Porting from RTX64 4.x to MaxRT wRTOS®

USER GUIDE

IntervalZero

MaxRT wRTOS

Porting from RTX64 4.x to MaxRT wRTOS

IZ-DOC-MaxRT-wRTOS-007

IntervalZero

200 Fifth Avenue, FL 6, STE 6020
Waltham, MA 02451
Phone: 781-996-4481
www.intervalzero.com

Contents

About this Porting Guide **1**

Product Comparison **2**

Product Packages 2

RTX64 Vision Integration 3

KINGSTAR EtherCAT MainDevice Integration 3

Licensing, Activation, and Core Configuration **4**

Product Codes 4

Activating Product Components and Configuring Cores 6

Dongle Activation Utilities 7

License Types 7

Configuring and Controlling the Runtime **8**

Configuring and Controlling RTX64 8

Configuring and Controlling wRTOS 8

Tool Comparison **10**

Activating, Configuring, and Controlling Components 10

Analyzing and Logging 11

Controlling Applications and Viewing Output 12

Networking 13

EtherCAT Devices 13

Measuring Performance and Latency 14

Project Settings **15**

API Headers and Libraries 15

Managed Libraries 19

SDK Code Changes **21**

New APIs 21

New Real-Time APIs 21

New Winsock APIs ... 39
New Configure and Control APIs ... 39
New GigE Vision (RTGV) APIs ... 42
New Managed APIs ... 42
Enhanced APIs ... 60
Enhanced Real-Time APIs ... 61
Breaking API Changes ... 62
Breaking Changes to Real-Time APIs ... 62
Breaking Changes to Configure and Control (RTFW) APIs ... 64
Breaking Changes to Managed APIs ... 65
Breaking Changes to Error Codes ... 67
Removed/Deprecated APIs ... 68
Removed/Deprecated Real-Time APIs ... 68
Removed/Deprecated Winsock APIs ... 73
Removed/Deprecated Configure and Control (RTFW) APIs ... 73
Removed/Deprecated Managed APIs ... 77
Removed Error Codes ... 81
Networking ... **82**
Network Components ... 82
RTX64 Network Components ... 82
wRTOS Network Components ... 83
Configuring and Controlling the Network ... 85
Configuring and Controlling the RTX64 Network ... 85
Configuring and Controlling the wRTOS Network ... 85
Porting a NAL Driver to the NL2 ... 87
Header and Library Requirements ... 88
Rework the Startup Logic ... 88
Rework the Shutdown Logic ... 88
Use NL2 Function Pointers instead of Direct Function Calls ... 89
Rework the Interrupts Handling ... 90

Rework the Link Status Monitoring Logic 91
Rework the Frame Transmission Logic 92
Rework the Frame Reception Logic 94
Remove Unnecessary Locks 95
Replace the RtndIoctl Function 96
Replace the RtndRequest Function 98
Implement Frame Buffer Allocation Functions 99
Support **100**
Contacting Technical Support by Phone 100
Online Resources 101

About this Porting Guide

This guide details the major differences between the RTX64 4.x and MaxRT wRTOS 1.x products, and describes the steps required to port an existing RTX64 application to MaxRT wRTOS.

This guide contains active links to the IntervalZero website and product documentation. To take full advantage of this guide, we recommend having an Internet connection and installation of wRTOS.

1

Product Comparison

Like RTX64, the MaxRT wRTOS Subsystem is architected and built as a set of 64-bit binaries and can run on 64-bit Windows operating systems; taking full advantage of 64-bit optimizations and features. See the *RTX64/MaxRT wRTOS Comparison Guide*, available from the Customer Center, for a comparison of RTX64 and wRTOS products.

Product Packages

RTX64 and MaxRT wRTOS components are available in these product packages:

RTX64

- Runtime (can be installed silently) includes these components:
  - Network Abstraction Layer (NAL)
  - TCP/IP Stack (must be purchased separately)
- SDK (can be installed silently)
- Runtime Merge Modules

MaxRT wRTOS

- wRTOS Runtime (can be installed silently) includes this component:
  - Network Link Layer (NL2, replaces the NAL in RTX64)
  - Virtual Network
  - Network Relay

  These product packages and components require wRTOS Runtime and can be installed with wRTOS Runtime but must be purchased separately:

  - Basic Networking includes these components:
    - TCP/IP Stack
    - GigE Vision
  - Fieldbuses includes this component:

- E-CAT
- SDK (can be installed silently)
- Runtime Merge Modules

RTX64 Vision Integration

GigE Vision provides functionality for controlling GigE Vision Cameras within the wRTOS environment. It also allows you to discover cameras on the network, query different camera configuration settings (such as image width, height, pixel formats, etc.), and acquire image data.

In RTX64, Vision is a separate, purchasable product. GigE Vision is integrated with MaxRT wRTOS and can be installed with wRTOS Runtime. However, to enable this feature, you must purchase and activate the wRTOS GigE Vision and wRTOS Basic Networking packages (see Product Packages above). Contact IntervalZero Sales to purchase product licenses.

wRTOS SDK includes an RtGigEVision sample program that shows the usage of the RtGigEVision APIs, and how one can use them to control and receive data from a GigE Vision Camera.

> **Note:** OpenCV is not included with wRTOS. You can download an OpenCV version that supports wRTOS from the Support Site.

KINGSTAR EtherCAT MainDevice Integration

MaxRT wRTOS incorporates the KINGSTAR EtherCAT master functionality in a wRTOS E-CAT component in the wRTOS Fieldbus product package (see above).

MaxRT wRTOS SDK includes E-CAT Fieldbus (RTECAT) functions and data types related to the wRTOS E-CAT master used for EtherCAT communication.

- To build RTSS applications, include the header **RtecatApi.h** in your project and link with the library **RtecatApi.lib**.
- To build Windows applications, include the header **RtecatApi.h** in your project and link with the library **RtecatApi_W64.lib**.

See the *Porting from KINGSTAR 4.x to wRTOS* guide for more KINGSTAR-specific porting information.

2

Licensing, Activation, and Core Configuration

This chapter compares product codes, licensing concepts, and activation and configuration methods in RTX64 and MaxRT wRTOS.

Product Codes

RTX64 and MaxRT wRTOS use different product codes to identify product components, as listed below.

RTX64 Product Codes

Product code	Description
BLD64	Rebuild support.
RTX64	The RTX64 Subsystem.
SDK64	The RTX64 Software Development Kit.
TCP64	The RTX64 TCP/IP Stack.

wRTOS Product Codes

Product code	Description
WBLD64	Allows you to build applications. This feature does not include debugging tools.

Product code	Description
WECR64	wRTOS E-CAT Cable Redundancy allows for the EtherCAT MainDevice to support Cable Redundancy.
WEHC64	wRTOS E-CAT Hot Connect allows the EtherCAT MainDevice to support Hot Connect.
WEHST64	wRTOS E-CAT High Speed Timer allows the EtherCAT MainDevice to use a High-Speed Timer.
WEMM64	wRTOS E-CAT Multiple MainDevices supports running multiple EtherCAT MainDevices. These require the wRTOS Fieldbus (WFBS64) component package.
WFBS64	wRTOS Fieldbus supports the use of the EtherCAT MainDevice and future fieldbuses.
WNET64	wRTOS Basic Networking supports the use of TCP/IP Stack applications above the NL2.
WRTOS64	wRTOS Runtime includes the Real-time Subsystem and Network Link Layer (NL2), Network Relay, and Virtual Network. See Runtime Editions for a list of the wRTOS Runtime editions and the maximum number of Real-Time Subsystem (RTSS) cores they support.
WSDK64	wRTOS Software Development Kit (SDK) is used to develop, build, and debug applications.
WVIS64	wRTOS GigE Vision allows the use of GigE and GeniCam drivers. This component also requires the wRTOS Basic Networking (WNET64) component package.

Activating Product Components and Configuring Cores

Product components must be activated with a valid license before you can use them. You can lock product components to a machine or IntervalZero-provided dongle. RTX64 and MaxRT wRTOS provide different activation options, as described below.

RTX64 Activation and Configuration

RTX64 provides these options for activating product components and configuring system cores:

- Use RTX64 Activation and Configuration GUI utility immediately after RTX64 installation, to activate product components, view license information, and assign system cores to Windows and RTX64.
- Use the Rtx64ActivationUtil.exe command line utility to silently activate product components and assign system cores to Windows and RTX64.

For more information, see the RTX64 Help, *RTX64 Runtime Install Guide*, or *RTX64 Deployment Guide*.

wRTOS Activation and Configuration

MaxRT wRTOS provides these options for activating product components and configuring system cores:

- wRTOS Settings:
    - On the Licensing and Activation page, which appears immediately after MaxRT wRTOS installation, you can activate product components and view license information.
    - On the Core Configuration page, you can view the total system cores available, set or change the RTSS configuration, and view the core assignments for wRTOS processes and components. When wRTOS Runtime is installed, you can assign system cores to Windows or wRTOS.
- You can use the MaxRTActivationUtil.exe command line utility to silently activate wRTOS components and configure system processors.

For more information, see the MaxRT wRTOS Help, *MaxRT wRTOS Installation Guide*, or *MaxRT wRTOS Deployment Guide*.

Dongle Activation Utilities

IntervalZero provides standalone dongle activation utilities for RTX64 and MaxRT wRTOS that allow you to separate the network activation process. Instead of activating dongles on the production floor, you can activate them at another location and provide importable license files and dongles to your production environment.

- RTX64: Use the IntervalZero Dongle Activation Utility to license IntervalZero products to an IntervalZero-provided dongle. You can download the Dongle Activation Utility from the RTX64 space on the IntervalZero Support Site.
- MaxRT wRTOS: Use the MaxRT Dongle Activation Utility, available from the MaxRT wRTOS space on the IntervalZero Support Site.

License Types

A license string is generated from the Activation Key. This is the process by which one or more product components are activated, depending on the components you have purchased. Licenses are provided by the IntervalZero License Server and are node-locked, meaning they can only be used on a single machine or dongle.

RTX64 License Types

RTX64 offers these license types:

- An Evaluation license is a temporary license with an expiration date.
- A Retail license is perpetual and does not expire.

wRTOS License Types

MaxRT wRTOS offers these license types:

- An Evaluation license is a temporary license with an expiration date.
- Multiple Retail license types:
    - A perpetual Retail license does not expire.
    - A time-limited Retail license has an expiration date.

3

Configuring and Controlling the Runtime

RTX64 and MaxRT wRTOS provide tools for configuring and controlling the Runtime and its dependent components.

Configuring and Controlling RTX64

RTX64 provides a Control Panel with settings for configuring the default behavior for the Real-Time Subsystem (RTSS), RTSS applications, and networking features. It also allows you to view information on installed and licensed components and launch the Analyzer utility. Using RTX64 Control Panel, you can also control (Start, Stop, Restart) the Subsystem and other Runtime components.

For more information on RTX64 Control Panel, see the RTX64 Help and *Configuring RTX64* user guide.

Configuring and Controlling wRTOS

MaxRT wRTOS also provides a Control Panel, but it has a different purpose than the RTX64 Control Panel. wRTOS Control Panel allows you to control (Start, Stop, Restart) the wRTOS Subsystem and other Runtime components, including NL2, TCP/IP, and E-CAT. It also provides shortcuts to launch several other wRTOS tools. wRTOS Control Panel does not provide settings for configuring the Runtime and other components. wRTOS configuration settings are available in the wRTOS Settings tool.

wRTOS Settings allows you to activate product components, configure RTSS and component cores, and configure default behavior for the Real-Time Subsystem (RTSS), RTSS applications, networking, fieldbuses, and other features. You can also access other wRTOS tools and resources. If you've purchased and installed the wRTOS SDK, wRTOS Settings will display information on the Visual Studio versions installed on the system.

Zeroing Memory on Allocation

The memory allocated by **malloc** or similar C-Runtime functions is not initialized to zero according to the C99 specification, which Windows follows. RTX64 provides a *Zero memory on allocation* setting, which is enabled by default, in the Control Panel. wRTOS provides a similar setting in wRTOS Settings, though it's disabled by default in wRTOS.

4

Tool Comparison

RTX64 and MaxRT wRTOS provide an extensive set of tools to assist with various tasks. This chapter compares the tool sets for each product.

Activating, Configuring, and Controlling Components

Functionality	RTX64 4.x Tool	wRTOS 1.x Tool
Activate product components	Activation and Configuration utility `Rtx64ActivationUtil.exe` for silent activation	Settings (Licensing and Activation page) Use the `MaxRTActivationUtil.exe` command line utility to activate wRTOS components silently.
Configure cores and set the RTSS boot configuration	Activation and Configuration utility `Rtx64ActivationUtil.exe` for silent core configuration.	Settings (Core Configuration page) Use the `MaxRTActivationUtil.exe` command line utility to configure cores silently.

Functionality	RTX64 4.x Tool	wRTOS 1.x Tool
Configure Runtime behavior	Control Panel	Settings Configure default behavior for the Real-Time Subsystem (RTSS), RTSS applications, networking, fieldbuses, and other features. You can also view installed Visual Studio versions (requires wRTOS SDK) and configure remote access.
Control (start, stop, restart) Runtime components	Control Panel	Control Panel Settings
Set up remote services for debugging RTSS applications remotely in Visual Studio	RTX64RemoteConfig	Remote Config

Analyzing and Logging

Functionality	RTX64 4.x Tool	wRTOS 1.x Tool
Analyze and log system status	Analyzer	Analyzer
Display the current Subsystem state	System Tray	System Tray
View and record debug or log messages generated during the execution of real-time applications.	Not available	Message Viewer

Functionality	RTX64 4.x Tool	wRTOS 1.x Tool
Trace RTSS application behavior by recording significant events that occur during their execution	Monitor	Monitor
View and analyze application behavior in monitoring session data to see elements in your application that require optimization	Tracealyzer	Not available

Controlling Applications and Viewing Output

Functionality	RTX64 4.x Tool	wRTOS 1.x Tool
Run an RTSS process	RtssRun Task Manager	RtssRun Task Manager
Terminate an RTSS process forcibly	RtssKill	RtssKill
Display and control active RTSS processes and Windows applications linked to RTSS processes on the system	Task Manager	Task Manager
Display and log print messages from all RTSS applications and RTDLLs	Server	Server
Display information on RTSS processes and their associated objects, like events, semaphores, and loaded RTDLLs	Objects	Objects

Functionality	RTX64 4.x Tool	wRTOS 1.x Tool
Display local memory allocation spaces (MSpaces) for all RTSS processes (optionally including internal system processes and proxy processes)	MSpaces	MSpaces
Stamp an RTSS or RTDLL binary with licensing information so it can run on the Subsystem	StampTool	StampTool

Networking

Functionality	RTX64 4.x Tool	wRTOS 1.x Tool
Diagnose and configure network connections to other computers	Network Utilities	Network Utilities (RtArp, RtIpConfig, RtPing, and RtRoute)
Measure latency in a Network Link Layer (NL2) driver	Not available	Network Response Timer Measurement

EtherCAT Devices

Functionality	RTX64 4.x Tool	wRTOS 1.x Tool
Configure the EtherCAT devices (SubDevices) connected to the E-CAT MainDevice within an E-CAT component instance and check their status.	Not available	E-CAT Configuration Tool

Functionality	RTX64 4.x Tool	wRTOS 1.x Tool
Import EtherCAT SubDevice Information (ESI) files for your EtherCAT hardware devices and save the ESI data into the E-CAT component database. This enables the E-CAT component to connect to and interact with your EtherCAT hardware devices.	Not available	E-CAT ESI Import Tool

Measuring Performance and Latency

Functionality	RTX64 4.x Tool	wRTOS 1.x Tool
Measure timer latency	System Response Timer Measurement (SRTM) Kernel System Response Timer Measurement (KSRTM)	System Response Timer Measurement (SRTM) is a tool that measures timer latency observed by an application. The source for this application is provided as part of our Samples. Kernel System Response Timer Measurement (KSRTM) is a Windows driver and utility that measures HAL-level timer latencies on the RTSS cores and provides reports and histograms of the results.
Compare latency between Windows and RTSS cores	Latency View	Latency View

5

Project Settings

RTX64 and MaxRT wRTOS provide Visual Studio application templates that can be used to create a real-time application (RTSS) executable or real-time DLL (RTDLL). The RTX64 SDK and wRTOS SDK include different sets of headers and libraries.

> **Note:** Network functionality was rearchitected for wRTOS 1.0. Network-related APIs in RTX64 4.x and previous versions are incompatible with wRTOS 1.0.

API Headers and Libraries

API set	RTX64 headers and libraries	wRTOS headers and libraries
Real-Time APIs (RTAPI)	**Headers:** RtssApi.h, Rtapi.h **Libraries:** Rtx_Rtss.lib (RTSS), RtApi.lib (Windows)	**Headers:** RtssApi.h, RtApi.h **Libraries:** RtApi.lib (Windows), Startup.lib (RTSS) Most RTX64 RTAPI APIs are source code compatible.
Debug Message (RTDBG) APIs	Not available	**Header:** RtdbgApi.h **Libraries:** RtdbgApi_W64.lib (Windows), RtdbgApi.lib (RTSS)

API set	RTX64 headers and libraries	wRTOS headers and libraries
E-CAT Fieldbus (RTECAT) APIs	Not available	**Header:** RtecatApi.h **Libraries:** RtecatApi_W64.lib (Windows), RtecatApi.lib (RTSS)
Filter Driver (RTND) APIs	**Header:** RtnApi.h **Library:** RtTcpip.lib	Incompatible with wRTOS. The wRTOS Network Link Layer (NL2) replaces the RTX64 NAL. See these API sets: • NL2 Filter Driver (RTNDFLT) APIs • TCP/IP Filter Driver (RTTCPIPFLT) APIs
IP Helper APIs	**Headers:** ws2tcpip.h, iphlpapi.h **Library:** Rtx_RtTcpip.lib	**Headers:** ws2tcpip.h, iphlpapi.h **Library:** Rttcpip.lib (RTSS)
NAL (RTNAL) APIs	**Headers:** rtnapi.h, RtNalApi.h **Library:** RtNal.lib	Incompatible with wRTOS. The wRTOS Network Link Layer (NL2) replaces the RTX64 NAL.
Native Framework (RTFW) APIs	**Header:** RtfwAPI.h **Library:** RtfwAPI.lib	**Header:** RtfwApi.h **Library:** RtfwApi.lib (Windows)

API set	RTX64 headers and libraries	wRTOS headers and libraries
Network (RTN) APIs	**Header:** rtnapi.h **Libraries:** RTX64Nal.lib (to use with NAL), RtTcpip.lib (to use with TCP/IP)	Incompatible with wRTOS. The wRTOS Network Link Layer (NL2) replaces the RTX64 NAL. See these API sets: • NL2 (RTNL2) APIs • NL2 Filter Driver (RTNDFLT) APIs • TCP/IP (RTTCPIP) APIs • TCP/IP Filter Driver (RTTCPIPFLT) APIs
Network Device (RTND) APIs	**Headers:** RtNalApi.h, RtnApi.h **Library:** RTX64Nal.lib	Incompatible with wRTOS. The wRTOS Network Link Layer (NL2) replaces the RTX64 NAL.
NIC Driver (RTND) APIs	Not available	**Header:** Rtnd.h
NL2 (RTNL2) APIs	Not available	**Header:** Rtnl2Api.h **Library:** Rtnl2Api.lib
NL2 Filter Driver (RTNDFLT) APIs	Not available	**Header:** Rtnd.h
TCP/IP (RTTCPIP) APIs	Not available	**Header:** RttcpipApi.h **Library:** Rttcpip.lib, RttcpipApi.lib (start/stop component functions)

API set	RTX64 headers and libraries	wRTOS headers and libraries
TCP/IP Filter Driver (RTTCPIPFLT) APIs	Not available	**Header:** Rttcpipflt.h
Variable Database (RTVDB) APIs	Not available	**Header:** RtvdbApi.h **Libraries:** RtvdbApi_W64.lib (Windows), RtvdbApi.lib (RTSS)
Vision (RTGV) APIs	**Header:** RtGVApi.h **Library:** RtGigEVision.lib	**Header:** RtgvApi.h **Library:** RtgvApi.lib (RTSS)
Windows Driver IPC APIs (RTKAPI)	**Header:** RtkApi.h **Library:** RtkApi.lib	**Header:** RtkApi.h **Library:** RtkApi_W64.lib (Windows)
Windows Supported APIs	**Header:** windows.h **Library:** Rtx_Rtss.lib	**Header:** windows.h **Library:** wRTOS_rtss.lib (RTSS)
Winsock APIs	**Headers:** Winsock2.h, WS2Tcpip.h **Library:** RtTcpip.lib	**Headers:** ws2tcpip.h, winsock2.h **Library:** Rttcpip.lib

Managed Libraries

Namespace	RTX64 library	wRTOS library
Real-Time APIs (RTAPI)	IntervalZero.RTX64.dll	IntervalZero.MaxRT.wRTOS.dll
Config	IntervalZero.RTX64.dll	IntervalZero.MaxRT.wRTOS.dll IntervalZero.MaxRT.NL2.Config.dll (NL2) IntervalZero.MaxRT.Ecat.Config.dll (E-CAT) IntervalZero.MaxRT.Tcpip.Config.dll (TCP/IP)
Control	IntervalZero.RTX64.dll	IntervalZero.MaxRT.wRTOS.dll IntervalZero.MaxRT.NL2.Control.dll (NL2) IntervalZero.MaxRT.Ecat.Control.dll (E-CAT) IntervalZero.MaxRT.Tcpip.Control.dll (TCP/IP)
DebugMessage	Not available	IntervalZero.MaxRT.Libraries.dll
Ecat	Not available	IntervalZero.MaxRT.Ecat.dll IntervalZero.MaxRT.Ecat.Config.dll IntervalZero.MaxRT.Ecat.Control.dll
Monitor	IntervalZero.RTX64.dll	IntervalZero.MaxRT.wRTOS.dll

Namespace	RTX64 library	wRTOS library
NL2	Not available	IntervalZero.MaxRT.NL2.dll
		IntervalZero.MaxRT.NL2.Config.dll
		IntervalZero.MaxRT.NL2.Control.dll
Tcpip	IntervalZero.RTX64.dll	IntervalZero.MaxRT.Tcpip.dll
		IntervalZero.MaxRT.Tcpip.Config.dll
		IntervalZero.MaxRT.Tcpip.Control.dll
Tools (misc.)	IntervalZero.RTX64.dll	IntervalZero.MaxRT.Tools.dll
VariableDatabase	Not available	IntervalZero.MaxRT.Libraries.dll

6

SDK Code Changes

This topic outlines the APIs included in RTX64 4.x SDKs that were enhanced, underwent breaking changes, or were removed in wRTOS 1.x SDKs.

IN THIS CHAPTER:

- New APIs
- Enhanced APIs
- Breaking API Changes
- Removed/Deprecated APIs

New APIs

This section lists the new APIs in wRTOS 1.x SDKs that are not available in RTX64 4.x SDKs.

New Real-Time APIs

New Real-Time (RTAPI) APIs

- Added RTAPI function **RtGetDefaultIdealProcessorNumber**, which returns the default ideal processor number that will be assigned to any RTSS process that is not assigned a specific process affinity mask or ideal processor. (1749)
- Added RTAPI function **RtIsPerformanceCore**, which queries whether the specified core is a performance core. (9317)
- Added RTAPI function **RtIsComponentInstalled**, which returns the install status and specific version information of the specified product component. (4424)
- Added structure RT_COMPONENT_INFO, which holds the version and readability information of installable components used by functions **RtIsComponentInstalled** and **RtfwIsComponentInstalled**. (4424)

- Added RTAPI function **RtGetLicenseStatus**, which returns whether the specified version of a product code is installed and has a valid license. (914, 8922)
- Added enumerators to enumeration **RT_FEATURE_LICENSE_STATUS status** in structure **RT_LICENSE_INFO** to support time-limited licenses. (13627):
    - **RT_FEATURE_STATUS_RETAIL_VALID** indicates the license is a valid retail license (with or without an expiration date).
    - **RT_FEATURE_STATUS_RETAIL_EXPIRED** indicates the license is an expired retail license.
    - **RT_FEATURE_STATUS_EVAL_VALID** indicates the license is a valid evaluation license.
    - **RT_FEATURE_STATUS_RETAIL_INVALID_HOST_ID** indicates the license is a dongle-based retail license for which the host ID is not present on the system.
- Added field *ProcessFlags* to structure **RTPROCESS_INFORMATION**. This new field is bit-mask flag value that contains information about a running process. (10048)
- Added new error codes that can be returned by function **RtMonitorChangeState** (1530):
    - RT_ERROR_MONITORING_ENABLED An operation attempted to enable monitoring, but monitoring was already enabled.
    - RT_ERROR_MONITORING_DISABLED Monitoring was disabled when an operation was attempted that required it to be enabled.
    - RT_ERROR_MONITORING_STARTED Monitoring was started when an operation was attempted that required it to be stopped.
    - RT_ERROR_MONITORING_STOPPED Monitoring was stopped when an operation was attempted that required it to be started.
    - RT_ERROR_MONITORING_PAUSED Monitoring was paused when an operation was attempted that required it to be running.

New E-CAT (RTECAT) APIs

- Added new functions and data types for E-CAT (RTECAT) APIs. (876)
- Added new functions for E-CAT component (9064):
  - Function **RtecatGetComponentStatus** gets the E-CAT component status.
  - Function **RtecatGetInstanceStatus** gets an E-CAT MainDevice instance status.
  - Function **RtecatIsComponentLicensed** checks if the E-CAT component license is available on the system.
  - Function **RtecatStartComponent** starts all configured E-CAT MainDevice instances.
  - Function **RtecatStartInstance** starts an E-CAT MainDevice instance.
  - Function **RtecatStopComponent** stops all running E-CAT MainDevice instances.
  - Function **RtecatStopInstance** stops a running E-CAT MainDevice instance.

New Debug Message (RTDBG) APIs

- Added new functions and data types for sending messages to Message Viewer (393):
  - Function **RtdbgCloseBasicChannel** closes the Basic message channel for the current process.
  - Function **RtdbgCloseChannel** closes a Standard or Trace message channel.
  - Function **RtdbgCreateStandardChannel** creates a Standard message channel. This channel is identified by the combination of *VendorId*, *ProductCode*, and *ChannelId*. If a channel with the same combination already exists, it will be opened.
  - Function **RtdbgCreateTraceChannel** creates a Trace message channel. This channel is identified by the combination of *VendorId*, *ProductCode*, and *ChannelId*. If a channel with the same combination already exists, it will be opened.
  - Function **RtdbgGetBasicCategoryName** gets the display name of a message category.
  - Function **RtdbgGetBasicMessage** gets the first Basic message in the queue.
  - Function **RtdbgGetCategoryFilter** reads the current category filter value. This value is a bitmap indicating which categories are enabled.
  - Function **RtdbgGetIndexFilter** reads the current index filter value for a trace source, which is an array containing a set of four short values indicating which indexes are enabled for this source. If the first short value is set to -1, all indexes are enabled. If the value is set to **RTDBG_FILTER_DISABLED**, the filter is disabled and all indexes are allowed.
  - Function **RtdbgGetMessage** gets the first Standard or Trace message in the queue.
  - Function **RtdbgGetSeverityFilter** reads the current severity filter value. This value is a bitmap indicating which severity levels are enabled.

- Function **RtdbgGetSourceFilter** reads the current source filter value. This value is a bitmap indicating which sources are enabled in the channel.
- Function **RtdbgGetTypeFilter** reads the current type filter value. This value is a bitmap indicating which types are enabled in the channel.
- Function **RtdbgOpenBasicChannel** opens the Basic message channel for the current process.
- Function **RtdbgSendBasicMessage** sends a Basic message.
- Function **RtdbgSendBasicMessageIsr** sends a Basic message from an Interrupt Service Routine (ISR).
- Function **RtdbgSendMessage** sends a Standard or Trace message.
- Function **RtdbgSendMessageIsr** sends a Standard or Trace message from an Interrupt Service Routine (ISR).
- Function **RtdbgSetBasicCategoryName** sets the display name for a message category.
- Function **RtdbgSetCategoryFilter** writes the category filter value. This value is a bitmap indicating which categories are enabled.
- Function **RtdbgSetIndexFilter** writes the index filter value for a trace source. A set of four short values indicating which indexes are enabled for this source. To enable all indexes, set the first short value to -1. To enable all indexes and disable the filter, set the value to **RTDBG_FILTER_DISABLED**.
- Function **RtdbgSetSeverityFilter** writes the severity filter value. This value is a bitmap indicating which severity levels are enabled.
- Function **RtdbgSetSourceFilter** writes the source filter value. This value is a bitmap indicating which sources are enabled in the channel.
- Function **RtdbgSetTypeFilter** writes the type filter value. This value is a bitmap indicating which types are enabled in the channel.
- Function **RtdbgSynchronizeIsrCounter** resets the offset used to timestamp Interrupt Service Routine (ISR) messages.
- Data type **RTDBG_DATATYPE** identifies the type of a value.
- Data type **RTDBG_PRINTFORMAT** identifies the format used to print a value.
- Data type **RTDBG_SEVERITY** identifies severity of a Basic message.
- Data type **RTDBG_BASIC_MESSAGE_HEADER** specifies the header of a Basic message.
- Data type **RTDBG_CHANNEL** represents a handle to a channel.
- Data type **RTDBG_EXTENDED_MESSAGE_HEADER** specifies the header of a Standard or Trace message.

New NL2 (RTNL2) APIs

- Added new functions and data types for mapping the UDP ports to receive queues (3173):
  - Data type **RTNL2_UDP_PORT_DISPATCH_RULE** specifies a UDP Port Dispatch rule.
  - Function **Rtnl2AddUdpPortDispatchRule** adds a rule to dispatch incoming packets to a given Physical Receive Queue based on the value of their UDP Destination Port.
  - Function **Rtnl2DeleteUdpPortDispatchRule** deletes a dispatch rule previously set up by **Rtnl2AddUdpPortDispatchRule**.
  - Function **Rtnl2GetUdpPortDispatchRules** gets the list of UDP Port dispatch rules currently set up in the interface.
- Added new functions and data types to support the Credit-Based Shaper functionality (12807)
  - Structure **RTNL2_CBS_PARAMS** describes Credit-Based Sharper parameters.
  - Function **Rtnl2SetPhysicalTxQueueCbsParams** sets the Credit-Based Shaper parameters on a given Physical Transmit Queue.
- Added new functions and data types for mapping PCP (Priority Code Point) values to receive queues (3174):
  - Function **Rtnl2AddPcpDispatchRule** adds a rule to dispatch incoming packets to a given Physical Receive Queue based on the value of their PCP (Priority Code Point) field in the VLAN Tag.
  - Function **Rtnl2DeletePcpDispatchRule** deletes a dispatch rule previously set up by **Rtnl2AddPcpDispatchRule**.
  - Function **Rtnl2GetPcpDispatchRules** gets the list of PCP dispatch rules currently set up in the interface.
  - Structure **RTNL2_VLAN_TAG** represents the content an IEEE 802.1Q VLAN Tag.
  - Structure **RTNL2_PCP_DISPATCH_RULE** specifies a PCP (Priority Code Point) Dispatch rule.
- Added new functions for controlling the NL2 component (922):
  - Function **Rtnl2GetComponentStatus** returns the status of the Network Link Layer (NL2) component.
  - Function **Rtnl2IsComponentLicensed** returns the license status of the Network Link Layer (NL2) component.
  - Function **Rtnl2StartComponent** starts the Network Link Layer (NL2) component.
  - Function **Rtnl2StopComponent** stops the Network Link Layer (NL2) component.
- Added new functions and data types for configuring interrupt moderation on a given message ID (11586):
  - Function **Rtnl2GetMsixMessageConfig** retrieves the configuration of an MSI-X message.
  - Function **Rtnl2SetInterruptModeration** sets the interval value for interrupt moderation.

- Structure **TNL2_INTERFACE_CONFIG** specifies the configuration of an interface.
- Structure **RTNL2_INTERFACE_FEATURES** specifies the hardware features supported by an interface.
- Data type **RTNL2_HINTERFACE** represents a handle to an interface.
- Structure **RTNL2_MSIX_MESSAGE_CONFIG** is used by the function **Rtnl2GetMsixMessageConfig** to specify the configuration of an MSI-X message.

- Added new functions and data types for creating a logical transmit queue (12552):
  - Function **Rtnl2CreateLogicalTxQueue** creates a Logical Transmit Queue object above the hardware queue specified by the handle of the owning interface and the index of the Transmit Queue within that interface.
  - Function **Rtnl2DestroyLogicalTxQueue** destroys a Logical Transmit Queue object created by **Rtnl2CreateLogicalTxQueue**.
  - Data type **RTNL2_HLOGICAL_TX_QUEUE** represents a handle to a Logical Transmit Queue.
- Added these new functions and data types for sending frames over a logical transmit queue (12557):
  - Function **Rtnl2TransmitOverLogicalTxQueue** transmits one frame through a specified Logical Transmit Queue.
  - Structure **RTNL2_TRANSMIT_DESC** is used by function **Rtnl2TransmitOverLogicalTxQueue** to specify the frame to transmit over a Logical Transmit Queue.
- Added new functions and data types for configuring Logical Receive filters (12572):
  - Function **Rtnl2StartLogicalRxQueue** starts a Logical Receive Queue.
  - Function **Rtnl2StopLogicalRxQueue** stops a Logical Receive Queue.
  - Function **Rtnl2SetLogicalRxQueueEtherTypeFilter** configures the Logical Receive Queue EtherType filter of a Logical Receive Queue.
  - Function **Rtnl2SetLogicalRxQueueMode** enables or disables a setting of the interface mode.
  - Function **Rtnl2GetLogicalRxQueueMode** gets the current setting value of the interface mode.
- Added new functions and data types for receiving frames from a Logical Receive Queue (12567):
  - Function **Rtnl2ReceiveFromLogicalRxQueue** receives one frame from a specified Logical Receive Queue.
  - Structure **RTNL2_RECEIVE_DESC** is used by the function **RtNalReceiveFromLogicalRxQueue** to describe a receive operation over a Logical Receive Queue.
- Added new functions and data types for creating a Logical Receive Queue (12562):

- Function **Rtnl2CreateLogicalRxQueue** creates a Logical Receive Queue object on a Physical Receive Queue specified by the handle of the owning interface and the index of the Physical Receive Queue within that interface.
- Function **Rtnl2DestroyLogicalRxQueue** destroys a Logical Receive Queue object created by **Rtnl2CreateLogicalRxQueue**.
- Data type **RTNL2_HLOGICAL_RX_QUEUE** represents a handle to a Logical Receive Queue.

- Added new functions and data types for configuring the hardware timestamp (11605):
  - Function **Rtnl2EnableLogicalRxQueueTimestamping** enables the Ingress Timestamp reporting for the frames received through a given Logical Receive Queue.
  - Function **Rtnl2EnablePhysicalRxQueueTimestamping** enables the Ingress Timestamp reporting for the frames received through a given Physical Receive Queue.
  - Data type **RTNL2_TIMESTAMP** specifies a PTP timestamp, hardware clock time, or time offset.
- Added new functions for configuring Egress Timestamps (299):
  - Function **Rtnl2EnableLogicalTxQueueTimestamping** enables the Egress Timestamp logic on a given Logical Transmit Queue and creates an event to monitor Egress Timestamps.
  - Function **Rtnl2EnablePhysicalTxQueueTimestamping** enables the Egress Timestamp logic on a Physical Transmit Queue and creates an event to monitor Egress Timestamps.
  - Function **Rtnl2GetLogicalTxQueueTimestamp** gets the value of the last Egress Timestamp for the Logical Transmit Queue.
  - Function **Rtnl2GetPhysicalTxQueueTimestamp** gets the value of the last Egress Timestamp for the Physical Transmit Queue.
- Added new functions and data types for getting a handle to a network interface (11577):
  - Function **Rtnl2EnumInterface** returns the name of an interface specified by its index.
  - Function **Rtnl2OpenInterface** opens the interface object specified by its name.
  - Function **Rtnl2CloseInterface** closes a handle to an interface object.
  - Function **Rtnl2GetInterfaceConfig** gets the configuration of an interface.
  - Function **Rtnl2GetPhysicalTxQueueConfig** gets the configuration of a Physical Transmit Queue.
  - Function **Rtnl2GetPhysicalRxQueueConfig** gets the configuration of a Physical Receive Queue.
  - Function **Rtnl2GetInterfaceFeatures** gets the hardware features supported by an interface.
  - Function **Rtnl2GetMacAddress** gets the MAC address of an interface.

- Data type **RTNL2_PHYSICAL_TX_QUEUE_CONFIG** specifies the current configuration of a Physical Transmit Queue.
- Data type **RTNL2_PHYSICAL_RX_QUEUE_CONFIG** specifies the current configuration of a Physical Receive Queue.

- Added new functions and data types for allocating and releasing NL2 buffers (11582):
  - Function **Rtnl2GetPhysicalTxQueueBuffers** gets one or more NL2 buffers to be used with a specified physical transmit queue.
  - Function **Rtnl2ReturnPhysicalTxQueueBuffers** returns one or more NL2 Buffers that were received for the specified physical transmit queue.
  - Data type **RTNL2_BUFFER_HEADER** specifies a buffer allocated by **Rtnl2GetPhysicalTxQueueBuffers**.
- Added new functions and data types for mapping the EtherType to receive queues (11585):
  - Function **Rtnl2AddEtherTypeDispatchRule** adds a rule to dispatch incoming packets to a given Physical Receive Queue based on the value of their EtherType.
  - Function **Rtnl2DeleteEtherTypeDispatchRule** deletes a dispatch rule previously set up by **RtnI2AddEtherTypeDispatchRule**.
  - Function **Rtnl2GetEtherTypeDispatchRules** gets the list of EtherType dispatch rules currently set up in the interface.
  - Data type **RTNL2_ETHER_TYPE_DISPATCH_RULE** specifies an EtherType dispatch rule.
- Added new functions and data types for monitoring and getting the link status (11578):
  - Function **Rtnl2CreateLinkStatusChangeEvent** creates an event to be signaled by the NL2 process whenever the link status for the specified interface changes.
  - Function **Rtnl2DestroyLinkStatusChangeEvent** destroys a LinkStatusChange event previously created by **Rtnl2CreateLinkStatusChangeEvent**.
  - Function **Rtnl2GetLinkStatus** gets the current link status of an interface.
  - Data type **RTNL2_LINK_STATUS** specifies the status of the link.
- Added new functions and data types for sending frames over an acquired transmit hardware queue (681):
  - Function **Rtnl2AcquirePhysicalTxQueue** acquires a Physical Transmit Queue object specified by the handle of the owning interface and the index of the Physical Transmit Queue within that interface.
  - Function **Rtnl2SubmitToPhysicalTxQueue** submits NL2 buffers for transmission through a given Physical Transmit Queue.

- Function **Rtnl2CreatePhysicalTxQueueEvent** enables the hardware interrupt for a given Physical Transmit Queue and creates an event for the specified Physical Transmit Queue. This event is signaled by the NL2 each time the NIC triggers the hardware interrupt associated with that Physical Transmit Queue.
- Function **Rtnl2DestroyPhysicalTxQueueEvent** destroys a Transmit event previously created by **Rtnl2CreatePhysicalTxQueueEvent**, and disables the hardware interrupt associated with the Physical Transmit Queue.
- Function **Rtnl2ExtractFromPhysicalTxQueue** checks the DMA ring of the specified Physical Transmit Queue, and extracts buffers from the FIFO of consumed buffers.
- **Rtnl2ReleasePhysicalTxQueue** releases a Physical Transmit Queue object previously acquired by **Rtnl2AcquirePhysicalTxQueue**.
- Data type **RTNL2_HPHYSICAL_TX_QUEUE** represents a handle to a Physical Transmit Queue.

- Added new functions, data types and enumerations for receiving frames from an acquired receive hardware queue (829, 850):
  - Function **Rtnl2AcquirePhysicalRxQueue** acquires a Physical Receive Queue object specified by the handle of the owning interface and the index of the Physical Receive Queue within that interface.
  - Function **Rtnl2SetPhysicalRxQueueMode** enables or disables an interface mode setting.
  - Function **Rtnl2GetPhysicalRxQueueMode** gets the current setting value of the interface mode.
  - Function **Rtnl2SubmitToPhysicalRxQueue** submits NL2 buffers for reception through a given Physical Receive Queue.
  - Function **Rtnl2CreatePhysicalRxQueueEvent** enables the hardware interrupt for a given Physical Receive Queue and creates an event for the specified Physical Receive Queue. This event is signaled by the NL2 each time the NIC triggers the hardware interrupt associated with that Physical Receive Queue.
  - Function **Rtnl2DestroyPhysicalRxQueueEvent** destroys a Receive event previously created by **Rtnl2CreatePhysicalRxQueueEvent**, and disables the hardware interrupt associated with the Physical Receive Queue.
  - Function **Rtnl2ExtractFromPhysicalRxQueue** checks the DMA ring of the specified Physical Receive Queue, and extracts buffers from the FIFO of filled buffers.
  - Function **Rtnl2ReleasePhysicalRxQueue** releases a Physical Receive Queue object previously acquired by **Rtnl2AcquirePhysicalRxQueue**.
  - Data type **RTNL2_HPHYSICAL_RX_QUEUE** represents a handle to a Physical Receive Queue.
  - Enumeration **RTNL2_RX_MODE_SETTING** specifies a given setting of the interface mode.

- Added new functions for configuring multicast filters (714):
  - Function **RtnI2SetLogicalRxQueueMulticastFilter** configures the per-queue list of allowed Multicast Addresses for a given Logical Receive Queue.
  - Function **RtnI2SetPhysicalRxQueueMulticastFilter** configures the per-queue list of allowed Multicast Addresses for a given Physical Receive Queue.
- Added new functions and data types to support cross-timestamping (PTM) (9427):
  - Data type **RTNL2_HCLOCK** represents a handle to a clock.
  - Structure **RTNL2_READ_CLOCK_RESULT** describes the result of the operations made by the **RtnI2ReadClock** function.
  - Function **RtnI2OpenClock** opens a clock object specified by the handle of the owning interface and the index of the clock within that interface.
  - Function **RtnI2ReadClock** reads the NIC clock time and the CPU time simultaneously (cross-timestamp) and returns all timestamps.
  - Function **RtnI2AdjustClockTime** adjusts the time of a clock object by applying an offset to the current counter value.
  - Function **RtnI2SetClockRate** sets the rate of a clock object.
  - Function **RtnI2GetClockRate** gets the rate of a clock object.
  - Function **RtnI2CloseClock** closes a clock object opened by RtnI2OpenClock.
- Added function **RtnI2ControlDeviceSpecialFunction**, which controls a special function of a device.

New Network Relay (RTRLY) APIs

- Added new functions for controlling the Network Relay component (10581):
  - **RtrlyIsComponentLicensed** returns whether the Network Relay component has a valid license of the same major version as the currently installed wRTOS Runtime.
  - **RtrlyStartComponent** starts the Network Relay component.
  - **RtrlyStopComponent** stops the Network Relay component.
  - **RtrlyGetComponentStatus** returns the status of the Network Relay component.

New TCP/IP (RTTCPIP) APIs

- Added new functions for controlling the TCP/IP Stack component (933):
  - **RttcpipGetComponentStatus** returns the status of the TCP/IP Stack component.
  - **RttcpipIsComponentLicensed** returns the license status of the TCP/IP Stack component.
  - **RttcpipStartComponent** starts the TCP/IP Stack component.

- **RttcpipStopComponent** stops the TCP/IP Stack component.

- Added new functions for configuring the IP address of a network interface (333):
  - **RttcpipAddInterfaceAddress** adds an IPv4 address to a network interface.
  - **RttcpipDeleteInterfaceAddress** deletes the requested IP address of a network interface.
  - **RttcpipGetDefaultGateway** gets the IPv4 address of the default gateway associated with an interface.
  - **RttcpipGetInterfaceAddresses** retrieves a list of IPv4 and IPv6 addresses.
  - **RttcpipSetDefaultGateway** sets the IPv4 address of the default gateway associated with an interface.
  - **RttcpipUpdateInterfaceAddress** updates one of the IP addresses of a network interface.
  - **RTTCPIPINTERFACEADDRESS** defines the interface address format used for IPV4 and IPv6 addresses.

New TCP/IP Filter Driver (RTTCPIPFLT) APIs:

- Function **RttcpipfltConfigure** is called once for each NIC associated with the Filter Driver at TCP/IP Stack startup.
- Function **RttcpipfltReceive** is called every time the TCP/IP Stack receives an Ethernet frame from the network.
- Function **RttcpipfltReceiveEx** is called every time the TCP/IP Stack receives an Ethernet frame from the network. This function includes an additional parameter compared to **RttcpipfltReceive** which points to the network device that received the frame.
- Function **RttcpipfltTransmit** is called every time the TCP/IP Stack is about to send an Ethernet frame over the network.
- Function **RttcpipfltTransmitEx** is called every time the TCP/IP Stack is about to send an Ethernet frame over the network. This function includes an additional parameter compared to **RttcpipfltTransmit**, which is a pointer to the network device that should send the frame.
- Structure **RTTCPIPFLT_ETHERNET_HEADER** represents 14 bytes of an Ethernet frame header.

New NIC Driver (RTND) APIs

- Added Callback APIs that are meant to be called from NL2 drivers:
  - Structure **RTND_CALLBACKS** contains function pointers to Network Link Layer (NL2) callbacks.
  - Function **CreateRxBuffers** is meant to be called by a driver to allocate a new set of NL2 Receive Buffers.
  - Function **CreateTxBuffers** is meant to be called by a driver to allocate a new set of NL2 Transmit Buffers.

- Function **DestroyRxBuffers** is meant to be called by a driver to destroy one or several NL2 Receive Buffers that were previously created by **CreateRxBuffers**.
- Function **DestroyTxBuffers** is meant to be called by a driver to destroy one or more NL2 Transmit Buffers that were previously created by **CreateTxBuffers**.
- Function **GetVerbose** returns a verbose Boolean received by the Network Link Layer (NL2) process from the Windows registry at startup.
- Function **NotifyEgressTimestamp** is meant to be called by the IST of the driver when it detects that a new Egress Timestamp is available for one of the Transmit Queues of one of its managed interfaces.
- Function **NotifyLinkStatusChange** is meant to be called by the IST of the driver when it detects a link status change on one of its managed interfaces.
- Function **NotifyRxInterrupt** is meant to be called by the IST of the driver when a buffer is filled by a Receive Queue.
- Function **NotifyTxInterrupt** is meant to be called by the IST of the driver when a buffer is consumed by a Transmit Queue.

- Added data types of interface capabilities reported by the driver to the NL2. These data types describe the supported options and parameter ranges that the NL2 can set for an interface before starting it. (5066)
  - Enumeration **RTND_CAPABILITY_ID** identifies a given interface capability.
  - Structure **RTND_CAPABILITY_CLOCK** describes the interface's clock-related capability.
  - Structure **RTND_CAPABILITY_INTERRUPT** describes the interface's interrupt capability.
  - Structure **RTND_CAPABILITY_JUMBO** describes the interface's jumbo capability.
  - Structure **RTND_CAPABILITY_LINK** describes the interface's link-related capability.
  - Structure **RTND_CAPABILITY_RX** describes the interface's receive capability.
  - Structure **RTND_CAPABILITY_RX_RING** describes a Receive Queue's DMA ring capability.
  - Structure **RTND_CAPABILITY_TSN** describes the interface's TSN-related capability.
  - Structure **RTND_CAPABILITY_TX** describes the interface's transmit capability.
  - Structure **RTND_CAPABILITY_TX_RING** describes a Transmit Queue's DMA ring capability.
- Added structure **RTND_BUFFER_HEADER**, which describes both Transmit and Receive Buffers. This structure is only visible to the driver, not the application. (5087)
- Added structure **RTND_BUFFER_OBJECT**, which is an opaque data type that represents the buffer object descriptor used by the NL2 internally. (5087)
- Added data type **RTND_INTERFACE_OBJECT**, which is used in opaque pointers to identify Interface objects from the drivers. (5060)

- Added structure **RTND_MULTICAST_ENTRY**, which describes an entry of a linked list of Multicast Addresses. (5084)
- Added structure **RTND_TIMESTAMP**, which describes a PTP timestamp, hardware clock time, or time offset. (5081)
- Added structure **RTND_DISPATCHER_ETHER_TYPE_ENTRY**, which describes an entry of the EtherType Hardware Dispatcher. (5078)
- Added structure **RTND_DISPATCHER_UDP_PORT_ENTRY**, which describes an entry of the UDP Port Hardware Dispatcher. (3173)
- Added structure **RTND_LINK_STATUS**, which describes the link status. (5075)
- Added data types of interface settings that the NL2 can configure before starting the interface. (5069)
    - Enumeration **RTND_SETTING_ID** identifies a given interface setting for the **RtndSetInterface** function.
    - Structure **RTND_SETTING_CLOCK** describes the interface's clock setting.
    - Structure **RTND_SETTING_INTERRUPT** describes the interface's interrupt setting.
    - Structure **RTND_SETTING_JUMBO** describes the interface's jumbo setting.
    - Structure **RTND_SETTING_LINK** describes the interface's link setting.
    - Structure **RTND_SETTING_NON_QUEUE_MSIX_MAPPING** describes the interface's non-queue interrupt mapping setting.
    - Structure **RTND_SETTING_RX_QUEUE_MSIX_MAPPING** describes the Receive Queue's interrupt mapping setting.
    - Structure **RTND_SETTING_RX_QUEUE_TIMESTAMPING** describes the Receive Queue's timestamping setting.
    - Structure **RTND_SETTING_RX_RING** describes the Receive Queue's DMA ring setting.
    - Structure **RTND_SETTING_TX_QUEUE_MSIX_MAPPING** describes the Transmit Queue's interrupt mapping setting.
    - Structure **RTND_SETTING_TX_QUEUE_SCHEDULING** describes the Transmit Queue's scheduling algorithm.
    - Structure **RTND_SETTING_TX_QUEUE_TIMESTAMPING** describes the Transmit Queue's timestamping setting.
    - Structure **RTND_SETTING_TX_RING** describes the Transmit Queue's DMA ring setting.

- Added data types for features that are reported by the driver to the NL2. These features describe the supported options and parameter ranges that the NL2 can use while the interface is started. (5072)
  - Enumeration **RTND_FEATURE_ID** identifies a given interface feature.
  - Structure **RTND_FEATURE_BUFFERS** describes the buffers feature supported by an interface.
  - Structure **RTND_FEATURE_DISPATCH_RULES** describes the hardware dispatch rules interface feature.
  - Structure **RTND_FEATURE_INTERFACE_MODES** describes the dynamic modes supported by an interface.
  - Structure **RTND_FEATURE_INTERRUPT** describes the interrupt feature supported by an interface.
- Added structure **RTND_DRIVER_INFO**, which contains global driver information not specific to a particular interface. (4851)
- Added function **RtndShutdownInterface**, which shuts down an interface. After the function returns, the NIC will no longer generate a hardware interrupt. (5146)
- Added new functions for allocating and freeing transmit and receive data buffers (2396):
  - **RtndAllocateRxFrameDataBuffers** allocates a set of receive frame data buffers.
  - **RtndAllocateTxFrameDataBuffers** allocates a set of transmit frame data buffers.
  - **RtndFreeRxFrameDataBuffers** frees a set of receive frame data buffers previously allocated by **RtndAllocateRxFrameDataBuffers**.
  - **RtndFreeTxFrameDataBuffers** frees a set of transmit frame data buffers previously allocated by **RtndAllocateTxFrameDataBuffers**.
- Added new functions and data types to support the NL2 driver interface (17696):
  - Function **RtndInitDriver** provides drivers with pointers to the NL2 callback functions.
  - Function **RtndManageInterface** manages an interface on the PCI Bus.
  - Function **RtndQueryInterfaceCapability** gets a given capability of an interface.
  - Function **RtndSetInterface** writes a given setting to the interface.
  - Function **RtndStartInterface** starts an interface.
  - Function **RtndQueryMacAddress** gets the MAC address of an interface.
  - Function **RtndQueryInterfaceFeature** gets a given feature of an interface.
  - Function **RtndStopInterface** stops an interface.
  - Function **RtndUnmanageInterface** removes all associations made for a given interface.
  - Function **RtndQueryLinkStatus** gets the Link Status of an interface.

- Function **RtndSetPromiscuousMode** requests the interface to enable or disable promiscuous mode.
- Function **RtndSetPassBadFramesMode** requests the interface to enable or disable the Pass Bad Frames mode.
- Function **RtndSetInterruptModeration** requests the interface to change the value of the interval, in nanoseconds, for interrupt moderation. If the interrupt type is MSI-X, the NL2 will supply the message ID of the MSI-X message to throttle. Otherwise, the NL2 will set the *MessageId* parameter to zero (0).
- Function **RtndSetDispatcherEtherTypeEntry** updates an entry in the EtherType Hardware Dispatcher table.
- Function **RtndGetDispatcherEtherTypeEntry** gets an entry in the EtherType Hardware Dispatcher table.
- Function **RtndSetDispatcherUdpPortEntry** updates an entry in the UDP Port Hardware Dispatcher. (3173)
- Function **RtndGetDispatcherUdpPortEntry** gets an entry in the UDP Port Hardware Dispatcher table. (3173)
- Function **RtndGetDispatcherPcpEntry** gets an entry in the PCP Hardware Dispatcher table.
- Function **RtndSetDispatcherPcpEntry** updates an entry in the PCP Hardware Dispatcher table.
- Structure **RTND_CAPABILITY_VLAN** describes an interface's VLAN capability.
- Structure **RTND_VLAN_TAG** represents the content of an IEEE 802.1Q VLAN Tag.
- Structure **RTND_DISPATCHER_PCP_ENTRY** describes an entry of the PCP (Priority Code Point) Hardware Dispatcher.
- Function **RtndSetMulticastFilter** updates the hardware Multicast Hash Filter of an interface.
- Function **RtndEnableTxInterruptSource** enables or disables the transmit interrupt for a specific Transmit Queue.
- Function **RtndAttachTxQueue** attaches the specified Transmit Queue to the current application process.
- Function **RtndApplyTxBuffers** transmits all buffers inserted in the Transmit Queue with **RtndSubmitTxBuffer**.
- Function **tndExtractLastTxTimestamp** determines whether the Egress Timestamp register associated with a given Transmit Queue is valid and, if it is, returns the Egress Timestamp value.
- Function **RtndSubmitTxBuffer** inserts a buffer to the specified Transmit Queue.
- Function **RtndExtractTxBuffer** extracts a consumed buffer from the specified Transmit Queue.

- Function **RtndDetachTxQueue** detaches the specified Transmit Queue to the current application process.
- Function **RtndGetTxBuffers** gets multiple buffers from the driver's pool of available transmit buffers.
- Function **RtndReturnTxBuffers** returns multiple buffers to the driver's pool of available transmit buffers.
- Function **RtndStartRxQueue** starts the specified Receive Queue.
- Function **RtndStartTxQueue** starts the specified Transmit Queue.
- Function **RtndEnableRxInterruptSource** enables or disables the receive interrupt for a specific Receive Queue.
- Function **RtndStopRxQueue** stops the specified Receive Queue.
- Function **RtndStopTxQueue** stops the specified Transmit Queue.
- Function **RtndAttachRxQueue** attaches the specified Receive Queue to the current application process.
- Function **RtndApplyRxBuffers** fetches all buffers inserted in the Receive Queue with **RtndSubmitRxBuffer**.
- Function **RtndSubmitRxBuffer** inserts a buffer to the specified Receive Queue.
- Function **RtndExtractRxBuffer** extracts a filled buffer from the specified Receive Queue.
- Function **RtndDetachRxQueue** detaches the specified Receive Queue to the current application process.

- Added functions and data types to support cross-timestamping (PTM) (9427):
  - Structure **RTND_FEATURE_CLOCK** describes the clock feature of an interface.
  - Structure **RTND_READ_CLOCK_RESULT** describes the result of the operations made by the RtndReadClock function.
  - Function **RtndReadClock** reads a NIC clock time and CPU time simultaneously (cross-timestamp) and returns all timestamps.
  - Function **RtndAdjustClockTime** adjusts the time of a NIC clock by applying an offset to the current counter value.
  - Function **RtndSetClockRate** sets the rate of a NIC clock.
- Added function **RtndControlDeviceSpecialFunction**, which controls a special function of a device.
- Added new functions and data types to support the Credit-Based Shaper functionality (12807)
  - Structure **RTND_CBS_PARAMS** describes Credit-Based Shaper parameters.

- Function **RtndSetCbsParams** sets the Credit-Based Shaper parameters on a given Transmit Queue.

New NL2 Filter Driver (RTNDFLT) APIs

- Added new functions and data types that a Filter Driver must support to be used by the Network Link Layer (NL2) (5348):
  - Function **RtndfltInitDriver** initializes the filter driver and exchanges global information between the filter driver and the Network Link Layer (NL2).
  - Function **RtndfltEndDriver** ends a filter driver.
  - Function **RtndfltManageRxQueue** initializes an instance of the filter driver for the specified Rx Queue.
  - Function **RtndfltUnmanageRxQueue** removes all associations made for a given Rx Queue.
  - Function **RtndfltReceiveFrames** inspects a batch of NL2 Buffers (represented by a linked list of RTND Buffer Headers) corresponding to a burst of received frames on the specified Rx Queue, and determines for each whether it will be dropped or passed to the application.
  - Structure **RTNDFLT_CALLBACKS** contains function pointers to NL2 callbacks.
  - Structure **RTNDFLT_DRIVER_INFO** contains global driver information not specific to an Rx Queue.

New Variable Database (RTVDB) APIs

- Added new functions and data types for user applications to define variables exposed to MaxRT tools and servers (4094):
  - Function **RtvdbBrowseDatabases** gets the list of databases.
  - Function **RtvdbBrowseDirectory** browses a directory.
  - Function **RtvdbCloseDatabase** closes a database.
  - Function **RtvdbCloseDirectory** closes a directory.
  - Function **RtvdbCloseVariable** closes a variable.
  - Function **RtvdbCreateBufferVariable** creates and opens a buffer variable.
  - Function **RtvdbCreateDatabase** creates a database, opens it and its root directory.
  - Function **RtvdbCreateDirectory** creates and opens a sub-directory.
  - Function **RtvdbCreateVariable** creates and opens a variable.
  - Function **RtvdbDeleteDatabase** deletes a database.
  - Function **RtvdbDeleteDirectory** deletes a directory.
  - Function **RtvdbDeleteVariable** deletes a variable.
  - Function **RtvdbGetDatabaseStatus** gets the status of a database.

- Function **RtvdbInitialize** initializes the variable database library for the current process.
- Function **RtvdbOpenDatabase** opens a database and its root directory.
- Function **RtvdbOpenDirectory** opens a sub-directory.
- Function **RtvdbOpenVariable** opens a variable.
- Function **RtvdbRelease** closes the variable database library for the current process.
- Function **RtvdbRenameDirectory** renames a directory.
- Data type **RTVDB_DATABASE** represents a handle to a database.
- Data type **RTVDB_DATABASE_DESCRIPTION** describes a database and its state.
- Data type **RTVDB_DIRECTORY** represents a handle to a directory.
- Data type **RTVDB_VARIABLE** represents a handle to a variable.
- Data type **TVDB_VARIABLE_CONTROL** provides access to a variable.
- Data type **RTVDB_VARIABLE_DESCRIPTION** describes a variable.
- Data type **RTVDB_VARIABLE_DATATYPE** identifies the data type of a variable.

New Data Logger APIs

- Added new functions and data types for the wRTOS Scope tool to log variables (4099):
  - Function **RtvdbGetLogData** reads the logged data, automatically wrapping around the end of the data logger buffer. It stops when reaching the current log index. This function also reads data points. Each point has a value for each channel.
  - Function **RtvdbGetLogStatus** gets the current log status.
  - Function **RtvdbLogBackgroundProcess** initializes and releases the data logger.
  - Function **RtvdbLogCyclicProcess** triggers the data-logging engine's cyclic process. The real-time application's cyclic task calls this function to trigger data collection synchronized with its process.
  - Function **RtvdbStartLog** sends a request to start the log.
  - Function **RtvdbStopLog** cancels the current log request.
  - Data type **RTVDB_CHANNEL** provides access to a variable to log.
  - Data type **RTVDB_LOG_STATUS** identifies the status of a log command.
  - Data type **RTVDB_POINT** identifies the value of a data point.
  - Data type **RTVDB_TRIGGER** describes the logging trigger.
  - Data type **RTVDB_TRIGGER_TYPE** identifies the comparison between the current and reference values to trigger the log.

New Winsock APIs

New Winsock APIs

- Added new functions for converting the local index of a network interface to the ANSI name, and vice versa (333):
  - **if_indextoname** converts the local index for a network interface to the ANSI interface name.
  - **if_nametoindex** converts the ANSI interface name for a network interface to a local index for the interface.

New Configure and Control APIs

New Configure and Control APIs

- Added function **RtfwUnloadRtApi**, which unloads the RTAPI library. (6370)
- Added APIs for configuring E-CAT component instances (9064):
  - Structure **RTFWECAT_INSTANCE_CONFIGURATION** holds the E-CAT MainDevice instance configuration information.
  - Function **RtfwecatConfigureInstance** configures an E-CAT MainDevice instance.
  - Function **RtfwecatCreateInstance** defines a new E-CAT MainDevice instance.
  - Function **RtfwecatDeleteInstance** deletes an E-CAT MainDevice instance.
  - Function **RtfwecatGetAllInstances** gets the configuration of all E-CAT MainDevice instances.
  - Function **RtfwecatGetInstance** gets the current E-CAT MainDevice instance configuration.
- Added functions for setting and retrieving the default ideal processor (1749):
  - Function **RtfwSetDefaultIdealProcessor** sets the default processor where new processes will be executed by default.
  - Function **RtfwGetDefaultIdealProcessor** retrieves the default processor where new processes will be executed by default.
- Added APIs for configuring and querying configuration details for the Network Link Layer (NL2) and its interfaces: (5128, 896)
  - Function **Rtfwnl2CreateInterface** creates a new wRTOS network interface to be used by the Network Link Layer (NL2).
  - Function **Rtfwnl2DeleteInterface** deletes a Network Link Layer (NL2) interface.
  - Function **Rtfwnl2EnableInterface** enables or disables a Network Link Layer (NL2) interface.

- Function **RtfwnI2GetAllInterfaces** returns the configurations of all Network Link Layer (NL2) interfaces.
- Function **RtfwnI2GetClientProcessIDs** returns the IDs for all Network Link Layer (NL2) client processes.
- Function **RtfwnI2GetConfiguration** returns the configuration of the Network Link Layer (NL2).
- Function **RtfwnI2SetConfiguration** sets the configuration of the Network Link Layer (NL2).
- Function **RtfwnI2GetInterface** returns the configuration of a Network Link Layer (NL2) interface associated with a given name.
- Function **RtfwnI2RenameInterface** renames a Network Link Layer (NL2) interface.
- Function **RtfwnI2SetInterface** sets the configuration of a Network Link Layer (NL2) interface.
- Function **RtfwnI2GetVerbosity** returns whether verbosity is enabled or disabled for the Network Link Layer (NL2).
- Function **RtfwnI2SetVerbosity** enables or disables verbosity for the Network Link Layer (NL2).
- Structure **RTFW_INTERFACE_INFO** contains the name and enabled state of a Network Link Layer (NL2) interface.
- Structure **RTFW_NL2_CONFIGURATION** contains the configuration of the Network Link Layer (NL2).
- Structure **RTFW_NL2_INTERFACE** contains the configuration parameters of a Network Link Layer (NL2) interface.
- Structure **RTFW_NL2_INTERFACE_MSIXMESSAGE** contains the configuration of one MSI-X message for a Network Link Layer (NL2) interface.
- Structure **RTFW_NL2_INTERFACE_RXQUEUE** contains receive queue information for a Network Link Layer (NL2) interface.
- Structure **RTFW_NL2_INTERFACE_TXQUEUE** contains transmit queue information for a Network Link Layer (NL2) interface.

- Added functions for getting and setting the global TCP/IP configuration: (5209)
  - Function **RtfwtcpipGetConfiguration** returns the configuration of the TCP/IP Stack.
  - Function **RtfwtcpipSetConfiguration** sets the global configuration of the TCP/IP Stack.
- Added APIs for querying Network Link Layer (NL2) interfaces that have TCP/IP settings (5219, 5128):
  - Function **RtfwtcpipGetAllInterfaces** enumerates all Network Link Layer (NL2) interfaces that have TCP/IP settings.
- Added functions to set and return TCP/IP Stack verbosity: (5214)
  - Function **RtfwtcpipGetVerbosity** returns whether verbosity is enabled or disabled for the TCP/IP Stack.

    - Function **RtfwtcpipSetVerbosity** enables or disables verbosity for the TCP/IP Stack.
- Added APIs for configuring, setting, and querying TCP/IP interface values. (5224)
    - Function **RtfwtcpipGetInterfaceSettings** returns the configuration of a TCP/IP Stack interface associated with a given name.
    - Function **RtfwtcpipSetInterfaceSettings** sets the configuration of a TCP/IP Stack interface.
    - Structure **RTFW_TCPIP_INTERFACE** represents the configuration parameters of a TCP/IP Stack interface.
- Added function **RtfwtcpipEnableInterface**, which enables or disables TCP/IP functionality for a specified network interface. (8962)
- Added function **RtfwIsComponentInstalled**, which returns the install status and specific version information of the specified product component. (4424)
- Added function **RtfwGetLicenseStatus**, which returns whether the specified version of a product code has a valid license. (948, 8922)
- Added functions for controlling and querying the Network Link Layer (NL2) component. (953)
    - **Rtfwnl2GetComponentStatus** returns the status of the Network Link Layer (NL2) component.
    - **Rtfwnl2IsComponentLicensed** returns whether the Network Link Layer (NL2) component has a valid license of the same major version as the currently installed wRTOS Runtime.
    - **Rtfwnl2StartComponent** starts the Network Link Layer (NL2) component.
    - **Rtfwnl2StopComponent** stops the Network Link Layer (NL2) component.
- Added functions for controlling and querying the TCP/IP Stack component. (958)
    - **RtfwtcpipGetComponentStatus** returns the status of the TCP/IP Stack component.
    - **RtfwtcpipIsComponentLicensed** returns whether the TCP/IP Stack component has a valid license of the same major version as the currently installed wRTOS Runtime.
    - **RtfwtcpipStartComponent** starts the TCP/IP Stack component.
    - **RtfwtcpipStopComponent** stops the TCP/IP Stack component.
- Added functions for configuring the Network Relay component:
    - **RtfwrlyEnableInterface** enables or disables Network Relay functionality in a network interface.
    - **RtfwrlyGetAllInterfaces** returns the configurations of all Network Relay interfaces.
    - **RtfwrlyGetConfiguration** returns the Network Relay configuration.
    - **RtfwrlyGetInterfaceSettings** returns a Network Relay interface configuration associated with a given name.

- **RtfwrlyGetVerbosity** returns whether verbosity is enabled or disabled for the Network Relay component.
- **RtfwrlySetConfiguration** sets the Network Relay configuration.
- **RtfwrlySetInterfaceSettings** sets the configuration of a Network Relay interface.
- **RtfwrlySetVerbosity** enables or disables verbosity for the Network Relay component.
- **RTFW_RLY_CONFIGURATION** contains the configuration of the Network Relay component. See structure **RTFW_RLY_INTERFACE** for configuration parameters specific to a Network Relay interface.
- **RTFW_RLY_INTERFACE** represents the configuration parameters of a Network Relay interface.

New GigE Vision (RTGV) APIs

New GigE Vision (RTGV) APIs

- Added new functions to read/write a value from/to a register on the camera (9232):
  - Function **RtgvReadRegister** reads a value from a register on the camera.
  - Function **RtgvWriteRegister** writes a value to a register on the camera.

New Managed APIs

New Managed APIs

- Added **RtfwUnloadRtApi**, which unloads the RTAPI library. This is only necessary if you need to detach from the Subsystem after a Subsystem-dependent function is called. (6370)
- Added APIs for getting and setting the default core where a process's main thread will run unless an ideal processor is provided when running the process. (1749)
  - Property **Subsystem.DefaultIdealProcessor** gets or sets the default core that a process's main thread runs on unless an ideal processor is provided when the process runs.
  - Method **Subsystem.ResetDefaultIdealProcessor** resets property **DefaultIdealProcessor** to its default value (0). After this method is called, the real-time Subsystem must be restarted for the change to take effect.
- Added APIs for controlling and configuring the Network Link Layer (NL2) component. (991, 7302, 7307)
  - Namespace **IntervalZero.MaxRT.NL2.Control** contains types used to control the NL2.
    - Class **Component** encapsulates various control operations for the NL2.
      - Method **Component.Start** starts the NL2.

        - Method **Component.Stop** stops the NL2.
        - Method **Component.GetStatus** queries the status of the NL2.
        - Method **Component.GetClientProcessIDs** gets a list of client process IDs that are using the NL2 component.
    - Namespace **IntervalZero.MaxRT.NL2.Config** contains types used to configure the NL2.
        - Class **Component** encapsulates various configuration operations for the NL2.
            - Method **Component.IsLicensed** returns whether the NL2 component has a valid license of the same major version as the installed wRTOS Runtime.
            - Method **Component.ResetAll** resets the NL2 component to default values.
            - Method **Component.ResetAutoStart** resets the **AutoStart** property to its default value of false.
            - Method **Component.ResetExtMSpacePoolMinimumSize** resets the **ExtMSpacePoolMinimumSize** property to its default value of 1024.
            - Method **Component.ResetMainThreadIdealProcessor** resets the **MainThreadIdealProcessor** property to its default value of 0.
            - Method **Component.ResetMainThreadPriority** resets the **MainThreadPriority** property to it's default value of 66.
            - Method **Component.ResetMSpacePoolExpandable** resets the **MSpacePoolExpandable** property to its default value of 1.
            - Method **Component.esetMSpacePoolExpandSize** resets the **MSpacePoolExpandSize** property to its default value of 1024.
            - Method **Component.ResetVerbose** resets the **Verbose** property to its default value of false.
            - Property **Component.AutoStart** gets or sets whether the NL2 component should start automatically.
            - Property **Component.ExtMSpacePoolMinimumSize** gets or sets the minimum size of the external memory space pool for the NL2.
            - Property **Component.MainThreadIdealProcessor** gets or sets the ideal processor for the NL2’s main thread.
            - Property **Component.MainThreadPriority** sets the priority of the NL2’s main thread.
            - Property **Component.MSpacePoolExpandable** gets or sets whether the NL2's memory space pool is expandable.

- Property **Component.MSpacePoolExpandSize** sets the size the NL2's memory space pool should expand by.
- Property **Component.Verbose** gets or sets a Boolean indicating whether verbose logging is enabled.

- Class **InterfaceSettings** allows users to set the configuration of a network interface for the NL2 component.
  - Constructor **InterfaceSettings** initializes a new instance of the **InterfaceSettings** class to represent an NL2 interface.
  - Method **InterfaceSettings.AddMsixMessage** adds an **MsixMessage** object to the NL2 **InterfaceSettings** object.
  - Method **InterfaceSettings.AddRxQueue** adds an **RxQueue** object to the NL2 **InterfaceSettings** object.
  - Method **InterfaceSettings.AddTxQueue** adds a **TxQueue** object to the NL2 **InterfaceSettings** object.
  - Method **InterfaceSettings.CreateInterfaceSettings** creates an NL2 interface.
  - Method **InterfaceSettings.DeleteInterface** deletes the NL2 interface.
  - Method **InterfaceSettings.GetAllInterfaceSettings** returns a list of interfaces that are configured on the system for the TCP/IP component.
  - Method **InterfaceSettings.RemoveMsixMessage** removes an **MsixMessage** object from the NL2 **InterfaceSettings** object.
  - Method **InterfaceSettings.RemoveRxQueue** removes an **RxQueue** object from the NL2 **InterfaceSettings** object.
  - Method **InterfaceSettings.RemoveTxQueue** removes a **TxQueue** object from the NL2 **InterfaceSettings** object.
  - Property **InterfaceSettings.DeviceInstanceID** accesses the device Instance ID used by this NL2 interface.
  - Property **InterfaceSettings.Driver** accesses the driver's name for this NL2 interface.
  - Property **InterfaceSettings.Enabled** controls whether the interface is enabled or disabled.
  - Property **InterfaceSettings.FlowControl** accesses the **FlowControlType** enumeration specifying the flow control type used by this interface.
  - Property **InterfaceSettings.HardwareTimestampingEnabled** controls whether the interface is **HardwareTimestampingEnabled**.

- Property **InterfaceSettings.IngressTimestampingEtherType** gets or sets the EtherType if *HardwareTimestampingEnabled* is true.
- Property **InterfaceSettings.IngressTimestampingRule** accesses the **IngressTimeStampingRuleType** enumeration specifying the Ingress Timestamp rule type used by this interface.
- Property **InterfaceSettings.IngressTimestampingUDPPort** gets or sets the UDP port if *HardwareTimestampingEnabled* is true.
- Property **InterfaceSettings.Interrupt** accesses the **InterruptType** enumeration specifying the interrupt type used by this interface.
- Property **InterfaceSettings.IsVirtualNic** gets or sets a value indicating whether this interface is the wRTOS virtual network.
- Property **InterfaceSettings.JumboEnabled** controls whether jumbo frames are enabled for the interface.
- Property **InterfaceSettings.JumboMaxPacketSize** specifies the maximum jumbo packet size in bytes if jumbo frames are enabled.
- Property **InterfaceSettings.MsixMessages** gets or sets the MsixMessages list populated by the NL2 **InterfaceSettings** object.
- Property **InterfaceSettings.MsixNonQueueMessageId** gets or sets the message ID for non-queue (frame transmission or frame reception) interrupts if **InterruptType** is *RTIN_MSIX*.
- Property **InterfaceSettings.Name** accesses the unique interface name of this NL2 interface.
- Property **InterfaceSettings.NonMsixIstIdealProcessor** gets or sets the non-MSIX Interrupt Service Thread (IST) ideal processor if **InterruptType** is *RTIN_MSIX*.
- Property **InterfaceSettings.NonMsixIstPriority** gets or sets the Interrupt Service Thread (IST) priority for non-MSI-X interrupt types if **InterruptType** is *RTIN_MSIX*.
- Property **InterfaceSettings.PciBusLocation** accesses the PCI bus location for this NL2 interface.
- Property **InterfaceSettings.RxQueues** gets or sets the RxQueues list populated by the NL2 **InterfaceSettings** object.
- Property **InterfaceSettings.SpeedDuplex** accesses the **SpeedDuplexType** enumeration specifying the speed/duplex type used by this interface.
- Property **InterfaceSettings.TxQueues** gets or sets the TxQueue list populated by the NL2 **InterfaceSettings** object.

    - Event **InterfaceSettings.PropertyChanged** is triggered when a property value changes.
- Class **RxQueue** allows users to configure and manage a receive (Rx) queue in an NL2 interface. This class contains these members:
    - Property **RxQueue.BufferCount** gets or sets whether timestamping is enabled for the receive queue.
    - Property **RxQueue.Enabled** gets or sets whether the receive queue is enabled.
    - Property **FilterDriver** gets or sets an absolute pathname to a wRTOS filter driver RTDLL. This property is ignored when **FilterDriver** is false.
    - Property **FilterEnabled** gets or sets the receive queue's filter-enabled status.
    - Property **RxQueue.Index** gets or sets the receive queue index.
    - Property **RxQueue.ManagementThreadIdealProcessor** gets or sets the ideal processor for the receive queue's management thread.
    - Property **RxQueue.ManagementThreadPriority** gets or sets the priority of the receive queue's management thread.
    - Property **RxQueue.MSIXQueueMessageId** gets or sets the MSI-X message ID associated with the receive queue.
    - Property **RxQueue.TimestampingEnabled** gets or sets whether timestamping is enabled for the receive queue.
- Class **TxQueue** allows users to configure and manage a transmit (Tx) queue in an NL2 interface. This class contains these members:
    - Property **TxQueue.BufferCount** gets or sets whether timestamping is enabled for the transmit queue.
    - Property **TxQueue.Enabled** gets or sets whether the transmit queue is enabled.
    - Property **TxQueue.Index** gets or sets the transmit queue index.
    - Property **TxQueue.ManagementThreadIdealProcessor** gets or sets the ideal processor for the transmit queue's management thread.
    - Property **TxQueue.ManagementThreadPriority** gets or sets the priority of the transmit queue's management thread.
    - Property **TxQueue.MSIXQueueMessageId** gets or sets the MSI-X message ID associated with the transmit queue.

        - Property **TxQueue.TimestampingEnabled** gets or sets whether timestamping is enabled for the transmit queue.
    - Class **MsixMessage** allows users to configure and manage an MSI-X message in the NL2 interface. This class contains these members:
        - Property **MsixMessage.Enabled** gets or sets the enabled status of an MSI-X message.
        - Property **MsixMessage.IstIdealProcessor** gets or sets the ideal processor for the Interrupt Service Thread (IST).
        - Property **MsixMessage.IstPriority** gets or sets the priority of the Interrupt Service Thread (IST).
        - Property **MsixMessage.MsixMessageId** gets or sets the unique identifier of the MSI-X message.
- Added APIs for controlling and configuring the TCP/IP Stack component. (998, 7834, 7839, 7844)
    - Namespace **IntervalZero.MaxRT.Tcpip** contains types used to configure a TCP/IP IPv4 address and subnet mask.
    - Namespace **IntervalZero.MaxRT.Tcpip.Config** contains types used to configure the TCP/IP Stack.
        - Class **Component** encapsulates various configuration operations for the TCP/IP Stack.
            - Method **Component.IsLicensed** returns whether the TCP/IP Stack component has a valid license of the same major version as the installed wRTOS Runtime.
            - Method **Component.ResetAll** resets all TCP/IP configurations to their default values.
            - Method **Component.ResetAutoStart** resets the **AutoStart** parameter to its default value of zero (0).
            - Method **Component.ResetExtMSpacePoolMinimumSize** resets the **ExtMSpacePoolMinimumSize** property to its default value of 6272.
            - Method **Component.ResetIdealProcessor** resets the **IdealProcessor** property to its default value of zero (0).
            - Method **Component.ResetIPReassemblyTimeout** resets the **IPReassemblyTimeout** property to its default value of 60.
            - Method **Component.ResetMaxArpEntries** resets the **MaxArpEntries** property to its default value of 256.
            - Method **Component.ResetMaxConcurrency** resets the **MaxConcurrency** property to its default value of zero (0).

- Method **Component.ResetMaxSockets** resets the **Component.MaxSockets** property to its default value of 64.
- Method **Component.ResetMemory** resets the **Component.Memory** property to its default value of 1024 KB.
- Method **Component.ResetMSpacePoolExpandable** resets the **Component.MSpacePoolExpandable** property to its default value of 1.
- Method **Component.ResetMSpacePoolExpandSize** resets the **MSpacePoolExpandSize** property to its default value of 1024.
- Method **Component.ResetTimerExecutePriority** resets the **TimerExecutePriority** property to its default value of 66.
- Method **Component.ResetTimerIdealProcessor** resets the **TimerIdealProcessor** property to its default value of zero (0).
- Method **Component.ResetTimerInterval** resets the **TimerInterval** property to its default value of 100.
- Method **Component.ResetTimerPriority** resets the **TimerPriority** property to its default value of 66.
- Method **Component.ResetVerbose** resets the **Verbose** property to its default value of zero (0).
- Property **Component.AutoStart** gets or sets a Boolean indicating whether the TCP/IP Stack component should start automatically.
- Property **Component.ExtMSpacePoolMinimumSize** gets or sets the minimum size of the external memory space pool for the TCP/IP Stack.
- Property **Component.IdealProcessor** gets or sets the ideal processor for the TCP/IP Stack.
- Property **Component.Interfaces** gets a list of **NetworkInterface** objects representing the network interfaces enabled for TCP/IP.
- Property **Component.IPReassemblyTimeout** gets or sets the IP reassembly timeout value.
- Property **Component.MaxArpEntries** gets or sets the maximum number of ARP entries.
- Property **Component.MaxConcurrency** gets or sets the maximum concurrency level.
- Property **Component.MaxSockets** gets or sets the maximum number of sockets.
- Property **Component.Memory** gets or sets the memory allocation for the TCP/IP Stack.

        - Property **Component.MSpacePoolExpandable** gets or sets a Boolean indicating whether the memory space pool is expandable.
        - Property **Component.MSpacePoolExpandSize** gets or sets the size by which the memory space pool can expand.
        - Property **Component.RestartNeeded** indicates whether the TCP/IP Stack needs to be restarted for configuration changes made with this class to take effect.
        - Property **Component.TimerExecutePriority** gets or sets the execution priority of the TCP/IP Stack's timer thread.
        - Property **Component.TimerIdealProcessor** gets or sets the ideal processor for the TCP/IP Stack's timer thread.
        - Property **Component.TimerInterval** gets or sets the timer interval used by the TCP/IP Stack.
        - Property **Component.TimerPriority** gets or sets the timer priority used by the TCP/IP Stack.
        - Property **Component.Verbose** gets or sets a Boolean indicating whether verbose logging is enabled.
    - Namespace **IntervalZero.MaxRT.Tcpip.Control** contains types used to control the TCP/IP Stack.
        - Class **Component** encapsulates various control operations for the TCP/IP Stack.
            - Method **Component.Start** starts the TCP/IP component.
            - Method **Component.Stop** stops the TCP/IP component.
            - Method **Component.GetStatus** returns the status of the TCP/IP component.
            - Method **Component.GetClientProcessIDs** returns a list of client process IDs currently using the TCP/IP component.
        - Class **Ipv4Configuration** represents an IPv4 address and subnet mask.
            - Method **Ipv4Configuration.Equals** determines whether the specified **Ipv4Configuration** is equal to the current **Ipv4Configuration**.
            - Property **Ipv4Configuration.Ipv4Address** gets or sets the IPv4 address.
            - Property **Ipv4Configuration.SubnetMask** gets or sets the subnet mask.
        - Class **InterfaceSettings** represents the TCP/IP settings for an interface.
            - Property **InterfaceSettings.AreProtocolSettingsValid** indicates whether the protocol settings are valid.
            - Property **InterfaceSettings.Enabled** indicates whether the network interface is enabled.

- Property **InterfaceSettings.FilterDriver** gets or sets the name of the filter driver.
- Property **InterfaceSettings.FilterEnabled** indicates whether the filter is enabled.
- Property **InterfaceSettings.Gateway** gets or sets the gateway address for the network interface.
- Property **InterfaceSettings.IgnoreProtocolValidationError** indicates whether to ignore protocol validation errors.
- Property **InterfaceSettings.Ipv4Configurations** gets or sets a list of **Ipv4Configuration.** objects representing the IPv4 configurations.
- Property **InterfaceSettings.IPv6Address** gets or sets the IPv6 address of the network interface.
- Property **InterfaceSettings.IPv6Prefix** gets or sets the IPv6 prefix length.
- Property **InterfaceSettings.LinkStatusEnabled** indicates whether the link status is enabled.
- Property **InterfaceSettings.LinkStatusIdealProcessor** gets or sets the ideal processor for link status.
- Property **InterfaceSettings.LinkStatusPriority** gets or sets the priority for link status.
- Property **InterfaceSettings.Name** gets the name of the network interface.
- Property **InterfaceSettings.ReceiveIdealProcessor** gets or sets the ideal processor for TCP/IP Stack's receive thread.
- Property **InterfaceSettings.ReceivePriority** gets or sets the priority for the TCP/IP Stack's receive thread for the network interface.
- Property **InterfaceSettings.RxBufferCount** gets or sets the receive buffer count used by the TCP/IP Stack.
- Property **InterfaceSettings.RxQueueIndex** gets or sets the receive queue index used by the TCP/IP Stack.
- Property **InterfaceSettings.TxBufferCount** gets or sets the transmit buffer count used by the TCP/IP Stack.
- Property **InterfaceSettings.TxQueueIndex** gets or sets the transmit queue index used by the TCP/IP Stack.

- Added a managed **IntervalZero.MaxRT.Rly.Config** namespace containing types used to configure the Network Relay component and its interface settings.
  - Added a new managed **Rly.Config.Component** class that encapsulates various Network Relay configuration operations:

- Method **Reset All** resets all Network Relay component configurations to their default values.
- Method **ResetAutoStart** resets the AutoStart property to its default of manual start (false). After calling this method, the Network Relay component must be restarted for the changes to take effect.
- Method **ResetExtMSpacePoolMinimumSize** resets the ExtMSpacePoolMinimumSize property to its default value of 6,272 KB. After calling this method, the Network Relay component must be restarted for the changes to take effect.
- Method **ResetIdealProcessor** resets the IdealProcessor property to its default value of 0xFFFFFFFF, indicating that the system default processor should be used. After calling this method, the Network Relay component must be restarted for the changes to take effect.
- Method **ResetMSpacePoolExpandable** resets the MSpacePoolExpandable property to its default of enabled, which is a value of 1. After calling this method, the Network Relay component must be restarted for the changes to take effect.
- Method **ResetMSpacePoolExpandSize** resets the MSpacePoolExpandSize property to its default value (false). After calling this method, the Network Relay component must be restarted for the changes to take effect.
- Method **ResetPriority** resets the Priority property to its default priority of 66. After calling this method, the Network Relay component must be restarted for the changes to take effect.
- Method **ResetVerbose** resets the Verbose property to its default of disabled (false). After calling this method, the Network Relay component must be restarted for the changes to take effect.
- Property **AutoStart** gets or sets a Boolean indicating whether the Network Relay component should start automatically.
- Property **ExtMSpacePoolMinimumSize** gets or sets the minimum size of the Network Relay's external memory space pool.
- Property **IdealProcessor** gets or sets the ideal processor of the main thread.
- Property **MSpacePoolExpandable** gets or sets a Boolean indicating whether the memory space pool is expandable.
- Property **MSpacePoolExpandSize** gets or sets the size by which the memory space pool can expand.
- Property **Priority** gets or sets the priority of the main thread.
- Property **RestartNeeded** indicates whether the Network Relay component must be restarted for configuration changes made with this class to take effect.

        - Property **Verbose** gets or sets a Boolean indicating whether verbose logging is enabled.
    - Added a new managed **Rly.Config.InterfaceSettings** class that encapsulates various Network Relay interface settings:
        - Method **GetAllInterfaceSettings** returns a list of interfaces configured for the Network Relay component.
        - Method **CreateInterfaceSettings(string name)** creates a new InterfaceSettings object for the specified interface name.
        - Property **AreProtocolSettingsValid** is read only. If the value is false, the protocol settings conflict with another protocol interface setting. If the value is true, the protocol settings are valid.
        - Property **ConfigIdealProcessor** gets or sets the preferred processor for the configuration thread.
        - Property **ConfigPriority** gets or sets the priority of the configuration thread.
        - Property **Enabled** gets or sets whether the interface is enabled.
        - Property **IgnoreProtocolValidationError** determines whether changes will be validated against higher-level network protocol settings on the interface.
        - Property **Name** gets the name of the network interface.
        - Property **ReceiveIdealProcessor** gets or sets the preferred processor for the receive thread.
        - Property **ReceivePriority** gets or sets the priority of the receive thread.
        - Property **RxBufferCount** gets or sets the number of receive buffers.
        - Property **RxQueueIndex** gets or sets the receive queue index.
        - Property **TransmitIdealProcessor** gets or sets the preferred processor for the transmit thread.
        - Property **TransmitPriority** gets or sets the priority of the transmit thread.
        - Property **TxBufferCount** gets or sets the number of transmit buffers.
        - Property **TxQueueIndex** gets or sets the transmit queue index.
- Added a managed **IntervalZero.MaxRT.Rly.Control** namespace, which contains types used to control the Network Relay component.
    - Added a managed **Rly.Control.Component** class that encapsulates various control operations for the Network Relay component:
        - Method **GetStatus** queries the status of the Network Relay component.

        - Method **Start** starts the Network Relay component. If the Network Link Layer (NL2) is not started, the NL2 will also start.
        - Method **Stop** stops the Network Relay component.

- Added a managed **IntervalZero.MaxRT** namespace:
    - Added a managed **MaxRT.Device** class, which is the base class for all classes representing hardware devices on the current machine:
        - Event **PropertyChanged** occurs when a property value changes.
        - Property **ClassGuid** gets the Class GUID string of the device.
        - Property **Description** gets the Plug and Play friendly name string of the device.
        - Property **DeviceInstanceId** gets the Plug and Play instance ID string of the device.
        - Property **HardwareId** gets the Plug and Play hardware ID string of the device.
        - Property **OutOfDate** indicates whether the object is out-of-date with respect to the hardware device.
        - Property **OwnedByRtos** indicates whether the device is owned by MaxRT.
        - Property **PCIBusLocation** gets the PCI bus location for the device as a string.
    - Added a managed **MaxRT.RtDevice** Class, which represents a device used in the real-time environment.
    - Added a managed **MaxRT.InterfaceSettings** class for network interface configuration:
        - Property **Name** gets the name of a network interface.
    - Added a managed **MaxRT.NetworkInterfaceStatus** class, which represents the status of a network interface.
        - Property **ErrorCode** gets a Windows or wRTOS error code describing the error state.
        - Property **ErrorDescription** gets a description for the error code retrieved by the ErrorCode property.
        - Property **Name** gets the unifying identifier for the network interface.
        - Property **Status** gets the current status of the network interface.
    - Added a managed **MaxRT.MaxRTLock** class for lock objects in the MaxRT environment.
    - Added a managed **MaxRT.MaxRTNativeException** class representing native errors in the MaxRT environment.
        - Property **Code** gets the error code associated with the exception.

        - Property **DetailedMessage** gets a detailed message including the error code and description.
        - Property **ErrorCodeString** gets a string description of the error code retrieved by the Code property.
    - Added a managed **MaxRT.SystemErrorCode** enumeration that represents system error codes used in MaxRT native exceptions.
- Added new APIs for determining component installation status and version information (4424):
    - Method **Product.GetComponentInfo** returns component information.
    - Structure **Product.ComponentInfo** contains version information and a readable friendly name of the specified component. This structure contains these members:
        - Field **BuildNumber** represents the build number of the component. For example 4.5.1.**1234**.
        - Field **FriendlyName** represents the friendly name of the component.
        - Field **MajorVersion** represents the major version of the component. For example, **4**.5.1.1234
        - Field **MinorVersion** represents the minor version of the component. For example, 4.**5**.1.1234
        - Field **PatchVersion** represents the patch, fix, or update version of the component. For example, 4.5.**1**.1234
- Added new APIs for determining Subsystem exception handling behavior (11320):
    - Enumeration **ExceptionHandling** represents the available exception handling options. Enumerators from this enumeration can be assigned to property **Subsystem.ExceptionHandlingMode**.
    - Property **ExceptionHandlingMode** accesses the **Subsystem.ExceptionHandling** enumeration specifying the available exception handling options used by the Subsystem.
- Added new field **hasExpirationDate** to structure **Product.RT_LICENSE_INFO** that specifies whether the license has an expiration date. This field is non-zero if this license has an expiration date. Otherwise it is zero.
- Added new enumerators to enumeration **Product.RT_FEATURE_LICENSE_STATUS** to support time-limited licenses. (13627):
    - **RT_FEATURE_STATUS_RETAIL_VALID** indicates the license is a valid retail license (with or without an expiration date).
    - **RT_FEATURE_STATUS_RETAIL_EXPIRED** indicates the license is an expired retail license.
    - **RT_FEATURE_STATUS_EVAL_VALID** indicates the license is a valid evaluation license.
    - **RT_FEATURE_STATUS_RETAIL_INVALID_HOST_ID** indicates the license is a dongle-based retail license for which the host ID is not present on the system.
- Added new Data Logger APIs for wRTOS Scope tool to log variables (4099):

- Enumeration **LogState** identifies the status of a log command.
- Enumeration **TriggerType** identifies the trigger type.
- Class **Channel** represents a log channel. This class contains the following constructor and properties:
    - Constructor **Channel**
    - Property **Channel.DataType**
    - Property **Channel.Offset**
    - Property **Channel.Variable**
- Class **Log** represents a log request. This class contains the following constructor, methods and properties:
    - Constructor **Log**
    - Method **Log.GetData**
    - Method **Log.Stop**
    - Property **Log.BufferSize**
    - Property **Log.Channels**
    - Property **Log.Status**
- Class **LogData** represents the data read from the log buffer. This class contains the following constructor and properties:
    - Constructor **LogData**
    - Property **LogData.PeriodUs**
    - Property **LogData.Points**
    - Property **LogData.TimeStamp**
- Class **LogStatus** represents the state of a log request. This class contains the following constructor and properties:
    - Constructor **LogStatus**
    - Property **LogStatus.Index**
    - Property **LogStatus.State**
- Class **Point** represents the value of each channel at a specific time. This class contains the following constructor and property:
    - Constructor **Point**

        - Property **Point.Values**
    - Class **Trigger** represents a log trigger. This class contains the following constructor and properties:
        - Constructor **Trigger**
        - Property **Trigger.Channel**
        - Property **Trigger.Type**
        - Property **Trigger.Value**
- Added new Debug Message APIs for sending messages to Message Viewer (393):
    - Enumeration **DataType** identifies the type of a value.
    - Enumeration **PrintFormat** identifies the format used to print a value.
    - Enumeration **Severity** identifies the severity of a Basic message.
    - Class **BasicChannel** controls the Basic message channel. This class contains the following constructor, methods and properties:
        - Constructor **BasicChannel**
        - Method **BasicChannel.Close**
        - Method **BasicChannel.Dispose**
        - Method **BasicChannel.GetCategoryName**
        - Method **BasicChannel.GetMessage**
        - Method **BasicChannel.Open**
        - Method **BasicChannel.SendMessage**
        - Method **BasicChannel.SetCategoryName**
        - Property **BasicChannel.CategoryIndexFilter**
        - Property **BasicChannel.IsOpened**
        - Property **BasicChannel.SeverityFilter**
    - Class **BasicMessage** contains a basic message. This class contains the following constructor and fields:
        - Constructor **BasicMessage**
        - Field **BasicMessage.AppendValues**
        - Field **BasicMessage.Category**
        - Field **BasicMessage.Message**

        - Field **asicMessage.Severity**
        - Field **BasicMessage.Timestamp**
        - Field **BasicMessage.Value0**
        - Field **BasicMessage.Value1**
        - Field **BasicMessage.Value2**
        - Field **BasicMessage.Value3**
    - Class **ExtendedMessage** contains a Standard or Trace message. This class contains the following constructor and fields:
        - Constructor **ExtendedMessage**
        - Field **ExtendedMessage.Data**
        - Field **ExtendedMessage.Index**
        - Field **ExtendedMessage.Instance**
        - Field **ExtendedMessage.Message**
        - Field **ExtendedMessage.Source**
        - Field **ExtendedMessage.Format**
        - Field **ExtendedMessage.Timestamp**
        - Field **ExtendedMessage.Type**
    - Class **StandardChannel** controls a Standard message channel. This class contains the following constructor, methods and properties:
        - Constructor **StandardChannel**
        - Method **StandardChannel.Close**
        - Method **StandardChannel.Dispose**
        - Method **StandardChannel.GetMessage**
        - Method **StandardChannel.Open**
        - Method **StandardChannel.SendMessage**
        - Property **StandardChannel.IsOpened**
        - Property **StandardChannel.SourceIndexFilter**
        - Property **StandardChannel.TypeIndexFilter**

- Class **TraceChannel** controls a Trace message channel. This class contains the following constructor, methods and properties:
    - Constructor **TraceChannel**
    - Method **TraceChannel.Close**
    - Method **TraceChannel.Dispose**
    - Method **TraceChannel.GetIndexFilter**
    - Method **TraceChannel.GetMessage**
    - Method **TraceChannel.Open**
    - Method **TraceChannel.SendMessage**
    - Method **TraceChannel.SetIndexFilter**
    - Property **TraceChannel.IsOpened**
    - Property **TraceChannel.SourceIndexFilter**
    - Property **TraceChannel.TypeIndexFilter**
- Added new Variable Database APIs for user applications to define variables exposed to MaxRT tools and servers (4094):
    - Enumeration **DataType** identifies the type of a value.
    - Class **Database** controls a database. This class contains the following constructor, methods, and properties:
        - Constructor **Database**
        - Method **Database.CheckStatus**
        - Method **Database.Close**
        - Method **Database.Delete**
        - Method **Database.Dispose**
        - Property **Database.Invalidated**
        - Property **Database.IsOpened**
        - Property **Database.Name**
        - Property **Database.Root**
        - Property **Database.TimeStamp**

- Class **DatabaseDescription** describes a database. This class contains the following constructor and properties:
    - Constructor **DatabaseDescription**
    - Property **DatabaseDescription.Invalidated**
    - Property **DatabaseDescription.Name**
    - Property **DatabaseDescription.TimeStamp**
- Class **Directory** controls a directory. This class contains the following constructor, methods, and properties:
    - Constructor **Directory**
    - Method **Directory.Browse**
    - Method **Directory.Close**
    - Method **Directory.CreateDirectory**
    - Method **Directory.CreateVariable**
    - Method **Directory.Delete**
    - Method **Directory.Dispose**
    - Method **Directory.OpenDirectory**
    - Method **Directory.OpenVariable**
    - Method **Directory.Rename**
    - Property **Directory.IsOpened**
    - Property **Directory.Name**
    - Property **Directory.Size**
- Class **Library** controls the basic message channel. This class contains the following constructor, methods, and properties:
    - Constructor **Library**
    - Method **Library.BrowseDatabases**
    - Method **Library.Close**
    - Method **Library.CreateDatabase**
    - Method **Library.Dispose**
    - Method **Library.Open**

        - Method **Library.OpenDatabase**
        - Method **Library.StartLog**
        - Property **Library.IsOpened**
    - Class **Variable** controls a variable. This class contains the following constructor, methods, and properties:
        - Constructor **Variable**
        - Method **Variable.Close**
        - Method **Variable.Delete**
        - Method **Variable.Dispose**
        - Property **Variable.Description**
        - Property **Variable.IsOpened**
    - Class **Variable<T>** controls a variable with a value type. This class contains the following constructor and property:
        - Constructor **Variable<T>**
        - Property **Variable<T>.Value**
    - Class **VariableDescription** describes a variable. This class contains the following constructor and properties:
        - Constructor **VariableDescription**
        - Property **VariableDescription.Name**
        - Property **VariableDescription.Size**
        - Property **VariableDescription.Type**

New Managed E-CAT APIs

- Added new managed APIs for E-CAT control. See New Managed E-CAT APIs under the New APIs section in the Porting from KINGSTAR topic. (461)

Enhanced APIs

This section lists the APIs included in RTX64 4.x SDKs enhanced with non-breaking changes in wRTOS 1.x SDKs.

Enhanced Real-Time APIs

RTX64 4.x API	wRTOS 1.0 and later
RTPROCESS_INFORMATION	Added field **ProcessFlags**, a bit-mask flag value that contains information about a running process. `DWORD ProcessFlags`
RT_LICENSE_INFO	Added field **HostID**, which is the Host ID of the license. `TCHAR hostId[HOST_ID_MAX_LENGTH]`

Breaking API Changes

This section lists the APIs included in RTX64 4.x SDKs that underwent breaking changes in wRTOS 1.x SDKs.

> **IMPORTANT:** Existing real-time applications that call these APIs may need to be updated to ensure binary compatibility with wRTOS 1.x.

Breaking Changes to Real-Time APIs

Real-Time APIs (RTAPI)

RTX64 4.x API	wRTOS 1.0 and later
RtAttachShutdownHandler	Removed from header file **Rtapi.h** and added to **RtssApi.h**. Removed from library file **Rtapi.lib** and added to **Startup.lib**.
RtGetThreadPriority	Removed from header file **Rtapi.h** and added to **RtssApi.h**.
RtSetThreadPriority	Removed from header file **Rtapi.h** and added to **RtssApi.h**.
RT_LICENSE_INFO	Enumerator **RT_FEATURE_STATUS_EVAL** was renamed **RT_FEATURE_STATUS_EVAL_VALID** in field **RT_FEATURE_LICENSE_STATUS status**. Enumerator **RT_FEATURE_STATUS_INVALID_HOST_ID** was renamed **RT_FEATURE_STATUS_RETAIL_INVALID_HOST_ID** Removed field *isFeatureInstalled*.

RTX64 4.x API	wRTOS 1.0 and later
RT_MONITOR_COMPONENT	Replaced constant **RT_MONITOR_COMPONENT_NAL = 2** with **RT_MONITOR_COMPONENT_NL2 = 2**. Removed constant **RT_MONITOR_COMPONENT_ALL = 3**
RtReleaseShutdownHandler	Removed from header file **Rtapi.h** and added to **RtssApi.h**. Removed from library file **Rtapi.lib** and added to **Startup.lib**.
RTX64_MONITOR_CONTROL_OP	Renamed **WRTOS_MONITOR_CONTROL_OP**.
RTPROCESS_INFORMATION Structure	Removed field `EnableLocalMemory`.

Real-Time Network APIs

RTX64 4.x API	wRTOS 1.0 and later
RtnAddMultiRoute	Renamed **RttcpipAddMultiRoute**
RtnDeleteMultiRoute	Renamed **RttcpipDeleteMultiRoute**
RtnFrameAllocate	Renamed **RttcpipFrameAllocate**
RtnFrameFree	Renamed **RttcpipFrameFree**
RtnFrameTransmit	Renamed **RttcpipFrameTransmit**
RtnFrameTransmitInterface	Renamed **RttcpipFrameTransmitInterface**
RtnGetDeviceFromIpAddress	Renamed **RttcpipGetDeviceFromIpAddress**

RTX64 4.x API	wRTOS 1.0 and later
RtnGetDeviceName	Renamed **RttcpipGetDeviceName**
RtnGetDevicePtr	Renamed **RttcpipGetDevicePtr**
RtnInstallStaticRoute	Renamed **RttcpipInstallStaticRoute**
RtnIsDeviceOnline	Renamed **RttcpipIsDeviceOnline**
RtnQueryStackHeapUsage	Renamed **RttcpipQueryStackHeapUsage**
RtnRemoveStaticRoute	Renamed **RttcpipRemoveStaticRoute**

Breaking Changes to Configure and Control (RTFW) APIs

RTX64 4.x API	wRTOS 1.0 and later
RTFW_LOCAL_MEMORY_CONFIGURATION Structure	Removed member `EnableLocalMemory`.
RTFW_CONSOLE_CONFIGURATION_EX	Renamed **RTFW_CONSOLE_CONFIGURATION**
RtfwGetConsoleConfigurationEx	Renamed **RtfwGetConsoleConfiguration**
RtfwGetTCPIPClientProcessIDs	Renamed **RtfwtcpipClientProcessIDs**.
RTFW_PRIORITY_INVERSION_PROTOCOL	Constant **RTPI_NONE** was renamed **RTPI_DISABLED**. Constant **RTPI_TIERED_DEMOTION** was renamed **RTPI_ENABLED**.
RtfwSetConsoleConfigurationEx	Renamed **RtfwSetConsoleConfiguration**

Breaking Changes to Managed APIs

Note: Managed libraries and class layouts changed between RTX64 and wRTOS 1.x. See Project Settings earlier in this guide for more information on library changes.

RTX64 4.x API	wRTOS 1.0 and later
Class **Diagnostics.RTPerformance**	Renamed **Diagnostics.RtPerformance**
Class **Diagnostics.RTProcess**	Renamed **Diagnostics.RtProcess**
Interface **Diagnostics.IDiagnosticsRTProcess**	Renamed **Diagnostics.IDiagnosticsRtProcess**
Class **IO.RTBus**	Renamed **IO.RtBus**
Class **IO.RTInterruptHandler**	Renamed **IO.RtInterruptHandler**
Class **IO.RTMappedMemory**	Renamed **IO.RtMappedMemory**
Class **IO.RTPCISLot**	Renamed **IO.RtPCISLot**
Class **IO.RTPort**	Renamed **IO.RtPort**
Class **IO.RTPort8**	Renamed **IO.RtPort8**
Class **IO.RTSharedMemory**	Renamed **IO.RtSharedMemory**
Class **InteropServices.RTHandle**	Renamed **InteropServices.RtHandle**
Class **InteropServices.RTMarshal**	Renamed **InteropServices.RtMarshal**
Enumeration **InteropServices.RTHandleType**	Renamed **InteropServices.RtHandleType**
Class **Threading.RTEventWaitHandle**	Renamed **Threading.RtEventWaitHandle**

RTX64 4.x API	wRTOS 1.0 and later
Class **Threading.RTMutex**	Renamed **Threading.RtMutex**
Class **Threading.RTSemaphore**	Renamed **Threading.RtSemaphore**
Class **Threading.RTTimer**	Renamed **Threading.RtTimer**
Class **Threading.RTWaitHandle**	Renamed **Threading.RtWaitHandle**
Delegate **Threading.RTTimerCallback**	Renamed **Threading.RtTimerCallback**
Class **Threading.WaitHandles.RTSafeWaitHandle**	Renamed **Threading.WaitHandles.RtSafeWaitHandle**
Enumeration **Product.RT_FEATURE_LICENSE_STATUS**	Renamed enumerator **RT_FEATURE_STATUS_EVAL** to **RT_FEATURE_STATUS_EVAL_VALID**
Enumeration **Product.RT_FEATURE_LICENSE_STATUS**	Renamed enumerator **RT_FEATURE_STATUS_INVALID_HOST_ID** to **RT_FEATURE_STATUS_RETAIL_INVALID_HOST_ID**
Monitor.Subsystem.MonitorComponent Enumeration	Replaced member **NAL (2)** with **NL2 (2)**. Removed member **ALL (3)**
Method **Product.GetFeatureLicense**	Renamed **Product.GetLicenseInfo**
Property **Device.Rtx64Devices**	Renamed **Device.wRTOSDevices**
Property **Device.Rtx64NetworkDevices**	Renamed **Device.wRTOSNetworkDevices**
Property **Device.OwnedByRTX64**	Renamed **Device.OwnedByRtos**
Event **Config.RTX64Device**	Renamed **Config.wRTOSDevice**

RTX64 4.x API	wRTOS 1.0 and later
Interface **Config.IConfigRTX64Device**	Renamed **Config.IConfigwRTOSDevice**
Property **Subsystem.DisableSpeedStep**	Renamed **Subsystem.DisableWindowsIdleDetection**
Method **Subsystem.ResetDisableSpeedStep**	Renamed **Subsystem.ResetDisableWindowsIdleDetection**

Breaking Changes to Error Codes

Real-Time APIs

RTX64 4.x Error Code	wRTOS 1.0 and later
RT_ERROR_MONITORING_ALREADY_ENABLED	Renamed RT_ERROR_MONITORING_ENABLED
RT_ERROR_MONITORING_ALREADY_DISABLED	Renamed RT_ERROR_MONITORING_DISABLED
RT_ERROR_MONITORING_ALREADY_STARTED	Renamed RT_ERROR_MONITORING_STARTED
RT_ERROR_MONITORING_ALREADY_STOPPED	Renamed RT_ERROR_MONITORING_STOPPED
RT_ERROR_MONITORING_ALREADY_PAUSED	Renamed RT_ERROR_MONITORING_PAUSED

Removed/Deprecated APIs

This section lists RTX64 4.x APIs that were either removed from or are deprecated in wRTOS 1.x SDKs.

Removed/Deprecated Real-Time APIs

RTX64 4.x API	wRTOS 1.0 and later
RtAllocateLocalMemory	Removed. Use **RtAllocateLocalMemoryEx**.
RtAllocateLockedMemory	Removed. Use **RtAllocateLocalMemoryEx**.
RtFreeLockedMemory	Removed. Use **RtFreeLocalMemory**.
RtDisablePortIo	Removed.
RtEnablePortIo	Removed.
RtGetModuleFileName	Removed. Use **RtGetModuleFileNameEx**.
RtIsDefaultLocalMemory	Removed.
RtIsTcpStackLicensed	Removed.
RtGetThreadTimeQuantum	Removed. Use **RtGetTimeQuantum**.
RtGetThreadTimeQuantumEx	Removed. Use **RtGetTimeQuantum**.
RtMonitorEnableComponents	Removed.
RtMonitorGetEnabledComponents	Removed.
RtQueryComponent	Removed.
RtSetThreadTimeQuantum	Removed. Use **RtSetTimeQuantum**.

RTX64 4.x API	wRTOS 1.0 and later
RtSetThreadTimeQuantumEx	Removed. Use **RtSetTimeQuantum**.
RtSleepEx	Removed. Use **SleepEx**.
RtStartComponent	Removed. Use these component-specific functions: • Use **Rtnl2StartComponent** to start the Network Link Layer (NL2) component. • Use **RttcpipStartComponent** to start the TCP/IP Stack component. • Use **RtecatStartComponent** to start all configured E-CAT MainDevice instances. • Use **RtrlyStartComponent** to start the Network Relay component.
RtStopComponent	Removed. Use these component-specific functions: • Use **Rtnl2StopComponent** to stop the Network Link Layer (NL2) component. • Use **RttcpipStopComponent** to stop the TCP/IP Stack component.
RTSSCOMPONENT	Removed.

Windows Driver IPC APIs

RTX64 4.x API	wRTOS 1.0 and later
RtkIsNetworkLicensed	Removed.
RtkIsNetworkLicensedEx	Removed.

RTX64 4.x API	wRTOS 1.0 and later
RtkIsRuntimeLicensed	Removed.
RtkIsRuntimeLicensedEx	Removed.

Real-Time Network APIs

RTX64 4.x API	wRTOS 1.0 and later
RtnDecodePacket	Removed.
RtnDeleteCriticalLock	Removed.
RtnDisplayNbrCacheTable	Removed.
RtnDisplayRoutingTable	Removed.
RtnEnterCriticalLock	Removed.
RtnEnumPciCards	Removed.
RtnGetDataLong	Removed.
RtnGetIpAddress	Removed.
RtnGetMcastCount	Removed.
RtnGetMsPerTick	Removed.
RtnGetPacket	Removed.
RtnIndicateStatus	Removed.
RtnInitializeCriticalLock	Removed.

RTX64 4.x API	wRTOS 1.0 and later
RtnIOCTLDriver	Removed.
RtnIsStackOnline	Removed. Use **RttcpipGetComponentStatus** instead.
RtnLeaveCriticalLock	Removed.
RtnNotifyRecvQueue	Removed.
RtnNotifyTransmitQueue	Removed.
RtnRequest	Removed.
RtnSetDataLong	Removed.
RtnSetLinkAddress	Removed.
RtnSetDefaultGateway	Removed.
RtnTransmitCompleteCallback	Removed.

Real-Time Network Driver (RTND) APIs

RTX64 4.x API	wRTOS 1.0 and later
RtndAttachToReceiveQueue	Removed.
RtndAttachToTransmitQueue	Removed.
RtndConfigure	Removed.
RtndDetachReceiveQueue	Removed.
RtndDetachTransmitQueue	Removed.

RTX64 4.x API	wRTOS 1.0 and later
RtndInitDriverializeInterface	Removed.
RtndIoctl	Removed.
RtndReceive	Removed.
RtndReceiveWithCallback	Removed.
RtndRequest	Removed.
RtndServiceTransmitQueue	Removed.
RtndTransmit	Removed.
RtndTransmitEx	Removed.
RtndUpDown	Removed.
DrvISR	Removed.
RTND_MEDIA_CONNECT_STATE	Removed.
RTND_MEDIA_DUPLEX_STATE	Removed.
RTND_MEDIA_SPEED	Removed.
RTND_REQUEST	Removed.
RTND_STATISTICS_INFO	Removed.
RTND_STATUS	Removed.
RTNAL_ETHERNET_FILTER	Removed.

RTX64 4.x API	wRTOS 1.0 and later
RTNAL_FRAME	Removed.
RTNAL_HW_TIME	Removed.
RTNAL_INTERFACE	Removed.
RTNAL_PCI_LOCATION	Removed.
RTNAL_PTP_SET_TIMESTAMP_TYPE_ARGUMENTS	Removed.
RTNAL_PTP_MESSAGE_TYPE	Removed.
RTNAL_PTP_VERSION	Removed.
RTNAL_PHYSICAL_ADDRESS	Removed.

Removed/Deprecated Winsock APIs

RTX64 4.x API	wRTOS 1.0 and later
gethostbyname	Removed.

Removed/Deprecated Configure and Control (RTFW) APIs

RTX64 4.x API	wRTOS 1.0 and later
RtfwGetConsoleConfiguration	Removed. Use **RtfwGetConsoleConfiguration**.

RTX64 4.x API	wRTOS 1.0 and later
RtfwSetConsoleConfiguration	Removed. Use **RtfwSetConsoleConfiguration**.
RtfwCreateNetworkInterface	Removed. Use **Rtfwnl2SetInterfaceSettings**.
RtfwDeleteNetworkInterface	Removed. Use **Rtfwnl2DeleteInterface**.
RtfwGetAllNetworkInterfaces	Removed. Use **Rtfwnl2GetAllInterfaces** to return the configurations of all Network Link Layer (NL2) interfaces. Use **RtfwtcpipGetAllInterfaces** to enumerate all Network Link Layer (NL2) interfaces with TCP/IP settings.
RtfwGetNALClientProcessIDs	Removed. wRTOS 1.0 provides a Network Link Layer (NL2) that replaces the NAL from RTX64 4.x versions. Use **Rtfwnl2GetClientProcessIDs**.
RtfwGetNALConfiguration	Removed. wRTOS 1.0 provides a Network Link Layer (NL2) that replaces the NAL from RTX64 4.x versions. Use **Rtfwnl2GetConfiguration**.
RtfwGetNALLocalMemoryConfiguration	Removed. wRTOS 1.0 provides a Network Link Layer (NL2) component that replaces the NAL from RTX64 4.x versions. Use **Rtfwnl2GetConfiguration**.
RtfwGetNALStatus	Removed. wRTOS 1.0 provides a Network Link Layer (NL2) component that replaces the NAL from RTX64 4.x versions. Use **RtNL2GetComponentStatus**.
RtfwGetNetworkInterfaceByName	Removed. Use **Rtfwnl2GetInterface**.

RTX64 4.x API	wRTOS 1.0 and later
RtfwGetNetworkInterfacesStatus	Removed.
RtfwGetNetworkVerbosity	Removed. wRTOS 1.0 provides a Network Link Layer (NL2) component that replaces the NAL from RTX64 4.x versions. Use **Rtfwnl2GetVerbosity** or **RtfwtcpipGetVerbosity**.
RtfwGetTCPIPClientProcessIDs	Removed. Use **RtfwtcpipGetClientProcessIDs**.
RtfwGetTCPIPConfiguration	Renamed **RtfwtcpipGetConfiguration**.
RtfwGetTCPIPLocalMemoryConfiguration	Removed. Use **RtfwtcpipGetConfiguration**.
RtfwGetTCPIPStatus	Removed. Use **RttcpipGetComponentStatus**.
RtfwGetRTX64Devices	Removed. Use **RtfwGetwRTOSDevices**.
RtfwGetRTX64DevicesEx	Removed. Use **RtfwGetwRTOSDevices**.
RtfwSetNALConfiguration	Removed. wRTOS 1.0 provides a Network Link Layer (NL2) component that replaces the NAL from RTX64 4.x versions. Use **Rtfwnl2SetConfiguration**.
RtfwSetNALLocalMemoryConfiguration	Removed. wRTOS 1.0 provides a Network Link Layer (NL2) component that replaces the NAL from RTX64 4.x versions. Use **Rtfwnl2SetConfiguration**.
RtfwSetNetworkVerbosity	Removed. wRTOS 1.0 provides a Network Link Layer (NL2) component that replaces the NAL from RTX64 4.x versions. Use **Rtfwnl2SetVerbosity**.

RTX64 4.x API	wRTOS 1.0 and later
RtfwSetTCPIPConfiguration	Renamed **RtfwtcpipSetConfiguration**.
RtfwSetTCPIPLocalMemoryConfiguration	Removed. Use **RtfwtcpipSetConfiguration**.
RtfwStartNAL	Removed. wRTOS 1.0 provides a Network Link Layer (NL2) component that replaces the NAL from RTX64 4.x versions. Use **RtnI2StartComponent** in **RtfwnI2Api.dll**.
RtfwStartTCPIPStack	Removed. Use **RttcpipStartComponent** in **RtfwtcpipApi.dll**.
RtfwStopNAL	Removed. wRTOS 1.0 provides a Network Link Layer (NL2) component that replaces the NAL from RTX64 4.x versions. Use **RtnI2StopComponent** in **RtfwnI2Api.dll**.
RtfwStopTCPIPStack	Removed. Use **RttcpipStopComponent** in **RtfwtcpipApi.dll**.
RTFW_CONSOLE_CONFIGURATION	Removed. Use **RTFW_CONSOLE_CONFIGURATION**.
RTFW_DEVICE	Removed. Use **RTFW_DEVICEEX**.
RTFW_NAL_CONFIGURATION	Removed. wRTOS 1.0 provides a Network Link Layer (NL2) component that replaces the NAL from RTX64 4.x versions. Use **RTFW_NL2_CONFIGURATION**.
RTFW_NETWORK_INTERFACE	Removed. Use **RTFW_NL2_INTERFACE** for NL2 interfaces. Use **RTFW_TCPIP_INTERFACE** for TCP/IP interfaces.
RTFW_NETWORK_INTERFACE_STATUS	Removed.

Removed/Deprecated Managed APIs

RTX64 4.x API	wRTOS 1.0 and later
IntervalZero.MaxRT.wRTOS.RtApi.Runtime.InteropServices Namespace	Removed, along with: • **RtHandle** Class • **RtHandleType** Enumeration • **RtMarshal** Class
RTX64Object.CreationTime Property	Removed.
RTX64Object.Name Property	Removed.
Config.ServerConsole	These properties were removed: • **LogFileName** • **SuppressWarnings**
Config.Subsystem.FreezeFaultingProcessOnException property **Config.Subsystem.ExceptionBehavior** property **Config.Subsystem.UseStructuredExceptionHandling** property	Replaced by these new Managed APIs: ◦ Enumeration **wRTOS.Config.Subsystem.ExceptionHandling** represents the available exception handling methods. Enumerators from this enumeration can be assigned to property **Subsystem.ExceptionHandlingMode**. ◦ Property **wRTOS.Config.Subsystem.ExceptionHandlingMode** accesses the **Subsystem.ExceptionHandling** enumeration specifying the available exception handling methods.
Config.ScheduledProcess.UseLocalMemory Property	Removed.

RTX64 4.x API	wRTOS 1.0 and later
Config.IConfigNetwork Interface	Removed.
Config.IConfigNetworkFilter Interface	Removed.
Config.IConfigNetworkInterface Interface	Removed.
Config.IConfigTCPIP Interface	Removed.
Config.IConfigIPv4Configuration Interface	Removed.
Config.IPv4AddressNetMaskPair Structure	Removed.
Config.IPv4Configuration Class	Removed.
Config.Network Class	Removed.
Config.NetworkFilter Class	Removed.
Config.NetworkInterface Class	Removed.
Config.TCPIP Class	Removed.
Config.NetworkInterface.InterruptType Enumeration	Removed.
Config.NetworkInterface.SpeedDuplexType Enumeration	Removed.
Control.Subsystem.StartNAL	Removed. Use **NL2.Control.Start**.
Control.Subsystem.StopNAL	Removed. Use **NL2.Control.Stop**.
Control.Subsystem.GetNALStatus	Removed. Use **NL2.Control.GetStatus**.

RTX64 4.x API	wRTOS 1.0 and later
Control.Subsystem.StartTcpipStack	Removed. Use **Tcpip.Control.Start**.
Control.Subsystem.StopTcpipStack	Removed. Use **Tcpip.Control.Stop**.
Control.Subsystem.GetTCPIPStatus	Removed. Use **Tcpip.Control.GetStatus**.
Control.Subsystem.NetworkInterfaceStatus	Removed.
Control.Subsystem.GetNetworkInterfacesStatus	Removed.
Control.Subsystem.NetworkControlRegion	Removed.
Control.Subsystem.StateEx Property	Merged with and replaced by the **Control.Subsystem.State** property.
RtApi.Diagnostics.RTProcess.Start Method	Removed these overloads: • **Start** • **Start(Boolean)** • **Start(ProcessStartInfo)** • **Start(String)** • **Start(String, String)** • **Start(ProcessStartInfo, UInt64, UInt32)** • **Start(String, UInt64, UInt32)** • **Start(String, String, UInt64, UInt32)**
Monitor.Subsystem.EnableComponents Method	Removed.
Monitor.Subsystem.GetEnabledComponents Method	Removed.

RTX64 4.x API	wRTOS 1.0 and later
Monitor.MonitorEventContiguousMemoryAlloc Class	Removed the class and all its members.
Monitor.MonitorEventContiguousMemoryAllocFail Class	Removed the class and all its members.
Monitor.MonitorEventContiguousMemoryAllocSpecifyCache Class	Removed the class and all its members.
Monitor.MonitorEventContiguousMemoryAllocSpecifyCacheFail Class	Removed the class and all its members.
Monitor.MonitorEventContiguousMemoryFree Class	Removed the class and all its members.
Monitor.MonitorEventContiguousMemoryFreeFail Class	Removed the class and all its members.
Monitor.MonitorEventWindowsMemoryAlloc Class	Removed the class and all its members.
Monitor.MonitorEventWindowsMemoryAllocFail Class	Removed the class and all its members.
Monitor.MonitorEventWindowsMemoryFree Class	Removed the class and all its members.
Monitor.MonitorEventWindowsMemoryFreeFail Class	Removed the class and all its members.
Monitor.MonitorEventShutdownHandlerCall Class	Removed the class and all its members.
Monitor.MonitorEventLocalMemoryExpand Class	Removed the class and all its members.
Monitor.MonitorEventLocalMemoryShrink Class	Removed the class and all its members.
Config.Subsystem.EnableLocalMemory Property	Removed.

RTX64 4.x API	wRTOS 1.0 and later
Config.Subsystem.ResetEnableLocalMemory Method	Removed

Removed Error Codes

Real-Time APIs

RTX64 4.x Error Code	wRTOS 1.0 and later
RT_ERROR_MONITORING_IS_STARTED	Removed.
RT_ERROR_MONITORING_NOT_ENABLED	Removed.
RT_ERROR_MONITORING_INVALID_STATE_TRANSITION	Removed.

7

Networking

In this chapter, we compare the RTX64 and MaxRT wRTOS network architectures and highlight important functionality differences.

TOPICS:

- Network Components
- Configuring and Controlling the Network
- Porting a NAL Driver to the NL2

Network Components

This section compares the network components available in RTX64 and MaxRT wRTOS.

RTX64 Network Components

The RTX64 network has two primary components: a Network Abstraction Layer (NAL) and a TCP/IP protocol stack.

Network Abstraction Layer (NAL)

The NAL is a network layer that abstracts the network hardware and driver functions from the upper-level protocol stacks and provides management interfaces for those upper layers to query for and use available network assets easily. It is a separate protocol layer from the TCP/IP Stack. Using the NAL, you can use network functionality such as EtherCAT, TSN (Time Sensitive Networks), and PTP (Precision Time Protocol).

TCP/IP Stack

The TCP/IP Stack is an optional purchasable protocol stack by Treck that provides deterministic processing and networking capability. The TCP/IP Stack is dependent on the NAL. The TCP/IP component is always installed by the RTX64 Runtime but requires a separate purchasable license.

wRTOS Network Components

wRTOS provides networking capability through a base component called the Network Link Layer (NL2) and a set of optional protocol components stacked above the NL2. These components run within the RTSS environment.

The NL2 offers raw access to the Ethernet hardware, while the protocol components offer high-level functionality such as TCP/IP and E-CAT. Applications can use the services of one or more of these network components simultaneously.

Network Link Layer (NL2)

The Network Link Layer (NL2) software component provides real-time applications with abstract APIs to access network services at the Layer 2 of the OSI model, independent of the underlying hardware. This typically includes:

- Transmit and receive raw Ethernet packets
- Configure and get the link status
- Configure hardware timestamping
- Configure hardware filters
- Configure QoS

Using the NL2, you can easily integrate network functionality such as TCP/IP and E-CAT.

To view a list of network cards supported off-the-shelf by the NL2, see Supported Network Interface Cards in the Help.

> **Note:** The NL2 component is an optional feature in the wRTOS Runtime product package. All upper-level networking features require the NL2.

TCP/IP Stack

The TCP/IP Stack component is a protocol stack by Treck that provides processing and network capability within the RTSS environment. wRTOS provides an API that conforms to a subset of the functions defined in the Windows Sockets 2.0 (Winsock) specification for Windows. The wRTOS Stack is the same as the RTX64 Stack but with different underpinnings.

This component requires a license. TCP/IP is included in the wRTOS Basic Networking feature that is optionally installed with the wRTOS Runtime. However, to enable this feature, you must have an wRTOS Basic Networking license (WNET64). Contact IntervalZero Sales to purchase licenses.

The TCP/IP Stack relies on the NL2 for Layer 2 access.

Network Relay

The Network Relay component allows Windows and RTSS applications to use the same physical NIC at the same time

GigE Vision

GigE Vision provides functionality for using GigE Vision Cameras within the real-time wRTOS environment. Using Vision, you can quickly discover cameras on the network, query different camera configurations, and acquire image data. Through a provided communication library images can be passed between RTSS and Windows. Or a third-party vision library, such as OpenCV, can be layered on top of GigE Vision interface to provide additional image processing functionality within RTSS.

This component requires a license. GigE Vision can be installed with wRTOS Runtime. However, to enable this feature, you must purchase and activate the wRTOS GigE Vision (WVIS64) and wRTOS Basic Networking (WNET64) packages. Contact IntervalZero Sales to purchase product licenses.

E-CAT

The E-CAT component exposes Windows and real-time interfaces that offer support for CANopen over EtherCAT and simplifies the configuration of EtherCAT networks with its unique plug-and-play approach.

Optional add-on feature packages:

- High-Speed Timer
- Multiple MainDevice
- Hot Connect
- Cable Redundancy

This component requires a license. The E-CAT component requires a wRTOS Fieldbus license (WFBS64). Contact IntervalZero Sales to purchase licenses.

Configuring and Controlling the Network

This section compares the methods available in RTX64 and MaxRT wRTOS for configuring and controlling network components.

Configuring and Controlling the RTX64 Network

The RTX64 Control Panel provides several options for configuring NAL and TCP/IP Stack behavior and performance on the **Configure and control the network** page. You can also manually start, stop, and restart the NAL and/or TCP/IP Stack directly from this page.

From the **Manage interfaces** page in the Control Panel, you can add, delete, set properties for, and associate filters with RTX64 interfaces.

Note: If the optional TCP/IP Stack is in use, you can restart it independent from the NAL and network drivers if only TCP/IP-specific interface configuration settings, such as IP addresses, or the Stack itself, has changed. Otherwise, the NAL and TCP/IP Stack must be restarted together.

Configuring and Controlling the wRTOS Network

wRTOS Settings includes many options for configuring wRTOS network components and managing network interfaces.

The Network page (**Contents / Network**) contains links to these sub-pages:

Icon	Name	Description
	Interfaces	Configure network interface-specific settings.

Icon	Name	Description
	Network Link Layer (NL2)	Configure default Network Link Layer (NL2) settings. The NL2 software component provides real-time applications with abstract APIs to access network services at the Layer 2 of the OSI model, independent of the underlying hardware. **Note:** The NL2 component is included as an optional feature in the wRTOS Runtime product package. All upper-level networking features require the NL2.
	TCP/IP	Configure default TCP/IP settings. **This component requires a license.** TCP/IP is included in the wRTOS Basic Networking feature that is optionally installed with the wRTOS Runtime. However, to enable this feature, you must have a MaxRT wRTOS Basic Networking license (WNET64). Contact IntervalZero Sales to purchase licenses.

Icon	Name	Description
	E-CAT	Configure default E-CAT settings. E-CAT exposes Windows and real-time interfaces that offer support for CANopen over EtherCAT and simplifies configuration of EtherCAT networks with its unique plug-and-play approach. **This component requires a license.** E-CAT is included in the wRTOS Fieldbus feature that is optionally installed with the wRTOS Runtime. However, to enable this feature, you must have a MaxRT wRTOS Fieldbus license (WFB64). Contact IntervalZero Sales to purchase licenses.

You can start, stop, and restart network components using wRTOS Settings or the wRTOS Control Panel.

Porting a NAL Driver to the NL2

This section describes the steps required to port an RTX64 4.x NAL NIC driver to a NL2 NIC driver that can be used by the wRTOS Network Link Layer (NL2).

For information on NL2 NIC drivers, see Creating a NIC Driver in the Help.

IN THIS SECTION:

- Header and Library Requirements
- Rework the Startup Logic
- Rework the Shutdown Logic
- Use NL2 Function Pointers instead of Direct Function Calls
- Rework the Interrupts Handling
- Rework the Link Status Monitoring Logic
- Rework the Frame Transmission Logic
- Rework the Frame Reception Logic

- Remove Unnecessary Locks
- Replace the RtndIoctl Function
- Replace the RtndRequest function
- Implement Frame Buffer Allocation Functions

Header and Library Requirements

To integrate your driver with the NL2, include the **Rtnd.h** header file. Do not use **Rtnl2Api.h** to prevent errors. Remove the following header files and library from your driver if they were previously referenced:

- Headers: **rtNalApi.h**, **rtnapi.h**, **winsock2.h**, **ws2tcpip.h**
- Library: **RTX64Nal.lib**

Rework the Startup Logic

During its startup phase, the NAL loads the NIC drivers and calls the following functions for each enabled NIC:

- RtndInitDriverializeInterface
- RtndConfigure
- RtndUpDown

In contrast, the NL2 calls these functions:

- RtndInitDriver
- RtndManageInterface
- RtndQueryInterfaceCapability
- RtndSetInterface
- RtndStartInterface
- RtndQueryMacAddress
- RtndQueryInterfaceFeature

Rework the Shutdown Logic

During its shutdown phase, the NAL calls this function:

- RtndUpDown

In contrast, the NL2 calls these functions:

- RtndShutdownInterface
- RtndStopInterface
- RtndUnmanageInterface

Use NL2 Function Pointers instead of Direct Function Calls

NAL NIC drivers link with **RTX64Nal.lib** and request NAL services by calling the following functions exported by the NAL process:

- RtnEnumPciCards
- RtnGetDeviceName
- RtnSetLinkAddress
- RtnGetMcastCount
- RtnIndicateStatus
- RtnNotifyTransmitQueue
- RtnTransmitCompleteCallback
- RtnNotifyRecvQueue
- RtnSetDataLong
- RtnGetDataLong
- RtnGetPacket
- RtnDecodePacket
- RtnIntializeCriticalLock
- RtnDeleteCriticalLock
- RtnEnterCriticalLock
- RtnLeaveCriticalLock

In contrast, NL2 NIC drivers request NL2 services using the function pointers supplied by the NL2 in the call to RtndInitDriver:

- GetVerbose
- NotifyLinkStatusChange
- NotifyTxInterrupt
- NotifyRxInterrupt
- NotifyEgressTimestamp
- CreateTxBuffers
- DestroyTxBuffers
- CreateRxBuffers
- DestroyRxBuffers

Rework the Interrupts Handling

Interrupts Enabling

NAL NIC drivers are expected to enable the following interrupts at startup and keep them enabled until the NAL shuts down:

- Link Status Change interrupt
- Transmit interrupt
- Receive interrupt

NL2 NIC drivers are expected to enable the following interrupts at startup:

- Link Status Change interrupt
- Egress Timestamp interrupt (if the hardware supports Egress Timestamping)

The NL2 determines whether Transmit and Receive interrupts should be enabled by calling the following functions at run time:

- RtndEnableTxInterruptSource
- RtndEnableRxInterruptSource

Interrupts Servicing

In NAL NIC drivers, an IST typically does the following work:

- In case of a Link Status Change interrupt:
  - Set an event to wake up the Line Status thread
- In case of a Transmit interrupt:
  - Call **RtnNotifyTransmitQueue** (which wakes up the application's Transmit Complete thread) or set an event to wake up the driver's Transmit Complete thread
- In case of a Receive interrupt:
  - If the queue is attached to an application, update the count of received frames and call **RtnNotifyRecvQueue** (which signals the application's Receive event), otherwise, flush the queue (extract the filled buffers and re-submit them immediately)

> **Note:** NAL NIC drivers ignore Egress Timestamp interrupts.

In NL2 NIC drivers, an IST should typically do the following work:

- In case of a Link Status Change interrupt:
  - Call **RTND_CALLBACKS.NotifyLinkStatusChange**
- In case of a Transmit interrupt:
  - Call **RTND_CALLBACKS.NotifyTxInterrupt**
- In case of a Receive interrupt:
  - Call **RTND_CALLBACKS.NotifyRxInterrupt**
- In case of an Egress Timestamp interrupt:
  - Call **RTND_CALLBACKS.NotifyEgressTimestamp**

> **Note:** NL2 drivers should not access the DMA rings or use a lock in their ISTs.

Rework the Link Status Monitoring Logic

NAL NIC drivers have their Line Status Monitoring thread, which awakes every time a Link Status Change interrupt occurs. The thread performs the following tasks:

- Reads the new link status from the NIC registers
- Reconfigures some of the NIC registers, if required
- Calls **RtnIndicateStatus**

NL2 NIC drivers don't need to have their Line Status Monitoring thread because the NL2 itself implements the monitoring logic. NL2 NIC drivers only need to provide the RtndQueryLinkStatus function, which should do the following:

- Read the new link status from the NIC registers
- Reconfigure some of the NIC registers, if required
- Return the new link status to the NL2

Rework the Frame Transmission Logic

About Frame Transmission

In the NAL, the general rule is a dedicated Transmit Complete thread services frame transmission. Depending on the application's request, this thread can run within the context of the application process or the context of the driver (i.e. the context of the NAL process). When running in the application context, the Transmit Complete thread calls a callback provided by the application every time a buffer in the queue is consumed. When running in the context of the driver, the Transmit Complete thread simply extracts the consumed buffer and makes it available for future transmission.

In the NL2, frame transmission is always serviced at the initiative of the NL2, which calls RtndExtractTxBuffer.

Start/Stop the Queue

This is an optional feature that doesn't exist in NAL NIC drivers.

If a NIC supports dynamically starting/stopping a Transmit Queue independently of the other Transmit Queues, the NL2 NIC driver should implement the following functions:

- RtndStartTxQueue
- RtndStopTxQueue

The NL2 will call these functions to ensure that a Transmit Queue is in the same initial state each time a new application acquires it.

Attach Transmit Queues

When an application acquires a Transmit Queue and enables the Transmit Complete callback functionality, the NAL calls the following function of the NIC driver:

- RtndAttachToTransmitQueue

This function notifies the driver that Transmit interrupts for this queue should be serviced by the application's Transmit Complete thread rather than the driver's Transmit Complete thread.

For NL2 NIC drivers, the equivalent function is RtndAttachTxQueue, which is called from the context of any process that plans to use this queue.

Submit Buffers

When the application provides a buffer containing a frame to transmit, the NAL calls the following functions of the NIC driver:

- RtndTransmit
- RtndTransmitEx

In contrast, the NL2 calls these functions:

- RtndSubmitTxBuffer
- RtndApplyTxBuffers

Extract Buffers

When a Transmit Queue is acquired by an application that enabled the Transmit Complete Callback functionality, the NAL calls the following function from the application's Transmit Complete thread to extract the consumed buffers:

- RtndServiceTransmitQueue

In other cases, the NAL driver is responsible for automatically extracting the consumed buffers using its Transmit Complete thread.

In contrast, the extraction of consumed buffers by the NL2 NIC drivers is always done upon request from the NL2 using the following function:

- RtndExtractTxBuffer

Detach Transmit Queues

When an application releases a Transmit Queue that had the Transmit Complete callback functionality enabled, the NAL calls the following function of the NIC driver:

- RtndDetachTransmitQueue

This function notifies the driver that Transmit interrupts for this queue should now be serviced by the driver's Transmit Complete thread rather than the application's Transmit Complete thread.

For NL2 NIC drivers, the equivalent function is RtndDetachTxQueue, which is called from the context of any process that previously called .

Rework the Frame Reception Logic

About Frame Reception

In the NAL, frame reception is serviced either by the application that acquired the queue or by the driver's IST when no application has acquired the queue.

In the NL2, frame reception is always serviced at the initiative of the NL2, which calls RtndExtractRxBuffer.

Start/Stop the Queue

This is an optional feature that doesn't exist in NAL NIC drivers.

If a NIC supports dynamically starting/stopping a Receive Queue independently of the other Receive Queues, the NL2 NIC driver should implement the following functions:

- RtndStartRxQueue
- RtndStopRxQueue

The NL2 will call these functions to ensure that a Receive Queue is in the same initial state each time a new application acquires it.

Attach Receive Queues

When an application acquires a Receive Queue, the NAL calls the following function of the NIC driver:

- RtndAttachToReceiveQueue

This function notifies the driver that Receive interrupts for this queue must be serviced by the application rather than the driver's IST.

For NL2 NIC drivers, the equivalent function is RtndAttachRxQueue, which is called from the context of any process that plans to use this queue.

Extract Buffers

When an application acquires a Receive Queue, the NAL calls the following functions from the application's process to extract the filled buffers:

- RtndReceive
- RtndReceiveWithCallback

In other cases, the NAL driver is responsible for automatically extracting the filled buffers in the IST and re-submitting them immediately to the queue.

For NL2 NIC drivers, buffer extraction is always done upon request from the NL2 using the following function:

- RtndExtractRxBuffer

Submit Buffers

NAL NIC drivers automatically re-submit the extracted buffers after they have been processed (either after they have been copied to the application's buffer by **RtndReceive** or after they have been supplied to the application through the Receive Callback function).

In contrast, NL2 NIC drivers do not re-submit automatically, they wait for the NL2 to call the following functions:

- RtndSubmitRxBuffer
- RtndApplyRxBuffers

Detach Receive Queues

When an application releases a Receive Queue, the NAL calls the following function of the NIC driver:

- RtndDetachReceiveQueue

This function notifies the driver that the driver's IST should now service Receive interrupts for this queue rather than the application.

For NL2 NIC drivers, the equivalent function is RtndDetachRxQueue, which is called from the context of any process that previously called RtndAttachRxQueue.

Remove Unnecessary Locks

NAL NIC drivers generally need to use their locks for at least the following purposes:

- One lock per Transmit Queue, used by the driver's Transmit Complete thread and by the following functions to protect concurrent access to the queue registers: **RtndAttachToTransmitQueue**, **RtndDetachTransmitQueue**, **RtndTransmit**, **RtndTransmitEx**, **RtndServiceTransmitQueue**.
- One lock per Receive Queue, used by the IST and by the following functions to protect concurrent access to the queue registers: **RtndAttachToReceiveQueue**, **RtndDetachReceiveQueue**, **RtndReceive**, **RtndReceiveWithCallback**, **RtndIoctl**.
- One general lock, used by the following function to protect concurrent access to the general NIC registers: **RtndIoctl**.

The NL2 implements its locks and ensures that specific functions are called from within a single thread only. Therefore, NL2 NIC drivers don't require additional locks.

Replace the RtndIoctl Function

The NAL calls the **RtndIoctl** function for different purposes as the command code specifies. The table below lists the equivalent functions of the NL2 NIC driver that provide the same services:

RtndIoctl command code	NL2 NIC driver equivalent function
ENIOCPROMISC	RtndSetPromiscuousMode
ENIOCALL, ENIOCNORMAL	RtndSetPassBadFramesMode
ENIOADDMULTI, ENIODELMULTI	RtndSetMulticastFilter
ENIOLINKSTATUS	RtndQueryLinkStatus
RTNAL_IOCTL_RX_MONITOR	There is no equivalent; this functionality doesn't exist in the NL2.
RTNAL_IOCTL_TX_MONITOR	There is no equivalent; this functionality doesn't exist in the NL2.
RTNAL_IOCTL_READ_TX_HW_TIMESTAMP	RtndExtractLastTxTimestamp

Rtndloctl command code	NL2 NIC driver equivalent function
RTNAL_IOCTL_READ_RX_HW_TIMESTAMP	There is no equivalent; this functionality doesn't exist in the NL2.
RTNAL_IOCTL_READ_HW_TIMER	There is no equivalent; this functionality doesn't exist in the NL2.
RTNAL_IOCTL_READ32_REG	There is no equivalent; this functionality doesn't exist in the NL2.
RTNAL_IOCTL_WRITE32_REG	There is no equivalent; this functionality doesn't exist in the NL2.
RTNAL_IOCTL_READ_RX_PACKET_COUNT	There is no equivalent; this functionality doesn't exist in the NL2.
RTNAL_IOCTL_READ_TX_PACKET_COUNT	There is no equivalent; this functionality doesn't exist in the NL2.
RTNAL_IOCTL_USE_RX_NOTIFICATIONS	There is no equivalent; this functionality doesn't exist in the NL2.
RTNAL_IOCTL_SET_RX_POLLING	There is no equivalent; this functionality doesn't exist in the NL2.
RTNAL_IOCTL_CLEAR_RX_POLLING	There is no equivalent; this functionality doesn't exist in the NL2.
RTNAL_IOCTL_GET_RX_ETHERTYPE_FILTER	RtndGetDispatcherEtherTypeEntry
RTNAL_IOCTL_SET_RX_ETHERTYPE_FILTER, RTNAL_IOCTL_CLEAR_RX_ETHERTYPE_FILTER	RtndSetDispatcherEtherTypeEntry

Rtndloctl command code	NL2 NIC driver equivalent function
RTNAL_IOCTL_SET_RX_MESSAGE_TYPE_TO_TIMESTAMP	There is no equivalent; this functionality doesn't exist in the NL2.
RTNAL_IOCTL_SET_TX_MESSAGE_TYPE_TO_TIMESTAMP	There is no equivalent; this functionality doesn't exist in the NL2.
RTNAL_IOCTL_GET_SYSTEM_TIMER	There is no equivalent; this functionality doesn't exist in the NL2.
RTNAL_IOCTL_SET_INTERRUPT_MODERATION	RtndSetInterruptModeration
RTNAL_IOCTL_GET_PCI_BUS_LOCATION	There is no equivalent; this functionality doesn't exist in the NL2.

Replace the RtndRequest Function

The NAL calls the **RtndRequest** function for different purposes as specified by the OID. The table below lists the equivalent functions of the NL2 NIC driver that provide the same services:

RtndRequest OID	NL2 NIC driver equivalent function
RTND_OID_GEN_MAC_ADDRESS	
RTND_OID_GEN_MEDIA_CONNECT_STATUS, RTND_OID_GEN_MEDIA_DUPLEX_STATE, RTND_OID_GEN_LINK_SPEED	
All other OIDs	There is no equivalent; these features don't exist in the NL2.

Implement Frame Buffer Allocation Functions

The following functions are specific to the NL2 and must be implemented only in NL2 NIC drivers. They allow the NL2 to allocate or release a batch of buffers used to hold Ethernet frames for a specific Transmit or Receive Queue. These functions are necessary because each NIC may have special requirements for allocating frame buffers, making it the driver's responsibility to handle these allocations.

The functions to implement are as follows:

- RtndAllocateTxFrameDataBuffers
- RtndFreeTxFrameDataBuffers
- RtndAllocateRxFrameDataBuffers
- RtndFreeRxFrameDataBuffers

Support

For help with wRTOS, contact IntervalZero Technical Support by phone or access the online support resources available at https://www.intervalzero.com/en-support/en-customer-service/

Contacting Technical Support by Phone

> **Note:** If you purchased an IntervalZero product through a third-party reseller, contact the reseller for support.

Location	Number	Hours
United States	1-781-996-4481 At the prompt, press 3 for Support.	Monday - Friday, 8:30 a.m. – 5:30 p.m. US Eastern Time (GMT-500), excluding holidays.
R.O.C. Taiwan	+ 886-2-2556-8117	Monday - Friday, 9:00 a.m. – 5:00 p.m. Taipei Standard Time (GMT+8), excluding holidays.

Before Calling Technical Support

Please have the following information ready before calling IntervalZero Technical Support:

- **Your Support ID**: Customers who purchase direct support receive an e-mail address and password for accessing the IntervalZero Customer Support Portal.
- **The version number of your wRTOS software**

> **Note:** Make sure you have a valid maintenance contract.

Online Resources

Visit https://www.intervalzero.com/en-support/en-customer-service/ to log in to the Customer Support Portal (valid credentials are required), access online product Help, and view Support and Lifecycle policies and Product Release Notices.

Index

A

about 1
activating 4, 6
    to dongles 7
APIs
    deprecated 68
    enhanced 60
    headers 15
    libraries 15
    managed 19
    new 21
    removed 68

B

breaking API changes 62
breaking changes to error codes 67

C

code changes
    breaking 62, 67
    enhanced APIs 60
    new APIs 21
configuring
    cores 6
    Runtime 8
controlling
    components 8
    network 85
cores
    configuring 4

D

deprecated APIs 68

E

E-CAT 84
error codes
    removed 81
EtherCAT 84

F

Framework APIs
    breaking changes 64
    removed or deprecated 73

G

GigE Vision 84

L

libraries
    managed 19
licenses 7
    types 7
licensing 4

M

Managed APIs
    breaking changes 65
    removed or deprecated 77

N

NAL 82
network 82
    components 82
    configuring 85
    controlling 85
    E-CAT 84

GigE Vision 84
NAL 82
Network Link Layer (NL2) 83
NIC sharing 84
TCP/IP 82-83
Network Link Layer (NL2) 83
NIC Sharing 84
NL2
porting a NAL driver to 87

P

porting
NAL driver to NL2 87
product
comparison 2
packages 2
product codes 4
products
activating 6
project
configurations 15
settings 15

R

Real-Time APIs
breaking changes 62
enhancements 61
removed or deprecated 68
removed APIs 68
removed error codes 81
Runtime
configuring 8
controlling 8

S

SDK
code changes 21
headers 15
libraries 15
managed libraries 19

T

TCP/IP 82-83
tools 10
for activating components 10
for analyzing 11
for configuring components 10
for controlling applications 12
for controlling components 10
for EtherCAT 13
for logging 11
for measuring latency 13-14
for monitoring 11
for networking 13
for viewing application output 12