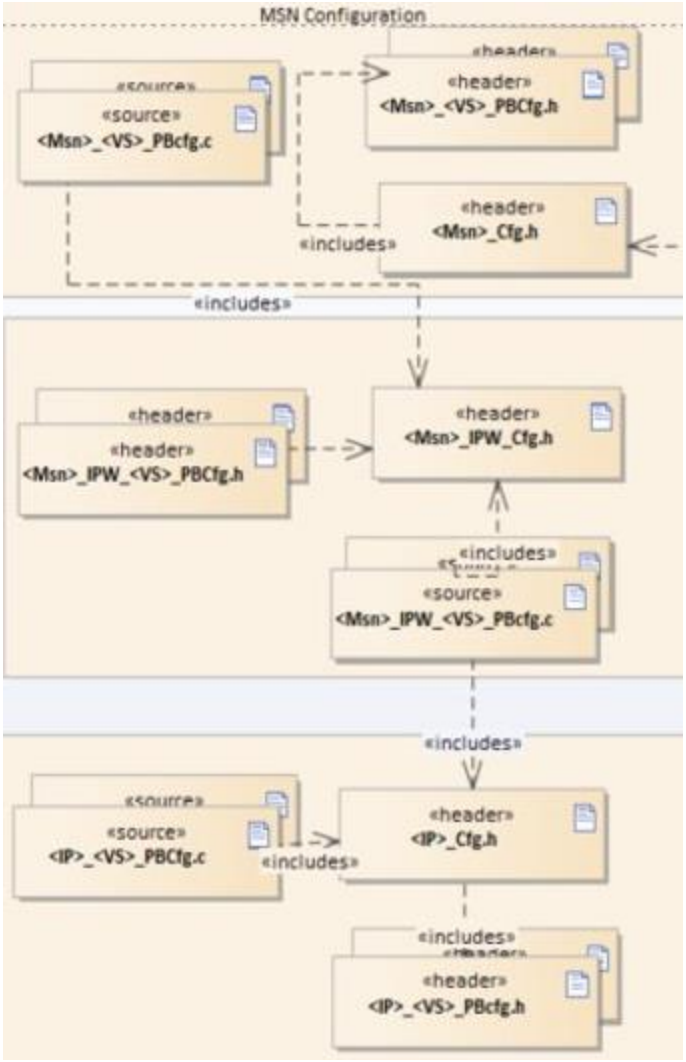

Figure 6. **Driver configuration file structure**

3.3. Functionality impact

The RTD abstract all the hardware functionalities and offers standardized interfaces across platforms. Due to changes in architecture and naming convention, SDK APIs and data types are changed to support the current approach, that will imply that even though from functional perspective the RTD offers same functionalities as the SDK, the migration will not be transparent to the customer.

The naming convention for the RTD implies consistent changes for the data types and APIs, as presented below.

3.3.1. Data types

Data types in RTD use the following naming convention:

<Prefix>_<TypeName>Type

Where:

- **<TypeName>** shall follow the so called CamelCase convention (first letter of each word is uppercase, consequent letters are lowercase).
- **<Prefix>** is
    - HLD: <MSN> ModuleShortName
    - IPW: <MSN>_Ipw (*1)
    - Shared IPs: <Ip>_<MSN>_Ip
    - Non-Shared IPs: <Ip>_Ip
    - Example: typedef uint16 Spi_NumberOfDataType;

NOTE

"IPW" symbols are private and not intended to be used by an application.

3.3.2. API (function) names

APIs in RTD have the following naming convention:

- **HL interface**: <MSN>_<Function>() (ex: Wdg_Init())
- **IP interface**:
    - **<IP>_<MSN>_IP_<Function>()** for all shared IPs (e.g. eTimer_Icu_Ip_StartSignalMeasurement())
    - **<Ip>_Ip_<Function>** for all IPs that are not shared (e.g. Swt_Ip_Init())

NOTE

"IPW" symbols are private and not intended to be used by an application.

The differences between SDK API names and RTD names are summarized in the following table.

Table 2. **SDK and RTD name differences**

Designation	SDK	RTD
Data types	*_t* suffix & *snake_case* style	*Type* suffix & *CamelCase* style
IP layer APIs	*<MDL>_DRV_FunctionName*	*<MDL>_IP_FunctionName*
High Level APIs	*<MDL>_PAL_FunctionName*	*<MDL>FunctionName*

NOTE

There is no implication that these formal changes in the API are the only required updated for porting from SDK to RTD. The APIs in RTD also contain semantic changes, as required by the integration with higher levels. From a functional perspective, RTD APIs map logically on former SDK APIs (no regression/degraded functionality); however, the meaning of some function names/parameters may differ. It is application's responsibility to use the proper APIs for the required functionality, after carefully studying the user manuals.

3.4. Memory mapping

The RTD introduce mechanisms for the mapping of code and data to specific memory sections in order to support avoidance of waste of RAM, usage of specific RAM properties, usage of specific ROM properties, usage of the same source code of a module for boot loader and application, encapsulation and isolation.

Default memory sections are provided inside the RTD package, therefore migrating from SDK to RTD implies only adding to the project the <Driver>_MemMap.h files provided into Base plugin and update the linker files.

The <Driver>_MemMap.h file stubs are provided, which is expected to be updated in AUTOSAR context and can be used as it is in Non-AUTOSAR context.

3.5. Expose interface

The RTD are designed to satisfy both AUTOSAR and non-AUTOSAR (former SDK) use cases. The RTD provide two sets of interfaces:

- Standardized interface generic across platforms
- IP interface generic across platforms with the same set of IP features

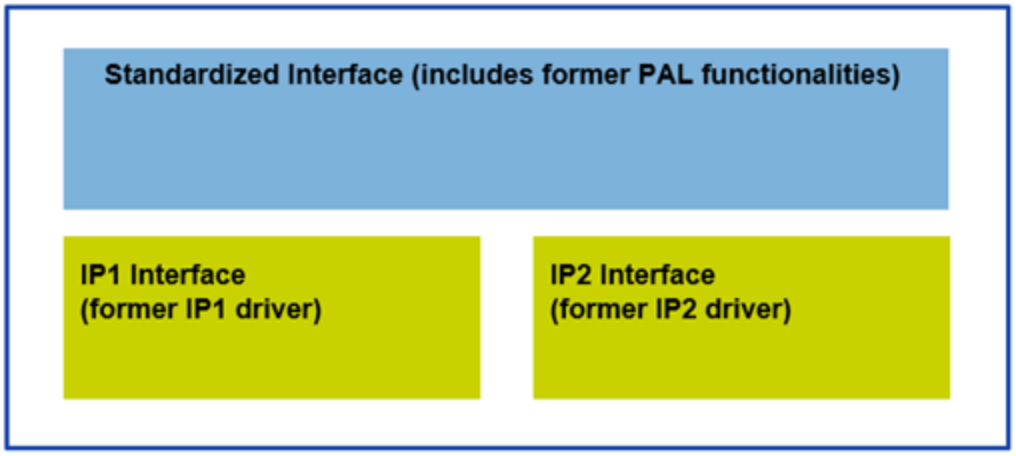

Figure 7. **Exposed interfaces**

The following limitations are still applicable when migrating the application from SDK to RTD,:

- It is forbidden to use the same hardware instance in HL and IP (E.g. if SPI1 is used in HL context, it cannot be used also through IPL)
- IP layer does not provide Tresos configuration. It can be configured only on Design Studio (CT).
- IP layer is not intended to be used in AUTOSAR applications, as it does not satisfy the ASR compliance constraints (DEM, DET, Multicore, etc.); IP layer is not intended to be a standalone AUTOSAR CDD.

3.6. Error management

The error detection and reporting mechanism for the RTD is tailored for the target application type. Error management concept incorporates reporting of the development errors and runtime errors, using different reporting mechanisms as described below.

Migrating applications that used the PAL functionalities implies architectural modification from error management perspective. For the high-level layer, error management follows the AUTOSAR specifications for Default Error Tracer and Diagnostics Event Manager. RTD provides reference code for the implementation of these modules, which can be used or overwritten by the customer application.

3.6.1. HL API

The HL APIs return Std_ReturnType (E_OK/E_NOT_OK) where synchronous reaction is needed.

For asynchronous reactions, when these are defined, the HL APIs will return an extra specific error, which can then be retrieved by calling the dedicated APIs in DEM/DET.

Default Error Tracer (DET) offers mechanisms to handle both development errors and runtime errors.

Diagnostic Event Manager (DEM) offers mechanisms to handle critical runtime errors,) in case these have a high impact on the application integrity.

The RTD provide a “stub” implementation of these ASR modules, which can be used or overwritten by the customer application.

3.6.2. IP API

The errors reported by the IP layer are still split in two categories:

- **Development errors**: Usually parameters checking, function call plausibility, etc. These errors are checked using DevAssert function. In case an error is detected, it halts the program execution in the default implementation. The default behavior of DevAssert function can also be overwritten by the application. This mechanism is almost identical to the DEV_ASSERT functionality in older SDK, the only improvement being that these statements are now enabled/disabled for each driver separately, as opposed to the SDK approach where this was a global configuration (refer to Figure 8).
- Runtime errors: As opposed to the SDK, where all runtime errors reported by drivers were grouped in the generic enumeration called status_t, the RTD define a set of runtime errors as per driver. The naming convention for these errors is <IP_Name>_Ip_StatusType.

Each driver defines the set of errors that can be reported by the controlled IP, these errors can either be used by the non-ASR application implemented on top of the IP layer for retrieving the status of the driver, or further fed into the high level state machine of the layers on top.

3.7. File structure

To comply with both AUTOSAR and Non-AUTOSAR configuration tools, the RTD use a modular approach on the file layout. While in S32SDK, drivers shared a common folder, but were distinct from PALs, RTD modules are contained in individual folders.

As such, in the RTD a module is the container for both IP layer and HL driver. Due to this fact, a module will have an IP folder, containing the implementations for the targeted IPs and two folders for the HL (include and src).

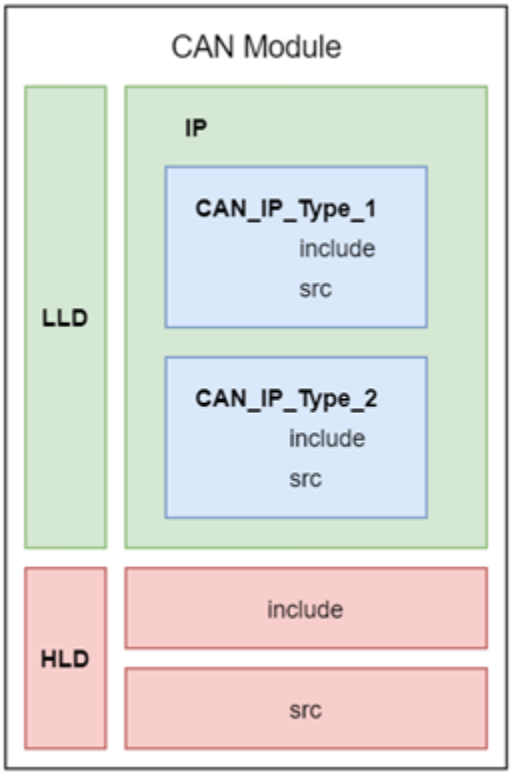

Figure 8. **RTD file structure**

3.8. Interrupt management

As opposed to the S32 SDK, the RTD do not manage interrupt requests at system level. It is an external assumption that the proper interrupts are enabled in the interrupt controller and the right handlers are present in the IVT for the drivers to work. It is thus application responsibility to configure the interrupt controller and define the right ISRs, as mentioned in each driver documentation.

From migration perspective, the RTD define a dedicated platform specific driver, called Platform_CDD, which provides the API and configuration support for setting up the interrupts. The configuration for Platform_CDD contains all the required information for the interrupt settings (enablement, priorities, handlers etc.). Calling the initialization function of this new driver sets the right configuration in place and is considered a prerequisite for the correct functionality of other drivers that require interrupt routines.

3.9. Timeout handling

The RTD do not support mutexes and semaphores for timeout handling. The timeout handling is simplified by offering asynchronous (interrupt driven) and synchronous(polling) services. For more details, review Timeout handling.

3.10. Safety

The RTD contain the safety analysis (FMEA) and the safety measures covering both exposed interfaces (HL + IP), including the safety measures and external assumptions for the application.

4. OS abstraction – OSIF

4.1. Migration from MCAL to RTD-OSIF

The OsIf module is added to the RTD to allow a more flexible approach for applications with regards to OS integration and also more options for users when configuring timeouts, for example, using an OS or a hardware timer for precise timing or loop counting for avoiding any additional resource usage.

The OsIf module needs to be configured within the Base component:

- The type of OS used
- The types of counters/timers to enable
- References to the Os counters or the Mcu clock, by case

Then, in each driver it is possible to select the type of timeout (precise timing in microseconds or loop counting).

4.2. Migrating from SDK to RTD-OSIF to OSIF

The OsIf module in the RTD is largely different from the SDK counterpart. Mutex and semaphores are no longer supported, and the timing services are mainly geared toward timeouts rather than measuring time or delays.

The project-level symbol (compiler -D options) for selecting the type of OS, -DUSING_OS_BAREMETAL or -DUSING_OS_FREERTOS remain the same. Additionally, -DUSING_OS_AUTOSAROS is now supported.

5. Multicore support

The RTD offers multicore support, by providing support for both former MCAL AUTOSAR multicore concept and SDK instance per core approach. See the following figure for more details.

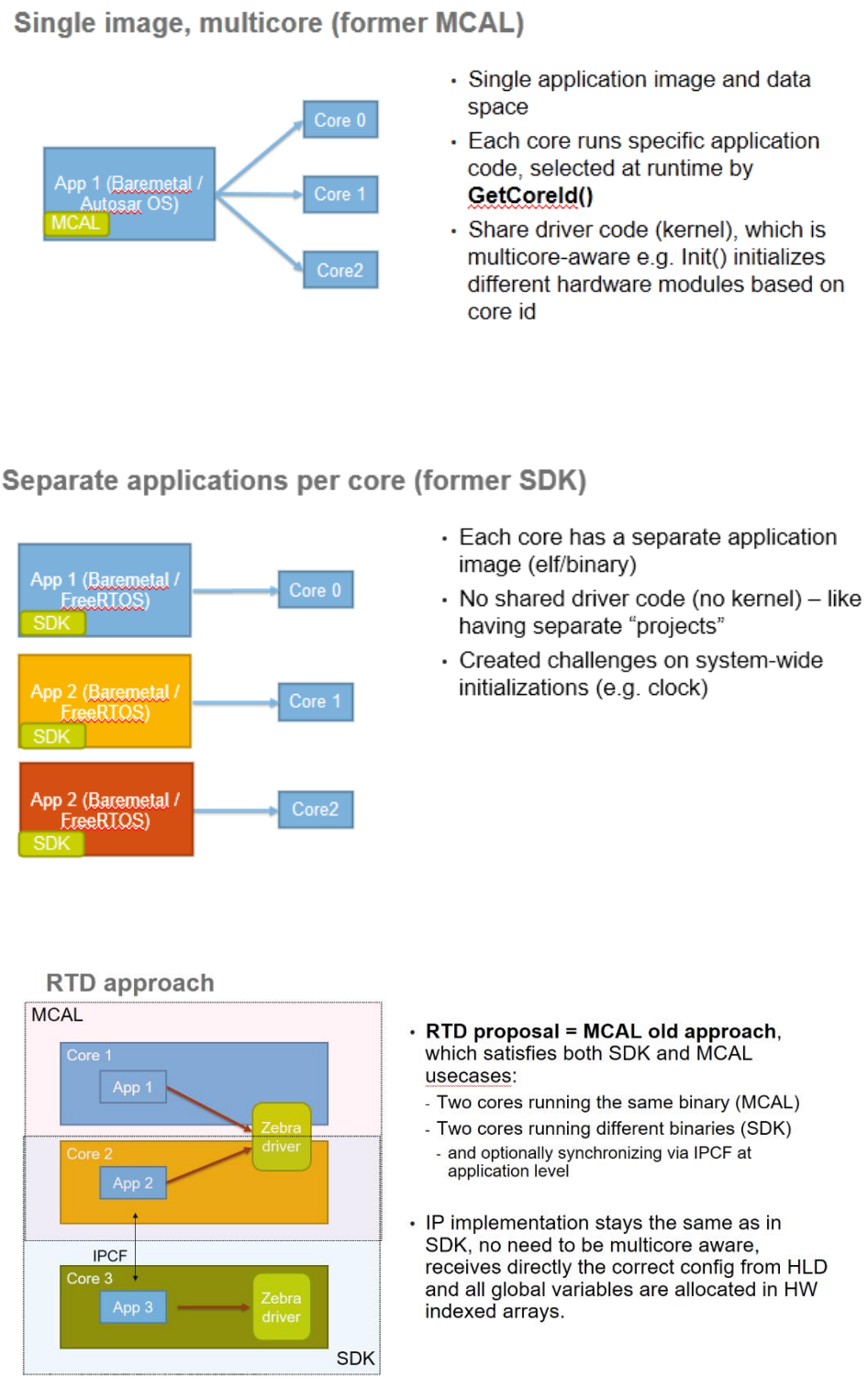

Figure 9. **Multicore support**

6. Release packaging

The RTD release package contains source code, Tresos configuration, S32CT configuration, examples, documentation.

From a delivery perspective, two methods are available:

- Design studio update site
- Installer published in Flexera

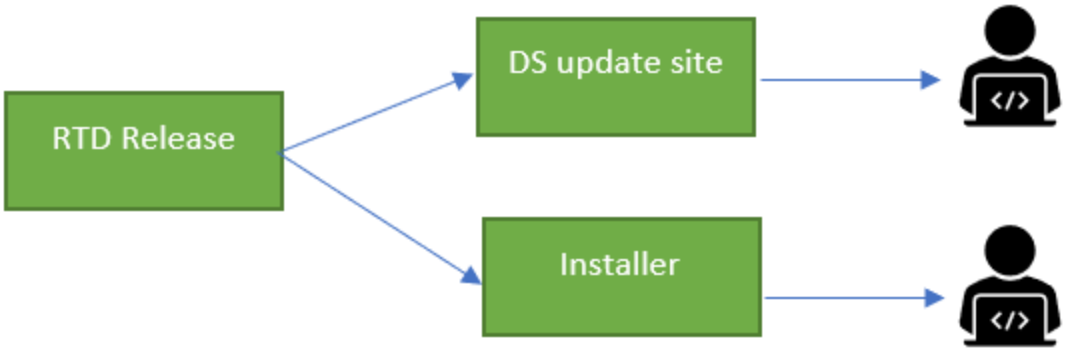

Figure 10. **RTD release packaging methods.**

From a former SDK user perspective, the deployment of the product as an S32 Design Studio update site will provide the same level of integration with configuration/build/debug tools, inherent from the IDE integration. The product is usable along the S32 configuration tool, that can configure both HL and IP layers; it also contains sample applications delivered as a ready-to-use DS project, highlighting the usage of the drivers and ready to be downloaded on target using built-in toolchain support, providing excellent “out of the box” experience for non-ASR customers.

Additionally, RTD integrates with other NXP software products which are bundled into S32 DS, including various non-ASR middleware and stacks which are ported on top of the drivers.

From MCAL perspective, the deployment of the product the installer published in Flexera will contain the same level of integration for RTD.

Appendix A. S32KXX product family

Chapter 1. Overview

Summary of the platform specific migration aspects, including the platform specific driver/IP mapping, the configuration tool.

Table 3. **Mapping of peripherals components and S32K3XX drivers**

Real Time Driver	S32K3XX IP	Comments
DEM	-	Diagnostics Event Manager Reference code provided by NXP to be used in Non AUTOSAR applications. To be replaced to AUTOSAR Standard Implementation for AUTOSAR applications
DET	-	Default Error Tracer Reference code provided by NXP to be used in Non AUTOSAR applications. To be replaced to AUTOSAR Standard Implementation for AUTOSAR applications
ECUC	-	Ecu Configuration – add support for multicore Reference code provided by NXP to be used in Non AUTOSAR applications. To be replaced to AUTOSAR Standard Implementation for AUTOSAR applications
ECUM	-	Ecu Manager Reference code provided by NXP to be used in Non AUTOSAR applications. To be replaced to AUTOSAR Standard Implementation for AUTOSAR applications
RTE	-	Run Time Environment – implements exclusive areas Reference code provided by NXP to be used in Non AUTOSAR applications To be replaced to AUTOSAR Standard Implementation for AUTOSAR applications
OsIf	-	OS Abstraction layer Integrates support for FreeRTOS and AUTOSAR OS

Resource	-	Resource driver – collection for all derivatives features
BASE	-	Base driver Collection of header files
	REG_PROT	A subset of driver's files provided by NXP can be used in Non AUTOSAR applications A subset of driver's files provided by NXP can be replaced to AUTOSAR Standard Implementation for AUTOSAR applications
MCU	MC_CGM	Microcontroller Unit Driver Provides services for basic microcontroller initialization, mode management and clock management
	FIRC	
	SIRC	
	PLLDIG	
	FXOSC	
	MC_RGM	
	MC_ME	
	SXOSC	
	CMU	
	MC_PCU	
	PFLASH	
	PRAMC	
	PMC	
PLATFORM	MSCM	Platform – Complex Device Driver Integrates functionalities specific to platform and interrupt management
	NVIC	
	INTM	
	MCM	
RM	MPU	Resource Manager – Complex Device Driver
	XRDC	
	SEMA42	
	PFLASH	

	VIRT_Wrapper	
	XBIC - MCR reg	
	XBAR	
MCL	eDMA	Microcontroller Library – Complex Device Driver Offers DMA and Cache functionalities
	LCU	
	DMAMUX	
	TRGMUX	
	Cache_M7	
PORT	SIUL2	PORT Driver Provides the services for initializing the whole structure of PORT driver
DIO		Digital Input Output Driver Provides services for accessing the microcontroller's hardware pins
EEP	FLEXRAM	EEPROM Driver Provides services for reading, writing, erasing to/from an EEPRM
	FLEXNVM	
FLS	QuadSPI	Flash Driver
	C40	
	PFLASH	
ICU	eMIOS	Input Capture Unit Driver
	SIUL2	
	LPCMP	
	WKPU	
OCU	eMIOS	Output Capture Unit Driver
GPT	eMIOS	General Purpose Timer Driver
	PIT(-RTI)	

	RTC	
	STM	
PWM	eMIOS	Pulse Width Modulation Driver
	FlexIO_PWM	
QD	eMIOS	Quadrature Complex Device Driver
I2C	LPI2C	I2C Complex Device Driver
	FlexIO_I2C	
ETH	ENET	Ethernet Driver
UART	LPUART	UART Complex Device Driver
	FlexIO_UART	
LIN	LPUART	Lin Driver
SPI	LPSPI	Serial Parallel Interface Driver
	FlexIO_SPI	
CAN	FlexCan	Can Driver
ADC	ADC	Analog Digital Comparator Driver
	BCTU	
Crypto	HSE_M	Crypto Driver
	MU	
WDG	SWT	Watchdog Driver
SENT	FlexIO_SENT	SENT driver
CRC	CRCU	CRC Complex Device Driver
SAI	SAI	SAI Complex Device Driver

Chapter 2. AUTOSAR Configuration and version impact.

The standard AUTOSAR MCAL modules that are part of the Real Time Drivers releases will be implemented following the AUTOSAR 4.4 requirements, therefore the interface and the configuration for those drivers will be compliant with the standard.

S32K1 and S32K2 MCAL projects were developed according to AUTOSAR 4.3, S32K3 is developed according to AUTOSAR 4.4. As a result, there is an impact in the AUTOSAR specific parameters which have been updated between these revisions. The expected impact is small, considering that the AUTOSAR 4.4 was an incremental updated with no disruptive change over AUTOSAR 4.3. Impacted modules are: CRYPTO and all the modules which performed configuration updates to accommodate the AUTOSAR Multicore Concept.

The AUTOSAR compliant configuration tools can leverage the import of all the parameters which have kept the correspondence from MCAL to RTD schema. Most of the AUTOSAR MCAL standard parameters and most of the Vendor specific extensions are expected to be compatible and importable in a new RTD project.

For all the new AUTOSAR RTD CDDs and a small subset of Vendor specific parameters, the configuration needs to be created from scratch. Default configuration files are provided as a starting point, to reduce the effort.

The same configuration tools and configuration flows (including default configurations) will be supported.

Chapter 3. SDK Configuration and tool impact

From SDK perspective, S32 Configuration Tool, that is part of S32 Design Studio, is used to configure the drivers. In the S32 Configuration Tool both HL and IP interfaces of the driver can be configured, in order to maintain the functionality already provided in SDK.

On S32K3 the same configuration tool (S32CT) will be supported.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: AN13435
Rev. 0
10/2021