HighIntegritySystems

WITTENSTEIN

Real-Time Drivers and SAFERTOS® for Safety Critical Applications

How to migrate from SDK/MCAL to Real-Time Drivers and from FreeRTOS to SAFERTOS® for Safety Critical Applications.

Issue 1.0 - June 21, 2023

Written in Partnership with NXP

WITTENSTEIN high integrity systems
Americas: +1 408 625 4712
ROTW: +44 1275 395 600

email: sales@highintegritysystems.com
web: www.highintegritysystems.com

WITTENSTEIN

CONTENTS

List of Figures

List of Notation

API	Application Programming Interface
BSP	Board Support Package
CAN	Controller Area Network
CDDs	Complex Device Drivers
CPU	Central Processing Unit
CSEc	Cryptographic Hardware Accelerator
DAP	Design Assurance Pack
DEM	Diagnostic Event Manager
DET	Default Error Tracer
DHF	Design History File
ECC	Error Correcting Code
ECU	Electronic Control Unit
HDQFP	High Density QFP
IP	Internet Protocol
HLI	High-Level Interface
HSE	Hardware Security Engine
IDE	Integrated Development Environment
IPW	IP Wrapper
IVT	Interrupt Vector Table
LIN	Local Interconnect Network

LLCE	Low Latency Communication Engine
LLI	Low-Level Interface
MCAL	Microcontroller Abstraction Layer
MCU	Microcontroller Unit
MMU	Memory Management Unit
MPU	Memory Protection Unit
OS	Operating System
OTA	Over-The-Air
PAL	Peripheral Abstraction Layer
PFE	Packet Forwarding Engine
RM	Resource Manager
RTD	Real Time Drivers
RTOS	Real Time Operating System
SAF	Safety Software Framework
SBCs	System Basis Chips
SCST	Structural Core Self-test
SDK	Software Development Kit
SEooC	Safety Element out of Context
SIL	Safety Integrity Level
SPD	Safety Peripheral Drivers
SPI	Serial Peripheral Interface
TCB	Task Control Block
UART	Universal Asynchronous Receiver-Transmitter

WITTENSTEIN

CHAPTER 1 Introduction

Functional safety is defined as the absence of unreasonable risk due to hazards caused by malfunctioning behaviour of electrical or electronic systems. Its focus within the automotive industry originated mainly in powertrain applications such as braking, chassis and steering control. Today, increased ECU content in vehicles and advanced safety features have contributed to its growth across numerous sub-systems including body control, battery management, and zonal control to name a few. NXP Semiconductor's S32K1 and S32K3 families of automotive microcontrollers (MCUs) comprise single and multi-core devices that are suitable for applications that require ASIL B to ASIL D levels of functional safety. Safety hardware features include but are not limited to: ECC on Flash and RAM memories, clock and voltage monitors, and a Fault Collection and Control Unit (S32K3 only). Supporting these is a collection of NXP software – Real-Time Drivers (RTD) for AUTOSAR™ and non-AUTOSAR, Structural Core Self-test (SCST) library, Safety Peripheral Drivers (SPD) and a Safety Software Framework (SAF).

NXP's S32K1, S32K3 and S32G processor families are supported by SAFE**RTOS**® from WITTENSTEIN high integrity systems (WHIS). Used widely in automotive and other safety-critical applications, developers using SAFE**RTOS**® will often prototype with open source FreeRTOS on the target hardware before upgrading to SAFE**RTOS**® at the start of formal development.

This paper will examine the development of a safety application in the context of migrating from NXP's Microcontroller Abstraction Layer (MCAL) or Software Development Kit (SDK) drivers to RTD, and from FreeRTOS to SAFE**RTOS**®.

CHAPTER 2 Real-Time Drivers (RTD)

2.1 Introduction to Real-Time Drivers

Real-Time Drivers (RTD) combine AUTOSAR MCAL drivers with a set of Complex Device Drivers (CDDs) and are designed for automotive processors featuring Arm® Cortex®-M cores. RTD combine elements of NXP's SDK and MCAL drivers into a single software product that provides full IP and feature coverage for both AUTOSAR and non-AUTOSAR applications. Developed as a Safety Element out of Context (SEooC) using an ISO 2626 compliant process, RTD enhance and extend the previous generation of drivers and are suitable for use in applications up to ASIL D level. NXP provides RTD for System Basis Chips (SBCs), Ethernet and CAN PHY devices delivered as separate packages.

RTD are available as a standalone distribution or integrated within NXP's S32 Design Studio (S32DS) IDE. RTD can be configured using the S32 Configuration Tool (S32CT), Elektrobit's EB tresos Studio™ or any other AUTOSAR compliant configurator (e.g. Vector's DaVinci Configurator Pro™). The RTD installation package includes source code, comprehensive documentation and multiple examples/demos.

There are two APIs within RTD: a High-Level Interface (HLI) for AUTOSAR and non-AUTOSAR applications and a Low-Level Interface (LLI) for non-AUTOSAR applications. Both have functionality for MCU, memory, communication, I/O and complex device drivers containing aspects for analogue, safety, security, header files and others. Safety Peripheral Drivers are delivered as part of NXP's broader Safety Software Framework (SAF) or Safety Peripheral Drivers (SPD) packages.

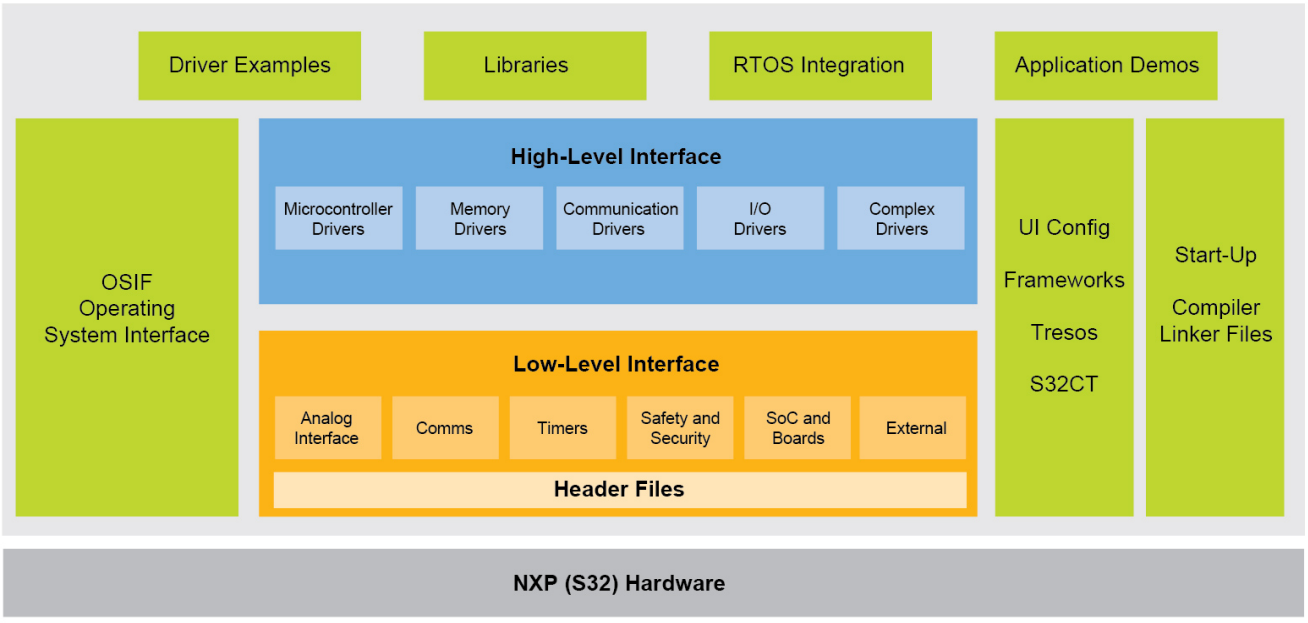

Figure 2-1 *Real-Time Drivers*

2.2 Supported Devices with RTD

RTD are available for a wide range of NXP automotive processors. No license cost is required for use in AUTOSAR or non-AUTOSAR applications.

2.2.1 S32K1/K3 MCUs

S32K is a scalable portfolio of Arm® Cortex®-M core based MCUs targeted at general purpose automotive and industrial applications. All devices are AEC-Q100 qualified and include next generation safety, security and connectivity features with extensive software and tool support from NXP and its partners. Included in NXP's product longevity program, S32K MCUs are available for a minimum of 15 years of production.

The S32K1 family consists of 8 single core Arm® Cortex®-M0+ and M4F core MCUs with flash memory ranging from 128kB to 2MB and ASIL B level safety features. Additional features include CAN-FD, 100Mpbs Ethernet, a cryptographic hardware accelerator (CSEc) with NXP firmware, and a range of low power operating modes. Software includes MCAL drivers and a production-grade Software Development Kit (SDK) containing low-level drivers. While still available, these driver packages are now superseded by RTD. FreeRTOS™ OS example code is included with RTD as part of NXP's Reference Software package.

The S32K3 family extends the portfolio scalability by introducing Arm® Cortex®-M7-based MCUs in single, dual, triple and lockstep core configurations supporting up to ASIL D applications. Flash memory range is extended up to 8MB with symmetric and asymmetric cryptography provided by the Hardware Security Engine (HSE). Full hardware over-the-air (OTA) update capability and Gbps Ethernet with TSN support further extend the feature set with devices available in BGA and new High Density QFP (HDQFP) packages.

2.2.2 S32G2/G3 processors

The S32G2/3 families of vehicle network processors are designed to enable the development of high performance, domain-based vehicle architectures while providing reduced software complexity and enhanced security and safety. Built upon up to eight 1.3GHz Arm Cortex-A53 cores, S32G devices include a hardware-accelerated Low Latency Communication Engine (LLCE) for de-coupling low-level CAN, LIN, FlexRay and SPI data processing from the main cores. Ethernet packets are accelerated using a Packet Forwarding Engine (PFE) that offloads computationally-intensive routing and address translation tasks from the main cores. S32G processors share many of the same software and tools as the S32K1/3 MCUs but add a Linux BSP and a range of cloud-connected products from NXP partner companies.

2.3 Downloading RTD

You can find links to download RTD at nxp.com/RTD with versions available for each supported processor family.

2.4 Migrating from SDK/MCAL to RTD

RTD provide multiple software features as extensions to the AUTOSAR standard with full coverage of all hardware features and peripherals. RTD contain examples with default configurations that are offered for the NXP S32CT and EB tresos configuration tools.

RTD integrate use cases from AUTOSAR and non-AUTOSAR environments therefore from the developer perspective the SDK and MCAL experience is maintained. Each extension from the standard package can be enabled or disabled.

From an MCAL developer's perspective, RTD extend the standardised functionality as defined by the standard and provide full coverage of on-chip hardware. In AUTOSAR applications the standardized interface is available and it is recommended to use this to maintain portability across applications.

From an SDK customer perspective, RTD extend the complete peripheral functionality while adding support for multi-core and user mode, and memory mapping of code and data to specific memory sections. For non-AUTOSAR applications, both High and Low Level interfaces (standardized and IP) are available and can be used.

From the configuration point of view, S32CT can be used to configure both the driver and the peripheral layer independent of the driver. The use of the standardized interface and peripheral layer interface is exclusive on the same hardware unit. EB tresos can be used for High Level driver configuration only.

2.4.1 MCAL Migration Guide to RTD

2.4.1.1 AUTOSAR version and configuration impact

The standard AUTOSAR MCAL modules that are part of RTD are implemented following the AUTOSAR 4.4 version or newer (please refer to Release Notes for details), hence the interface and the configuration for those drivers are compliant with the standard.

Default configuration files have been replaced in RTD with driver examples. No changes are required for currently 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.

Additionally, the drivers are supported by the S32 Configuration Tool inside S32DS. This allows configuring of the entire driver (i.e. its configuration in tresos) with the configuration being independent of the peripheral interface. For example, configuring the AUTOSAR CAN driver (AUTOSAR interface), or configuring only the non-AUTOSAR FlexCAN module (peripheral interface). Details for each specific platform migration are available in the platform specific appendix.

From the AUTOSAR perspective, RTD include additional functional extensions and new CDDs to address most of the hardware features on top of the already standardized AUTOSAR functionalities.

Migrating a project that utilizes 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 new APIs to extend the hardware functionality coverage. The graphic interface is updated to offer the possibility to configure those features.

2.4.1.2 CDD functionalities impact

New CDDs are added to enable the hardware peripherals that were not covered by the previous MCAL releases (e.g. UART, Quadrature, RM, Platform). A new feature is available for bare-metal applications through a Platform CDD which allows configuring and handling the interrupts for the application. The usage of this CDD is optional and interrupt management can be customised for the application. Previous functionalities that were available only in MCAL are now migrated to RM (Resource Manager CDD) and MCL to improve coverage for hardware platform features and to split the functionalities.

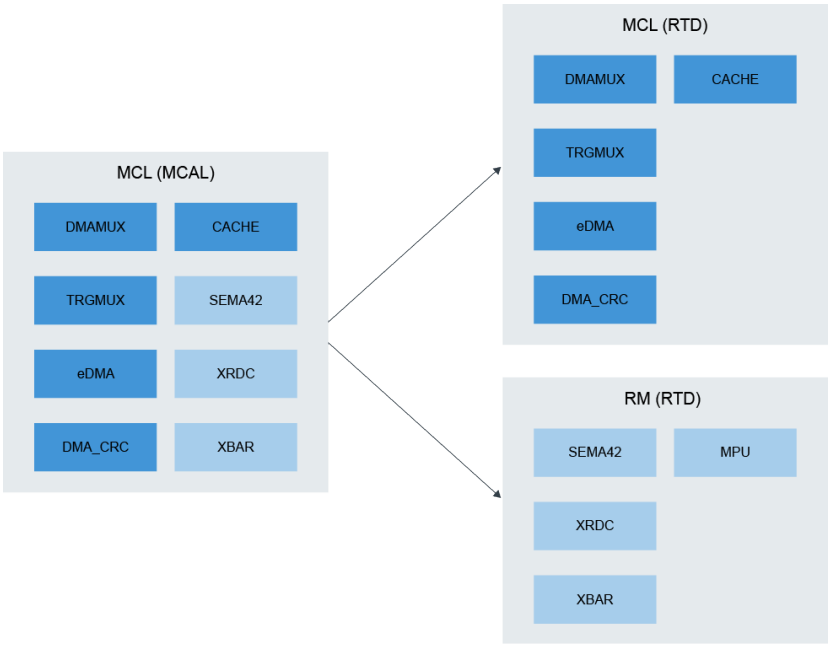

Figure 2-2 *MCAL to RTD*

2.4.1.3 File structure impact: Plugin structure

For the MCAL user perspective there is limited impact in the RTD plug-in 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 plug-in). 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 HLI files, so nothing changes in the application flow.

All modules require configuration and generation of configuration files prior to their usage. Example projects, included in the plugins, are provided to serve as a starting point.

2.4.1.4 Timeout handling

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

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

2.4.1.5 Compiler abstraction

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

2.4.1.6 Driver configuration and build

Due to changes in the hardware peripherals, migrating the driver configurations should be accomplished via the following steps:

1. All the general driver configurations 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. Configuration examples can be used as a starting point.
3. RTD updated/new parameters need to be re-configured. The provided example projects mitigate the migration effort by providing a default configuration as a starting point.

For driver build, the following steps need to be completed.

1. File structure update, to include the new sources, wherever required.
2. Mapping the changes for rearchitected drivers’ configuration, functionality and symbol names.

2.4.2 SDK Migration Guide to RTD

RTD offer the functionalities that are available in the SDK, extending the Peripheral Abstraction Layer (PAL) with AUTOSAR implementation and support for use-cases addressing both AUTOSAR and non-AUTOSAR applications. RTD abstract all the hardware functionalities as supported in the SDK. The RTD architecture enables decoupling of the peripheral layer in order to be used standalone by providing the peripheral interfaces. Migrating an application from SDK to RTD requires several changes as imposed by the RTD architecture. From the SDK perspective S32CT is used to configure the drivers. In the S32CT, both HL and IP interfaces of the driver can be configured to maintain the functionality already provided in SDK.

2.4.2.1 Driver configuration changes

The driver configuration files follow the same layered architecture, split between HL and IP layers, with the internal IP Wrapper (IPW) glue-layer in between.

Note:
The main difference between SDK and RTD configuration scheme is the usage of dynamic (configurable) pre-compiled configuration parameters. These allow 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.

2.4.2.2 Configuration classes and variant support

The RTD configuration follows the AUTOSAR 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 and post-build time. Each driver has support for a predefined one, several, or all configuration classes.

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). This applies only to the High Level Interfaces.

No default configuration is stored as internal driver global variables, instead, a 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. Configuration examples can be used as a starting point and need to be processed by the configuration tool before module initialization can be performed.

2.4.2.3 Functionality impact

RTD abstract all the hardware functionalities and offer 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 RTD offers same functionalities as the SDK, the migration will not be transparent to the customer.

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

Data types in RTD use the following naming convention:

`<Prefix>_<TypeName>Type`

Where:

- `<TypeName>` shall follow the so called PascalCase convention (first letter of each word is uppercase, consequent letters are lowercase).
- `<Prefix>`:
  - For HL interface: `<MSN>_[<vi>_<va>_]`
  - For IPW: `<MSN>_[<vi>_<va>_]_Ipw`
  - For IP interface:
    `<Ip>_<MSN>_Ip` for all shared IPs
    `<Ip>_Ip` for IPs that are not shared
  - Where:
    `<MSN>` Module Name
    `<vi>` Vendor ID (=43 for NXP)
    `<va>` Vendor Api Infix
    `<Ip>` Name of the IP

Example: `typedef uint16 Spi_NumberOfDataType;`

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

APIs in RTD have the following naming convention:

`<Prefix>_<APIName>()`

Where:

- `<APIName>` shall follow the so called PascalCase convention (first letter of each word is uppercase, consequent letters are lowercase).
- `<Prefix>`:
  - For HL interface: `<MSN>_[<vi>_<va>_]` (example : `Lin_SendHeader()`)
  - For IPW: `<MSN>_[<vi>_<va>_]_Ipw`
  - For IP interface:
    `<Ip>_<MSN>_Ip` for all shared IPs (example: `eTimer_Icu_Ip_StartSignalMeasurement()`)
    `<Ip>_Ip` for Ips that are not shared (example: `Swt_Ip_Init()`)
  - Where:
    `<MSN>` Module Name
    `<vi>` Vendor ID (=43 for NXP)
    `<va>` Vendor Api Infix
    `<Ip>` Name of the IP

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 (for drivers that do not use Vendor ID and Vendor Api Infix, and for peripherals that are not shared between drivers).

Designation	SDK	RTD
Data types	`_t` suffix & `snake_case` style	`Type` suffix & `PascalCase` style
IP layer APIs	`<MDL>_DRV_FunctionName`	`<MDL>_Ip_FunctionName`
High Level APIs	`<MDL>_PAL_FunctionName`	`<MDL>_FunctionName`

Table 2-1 *SDK and RTD name differences*

Note:
There is no implication that these formal changes in the API are the only required updates 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.

2.4.2.4 Memory mapping

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 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.

2.4.2.5 Expose interface

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

- High Level: Standardized interface generic across platforms
- Low Level: IP interface generic across platforms with the same set of IP features

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 using the S32 Configuration Tool (S32CT) inside S32 Design Studio.
- IP layer is not intended to be used in AUTOSAR applications, as it does not satisfy the AUTOSAR compliance constraints (DEM, DET, Multicore, etc.); IP layer is not intended to be a standalone AUTOSAR CDD.

2.4.2.6 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.

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 AUTOSAR modules, which can be used or overwritten by the customer application.

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 3).
- **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.

2.4.2.7 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.

2.4.2.8 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 the application's responsibility to configure the interrupt controller and define the right ISRs as mentioned in each driver's documentation.

From migration perspective, the RTD define a dedicated platform specific CDD, called Platform, which provides the API and configuration support for setting up the interrupts. The configuration for Platform 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.

2.4.2.9 Exclusive areas

Thread-safety is ensured by means of Exclusive Areas, implemented in all RTD layers. As defined by AUTOSAR, all locks (critical sections) are implemented via SchM calls (using the `SchM_Enter_<ModulePrefix>` and `SchM_Exit_<ModulePrefix>` functions). RTD provide only a reference implementation for the exclusive areas. Depending on the use case the developer decides which exclusive areas to implement and with what method e.g. disabling interrupts, mutexes, etc.

CHAPTER 3 Migrating from FreeRTOS™ to SAFERTOS®

SAFE**RTOS**® is a safety critical RTOS from WITTENSTEIN high integrity systems (WHIS), available pre-certified to IEC 61508 SIL3, and ISO 26262 ASILD. It is commonly used in the automotive industry and supports a wide range of NXP platforms.

SAFE**RTOS**® and the NXP S32K/G processor families are well suited for non-AUTOSAR automotive applications due to their high performance and safety credentials. SAFE**RTOS**® is fully compatible with NXP's RTD.

The advantages of SAFE**RTOS**® include but are not limited to:

- ISO 26262-6 ASIL D & IEC 61508-3 SIL 3 pre-certified Real Time Operating System;
- Dependable, deterministic Task scheduling;
- Specific Processor/Compiler combination;
- Tools to enable spatial separation;
- Easy integration into development environment;
- Smooth path to certification;
- Low cost of ownership over product lifetime;
- Multiple adoption models;
- Part of a stable and mature corporation;
- Developed by WITTENSTEIN high integrity systems, a safety systems company.

3.1 Comparing FreeRTOS™ and SAFERTOS®

3.1.1 A popular migration route

SAFE**RTOS**® is based on the functional model of the FreeRTOS kernel, a market leading embedded RTOS for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of IoT libraries suitable for use across all industry sectors.

Although SAFE**RTOS**® and FreeRTOS share a functional model, SAFE**RTOS**® is not FreeRTOS. It has been completely redesigned by WHIS' team of safety experts. WHIS engineers took the FreeRTOS kernel functional model, subjected it to a full HAZOP, identified all areas of weakness within the functional model and API, and generated a set of safety requirements. The resulting functional and safety requirements sets were put through an IEC 61508-3 SIL 3 development life cycle, the highest possible for a software only component, creating the SAFE**RTOS**® code base and Design Assurance Pack.

As the FreeRTOS kernel and SAFE**RTOS**® share the same functional model, upgrading is easy. Many WHIS customers prototype using the FreeRTOS kernel and convert to SAFE**RTOS**® at the start of their formal development phase. An experienced engineer can migrate a typical FreeRTOS application from FreeRTOS to SAFE**RTOS**® within one day. For engineers less experienced with this upgrade path, the migration would take approximately three days.

3.1.2 A Shared History

Richard Barry created FreeRTOS while an employee at WHIS. Seeing its potential, WHIS engineers worked closely with Richard. They took the FreeRTOS functional model, exposed it to a full HAZOP, identified all areas of weakness within the functional model and API, mitigated all areas of weakness, and took the resulting requirements set through an IEC 61508 SIL 3 development life cycle, the highest possible for a software only component.

In doing so SAFE**RTOS**® and its accompanying Design Assurance Pack were created: the renowned safety critical RTOS that delivers superior performance and safety critical dependability whilst consuming minimal resources.

The success of these endeavors can be judged by the fact SAFE**RTOS**® was independently certified on the first iteration by TÜV SÜD back in 2007. SAFE**RTOS**® has continuously evolved since its initial creation and the list of supported processors and toolsets is constantly expanding.

As the FreeRTOS kernel and SAFE**RTOS**® share the same functional model, upgrading is easy. Many developers prototype using the FreeRTOS kernel and convert to SAFE**RTOS**® at the start of their formal development phase.

3.1.3 Key Differences

SAFE**RTOS**® and FreeRTOS share a functional model, so feel very similar to use. There are however some key differences. Compared with FreeRTOS, SAFE**RTOS**®:

- Has been completely re-engineered to meet safety critical software needs;
- Has fewer Application Programming Interface (API) functions;
- Does more error checking;
- Returns a status code from most API calls (with other return data passed by reference);
- Strongly encourages 100% static allocation and provides no “heap” functions;
- Requires the application to supply all stack, task control block and queue buffers;
- Uses a processor’s Memory Protection Unit (MPU) by default.

Because of this, when migrating a FreeRTOS project to SAFE**RTOS**® there is some work to do to get the kernel up and running.

FreeRTOS hides much routine memory management “under the hood”, dynamically allocating stack buffers as tasks are created, allocating key kernel buffers when the kernel is started, and so on. It is possible to configure FreeRTOS to have all these things statically allocated and supplied by the application, but most people settle for the simpler approach of letting FreeRTOS do it.

FreeRTOS also provides a lot of compile-time options and allows the application designer to insert extra functionality into the kernel via strategically placed “hook macros”, so that code can be run as tasks are switched in or out, for example, or when they are deleted or created.

3.1.3.1 API Differences

WHIS offer a free Application Note that goes into some detail about the main differences between the FreeRTOS and SAFE**RTOS**® API, that is freely available from the WHIS website Download Centre www.highintegritysystems.com

For a complete description of the SAFE**RTOS**® API, the product- specific manual that ships with every licensed copy of SAFE**RTOS**® is definitive. The RTOS-defined abstract type names differ between the two platforms, as do the necessary RTOS #include files. With FreeRTOS, an application file needs to #include headers for each of the main parts of the API that it uses – task functions, queue functions, semaphore functions, etc. – while with SAFE**RTOS**®, an application file need only #include a single RTOS header file, `SafeRTOS_API.h`.

3.1.3.2 Static Allocation and the MPU

SAFE**RTOS**® requires the application engineer to supply all the buffers that are needed to maintain task and object state. For example, both FreeRTOS and SAFE**RTOS**® create a kernel "idle task" at the lowest possible task priority to ensure that there is never a situation where there is no ready task that can be scheduled. In both cases, the idle task must never block. Additionally there are mechanisms by which the idle task can be used for "background" application functions, for power saving measures or other application-specific purposes.

The idle task needs a task stack, and a task control block. FreeRTOS allocates these from heap (unless explicitly configured for static allocation) while SAFE**RTOS**® requires the application to supply these (and other) buffers. Static allocation is generally preferred in safety critical systems as it's easier to make a case that there is sufficient memory for all run-time circumstances.

Working with the MPU implies that the application designer has a conscious oversight of the exact placement in address space of all memory structures – and the kernel task and queue buffers are an important part of that. Moreover, an MPU usually imposes restrictions on the alignment and size of protection regions, and the application engineer needs the flexibility to meet these restrictions, which often means arranging things carefully to avoid inefficient use of the available space.

With FreeRTOS, it's usually enough to make sure there's plenty of free space on the heap before calling `xTaskCreate`. With SAFE**RTOS**®, you need to pre-allocate and explicitly locate a carefully sized and aligned stack buffer, and statically allocate and locate a buffer for the task TCB, then pass pointers to these structures into xTaskCreate along with the usual task parameters.

3.1.3.3 Task Privilege and Kernel Function Wrappers

Each SAFE**RTOS**® task is assigned an operating privilege, of which there are usually two: "Privileged" and "Unprivileged". "Privileged" tasks have the same basic powers as kernel code. Most CPUs that allow privileged and unprivileged modes have a restricted range of instructions available in unprivileged modes, and a limited mechanism – usually some kind of software trap, exception or interrupt – for moving between unprivileged and privileged mode operation.

It's generally regarded as good practice to code as much of an application as possible to run in unprivileged mode, and to that end, each task is provided with a set of MPU parameters that are switched in and out of a fixed range of MPU region descriptor slots whenever the task is switched in and out by the scheduler.

The SAFE**RTOS**® task creation adds an additional MPU region for each task that grants it access to its own stack, and this is automatic, although the application programmer still must ensure that the task's stack buffer conforms to the restrictions imposed by the MPU regarding alignment and size.

Unprivileged tasks also need to have execute access to the kernel API functions. Since these are usually expected to be called only from privileged mode, every API function with SAFE**RTOS**® has a privilege-escalating wrapper. Typically, this works by raising an exception of some kind – most often using a "system call" – a synchronous interrupt or trap facility provided by the CPU for this purpose. The API wrapper will raise a task's privilege level, if need be, execute the API function, then lower the privilege again if it was raised.

In SAFE**RTOS**®, we presume that application tasks will normally run unprivileged. With the FreeRTOS MPU-aware ports, tasks are presumed generally to run at kernel privilege, as they do in non-MPU ports, but tasks may optionally be created "restricted", that is, unprivileged.

3.2 Migrating from FreeRTOS™ to SAFE**RTOS**®

Two Stage Approach

The approach used to convert a FreeRTOS project into a SAFE**RTOS**® one is typically done in two basic stages:

1. Convert the project to SAFE**RTOS**®, with everything running in privileged mode.
2. "Lock down" tasks that needn't run in privileged mode.

Part 1: Converting the Project to use SAFE**RTOS**®

1. Replace the FreeRTOS kernel code files with the SAFE**RTOS**® ones;
2. Edit the linker file to export symbols required by the SAFE**RTOS**® kernel;
3. Ensure that the SAFE**RTOS**® interrupt and exception handlers are installed;
4. Create mandatory hook functions and static buffers for kernel tasks and queues;
5. Create a scheduler initialization struct and function;
6. Create required static buffers for application task TCBs, stacks, queues, etc.;
7. Convert FreeRTOS API calls to equivalent SAFE**RTOS**® ones;
8. Convert FreeRTOS-specific types to SAFE**RTOS**® ones;
9. Make all tasks privileged and set default MPU parameters.

Note that steps 1 through 6 inclusive are essentially unavoidable one-off overhead for any SAFE**RTOS**® application, whether converted from FreeRTOS or not.

Steps 7 and 8, particularly 8, can to some extent be speeded up using an IDE's search & replace functions, though beware of fundamental differences between the APIs that mean care is needed.

Part 2: Making Tasks Unprivileged

Making tasks run in an unprivileged manner varies in complexity according to the tasks. It can be as simple as marking a task unprivileged, or it can involve considerable extra work.

3.2.1 Convert the Application to use SAFE**RTOS**®

3.2.1.1 Replace the FreeRTOS Code

The first step is straightforward, we just delete everything under amazon-freertos and replace it with the SAFE**RTOS**® kernel source. Or, in the case of this web-demo version of SAFE**RTOS**®, replace it with the SAFE**RTOS**® headers and library file. Here, we'll create a folder called SAFE**RTOS**® containing subfolder kernel.

In the project options, all references in include paths to `amazon-freertos/freertos_kernel` need to be replaced by SAFE**RTOS**®/kernel. There are the expected entries in the compiler and assembler include paths under C/C++ Build/ Settings.

In a wizard-generated project like this there are also some references under C/C++ General/Paths and Symbols.

3.2.1.2 Edit the Linker File

SAFE**RTOS**® requires some named sections to be defined in, and various symbols to be exported from, the linker file in order to set up initial MPU regions to protect kernel data and to allow access to general code.

The sections and symbols required are port specific, but you can see the MPU settings (and hence the required symbols) in the kernel portable layer header file `portmpu.h`.

We can see we need a section for kernel functions called "`kernel_func`" and another for kernel data called "`kernel_data`". The kernel functions and data are placed in these by section attributes.

The kernel functions are typically placed alongside the vector table so that both can be included in the MPU region that protects them, and the kernel data are placed somewhere in RAM that can be readily aligned to the needs of the MPU. In this case, the NXP MPU doesn't have especially strict alignment requirements – regions need to be a minimum of 32 byte aligned – so the placement and size are quite flexible.

In addition to the linker sections, the linker file needs to export symbols indicating where these sections start and finish, and we also need symbols for the start and end or size of ROM and RAM. Again, from the portable layer we can see we need:

- `lnkStartFlashAddress`
- `lnkEndFlashAddress`
- `lnkStartKernelFunc`
- `lnkEndKernelFunc`
- `lnkStartKernelData`
- `lnkEndKernelData`
- `lnkRAMEnd`
- `lnkRAMStart`

3.2.1.3 Exception and interrupt handlers

SAFE**RTOS**®, like FreeRTOS, makes use of certain interrupts and exceptions. The S32K product variants needs to use the SysTick, PendSV and SVC interrupts, all of which are generally hooked into simply by using the CMSIS names for the handlers for these. If using CMSIS compliant platform support, these symbols are weakly linked and appear in the default vector table where they need to be, so that the RTOS simply has to name its own handlers according to the CMSIS convention.

In SAFE**RTOS**®, we generally achieve this by simply #defining the SAFE**RTOS**® names for the necessary handlers to be the CMSIS names instead, and we typically do this in `SafeRTOSConfig.h`. However, for this example we're using a library build of SAFE**RTOS**®, so the handler names are already fixed in the library, so we just reverse the trick and redefine the CMSIS names in the startup file where the vector table is defined, instead.

In this example project, the vector table is defined in `startup_mk64f12.c`, and we simply insert the following few lines near the beginning of that file:

```
/* SAFERTOS system tick, SVC and PendSV handlers */
#define SysTick_Handler      vTaskProcessSystemTickFromISR
#define  SVC_Handler         vSafeRTOSSVCHandler
#define PendSV_Handler       vSafeRTOSPendSVHandler
```

Unlike FreeRTOS, SAFE**RTOS**® checks the vector table entries at startup and will not run if they aren't present and correct.

3.2.1.4 Kernel hook Functions

SAFE**RTOS**® doesn't have FreeRTOS' hook macros in the kernel. Instead, it provides for a limited set of "hook functions" that, if present, are called by the kernel when, for example, a task is deleted or the idle task runs or the tick interrupt fires.

Of the various hook functions, only one is mandatory, and that is the Error Hook Function. It's a requirement of safety critical systems that they can enter a safe failure state if an unrecoverable error is detected. If the kernel detects an error condition like a corrupted task control block or an overflowed stack, then it calls the Error Hook Function, which is responsible for making the system safe overall – turning off any motors or H-bridges, for example – and must not return. For our simple example application, we'll have an error hook function that simply loops indefinitely having made information about the error accessible to the debugger.

3.2.1.5 Kernel Task Buffers, Kernel Configuration and Startup

In addition to the address of one or more hook functions, we need to pass in the address and size of a stack buffer for use by the kernel idle task. There's also a second stack buffer required for the kernel timer task. Both the idle and timer tasks have some additional parameters that must be passed in, including their own MPU parameters, and there's also a timer command queue that needs a queue buffer of sufficient size, alignment etc.

To pass in all of the parameters that are needed to configure SAFE**RTOS**®, the structure `xPORT_INIT_PARAMETERS` is defined. A pointer to this structure is used as the parameter to the `xTaskInitializeScheduler() function`.

Arguably, the most complicated part of the move to SAFE**RTOS**® with a minimal application is getting the scheduler configured and started. To avoid obfuscating the tiny demo application with so much more code, we've chosen to place the error hook function in a "hook functions" source file, and all the kernel configuration, including the stack and TCB allocations for kernel tasks, into another source file which we've named `SafeRTOSConfig.c`, since it's a sort of extension, in a sense, of the kernel configuration header file `SafeRTOSConfig.h`.

SAFE**RTOS**® API functions almost always return either `pdPASS` (to signify that the operation was successful) or an error code. This means that whenever a SAFE**RTOS**® API function is used, the caller should save the return value so that the cause of the failure can be identified. All of the error definitions can be found in the header file projdefs.h.

The error codes for the entire API are listed, making it relatively easy to work out what went wrong if the kernel didn't start, for example. Usually if this happens it's because the proper IRQ handlers haven't been set up in the vector table(s), or one or other of the static kernel task structures – stack of TCB – is mis-aligned or too small, or a non-optional hook function – usually only the Error Hook – is missing, misaligned or mis-located.

3.2.1.6 Global MPU Regions

One final thing worth doing for this particular application is to use one or other of the "spare" MPU regions (that many SAFE**RTOS**® ports provide) to set up a global, permissive region that allows access to the GPIOs that we're using to drive the on-board LEDs. This saves us having to grant explicit access to those to every task or call-back that might want to use them – the number of task- specific MPU regions is limited. If you look at the kernel configuration and startup code in SafeRTOSConfig.c, you'll see where we've done just that. There's a call to `xMPUConfigureGlobalRegion()` that sets up a "global" MPU region granting read/write access to the GPIO register space. (Which is the majority of peripheral address space in this case.)

3.2.1.7 Application Task TCBs and Stacks

We next need to set up the stack and TCB buffers for our individual application tasks, and to fill in the other task parameters. Again, this involves allocating static buffers, having a suitable size and alignment for their intended use given MPU and other port-specific considerations.

In FreeRTOS, when we create tasks, we just pass all the necessary task parameters via six arguments to the `xTaskCreate()` function, which returns either the handle of the newly created task, or pdFAIL if it can't create the task. With SAFE**RTOS**®, there are more task parameters, and some of them are port-specific, so they're passed to `xTaskCreate()` using single argument pointing to a port-specific `xTaskParameters` structure. Like most of the SAFE**RTOS**® API, `xTaskCreate()` returns either `pdPASS` or an error code, so a second argument gives the address of a task handle variable to receive the handle of the newly-created task. If you don't need the task handle, this argument can be NULL.

In our example, many of the task parameters remain the same for all four of the tasks our application creates, so we save some space and time by re-using the same parameter's structure and just changing the parameters that need to change. This isn't best-practice as it would be relatively easy to overlook some vital change – for instance, it is imperative that each task has unique stack and TCB buffers (for obvious reasons).

Here, we see that the task name and parameter differ in each case, too, as in the FreeRTOS version. Also among the task parameters are each task's privilege level and MPU region settings. As described earlier, in the first place we're going to make all tasks privileged, so none of them require any additional MPU parameters – not with this application on this target anyway – so we leave those set to zero/null.

3.2.1.8 API Function Call Changes

More or less the last thing that needs doing is for every API call, to check to see that the SAFE**RTOS**® version has the same name and whether, or how, its prototype differs. In most cases, void functions will have become functions returning the architecture's signed base type, in order to report an error or pass code. Where the equivalent FreeRTOS function also returns a value, typically the SAFE**RTOS**® version will return the value via a pointer type argument, and the function return value will usually be a status code. Some of the function names will have changed, and some of the FreeRTOS functions won't be available, so the application programmer will sometimes have to work around such omitted functions by using the underlying core functions in perhaps a slightly more long- winded way.

For the sake of form, where functions in SAFE**RTOS**® return a value that we discard, the function should generally be prefixed with a void cast to indicate that this is deliberate.

3.2.1.9 Type Name Changes

As well as some of the API functions having different names, many of the port-layer defined type names also differ. Some are common to both FreeRTOS and SAFE**RTOS**®, but many of the longer- established ones are not.

If developing FreeRTOS applications with a specific view to converting to SAFE**RTOS**®, picking the type name options that are shared between the two, or are at least readily search-and-replaceable, will save some time.

3.2.1.10 Wrapping Up

After following the steps to now, we should have arrived at a point where the application will compile again and will run on the target system essentially the same as the FreeRTOS version.

If, as often happens in practice, the kernel refuses to start at this point, or a task creation call fails, the error codes returned should help to identify the problem. These codes are listed in the kernel header file projdefs.h. As an example, the function that starts the kernel, `xTaskStartScheduler()`, is not supposed to return. If, rather than starting the scheduler, it instead returned the value -27, we could refer to projdefs.h and see that this is the code for `ERROR_IN_VECTOR_TABLE`, which means that the kernel start-up configuration checks have determined that one of the interrupt vectors essential to the SAFE**RTOS**® kernel is misconfigured in some way.

3.2.2 Making Application Tasks Unprivileged

In the final stage of conversion, we'll look at "locking down" the privileged tasks so that they can run unprivileged. We've already granted access for any task to the GPIO registers we're using, via a "global MPU region." Now let's see what else the tasks might need to have explicit access granted for if they are converted to unprivileged.

One worthwhile exercise is simply to mark one or other of the tasks as unprivileged and then load and run the application and note that (in this case) it winds up in the MPU fault handler. You can examine registers with the debugger to identify the faulting address, and you could try using the debugger to force a return from the handler as a quick way to identify the instruction that gave rise to the MPU fault. (Though with the fault uncorrected, the CPU will likely machine-check at that point and probably drop the debug connection.)

In this case, we can see we need the individual LED tasks need access to the shared "brightness request" array, and also to the handle of the mutex that is used to control access to said array. The mutex buffer, on the other hand, is only ever accessed via privileged kernel code so can remain out of reach of the tasks.

By the same token, if we'd used a queue to pass information between the tasks, rather than shared memory, we would need no additional MPU settings to grant access as the queue itself would be accessed only in privileged mode (by the kernel API) and the queue handle could be communicated via each task's parameter or thread-local storage pointer.

Recall that earlier on in this document, we commented that the shared memory communication between tasks was less than ideal but allowed us to illustrate a technique? Well, this is it. Given a need to access a shared region of RAM, how do we do it?

We go back to the linker file and create another named section, complete with exported start and size symbols, somewhat similar to the kernel sections we looked at earlier. Then we add placement attributes to the code. Syntax between different compilers vary, but with GCC, we typically use:

```
__attribute__ ( ( section ( "section_name")))
```

To specify the placement of a variable or function.

Finally, we use the symbols from the linker file to define MPU region specifications that can be added to task MPU parameters in order to gain access.

Remember that the exported linker symbols are addresses, so we can take the symbol's address after declaring it as an extern variable, or we can declare it as an array type whereby its unadorned name will be an address automatically.

CHAPTER 4 Using RTD and SAFE**RTOS**® in a Safety Critical Application

RTD is one of several offerings in the S32 software enablement to maximize software reuse across processor platforms and broaden AUTOSAR access for mass-market developers.

The use of an ISO 26262-compliant development process for RTD allows any automotive or industrial application to implement functional measures in both hardware and software. Along with the safety compliant RTD, SAFE**RTOS**® delivers source code and a Design Assurance Pack (DAP) tailored to the processor and compiler being used. The DAP guarantees easy installation of SAFE**RTOS**® integration into the development environment being used and that there will be no hardware retesting. Together SAFE**RTOS**® and RTD reduce development costs and accelerate time to market while achieving required safety targets.

CHAPTER 5 Conclusion

As software-defined vehicles emerge, automotive software has become the central development challenge to achieve safety requirements. The RTD software offering helps customers address this challenge while covering the S32 Automotive platform's common architecture and functional safety.

SAFE**RTOS**® provides responsive, feature rich functionality within a networked environment with a 100% success rate in certifying SAFE**RTOS**® in applications, making it ideal for automotive projects.

Together the RTD software and SAFE**RTOS**® make a compelling solution for automotive developers.

WITTENSTEIN

RESOURCES

Video Tutorials on Using the SAFE**RTOS**® Demo

WHIS offer a variety of useful videos on the WHIS YouTube Channel. In particular, the playlist "Using the SAFE**RTOS**® Demo" is a three-part series of videos that will help walk you through how to use the free demos on the WHIS website.

- https://www.youtube.com/user/HighIntegritySystems/videos

FreeRTOS

Developed in partnership with the world's leading chip companies over a 15-year period, and now downloaded every 170 seconds, FreeRTOS is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors. FreeRTOS is stewarded by Amazon Web Services (AWS). Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of IoT libraries suitable for use across all industry sectors. FreeRTOS is built with an emphasis on reliability and ease of use.

- https://www.freertos.org/
- https://github.com/FreeRTOS

SAFE**RTOS**®

SAFE**RTOS**® from WITTENSTEIN high integrity systems is a pre-certified safety Real-Time Operating System (RTOS) for embedded processors. It delivers superior performance and pre-certified dependability, whilst utilizing minimal resources. 8-hour binary demos (simply reset the board at the end of 8 hours to restart the execution) are freely available from the WHIS website, or 30 day evaluation packages are available by contacting the sales team.

- https://www.highintegritysystems.com/safertos/

RTD

Real-Time Drivers (RTD) combine AUTOSAR MCAL drivers with a set of Complex Device Drivers (CDDs) and are designed for automotive processors featuring Arm® Cortex®-M cores. RTD combine elements of NXP's SDK and MCAL drivers into a single software product that provides full IP and feature coverage for both AUTOSAR and non-AUTOSAR applications. Developed as a Safety Element out of Context (SEooC) using an ISO2626 compliant process, RTD enhance and extend the previous generation of drivers and are suitable for use in applications up to ASIL D level. NXP provides RTD for System Basis Chips (SBCs), Ethernet and CAN PHY devices delivered as separate packages.

- www.nxp.com/RTD

CONTACT INFORMATION

User feedback is essential to the continued maintenance and development of SAFE**RTOS**®. Please provide all software and documentation comments and suggestions to the most convenient contact point listed below.

Contact WITTENSTEIN high integrity systems

Address: WITTENSTEIN high integrity systems
Brown's Court, Long Ashton Business Park
Yanley Lane, Long Ashton
Bristol, BS41 9LB
England

Phone: +44 (0)1275 395 600
Email: support@HighIntegritySystems.com

Website www.HighIntegritySystems.com

Contact NXP

Address: NXP Semiconductors
High Tech Campus 60
5656 AG Eindhoven
The Netherlands

Phone: +31 40 272 9999
Support: www.nxp.com/support

Website www.nxp.com