ETAS

RTA-OS RH850x2/GHS V5.0.3

Port Guide
Status: Released

DRIVING | EMBEDDED EXCELLENCE

Copyright

The data in this document may not be altered or amended without special notification from ETAS GmbH. ETAS GmbH undertakes no further obligation in relation to this document. The software described in it can only be used if the customer is in possession of a general license agreement or single license. Using and copying is only allowed in concurrence with the specifications stipulated in the contract. Under no circumstances may any part of this document be copied, reproduced, transmitted, stored in a retrieval system or translated into another language without the express written permission of ETAS GmbH.

©Copyright 2008-2019 ETAS GmbH, Stuttgart.

The names and designations used in this document are trademarks or brands belonging to the respective owners.

Document: 10806-PG-5.0.3 EN-06-2019

Safety Notice

This ETAS product fulfills standard quality management requirements. If requirements of specific safety standards (e.g. IEC 61508, ISO 26262) need to be fulfilled, these requirements must be explicitly defined and ordered by the customer. Before use of the product, customer must verify the compliance with specific safety standards.

Contents

1 Introduction **7**
1.1 About You . . . 7
1.2 Document Conventions . . . 8
1.3 References . . . 8

2 Installing the RTA-OS Port Plug-in **9**
2.1 Preparing to Install . . . 9
2.1.1 Hardware Requirements . . . 9
2.1.2 Software Requirements . . . 9
2.2 Installation . . . 10
2.2.1 Installation Directory . . . 10
2.3 Licensing . . . 11
2.3.1 Installing the ETAS License Manager . . . 11
2.3.2 Licenses . . . 12
2.3.3 Installing a Concurrent License Server . . . 13
2.3.4 Using the ETAS License Manager . . . 14
2.3.5 Troubleshooting Licenses . . . 16

3 Verifying your Installation **19**
3.1 Checking the Port . . . 19
3.2 Running the Sample Applications . . . 19

4 Port Characteristics **21**
4.1 Parameters of Implementation . . . 21
4.2 Configuration Parameters . . . 21
4.2.1 Stack used for C-startup . . . 21
4.2.2 Stack used when idle . . . 22
4.2.3 Stack overheads for ISR activation . . . 22
4.2.4 Stack overheads for ECC tasks . . . 22
4.2.5 Stack overheads for ISR . . . 22
4.2.6 Orti22/Lauterbach . . . 23
4.2.7 Orti23/Lauterbach . . . 23
4.2.8 ORTI Stack Fill . . . 23
4.2.9 Direct Vector mode . . . 24
4.2.10 EBASE register . . . 24
4.2.11 Linker Model . . . 24
4.2.12 Enable untrusted stack check . . . 24
4.2.13 GetAbortStack always . . . 25
4.2.14 Handle FPU context . . . 25
4.2.15 Cache Core ID . . . 25
4.2.16 Core Specific Wrappers . . . 26
4.2.17 misalign_pack . . . 26
4.2.18 stack_protector . . . 27
4.2.19 SDA Threshold . . . 27
4.2.20 Optimizer setting . . . 27
4.2.21 SDA size setting . . . 27
4.2.22 Option set 1 . . . 28

4.2.23 Enhanced Isolation 28
4.3 Generated Files 29

5 Port-Specific API 30
5.1 API Calls 30
5.1.1 Os_AwaitStartup 30
5.1.2 Os_CacheCoreID 31
5.1.3 Os_InitializeVectorTable 32
5.2 Callbacks 33
5.2.1 Os_Cbk_GetAbortStack 33
5.2.2 Os_Cbk_StartCore 34
5.3 Macros 36
5.3.1 CAT1_ISR 36
5.3.2 Os_Clear_x 36
5.3.3 Os_DisableAllConfiguredInterrupts_Corex 36
5.3.4 Os_Disable_x 37
5.3.5 Os_EnableAllConfiguredInterrupts_Corex 37
5.3.6 Os_Enable_x 37
5.3.7 Os_IntChannel_x 38
5.4 Type Definitions 38
5.4.1 Os_StackSizeType 38
5.4.2 Os_StackValueType 38

6 Toolchain 39
6.1 Compiler Versions 39
6.1.1 Green Hills v2018.1.5 39
6.2 Options used to generate this guide 39
6.2.1 Compiler 39
6.2.2 Assembler 41
6.2.3 Librarian 42
6.2.4 Linker 42
6.2.5 Debugger 43

7 Hardware 45
7.1 Supported Devices 45
7.2 Register Usage 45
7.2.1 Initialization 45
7.2.2 Modification 46
7.3 Starting Cores in Multicore Configurations 47
7.4 Core/Processor State when StartOS() is called 47
7.5 Core ID Caching 48
7.6 Interrupts 48
7.6.1 Interrupt Priority Levels 48
7.6.2 Allocation of ISRs to Interrupt Vectors 49
7.6.3 Cross-core Interrupts 49
7.6.4 CPU Vectors and the ProtectionHook or Default Interrupt . 49
7.6.5 Recovering from CPU Exceptions/Interrupts that call the ProtectionHook 50
7.6.6 Interrupt Priority Level Extension 50

7.6.7 Register Banks 50
7.6.8 Vector Table 50
7.6.9 Using Raw Exception Handlers 52
7.6.10 Writing Category 1 Interrupt Handlers 52
7.6.11 Writing Category 2 Interrupt Handlers 52
7.6.12 Default Interrupt 53
7.7 Memory Model 53
7.8 Processor Modes 53
7.9 Stack Handling 53

8 Enhanced Isolation **54**

9 Performance **55**
9.1 Measurement Environment 55
9.2 RAM and ROM Usage for OS Objects 55
9.2.1 Single Core 56
9.2.2 Multi Core 56
9.3 Stack Usage 56
9.4 Library Module Sizes 57
9.4.1 Single Core 57
9.4.2 Multi Core 59
9.5 Execution Time 62
9.5.1 Context Switching Time 63

10 Finding Out More **65**

11 Contacting ETAS **66**
11.1 Technical Support 66
11.2 General Enquiries 66
11.2.1 ETAS Global Headquarters 66
11.2.2 ETAS Local Sales & Support Offices 66

1 Introduction

RTA-OS is a small and fast real-time operating system that conforms to both the AUTOSAR OS (R3.0.1 -> R3.0.7, R3.1.1 -> R3.1.5, R3.2.1 -> R3.2.2, R4.0.1 -> R4.3.1) and OSEK/VDX 2.2.3 standards (OSEK is now standardized in ISO 17356). The operating system is configured and built on a PC, but runs on your target hardware.

This document describes the RTA-OS RH850x2/GHS port plug-in that customizes the RTA-OS development tools for the Renesas RH850x2 with the GREENHILLS compiler. It supplements the more general information you can find in the *User Guide* and the *Reference Guide*.

The document has two parts. Chapters 2 to 3 help you understand the RH850x2/GHS port and cover:

- how to install the RH850x2/GHS port plug-in;
- how to configure RH850x2/GHS-specific attributes;
- how to build an example application to check that the RH850x2/GHS port plug-in works.

Chapters 4 to 9 provide reference information including:

- the number of OS objects supported;
- required and recommended toolchain parameters;
- how RTA-OS interacts with the RH850x2, including required register settings, memory models and interrupt handling;
- memory consumption for each OS object;
- memory consumption of each API call;
- execution times for each API call.

For the best experience with RTA-OS it is essential that you read and understand this document.

1.1 About You

You are a trained embedded systems developer who wants to build real-time applications using a preemptive operating system. You should have knowledge of the C programming language, including the compilation, assembling and linking of C code for embedded applications with your chosen toolchain. Elementary knowledge about your target microcontroller, such as the start address, memory layout, location of peripherals and so on, is essential.

You should also be familiar with common use of the Microsoft Windows operating system, including installing software, selecting menu items, clicking buttons, navigating files and folders.

1.2 Document Conventions

The following conventions are used in this guide:

Choose **File > Open**.	Menu options appear in **bold, blue** characters.
Click **OK**.	Button labels appear in **bold** characters
Press <Enter>.	Key commands are enclosed in angle brackets.
The “Open file” dialog box appears	GUI element names, for example window titles, fields, etc. are enclosed in double quotes.
`Activate(Task1)`	Program code, header file names, C type names, C functions and API call names all appear in a `monospaced typeface`.
See Section 1.2.	Internal document hyperlinks are shown in blue letters.
	Functionality in RTA-OS that might not be portable to other implementations of AUTOSAR OS is marked with the RTA-OS icon.
	Important instructions that you must follow carefully to ensure RTA-OS works as expected are marked with a caution sign.

1.3 References

OSEK is a European automotive industry standards effort to produce open systems interfaces for vehicle electronics. OSEK is now standardized in ISO 17356. For details of the OSEK standards, please refer to:

`http://www.osek-vdx.org`

AUTOSAR (AUTomotive Open System ARchitecture) is an open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers and tool developers. For details of the AUTOSAR standards, please refer to:

`http://www.autosar.org`

2 Installing the RTA-OS Port Plug-in

2.1 Preparing to Install

RTA-OS port plug-ins are supplied as a downloadable electronic installation image which you obtain from the ETAS Web Portal. You will have been provided with access to the download when you bought the port. You may optionally have requested an installation CD which will have been shipped to you. In either case, the electronic image and the installation CD contain identical content.

Integration Guidance 2.1:*You must have installed the RTA-OS tools before installing the RH850x2/GHS port plug-in. If you have not yet done this then please follow the instructions in the* Getting Started Guide*.*

2.1.1 Hardware Requirements

You should make sure that you are using at least the following hardware before installing and using RTA-OS on a host PC:

- 1GHz Pentium Windows-capable PC.
- 2G RAM.
- 20G hard disk space.
- CD-ROM or DVD drive (Optional)
- Ethernet card.

2.1.2 Software Requirements

RTA-OS requires that your host PC has one of the following versions of Microsoft Windows installed:

- Windows 7
- Windows 8
- Windows 10

Integration Guidance 2.2:*The tools provided with RTA-OS require Microsoft's .NET Framework* v2.0 *(included as part of .NET Framework* v3.5*) and* v4.0 *to be installed. You should ensure that these have been installed before installing RTA-OS. The .NET framework is not supplied with RTA-OS but is freely available from* `https://www.microsoft.com/net/download`*. To install .NET 3.5 on Windows 10 see* `https://docs.microsoft.com/en-us/dotnet/framework/install/dotnet-35-windows-10`*.*

The migration of the code from *v2.0* to *v4.0* will occur over a period of time for performance and maintenance reasons.

2.2 Installation

Target port plug-ins are installed in the same way as the tools:

1. Either
   - Double click the executable image; or
   - Insert the RTA-OS RH850x2/GHS CD into your CD-ROM or DVD drive.

     If the installation program does not run automatically then you will need to start the installation manually. Navigate to the root directory of your CD/DVD drive and double click `autostart.exe` to start the setup.
2. Follow the on-screen instructions to install the RH850x2/GHS port plug-in.

By default, ports are installed into `C:\ETAS\RTA-OS\Targets`. During the installation process, you will be given the option to change the folder to which RTA-OS ports are installed. You will normally want to ensure that you install the port plug-in in the same location that you have installed the RTA-OS tools. You can install different versions of the tools/targets into different directories and they will not interfere with each other.

Integration Guidance 2.3:*Port plug-ins can be installed into any location, but using a non-default directory requires the use of the `--target_include` argument to both **rtaosgen** and **rtaoscfg**. For example:*

```
rtaosgen --target_include:<target_directory>
```

2.2.1 Installation Directory

The installation will create a sub-directory under `Targets` with the name `RH850x2GHS_5.0.3`. This contains everything to do with the port plug-in.

Each version of the port installs in its own directory - the trailing `_5.0.3` is the port's version identifier. You can have multiple different versions of the same port installed at the same time and select a specific version in a project's configuration.

The port directory contains:

RH850x2GHS.dll - the port plug-in that is used by **rtaosgen** and **rtaoscfg**.

RTA-OS RH850x2GHS Port Guide.pdf - the documentation for the port (the document you are reading now).

RTA-OS RH850x2GHS Release Note.pdf - the release note for the port. This document provides information about the port plug-in release, including a list of changes from previous releases and a list of known limitations.

There may be other port-specific documentation supplied which you can also find in the root directory of the port installation. All user documentation is distributed in PDF format which can be read using Adobe Acrobat Reader. Adobe Acrobat Reader is not supplied with RTA-OS but is freely available from `http://www.adobe.com`.

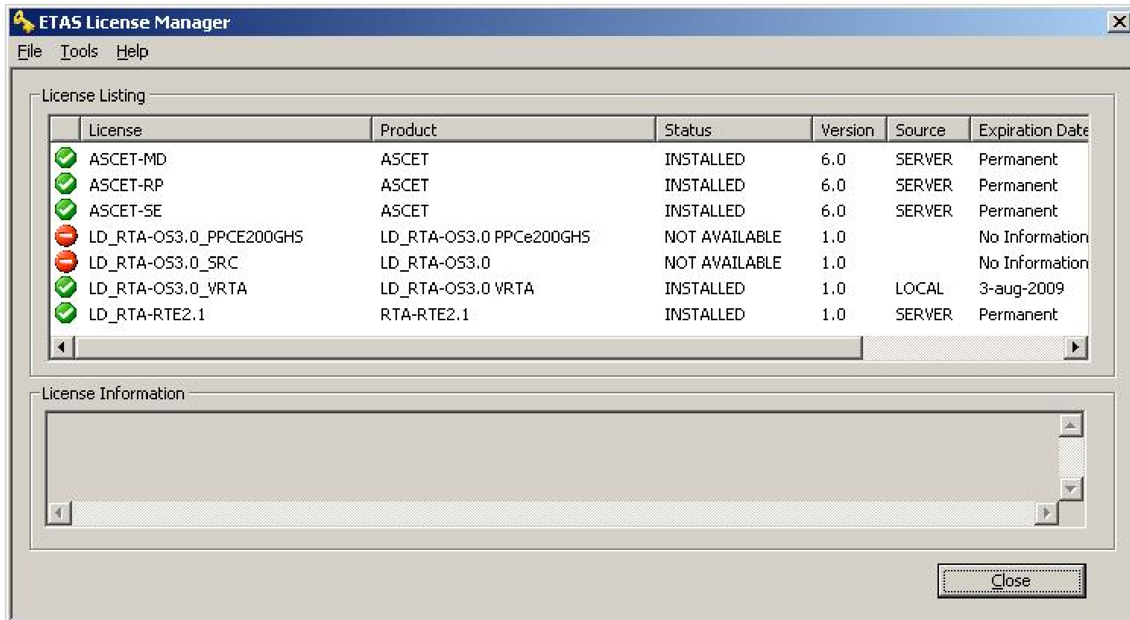


Figure 2.1: The ETAS License manager

2.3 Licensing

RTA-OS is protected by FLEXnet licensing technology. You will need a valid license key in order to use RTA-OS.

Licenses for the product are managed using the ETAS License Manager which keeps track of which licenses are installed and where to find them. The information about which features are required for RTA-OS and any port plug-ins is stored as license signature files that are stored in the folder `<install_folder>\bin\Licenses`.

The ETAS License Manager can also tell you key information about your licenses including:

- Which ETAS products are installed
- Which license features are required to use each product
- Which licenses are installed
- When licenses expire
- Whether you are using a local or a server-based license

Figure 2.1 shows the ETAS License Manager in operation.

2.3.1 Installing the ETAS License Manager

Integration Guidance 2.4: *The ETAS License Manager must be installed for RTA-OS to work. It is highly recommended that you install the ETAS License Manager during your installation of RTA-OS.*

The installer for the ETAS License Manager contains two components:

1. the ETAS License Manager itself;
2. a set of re-distributable FLEXnet utilities. The utilities include the software and instructions required to setup and run a FLEXnet license server manager if concurrent licenses are required (see Sections 2.3.2 and 2.3.3 for further details)

During the installation of RTA-OS you will be asked if you want to install the ETAS License Manager. If not, you can install it manually at a later time by running `<install_folder>\LicenseManager\LicensingStandaloneInstallation.exe`.

Once the installation is complete, the ETAS License Manager can be found in `C:\Program Files\Common Files\ETAS\Licensing`.

After it is installed, a link to the ETAS License Manager can be found in the Windows Start menu under **Programs→ ETAS → License Management → ETAS License Manager**.

2.3.2 Licenses

When you install RTA-OS for the first time the ETAS License Manager will allow the software to be used in *grace mode* for 14 days. Once the grace mode period has expired, a license key must be installed. If a license key is not available, please contact your local ETAS sales representative. Contact details can be found in Chapter 11.

You should identify which type of license you need and then provide ETAS with the appropriate information as follows:

Machine-named licenses allows RTA-OS to be used by any user logged onto the PC on which RTA-OS and the machine-named license is installed.

A machine-named license can be issued by ETAS when you provide the host ID (Ethernet MAC address) of the host PC

User-named licenses allow the named user (or users) to use RTA-OS on any PC in the network domain.

A user-named license can be issued by ETAS when you provide the Windows username for your network domain.

Concurrent licenses allow any user on any PC up to a specified number of users to use RTA-OS. Concurrent licenses are sometimes called *floating* licenses because the license can *float* between users.

A concurrent license can be issued by ETAS when you provide the following information:

1. The name of the server
2. The Host ID (MAC address) of the server.
3. The TCP/IP port over which your FLEXnet license server will serve licenses. A default installation of the FLEXnet license server uses port 27000.

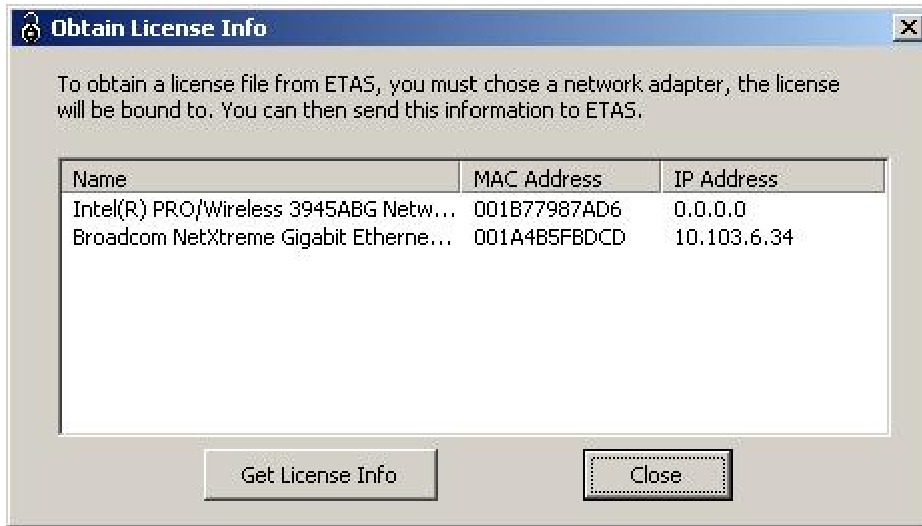


Figure 2.2: Obtaining License Information

You can use the ETAS License Manager to get the details that you must provide to ETAS when requesting a machine-named or user-named license and (optionally) store this information in a text file.

Open the ETAS License Manager and choose **Tools → Obtain License Info** from the menu. For machine-named licenses you can then select the network adaptor which provides the Host ID (MAC address) that you want to use as shown in Figure 2.2. For a user-based license, the ETAS License Manager automatically identifies the Windows username for the current user.

Selecting “Get License Info” tells you the Host ID and User information and lets you save this as a text file to a location of your choice.

2.3.3 Installing a Concurrent License Server

Concurrent licenses are allocated to client PCs by a FLEXnet license server manager working together with a vendor daemon. The vendor daemon for ETAS is called `ETAS.exe`. A copy of the vendor daemon is placed on disk when you install the ETAS License Manager and can be found in:

```
C:\Program Files\Common Files\ETAS\Licensing\Utility
```

To work with an ETAS concurrent license, a license server must be configured which is accessible from the PCs wishing to use a license. The server must be configured with the following software:

- FLEXnet license server manager;
- ETAS vendor daemon (`ETAS.exe`);

It is also necessary to install your concurrent license on the license server.

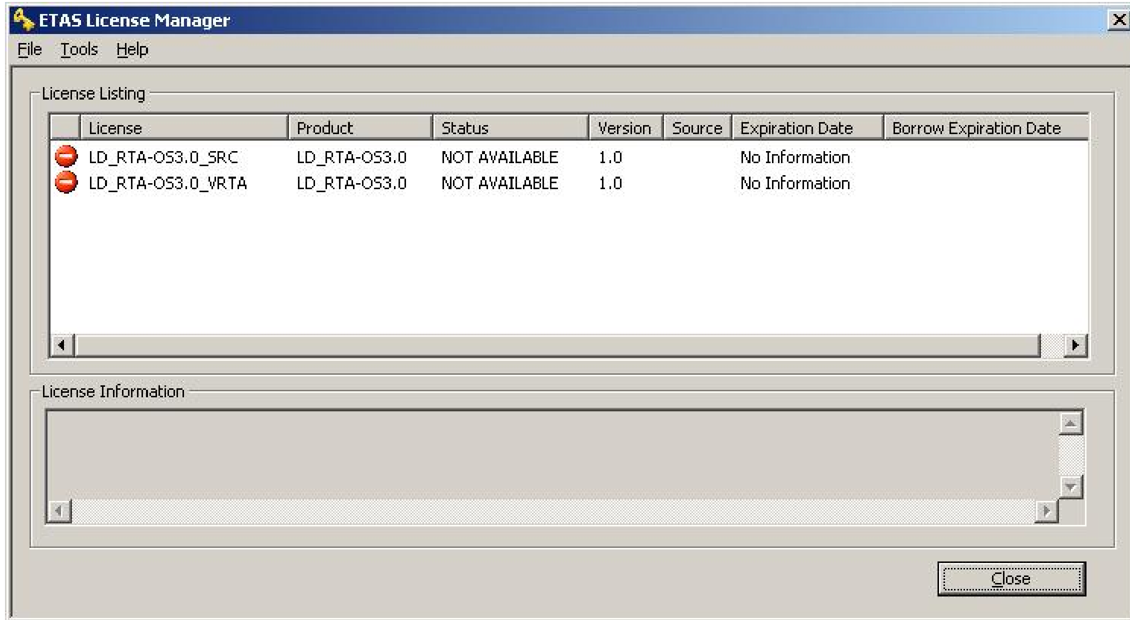


Figure 2.3: Unlicensed RTA-OS Installation

In most organizations there will be a single FLEXnet license server manager that is administered by your IT department. You will need to ask your IT department to install the ETAS vendor daemon and the associated concurrent license.

If you do not already have a FLEXnet license server then you will need to arrange for one to be installed. A copy of the FLEXnet license server, the ETAS vendor daemon and the instructions for installing and using the server (`LicensingEndUserGuide.pdf`) are placed on disk when you install the ETAS License manager and can be found in:

`C:\Program Files\Common Files\ETAS\Licensing\Utility`

2.3.4 Using the ETAS License Manager

If you try to run the RTA-OS GUI **rtaoscfg** without a valid license, you will be given the opportunity to start the ETAS License Manager and select a license. (The command-line tool **rtaosgen** will just report the license is not valid.)

When the ETAS License Manager is launched, it will display the RTA-OS license state as `NOT AVAILABLE`. This is shown in Figure 2.3.

Note that if the ETAS License Manager window is slow to start, **rtaoscfg** may ask a second time whether you want to launch it. You should ignore the request until the ETAS License Manager has opened and you have completed the configuration of the licenses. You should then say yes again, but you can then close the ETAS License Manager and continue working.

License Key Installation

License keys are supplied in an ASCII text file, which will be sent to you on completion of a valid license agreement.

If you have a machine-based or user-based license key then you can simply install the license by opening the ETAS License Manager and selecting **File → Add License File** menu.

If you have a concurrent license key then you will need to create a license stub file that tells the client PC to look for a license on the FLEXnet server as follows:

1. create a copy of the concurrent license file
2. open the copy of the concurrent license file and delete every line *except* the one starting with SERVER
3. add a new line containing USE_SERVER
4. add a blank line
5. save the file

The file you create should look something like this:

```
SERVER <server name> <MAC address> <TCP/IP Port>¶
USE_SERVER¶
¶
```

Once you have create the license stub file you can install the license by opening the ETAS License Manager and selecting **File → Add License File** menu and choosing the license stub file.

License Key Status

When a valid license has been installed, the ETAS License Manager will display the license version, status, expiration date and source as shown in Figure 2.4.

Borrowing a concurrent license

If you use a concurrent license and need to use RTA-OS on a PC that will be disconnected from the network (for example, you take a demonstration to a customer site), then the concurrent license will not be valid once you are disconnected.

To address this problem, the ETAS License Manager allows you to temporarily borrow a license from the license server.

To borrow a license:

1. Right click on the license feature you need to borrow.
2. Select “Borrow License”
3. From the calendar, choose the date that the borrowed license should expire.
4. Click “OK”

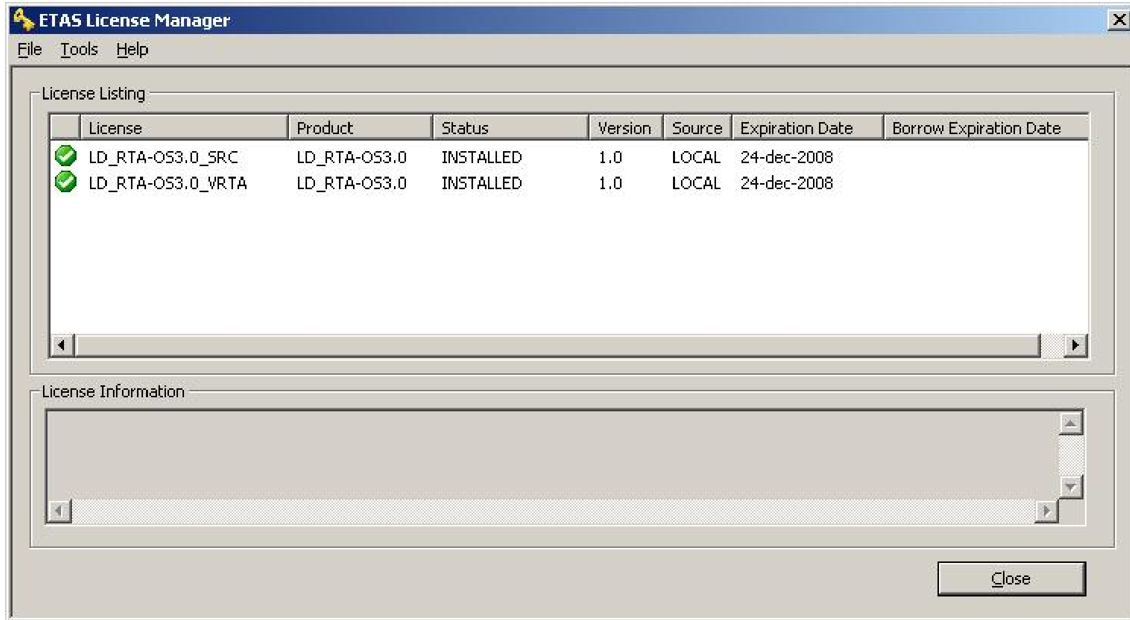


Figure 2.4: Licensed features for RTA-OS

The license will automatically expire when the borrow date elapses. A borrowed license can also be returned before this date. To return a license:

1. Reconnect to the network;
2. Right-click on the license feature you have borrowed;
3. Select “Return License”.

2.3.5 Troubleshooting Licenses

RTA-OS tools will report an error if you try to use a feature for which a correct license key cannot be found. If you think that you should have a license for a feature but the RTA-OS tools appear not to work, then the following troubleshooting steps should be followed before contacting ETAS:

Can the ETAS License Manager see the license?

The ETAS License Manager must be able to see a valid license key for each product or product feature you are trying to use.

You can check what the ETAS License Manager can see by starting it from the **Help → License Manager...** menu option in **rtaoscfg** or directly from the Windows Start Menu - **Start → ETAS → License Management → ETAS License Manager**.

The ETAS License Manager lists all license features and their status. Valid licenses have status `INSTALLED`. Invalid licenses have status `NOT AVAILABLE`.

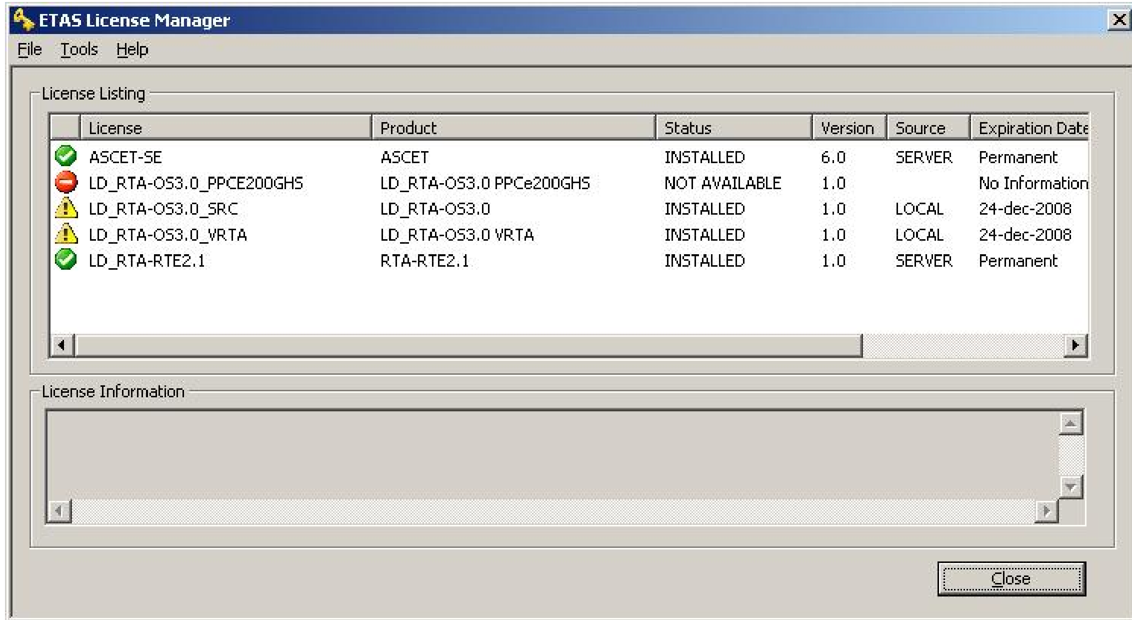


Figure 2.5: Licensed features that are due to expire

Is the license valid?

You may have been provided with a time-limited license (for example, for evaluation purposes) and the license may have expired. You can check that the Expiration Date for your licensed features to check that it has not elapsed using the ETAS License Manager.

If a license is due to expire within the next 30 days, the ETAS License Manager will use a warning triangle to indicate that you need to get a new license. Figure 2.5 shows that the license features `LD_RTA-OS3.0_VRTA` and `LD_RTA-OS3.0_SRC` are due to expire.

If your license has elapsed then please contact your local ETAS sales representative to discuss your options.

Does the Ethernet MAC address match the one specified?

If you have a machine based license then it is locked to a specific MAC address. You can find out the MAC address of your PC by using the ETAS License Manager (**Tools → Obtain License Info**) or using the Microsoft program **ipconfig /all** at a Windows Command Prompt.

You can check that the MAC address in your license file by opening your license file in a text editor and checking that the HOSTID matches the MAC address identified by the ETAS License Manager or the *Physical Address* reported by **ipconfig /all**.

If the HOSTID in the license file (or files) does not match your MAC address then you do not have a valid license for your PC. You should contact your local ETAS sales representative to discuss your options.

Is your Ethernet Controller enabled?

If you use a laptop and RTA-OS stops working when you disconnect from the network then you should check your hardware settings to ensure that your Ethernet controller is not turned off to save power when a network connection is not present. You can do this using Windows Control Panel. Select **System → Hardware → Device Manager** then select your Network Adapter. Right click to open **Properties** and check that the Ethernet controller is not configured for power saving in **Advanced** and/or **Power Management** settings.

Is the FlexNet License Server visible?

If your license is served by a FlexNet license server, then the ETAS License Manager will report the license as `NOT AVAILABLE` if the license server cannot be accessed.

You should contact your IT department to check that the server is working correctly.

Still not fixed?

If you have not resolved your issues, after confirming these points above, please contact ETAS technical support. The contact address is provided in Section 11.1. You must provide the contents and location of your license file and your Ethernet MAC address.

3 Verifying your Installation

Now that you have installed both the RTA-OS tools and a port plug-in and have obtained and installed a valid license key you can check that things are working.

3.1 Checking the Port

The first thing to check is that the RTA-OS tools can see the new port. You can do this in two ways:

1. use the **rtaosgen** tool

   You can run the command **rtaosgen −−target:?** to get a list of available targets, the versions of each target and the variants supported, for example:

   ```
   RTA-OS Code Generator
   Version p.q.r.s, Copyright © ETAS nnnn
   Available targets:
    TriCoreHighTec_n.n.n [TC1797...]
    VRTA_n.n.n [MinGW,VS2005,VS2008,VS2010]
   ```

2. use the **rtaoscfg** tool

   The second way to check that the port plug-in can be seen is by starting **rtaoscfg** and selecting **Help → Information...** drop down menu. This will show information about your complete RTA-OS installation and license checks that have been performed.

Integration Guidance 3.1:*If the target port plug-ins have been installed to a non-default location, then the `--target_include` argument must be used to specify the target location.*

If the tools can see the port then you can move on to the next stage – checking that you can build an RTA-OS library and use this in a real program that will run on your target hardware.

3.2 Running the Sample Applications

Each RTA-OS port is supplied with a set of sample applications that allow you to check that things are running correctly. To generate the sample applications:

1. Create a new *working* directory in which to build the sample applications.

2. Open a Windows command prompt in the new directory.

3. Execute the command:

   ```
   rtaosgen --target:<your target> --samples:[Applications]

   e.g.

   rtaosgen --target:[MPC5777Mv2]PPCe200HighTec_5.0.8
       --samples:[Applications]
   ```

You can then use the build.bat and run.bat files that get created for each sample application to build and run the sample. For example:

```
cd Samples\Applications\HelloWorld
build.bat
run.bat
```

Remember that your target toolchain must be accessible on the Windows PATH for the build to be able to run successfully.

Integration Guidance 3.2: *It is strongly recommended that you build and run at least the* Hello World *example in order to verify that RTA-OS can use your compiler toolchain to generate an OS kernel and that a simple application can run with that kernel.*

For further advice on building and running the sample applications, please consult your *Getting Started Guide*.

4 Port Characteristics

This chapter tells you about the characteristics of RTA-OS for the RH850x2/GHS port.

4.1 Parameters of Implementation

To be a valid OSEK (ISO 17356) or AUTOSAR OS, an implementation must support a minimum number of OS objects. The following table specifies the *minimum* numbers of each object required by the standards and the *maximum* number of each object supported by RTA-OS for the RH850x2/GHS port.

Parameter	Required	RTA-OS
Tasks	16	1024
Tasks not in SUSPENDED state	16	1024
Priorities	16	1024
Tasks per priority	-	1024
Queued activations per priority	-	4294967296
Events per task	8	32
Software Counters	8	4294967296
Hardware Counters	-	4294967296
Alarms	1	4294967296
Standard Resources	8	4294967296
Linked Resources	-	4294967296
Nested calls to `GetResource()`	-	4294967296
Internal Resources	2	no limit
Application Modes	1	4294967296
Schedule Tables	2	4294967296
Expiry Points per Schedule Table	-	4294967296
OS Applications	-	4294967295
Trusted functions	-	4294967295
Spinlocks (multicore)	-	4294967295
Register sets	-	4294967296

4.2 Configuration Parameters

Port-specific parameters are configured in the **General → Target** workspace of **rtaoscfg**, under the “Target-Specific” tab.

The following sections describe the port-specific configuration parameters for the RH850x2/GHS port, the name of the parameter as it will appear in the XML configuration and the range of permitted values (where appropriate).

4.2.1 Stack used for C-startup

XML name SpPreStartOS

Description

The amount of stack already in use at the point that StartOS() is called. This value is simply added to the total stack size that the OS needs to support all tasks and interrupts at run-time. Typically you use this to obtain the amount of stack that the linker must allocate. The value does not normally change if the OS configuration changes.

4.2.2 Stack used when idle

XML name SpStartOS

Description

The amount of stack used when the OS is in the idle state (typically inside Os_Cbk_Idle()). This is just the difference between the stack used at the point that Os_StartOS() is called and the stack used when no task or interrupt is running. This can be zero if Os_Cbk_Idle() is not used. It must include the stack used by any function called while in the idle state. The value does not normally change if the OS configuration changes.

4.2.3 Stack overheads for ISR activation

XML name SpIDisp

Description

The extra amount of stack needed to activate a task from within an ISR. If a task is activated within a Category 2 ISR, and that task has a higher priority than any currently running task, then for some targets the OS may need to use marginally more stack than if it activates a task that is of lower priority. This value accounts for that. On most targets this value is zero. This value is used in worst-case stack size calculations. The value may change if significant changes are made to the OS configuration. e.g. STANDARD/EXTENDED, SC1/2/3/4.

4.2.4 Stack overheads for ECC tasks

XML name SpECC

Description

The extra amount of stack needed to start an ECC task. ECC tasks need to save slightly more state on the stack when they are started than BCC tasks. This value contains the difference. The value may change if significant changes are made to the OS configuration. e.g. STANDARD/EXTENDED, SC1/2/3/4.

4.2.5 Stack overheads for ISR

XML name SpPreemption

Description

The amount of stack used to service a Category 2 ISR. When a Category 2 ISR interrupts a task, it usually places some data on the stack. If the ISR measures the stack to determine if the preempted task has exceeded its stack budget, then it will overestimate the stack usage unless this value is subtracted from the measured size. The value is also used when calculating the worst-case stack usage of the system. Be careful to set this value accurately. If its value is too high then when the subtraction occurs, 32-bit underflow can occur and cause the OS to think that a budget overrun has been detected. The value may change if significant changes are made to the OS configuration. e.g. STANDARD/EXTENDED, SC1/2/3/4.

4.2.6 Orti22/Lauterbach

XML name Orti22Lauterbach

Description

Select ORTI 2.2 generation for the Lauterbach debugger.

Settings

Value	Description
true	Generate ORTI
false	No ORTI (default)

4.2.7 Orti23/Lauterbach

XML name Orti23Lauterbach

Description

Select ORTI 2.3 generation for the Lauterbach debugger.

Settings

Value	Description
true	Generate ORTI
false	No ORTI (default)

4.2.8 ORTI Stack Fill

XML name OrtiStackFill

Description

Expands ORTI information to cover stack address, size and fill pattern details to support debugger stack usage monitoring

Settings

Value	Description
true	Support ORTI stack tracking
false	ORTI stack tracking unsupported (default)

4.2.9 Direct Vector mode

XML name DirectVectors

Description

Select Direct Vector Method for handling all EI maskable interrupts with a single vector address (please refer to the Renesas documentation for more details on the vector methods).

Settings

Value	Description
true	Use Direct Vector method
false	Use Table Reference method (default)

4.2.10 EBASE register

XML name EBASE_register

Description

Controls what the EBASE register is set to in Os_InitializeVectorTable(). Valid values are: 'true' to set EBASE to the address of the RTA-OS generated vector table (Os_interrupt_vectors) (default), 'false' to not set EBASE at all, a literal hexadecimal address (e.g. '0x00000200'), or a label whose address will be used (e.g. 'my_vectors' to set it to the address of my_vectors).

4.2.11 Linker Model

XML name Linker_Model

Description

Controls whether code gets placed in the RAM or ROM.

Settings

Value	Description
FLASH	Place code in the user FLASH areas, data in cluster RAM and stack in core-local RAM (default)
RAM	Place code/data in cluster RAM and stack in core-local RAM

4.2.12 Enable untrusted stack check

XML name DistrustStacks

Description

To help protect against untrusted code corrupting the stack pointer, extra code can be run before the interrupt handlers for EI interrupts (but not FE interrupts) to detect when untrusted code has an illegal stack pointer value, and in addition, call the Os_Cbk_GetAbortStack() callback to set up a safe area of memory to use as a stack when executing the ProtectionHook (please refer to the documentation for Os_Cbk_GetAbortStack()). This has a small performance overhead, so is made optional.

Settings

Value	Description
true	Perform the checks
false	Do not check (default)

4.2.13 GetAbortStack always

XML name always_call_GetAbortStack

Description

Controls whether to always use the Os_Cbk_GetAbortStack() callback to set up a safe area of memory to use as a stack when executing the ProtectionHook (please refer to the documentation for Os_Cbk_GetAbortStack()).

Settings

Value	Description
true	Always call Os_Cbk_GetAbortStack()
false	Only call Os_Cbk_GetAbortStack() when the ’Enable untrusted stack check’ target option is selected (default)

4.2.14 Handle FPU context

XML name handle_FPU_context

Description

Extra code can be added during context switches to additionally handle the FPSR and FPEPC registers for tasks and ISRs. This option should only be used if the majority of Tasks and ISRs in the application contain FPU instructions otherwise register sets should be used. This has a small performance overhead, so is made optional.

Settings

Value	Description
true	Save FPU context
false	Do not save FPU context (default)

4.2.15 Cache Core ID

XML name cache_core_ID

Description

When untrusted code is present, reading the core ID may require an expensive switch in and out of trusted mode. To avoid this switch, the core ID may be cached in the CTPC register. If this is done the Os_CacheCoreID() function must be called before any other RTA-OS API is used. Please note that when the core ID is cached, untrusted code may cause RTA-OS to behave in an undefined way if it corrupts the value in CTPC.

Settings

Value	Description
true	Cache the core ID
false	Do not cache the core ID (default)

4.2.16 Core Specific Wrappers

XML name core_wrappers

Description

By default, the outer and mid interrupt wrappers for EIINT ISRs are located in the section Os_primitives irrespective of the core to which the ISRs are assigned. If this option is set to true, then in Table-Reference mode multicore configurations, the outer and mid interrupt wrappers for EIINT ISRs that are assigned to core number X will be located in section Os_primitivesX. E.g. the wrappers for ISRs assigned to core 0 will be locted in Os_primitives0 and the wrappers for ISRs assigned to core 2 will be located in Os_primitives2. Enabling this option results in there being multiple copies of the mid wrappers and you need to locate the Os_primitiveX sections.

Settings

Value	Description
true	Generate core specific wrappers
false	Do not generate core specific wrappers (default)

4.2.17 misalign_pack

XML name misalign_pack

Description

Controls the -misalign_pack option (see the compiler documentation for more details). This option assumes that misaligned data access is handled by the target, and does not generate extra code to handle packed references.

Settings

Value	Description
true	Use -misalign_pack (default)
false	Use -no_misalign_pack

4.2.18 stack_protector

XML name stack_protector

Description

Controls the -stack_protector option. This option provides protection against stack smashing attacks by placing a canary in functions with variables greater than 8 bytes in length.

Settings

Value	Description
true	Use -stack_protector
false	Use -no_stack_protector (default)

4.2.19 SDA Threshold

XML name sda_setting

Description

Sets the value used for the -sda compiler option, as per the compiler documentation. Valid values are: ’0’ (default), ’none’, ’all’, ’never’, or the threshold value in bytes.

4.2.20 Optimizer setting

XML name Optimizer_setting

Description

Controls the optimizer strategy compiler option (see the compiler documentation for more details).

Settings

Value	Description
general	Improve both code size and performance, not favoring one over the other (default)
size	Improve code size over performance
speed	Improve code performance over size

4.2.21 SDA size setting

XML name SDA_size

Description

Controls the maximum size of the offset used by the compiler from the small data area (SDA) base register when accessing SDA data with load and store instructions.

Settings

Value	Description
16-bit	Only use 4-byte load/store instructions
23-bit	Extend SDA addressing to allow use of 6-byte load/store instructions (default)

4.2.22 Option set 1

XML name option_set_1

Description

Selects an alternative set of compiler options, as follows: -Ogeneral - Optimizer strategy, can be modified via target option; -G - Generate debug information; -cpu=rh850g4mh - Specify the target processor instruction set; -dwarf2 - Use DWARF 2 format in debug information; –no_commons - Allocate uninitialized global variables to a section and initialize them to zero at program startup; -prepare_dispose - Allow V850E prepare and dispose instructions; -no_callt - Disable use of the callt instruction; -reserve_r2 - Reserve r2 as an unmodified register; -shorten_loads - Convert 23 bit SDA relocations to 16 bit load/store instructions where possible; -sda=0 - Set the SDA threshold, can be modified via target option; -large_sda - Generate 23 bit SDA relocations for load/store instructions, can be modified via target option; –short_enum - Store enumerations in the smallest possible type; -list - Generate listings; -ignore_callt_state_in_interrupts - Do not save CTPSW and CTPC in interrupt routines; -frigor=accurate - Favor accuracy over performance in floating point operations. -fhard - Use hardware floating-point. Note that an error will be generated if this option set is used when an incompatible target option is also specified. This includes the -misalign_pack option, which defaults to the incompatible setting of true. Please note that this option set may only be used if you have a source code license.

Settings

Value	Description
true	Use this option set
false	Use the standard option set (default)

4.2.23 Enhanced Isolation

XML name EnhancedIsolation

Description

Use to enforce additional checks to prevent errors in untrusted code from affecting any other part of the system. Refer to the documentation in the User and Reference Guides.

Settings

Value	Description
true	Support Enhanced Isolation
false	Normal behavior (default)

4.3 Generated Files

The following table lists the files that are generated by **rtaosgen** for all ports:

Filename	Contents
Os.h	The main include file for the OS.
Os_Cfg.h	Declarations of the objects you have configured. This is included by Os.h.
Os_MemMap.h	AUTOSAR memory mapping configuration used by RTA-OS to merge with the system-wide MemMap.h file in AUTOSAR versions 4.0 and earlier. From AUTOSAR version 4.1, Os_MemMap.h is used by the OS instead of MemMap.h.
RTAOS.<lib>	The RTA-OS library for your application. The extension <lib> depends on your target.
RTAOS.<lib>.sig	A signature file for the library for your application. This is used by **rtaosgen** to work out which parts of the kernel library need to be rebuilt if the configuration has changed. The extension <lib> depends on your target.
<projectname>.log	A log file that contains a copy of the text that the tool and compiler sent to the screen during the build process.

5 Port-Specific API

The following sections list the port-specific aspects of the RTA-OS programmers reference for the RH850x2/GHS port that are provided either as:

- additions to the material that is documented in the *Reference Guide*; or
- overrides for the material that is documented in the *Reference Guide*. When a definition is provided by both the *Reference Guide* and this document, the definition provided in this document takes precedence.

5.1 API Calls

5.1.1 Os_AwaitStartup

Routine used on a multicore variant by a non-master core to wait until the master core is ready for the non-master core to start.

Syntax

```
FUNC(void, OS_APPL_CODE) Os_AwaitStartup(void)
```

Description

This target port assumes that when the RH850 processor comes out of reset all cores start automatically. The Os_AwaitStartup() function is called by non-master cores after they have setup the stack and the small-data area registers to suspend execution until the master core calls Os_Cbk_StartCore() to start the non-master core.

RTA-OS provides a default implementation of Os_AwaitStartup() that will be appropriate for most normal situations. In the RTA-OS samples, Os_AwaitStartup() is called from the code in reset.s that runs before the main() function is called. It can also be called in OS_MAIN().

Note that with the default implementations of Os_AwaitStartup() and Os_Cbk_StartCore() provided, your code is responsible for calling Os_AwaitStartup().

If you provide your own version of Os_AwaitStartup() you will most likely need to provide your own version of Os_Cbk_StartCore() as well.

Os_AwaitStartup() should not be called for core 0 as this is the master core. core 0 is the core whose PEID register contains the value 0.

Example

```
#define OS_START_CORE_KEY (0xAA551234UL) /* Pattern that will not match a
    hardware self test pattern. */

volatile uint32 Os_CoresStarted[..];

FUNC(void, OS_APPL_CODE) Os_AwaitStartup(void) {
  uint32 core = OS_STSR(OS_PEID_REGID, OS_PEID_SELID);

  /* Wait until activated by the master core. */
  while (Os_CoresStarted[core] != OS_START_CORE_KEY) {
    OS_SNOOZE();
  }

  /* Clear Os_CoresStarted[core] so that this function will
   * have to wait for Os_Cbk_StartCore(core) to be called
   * again if the core is reset and memory is not zeroed. */
   Os_CoresStarted[core] = 0U;
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupTaskHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

StartCore
StartNonAutosarCore
StartOS
Os_Cbk_StartCore

5.1.2 Os_CacheCoreID

Routine used to cache the core ID into the CTPC register.

Syntax

```
FUNC(void, OS_APPL_CODE) Os_CacheCoreID(void)
```

Description

If the ’Cache Core ID’ target option is set to true, this target port assumes that in multicore configurations the core ID is stored in the CTPC register. The core ID is stored in CTPC because CTPC is accessible in user mode, whereas the PEID register is only accessible in supervisor mode. This API functions copies PEID into CTPC. This API must be called on every core after the stack and small-area registers have been set up but

before any other RTA-OS API is used. Please note that when the core ID is cached, untrusted code may cause RTA-OS to behave in an undefined way if it corrupts the value in CTPC.

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupTaskHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

StartCore
StartNonAutosarCore
StartOS
Os_AwaitStartup

5.1.3 Os_InitializeVectorTable

Initialize interrupt handling.

Syntax

```
void Os_InitializeVectorTable(void)
```

Description

Os_InitializeVectorTable() initializes the interrupt controller (INTC1 and INTC2) registers and priorities according to the requirements of the project configuration.

The PSW.EBV bit is set so that EBASE is used to contain the address of the interrupt vector table. The EBASE register is set to contain the location of the interrupt vector table (the default is Os_interrupt_vectors but this can be changed by a target option). If Table Reference interrupt vectors are used then the INTBP register is set to contain the location of Os_EI_vectors.

The EICn registers of EIINT interrupts declared in the project configuration are modified so that their priority matches that expected by RTA-OS. If Table Reference method interrupt mode is selected then the EITB bits are set for the configured interrupts. The EICn mask bits are cleared so that configured interrupts are unmasked. For INTC2 interrupts the EIBDn registers of EIINT interrupts declared in the project configuration are modified to route the interrupt to the core specified in the configuration. The EICn and EIBDn registers of interrupt channels not declared in the project configuration are not modified by this function.

Modification of EICn and EIBDn registers for INTC2 interrupts is only carried out when Os_InitializeVectorTable() is called on the master core. Therefore the master core must call Os_InitializeVectorTable() before any other core calls StartOS().

The PLMR register is set to block all Category 2 interrupts (as these should only trigger after StartOS()) but to allow Category 1 interrupts to fire. The PSW.ID, PSW.EP and PSW.NP bits are cleared to allow EI and FE interrupts. The INTCFG.ISPC bit is set to disable hardware management of the current interrupt priority.

For multicore builds when core ID caching is enabled, the core ID is stored in CTPC.

Os_InitializeVectorTable() should be called before StartOS(). It should be called even if 'Suppress Vector Table Generation' is set to TRUE.

In multicore applications Os_InitializeVectorTable() should be called by all cores.

Example

```
Os_InitializeVectorTable();
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupTaskHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

StartOS

5.2 Callbacks

5.2.1 Os_Cbk_GetAbortStack

Callback routine to provide the start address of the stack to use to handle exceptions.

Syntax

```
FUNC(void *,{memclass}) Os_Cbk_GetAbortStack(void)
```

Return Values

The call returns values of type **void** *.

Description

Untrusted code can misbehave and cause a protection exception. When this happens, AUTOSAR requires that ProtectionHook is called and the task, ISR or OS Application must be terminated.

It is possible that at the time of the fault the stack pointer is invalid. For this reason, if 'Enable untrusted stack check' or 'GetAbortStack always' is configured, RTA-OS will call Os_Cbk_GetAbortStack() to get the address of a safe area of memory that it should use for the stack while it performs this processing.

Maskable interrupts will be disabled during this process so the stack only needs to be large enough to perform the ProtectionHook.

A default implementation of Os_Cbk_GetAbortStack() is supplied in the RTA-OS library.

In systems that use the Os_Cbk_SetMemoryAccess() callback, the return value is the last stack location returned in ApplicationContext from Os_Cbk_SetMemoryAccess(). This is to avoid having to reserve memory. Note that this relies on Os_Cbk_SetMemoryAccess() having been called at least once on that core otherwise zero will be returned. (The stack will not get adjusted if zero is returned.) Otherwise the default implementation returns the address of an area of static memory that is reserved for sole use by the abort stack.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_CALLOUT_CODE for AUTOSAR 4.0, OS_OS_CBK_GETABORTSTACK_CODE for AUTOSAR 4.1.

Example

```
FUNC(void *,{memclass}) Os_Cbk_GetAbortStack(void) {
  static uint32 abortstack[40U];
  return &abortstack[40U];
}
```

Required when

The callback must be present if 'Enable untrusted stack check' is configured and there are untrusted OS Applications. The callback is also present if the 'GetAbortStack always' target option is enabled.

5.2.2 Os_Cbk_StartCore

Callback routine used to start a non-master core on a multicore variant.

Syntax

```
FUNC(StatusType, OS_APPL_CODE)Os_Cbk_StartCore(
    uint16 CoreID
)
```

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	all	The core does not exist or can not be started.

Description

In a multicore application, the StartCore() or StartNonAutosarCore() OS APIs have to be called prior to StartOS() for each core that is to run. For this target port, these APIs make a call to Os_Cbk_StartCore() which is responsible for starting the specified core.

RTA-OS provides a default implementation of Os_Cbk_StartCore() that will be appropriate for most normal situations. Os_Cbk_StartCore() does not get called for core 0, because core 0 must start first.

If you provide your own version of Os_Cbk_StartCore() you will most likely need to provide your own version of Os_AwaitStartup() as well.

The CoreID argument corresponds to the value in a core's PEID register. For example when CoreID is 1, this function is being told to start the core whose PEID register contains the value 1.

Example

```
FUNC(StatusType, OS_APPL_CODE) Os_Cbk_StartCore(uint16 CoreID) {
  StatusType ret = E_OS_ID;

  if ((CoreID >= 1U) && (CoreID <= 5U)) {
    Os_CoresStarted[CoreID] = OS_START_CORE_KEY;
    ret = E_OK;
  }

  return ret;
}
```

Required when

Required for non-master cores that will be started.

See Also

StartCore
StartNonAutosarCore
StartOS
Os_AwaitStartup

5.3 Macros

5.3.1 CAT1_ISR

Macro that should be used to create a Category 1 ISR entry function. This macro exists to help make your code portable between targets. It should be used with both EI and FE interrupts.

Example

```
CAT1_ISR(MyISR) {...}
```

5.3.2 Os_Clear_x

Use of the Os_Clear_x() macro will clear the interrupt request bit in the interrupt control register for the specified EIINT interrupt channel. The EIINT interrupt can be specified using either the EIINT channel number or the RTA-OS configured vector name. For example, Os_Clear_EI_Channel_20() would specify EIINT interrupt channel 20, and Os_Clear_Millisecond() would specify the EIINT interrupt configured with the name Millisecond. To use the Os_Clear_x() macro the file Os_ConfigInterrupts.h must be included through the use of #include. The macro is provided so the interrupt channel can be cleared without corrupting the interrupt priority value configured by calling Os_InitializeVectorTable(). The macro may not be used by untrusted code. Please note that for EIINT interrupt channels 0 to 31 (i.e. INTC1 channels) this macro will only clear the interrupt request bit on the core that invokes the macro.

Example

```
Os_Clear_EI_Channel_20();
Os_Clear_Millisecond();
```

5.3.3 Os_DisableAllConfiguredInterrupts_Corex

Use of the Os_DisableAllConfiguredInterrupts_Corex() macro will disable all EIINT interrupt channels that have ISRs assigned to core x in the configuration. To use the Os_DisableAllConfiguredInterrupts_Corex() macro the file Os_ConfigInterrupts.h must be included through the use of #include. The macro is provided so the interrupt channels can be disabled without corrupting the interrupt priority values configured by calling Os_InitializeVectorTable(). It may not be used by untrusted code. Os_DisableAllConfiguredInterrupts_Corex() should only be invoked by core x.

x is the numeric value found in a core's PEID register. E.g. Os_DisableAllConfiguredInterrupts_Core0 disables interrupt channels on the core whose PEID register contains the value 0 and Os_DisableAllConfiguredInterrupts_Core1 disables interrupt channels on the core whose PEID register contains the value 1.

Example

```
Os_DisableAllConfiguredInterrupts_Core0();
...
Os_EnableAllConfiguredInterrupts_Core0();
```

5.3.4 Os_Disable_x

Use of the Os_Disable_x() macro will disable the specified EIINT interrupt. The EIINT interrupt can be specified using either the EIINT channel number or the RTA-OS configured vector name. For example, Os_Disable_EI_Channel_20() would specify EIINT interrupt channel 20, and Os_Disable_Millisecond() would specify the EIINT interrupt configured with the name Millisecond. To use the Os_Disable_x() macro the file Os_ConfigInterrupts.h must be included through the use of #include. The macro is provided so the interrupt channel can be masked without corrupting the interrupt priority value configured by calling Os_InitializeVectorTable(). The macro may not be used by untrusted code. Please note that the behavior of this macro is different for EIINT interrupt channels 0 to 31 (i.e. INTC1 channels) and channels 32 and greater (i.e. INTC2 channels). For channels 0 to 31 the interrupt will only be disabled on the core that invokes the macro. For channels 32 and greater the interrupt will be disabled on all cores.

Example

```
Os_Disable_EI_Channel_20();
Os_Disable_Millisecond();
```

5.3.5 Os_EnableAllConfiguredInterrupts_Corex

Use of the Os_EnableAllConfiguredInterrupts_Corex() macro will enable all EIINT interrupt channels that have ISRs assigned to core x in the configuration. To use the Os_EnableAllConfiguredInterrupts_Corex() macro the file Os_ConfigInterrupts.h must be included through the use of #include. The macro is provided so the interrupt channels can be enabled without corrupting the interrupt priority values configured by calling Os_InitializeVectorTable(). It may not be used by untrusted code. Os_EnableAllConfiguredInterrupts_Corex() should only be invoked by core x.

x is the numeric value found in a core's PEID register. E.g. Os_EnabledAllConfiguredInterrupts_Core0 enables interrupt channels on the core whose PEID register contains the value 0 and Os_EnableAllConfiguredInterrupts_Core1 enables interrupt channels on the core whose PEID register contains the value 1.

Example

```
Os_DisableAllConfiguredInterrupts_Corex();
...
Os_EnableAllConfiguredInterrupts_Corex();
```

5.3.6 Os_Enable_x

Use of the Os_Enable_x() macro will enable the specified EIINT interrupt. The EIINT interrupt can be specified using either the EIINT channel number or the RTA-OS

configured vector name. For example, Os_Enable_EI_Channel_20() would specify EIINT interrupt channel 20, and Os_Enable_Millisecond() would specify the EIINT interrupt configured with the name Millisecond. To use the Os_Enable_x() macro the file Os_ConfigInterrupts.h must be included through the use of #include. The macro is provided so the interrupt channel can be enabled without corrupting the interrupt priority value configured by calling Os_InitializeVectorTable(). The macro may not be used by untrusted code. Please note that the behavior of this macro is different for EIINT interrupt channels 0 to 31 (i.e. INTC1 channels) and channels 32 and greater (i.e. INTC2 channels). For channels 0 to 31 the interrupt will only be enabled on the core that invokes the macro. For channels 32 and greater the interrupt will be enabled on all cores.

Example

```
Os_Enable_EI_Channel_20();
Os_Enable_Millisecond();
```

5.3.7 Os_IntChannel_x

The Os_IntChannel_x macro can be used to get the EIINT interrupt channel number associated with an RTA-OS configured vector name. In the example, this is Os_IntChannel_Millisecond. To use the Os_IntChannel_x macro the file Os_ConfigInterrupts.h must be included through the use of #include.

Example

```
trigger_interrupt(Os_IntChannel_INTTAUD0I11);
trigger_interrupt(Os_IntChannel_Millisecond);
```

5.4 Type Definitions

5.4.1 Os_StackSizeType

An unsigned value representing an amount of stack in bytes.

Example

```
Os_StackSizeType stack_size;
stack_size = Os_GetStackSize(start_position, end_position);
```

5.4.2 Os_StackValueType

An unsigned value representing the position of the stack pointer (r3).

Example

```
Os_StackValueType start_position;
start_position = Os_GetStackValue();
```

6 Toolchain

This chapter contains important details about RTA-OS and the GREENHILLS toolchain. A port of RTA-OS is specific to both the target hardware and a specific version of the compiler toolchain. You must make sure that you build your application with the supported toolchain.

In addition to the version of the toolchain, RTA-OS may use specific tool options (switches). The options are divided into three classes:

kernel options are those used by **rtaosgen** to build the RTA-OS kernel.

mandatory options must be used to build application code so that it will work with the RTA-OS kernel.

forbidden options must not be used to build application code.

Any options that are not explicitly forbidden can be used by application code providing that they do not conflict with the kernel and mandatory options for RTA-OS.

Integration Guidance 6.1:*ETAS has developed and tested RTA-OS using the tool versions and options indicated in the following sections. Correct operation of RTA-OS is only covered by the warranty in the terms and conditions of your deployment license agreement when using identical versions and options. If you choose to use a different version of the toolchain or an alternative set of options then it is your responsibility to check that the system works correctly. If you require a statement that RTA-OS works correctly with your chosen tool version and options then please contact ETAS to discuss validation possibilities.*

6.1 Compiler Versions

This port of RTA-OS has been developed to work with the following compiler(s):

6.1.1 Green Hills v2018.1.5

Ensure that ccrh850.exe is on the path.

Tested on Release tests were performed on this version.

If you require support for a compiler version not listed above, please contact ETAS.

6.2 Options used to generate this guide

6.2.1 Compiler

Name ccrh850.exe
Version v2018.1.5 Release Date Thu Apr 19 23:03:47 PDT 2018

Options

Kernel Options

The following options were used to build the RTA-OS kernel for the configuration that was used to generate the performance figures in this document. If you select different target options, then the values used to build the kernel might change. You can run a Configuration Summary report to check the values used for your configuration.

-cpu=rh850g4mh Specify the target processor instruction set

-fsoft Floating-point operations performed in software (can be changed by target option Handle FPU context)

-ignore_callt_state_in_interrupts CTPSW and CTPC registers are not saved in interrupt routines

-misalign_pack No handling of misaligned data accesses

-nofarcalls Disable generation of far function calls

-no_callt Disable use of the callt instruction

-large_sda Generate 23 bit SDA relocations for load/store instructions

-Ogeneral Optimizer strategy

-Onounroll Prevent the optimizer from loop unrolling

-prepare_dispose Allow V850E prepare and dispose instructions

-registermode=32 No registers are reserved for the user

-reserve_r2 Reserve r2 as an unmodified register

-shorten_loads Convert 23 bit SDA relocations to 16 bit load/store instructions where possible

-Wshadow Warn local variable shadows

-Wundef Warn undefined symbols

--brief_diagnostics Brief error messages

--no_commons Allocate uninitialized global variables to a section and initialize them to zero at program startup

--no_wrap_diagnostics Do not wrap diagnostic error messages

--prototype_errors Report an error for functions with no prototype

--quit_after_warnings Treat all warnings as errors

--short_enum Store enumerations in the smallest possible type

-no_stack_protector Do not enable protection against stack smashing attacks

-sda=8 Set the SDA threshold

Mandatory Options for Application Code

The following options were mandatory for application code used with the configuration that was used to generate the performance figures in this document. If you select different target options, then the values required by application code might change. You can run a Configuration Summary report to check the values used for your configuration.

- The same options as for kernel compilation with the exception of the following which may be omitted from application code:

  -nofarcalls,

  -Onounroll,

  –brief_diagnostics,

  –no_wrap_diagnostics,

  –quit_after_warnings.

Forbidden Options for Application Code

The following options were forbidden for application code used with the configuration that was used to generate the performance figures in this document. If you select different target options, then the forbidden values might change. You can run a Configuration Summary report to check the values used for your configuration.

- Any options that conflict with kernel options for example:

`-callt` Use callt instruction in function prologues/epilogues

`-registermode=22` Restrict the compiler to using 22 general purpose registers

`-registermode=26` Restrict the compiler to using 26 general purpose registers

`-globalreg` Reserve register to hold global variable

`-ga` Use r28 as a frame pointer

`-r20has255` Register r20 is set to the value 255

`-r21has65535` Register r21 is set to the value 65535

`--no_short_enum` Store enumerations as integers

6.2.2 Assembler

Name	ccrh850.exe
Version	v2018.1.5 Release Date Thu Apr 19 23:03:47 PDT 2018

Options

Kernel Options

The following options were used to build the RTA-OS kernel for the configuration that was used to generate the performance figures in this document. If you select different target options, then the values used to build the kernel might change. You can run a Configuration Summary report to check the values used for your configuration.

- The same options as for compilation

Mandatory Options for Application Code

The following options were mandatory for application code used with the configuration that was used to generate the performance figures in this document. If you select different target options, then the values required by application code might change. You can run a Configuration Summary report to check the values used for your configuration.

- The same options as for compilation

Forbidden Options for Application Code

The following options were forbidden for application code used with the configuration that was used to generate the performance figures in this document. If you select different target options, then the forbidden values might change. You can run a Configuration Summary report to check the values used for your configuration.

- Any options that conflict with kernel options

6.2.3 Librarian

Name ax.exe
Version v2018.1.5 Release Date Thu Apr 19 23:02:01 PDT 2018

6.2.4 Linker

Name elxr.exe
Version v2018.1.5 Release Date Thu Apr 19 23:02:02 PDT 2018

Options

Kernel Options

The following options were used to build the RTA-OS kernel for the configuration that was used to generate the performance figures in this document. If you select different target options, then the values used to build the kernel might change. You can run a Configuration Summary report to check the values used for your configuration.

`-Man` Sort map file alphabetically and numerically.

-Ml Output locals to map file

-Mu Output undefined symbols to the map file

-Mx Output cross reference information to the map file

-keepmap Keep the map file if linking fails

-mapfile_type=2 Output detailed information to the map file

-globalcheck=normal Check definition and use of global variables

-strict_overlap_check It is an error if sections overlap

-Qn Do not print linker ID string

-delete Remove unused or unreferenced functions

--ignore_debug_references Ignore relocations from DWARF debug sections

-MD Generate dependence files

-no_map_eofn_symbols Disable end-of-file symbols in the map file

-lnk=-no_xda_modifications Disable linker modification SDA variables

-e Os_sample_reset Specify the application entry point

Mandatory Options for Application Code

The following options were mandatory for application code used with the configuration that was used to generate the performance figures in this document. If you select different target options, then the values required by application code might change. You can run a Configuration Summary report to check the values used for your configuration.

- The same options as for compilation

Forbidden Options for Application Code

The following options were forbidden for application code used with the configuration that was used to generate the performance figures in this document. If you select different target options, then the forbidden values might change. You can run a Configuration Summary report to check the values used for your configuration.

- Any options that conflict with kernel options

6.2.5 Debugger

Name	Lauterbach TRACE32
Version	Build 85076 or later

Notes on using ORTI with the debugger

ORTI Stack Fill with the Lauterbach debugger

The 'ORTI Stack Fill' target option is provided to extend the ORTI support to allow evaluation of unused stack space. The Task.Stack.View command can then be used in the Trace32 debugger.

The application must contain 32 bit constant values referencing the base and size of the stack(s). For each core <N> the constant OS_STACK<N>_BASE must contain the address of the core's stack base and the constant OS_STACK<N>_SIZE must contain the size of the core's stack. For example, assume there are two cores and the linker symbols Os_SP0_BASE and Os_SP1_BASE contain the cores' stack bases and the linker symbols Os_SP0_SIZE and Os_SP1_SIZE contain the cores' stack sizes:

```
extern const uint32 Os_SP0_BASE;
extern const uint32 Os_SP0_SIZE;
extern const uint32 Os_SP1_BASE;
extern const uint32 Os_SP1_SIZE;
const uint32 OS_STACK0_BASE = (uint32)&Os_SP0_BASE;
const uint32 OS_STACK0_SIZE = (uint32)&Os_SP0_SIZE;
const uint32 OS_STACK1_BASE = (uint32)&Os_SP1_BASE;
const uint32 OS_STACK1_SIZE = (uint32)&Os_SP1_SIZE;
```

If there were more than two cores you would extend the above example with OS_STACK2_BASE, OS_STACK2_SIZE, Os_SP2_BASE and Os_SP2_SIZE, etc.

The fill pattern used by the debugger must be contained within a 32 bit constant OS_STACK_FILL (e.g. for a fill pattern 0xCAFEF00D).

```
const uint32 OS_STACK_FILL = 0xCAFEF00D;
```

The stack must be initialized with this fill pattern either in the application start-up routines or during debugger initialization.

7 Hardware

7.1 Supported Devices

This port of RTA-OS has been developed to work with the following target:

Name: Renesas
Device: RH850x2

The following variants of the RH850x2 are supported:

- E2GM
- E2L
- E2M
- U2A
- U2A16
- U2A8

If you require support for a variant of RH850x2 not listed above, please contact ETAS.

7.2 Register Usage

7.2.1 Initialization

RTA-OS requires the following registers to be initialized to the indicated values before `StartOS()` is called.

Register	Setting
CTPC	If core ID caching is enabled (see the Cache Core ID target option), this register is used to store the core ID in multicore configurations. If core ID caching is enabled, in multicore configurations each core must set CTPC to the value stored in PEID before calling any RTA-OS API, including `StartOS()`. This can be done by calling `Os_CacheCoreID()`.
EBASE	The EBASE register must contain the address of the interrupt vector table. If using Direct Vector interrupt vectors the RINT bit should be set to configure a single vector entry (i.e. `EBASE.RINT` must be 1). This can be done by calling `Os_InitializeVectorTable()`.
EIBDn	The INTC interrupt core binding (in multicore parts) must be set to match the values declared in the configuration. This can be done by calling `Os_InitializeVectorTable()`.
EICn	The INTC priorities must be set to match the values declared in the configuration and configured interrupts must be unmasked. This can be done by calling `Os_InitializeVectorTable()`.
INTBP	The INTBP register must contain the address of the EIINT vector table when using Table Reference interrupt vectors. This can be done by calling `Os_InitializeVectorTable()`.
INTCFG.EPL	The interrupt function setting register must be set to disable interrupt level priority extension (i.e. `INTCFG.EPL` must be 0).This can be done by calling `Os_InitializeVectorTable()`.
INTCFG.ISPC	The interrupt function setting register must be set to disable updating of the ISPR register (i.e. `INTCFG.ISPC` must be 1).This can be done by calling `Os_InitializeVectorTable()`.
PSW.EBV	The EBASE register must be used to contain the address of the interrupt vector table (i.e. `PSW.EBV` must be 1). This can be ensured by calling `Os_InitializeVectorTable()`.
PSW.EP	Interrupt processing must not be in progress (i.e. `PSW.EP` must be 0). This can be ensured by calling `Os_InitializeVectorTable()`.
PSW.ID	EIINT interrupts must be enabled (i.e. `PSW.ID` must be 0). This can be ensured by calling `Os_InitializeVectorTable()`.
PSW.NP	FE interrupts must be enabled (i.e. `PSW.NP` must be 0). This can be ensured by calling `Os_InitializeVectorTable()`.
PSW.UM	The CPU must be in Supervisor mode (i.e. `PSW.UM` must be 0).
R4	The SDA registers must be initialized before calling any RTA-OS API, including `StartOS()` and `Os_AwaitStartup()`.
R5	The SDA registers must be initialized before calling any RTA-OS API, including `StartOS()` and `Os_AwaitStartup()`.
SP	The stack must be allocated and SP (R3) initialized before calling any RTA-OS API, including `StartOS()`, `Os_CacheCoreID()` and `Os_AwaitStartup()`.

7.2.2 Modification

The following registers must not be modified by user code after the call to `StartOS()`:

Register	Notes
CTPC	Application code may not directly change the CTPC Register if core ID caching is enabled.
INTCFG	Application code may not directly change the Interrupt function setting Register.
ISPR	Application code may not directly change the In-Service Priority Register.
PLMR	Application code may not directly change the Priority Level Mask Register.
PSW	Application code may not directly change the Program Status Word other than as a result of normal program flow.
R2	R2 is reserved for OS use by the Green Hills Compiler (-reserve_r2). RTA-OS does not use this register and does not preserve this register when handling interrupts.
SP	Application code may not change the stack pointer other than as a result of normal program flow.

7.3 Starting Cores in Multicore Configurations

The default implementations of `Os_AwaitStartup()` and `Os_Cbk_StartCore()` assume that all cores start executing when the RH850 comes out of reset.

7.4 Core/Processor State when StartOS() is called

When `StartOS()` is called RTA-OS expects the following to be true:

- The stack must be allocated and SP (R3) initialized.
- The SDA registers (R4 and R5) must be initialized.
- In multicore configurations when core ID caching is enable, the CTPC register must contain the core ID - i.e. the value in PEID.
- The CPU must be in Supervisor mode (i.e. `PSW.UM` must be 0).
- The EBASE register must be used to contain the address of the interrupt vector table (i.e. `PSW.EBV` must be 1).
- EIINT interrupts must be enabled (i.e. `PSW.ID` must be 0).
- FE interrupts must be enabled (i.e. `PSW.NP` must be 0).
- Interrupt processing must not be in progress (i.e. `PSW.EP` must be 0).
- The EBASE register must contain the address of the interrupt vector table. If using Direct Vector interrupt vectors the RINT bit should be set to configure a single vector entry (i.e. `EBASE.RINT` must be 1).
- The INTBP register must contain the address of the EIINT vector table when using Table Reference interrupt vectors.

- The interrupt function setting register must be set to disable updating of the ISPR register (i.e. `INTCFG.ISPC` must be 1).
- The interrupt function setting register must be set to disable interrupt priority level extension (i.e. `INTCFG.EPL` must be 0).
- The INTC priorities must be set to match the values declared in the configuration and configured interrupts must be unmasked.
- The INTC interrupt core binding (in multicore parts) must be set to match the values declared in the configuration.

If `StartOS()` detects that not all of the above are true it will call ShutdownHook. Please see the documentation for `Os_InitializeVectorTable()` and `Os_CacheCoreID()`.

7.5 Core ID Caching

If core ID caching is enabled (see the 'Cache Core ID' target option), then in multicore configurations RTA-OS expects to be able to read the current core's core ID from the CTPC register, as this is accessible from both User and Supervisor modes. The sample reset code provided (reset.s) initializes CTPC from the PEID register by calling the `Os_CacheCoreID()` function before the C runtime is initialized or `main()` is called. If you supply your own reset code for a multicore build, you should ensure that it either calls `Os_CacheCoreID()` or sets CTPC to the value in PEID on each core.

7.6 Interrupts

This section explains the implementation of RTA-OS's interrupt model on the RH850x2.

7.6.1 Interrupt Priority Levels

Interrupts execute at an interrupt priority level (IPL). RTA-OS standardizes IPLs across all targets. IPL 0 indicates task level. IPL 1 and higher indicate an interrupt priority. It is important that you don't confuse IPLs with task priorities. An IPL of 1 is higher than the highest task priority used in your application.

The IPL is a target-independent description of the interrupt priority on your target hardware. The following table shows how IPLs are mapped onto the hardware interrupt priorities of the RH850x2:

IPL	PLMR	Description
0	16	User (task) level. No interrupts are masked
1	15	Maskable EIINT Category 1 and 2 interrupts
2	14	Maskable EIINT Category 1 and 2 interrupts
...	...	Maskable EIINT Category 1 and 2 interrupts
14	2	Maskable EIINT Category 1 and 2 interrupts
15	1	Maskable EIINT Category 1 and 2 interrupts
16	0	Maskable EIINT Category 1 and 2 interrupts
17	N/A	All CPU vectors (0x0010..0x00F0)

Even though a particular mapping is permitted, all Category 1 ISRs must have equal or higher IPL than all of your Category 2 ISRs.

7.6.2 Allocation of ISRs to Interrupt Vectors

The following restrictions apply for the allocation of Category 1 and Category 2 interrupt service routines (ISRs) to interrupt vectors on the RH850x2. A ✓ indicates that the mapping is permitted and a ✗ indicates that it is not permitted:

Address	Category 1	Category 2
0x0010, SYSERR Category 1 trap not maskable	✓	✗
0x0020, Reserved Category 1 trap not maskable	✓	✗
0x0030, FETRAP Category 1 trap not maskable	✓	✗
0x0040, EITRAP0 Category 1 trap not maskable	✓	✗
0x0050, EITRAP1 Category 1 trap not maskable	✓	✗
0x0060, RIE Category 1 trap not maskable	✓	✗
0x0070, FPE/FXE Category 1 trap not maskable	✓	✗
0x0080, UCPOP Category 1 trap not maskable	✓	✗
0x0090, MIP/MDP Category 1 trap not maskable	✓	✗
0x00A0, PIE Category 1 trap not maskable	✓	✗
0x00B0, Reserved Category 1 trap not maskable	✓	✗
0x00C0, MAE Category 1 trap not maskable	✓	✗
0x00D0, Reserved Category 1 trap not maskable	✓	✗
0x00E0, FENMI Category 1 trap not maskable	✓	✗
0x00F0, FEINT Category 1 interrupt maskable through the PSW.NP bit	✓	✗
0x0100+4*n, Maskable EIINT channel n interrupt handler	✓	✓

CPU traps and interrupts (vectors 0x0010..0x00F0) always operate in direct vector mode. The address specified for a CPU interrupt in the RTA-OS configuration is its offset, e.g. EITRAP0 would be identified by vector address 0x0040. CPU traps and interrupts can only be configured as category 1 interrupts and they must have priority (IPL) 17. Maskable EIINT interrupts can operate in either direct vector or table reference mode (controlled by the target option 'Direct Vector mode'). In the RTA-OS configuration, EIINT interrupt channel n is specified by vector address 0x0100+n*4. For example EIINT channel 19 would be identified by vector address 0x014C.

7.6.3 Cross-core Interrupts

This target port uses the first free IPIR channel for cross-core interrupt generation. In multicore builds at least one IPIR channel must be free for RTA-OS to use. That is at least one of the vectors 0x0100, 0x0104, 0x0108 or 0x010C (EIINT channels 0, 1, 2 or 3) must not be configured as an interrupt in the RTA-OS configuration.

7.6.4 CPU Vectors and the ProtectionHook or Default Interrupt

If there is no default interrupt configured, or enhanced isolation is enabled, any CPU vector (vectors 0x0010..0x00F0) that does not have an ISR attached in the project

configuration is routed to the ProtectionHook. I.e if a CPU trap or interrupt occurs and no ISR has been configured for that trap/interrupt's vector then `ProtectionHook` will be called. If a default interrupt has been configured and enhanced isolation is not enabled then any CPU vector (vectors 0x0010..0x00F0) that does not have an ISR attached in the project configuration is routed to the default interrupt, except for vectors 0x0090 and 0x00A0 that are routed to the ProtectionHook.

7.6.5 Recovering from CPU Exceptions/Interrupts that call the ProtectionHook

If a CPU trap or interrupt (vectors 0x0010..0x00F0) occurs the RH850 will set `PSW.ID` and/or `PSW.EP` and/or `PSW.NP` to stop any other interrupts from occurring. RTA-OS does not clear `PSW.ID`, `PSW.EP`, `PSW.NP` or the source of the interrupt/trap (e.g. FEINT) before it calls `ProtectionHook`. Therefore if `ProtectionHook` restarts the OS after a CPU trap or interrupt, `PSW.ID`, `PSW.EP`, `PSW.NP` and the source of the trap/interrupt must be cleared before calling `StartOS()` again. `PSW.ID`, `PSW.EP` and `PSW.NP` can be cleared by calling `Os_InitializeVectorTable()`.

RTA-OS does not clear the source of the interrupt/trap as doing this may destroy state that is needed by `ProtectionHook` to work out what action to take.

7.6.6 Interrupt Priority Level Extension

Please note that this port does not support the interrupt priority level extension available on the U2Axx parts.

7.6.7 Register Banks

Please note that this port does not use the register bank mechanism.

7.6.8 Vector Table

rtaosgen normally generates an interrupt vector table for you automatically. You can configure “Suppress Vector Table Generation” as `true` to stop RTA-OS from generating the interrupt vector table.

Depending upon your target, you may be responsible for locating the generated vector table at the correct base address. The following table shows the section (or sections) that need to be located and the associated valid base address:

Section	Valid Addresses
Os_intvect	This section contains the interrupt vector table (`Os_interrupt_vectors`) generated by RTA-OS. The RTA-OS generated vector table does not contain the reset vector so this section must be aligned such that its first address is on a 512 + 16 byte boundary. See below for more details.
Os_EI_vect	This section contains the Table Reference mode EIINT vectors generated by RTA-OS. This section should be aligned on a 512 byte boundary so that its address can be stored in INTBP register when using Table Reference interrupt vectors.
Os_primitives	This section contains RTA-OS assembly code that may be run on any core.
Os_primitivesX	Where X is a single digit. These sections contains RTA-OS assembly code that is only run on core X. E.g. `Os_primitives0` contains code that will only be run on core 0 and `Os_primitives2` contains code that will only be run on core 2. These sections will only be used if the Core Specific Wrappers target option is set to "true".

The function `Os_InitializeVectorTable` should be called before `StartOS()` to set the EBASE register to the address of the vector table (`Os_interrupt_vectors` by default) aligned to a 512 byte boundary.

The RTA-OS generated vector table does not include the reset vector. The first entry in the RTA-OS generated vector table is for SYSERR (offset 0x10). This is to allow you to locate the reset vector immediately before the RTA-OS generated vector table (see the sample applications for an example of this). You should provide the reset vector and locate it as required.

If you use the RTA-OS generated vector table, it is essential that `Os_interrupt_vectors` is located 16 bytes after the value store in the EBASE register so that the first entry in the vector table (SYSERR) is 16 bytes after the address in EBASE. The EBASE register does not store the least significant 9 bits of the vector table address (i.e. when the value X is written to EBASE, the value stored in EBASE is X&0xFFFFFE00), therefore `Os_interrupt_vectors` must be located on a 512 + 16 byte boundary (`Os_interrupt_vectors` is in the `Os_intvec` section). See the sample applications for an example linker script that achieves this.

When the default interrupt is configured the RTA-OS generated vector table contains entries for all supported interrupts for the selected chip variant. If the default interrupt is not configured then entries are created up the highest configured interrupt.

RTA-OS reserves the EITRAP0 vector (i.e. 0x0040) for applications that use untrusted code.

RTA-OS currently supports two vector table formats: Table reference method and Direct vector.

When you supply the vector table (i.e. 'Suppress Vector Table Generation' is set to TRUE) the label `Os_interrupt_vectors` should be placed at the start of the vector table so that the function `Os_InitializeVectorTable()` can set the EBASE register to the correct address. Note that the minimum alignment of the EBASE register is 512 bytes so the vector table must be aligned accordingly.

7.6.9 Using Raw Exception Handlers

RTA-OS supports direct branches in the interrupt vector table for CPU vectors (vectors 0x0010..0x00F0). Normally RTA-OS produces wrapper code around the interrupt handler functions for CPU vectors to enforce the correct interrupt controller IPL settings. If interrupt handlers for CPU vectors are given names starting with "b_" then the interrupt vector table entry is an unconditional jump relative "jr" instruction to the handler function. When using these raw exception handlers it is the user's responsibility that:

- The correct register context is saved and restored.
- The correct return instruction is used.
- Interrupts are not re-enabled in these handlers.
- The RTA-OS API is not used in these handlers.

7.6.10 Writing Category 1 Interrupt Handlers

Raw Category 1 interrupt service routines (ISRs) must correctly handle the interrupt context themselves. RTA-OS provides an optional helper macro `CAT1_ISR` that can be used to make code more portable. Depending on the target, this may cause the selection of an appropriate interrupt control directive to indicate to the compiler that a function requires additional code to save and restore the interrupt context.

A Category 1 ISR therefore has the same structure as a Category 2 ISR, as shown below.

```
CAT1_ISR(Category1Handler) {
  /* Handler routine */
}
```

7.6.11 Writing Category 2 Interrupt Handlers

Category 2 ISRs are provided with a C function context by RTA-OS, since the RTA-OS kernel handles the interrupt context itself. The handlers are written using the `ISR()` macro as shown below:

```
#include <Os.h>
ISR(MyISR) {
  /* Handler routine */
}
```

You must not insert a return from interrupt instruction in such a function. The return is handled automatically by RTA-OS.

7.6.12 Default Interrupt

The 'default interrupt' is intended to be used to catch all unexpected interrupts. All unused interrupts have their interrupt vectors directed to the named routine that you specify. The routine you provide is not handled by RTA-OS and must correctly handle the interrupt context itself. The handler must use the `CAT1_ISR` macro in the same way as a Category 1 ISR (see Section 7.6.10 for further details).

7.7 Memory Model

The following memory models are supported:

Model	Description
Standard	The standard 32-bit EABI memory model is used.

7.8 Processor Modes

RTA-OS can run in the following processor modes:

Mode	Notes
Supervisor	All trusted code runs in Supervisor mode (`PSW.UM` is clear).
User	All untrusted code runs in User mode (`PSW.UM` is set).

7.9 Stack Handling

RTA-OS uses a single stack for all tasks and ISRs.

RTA-OS manages the stack (via register R3).

8 Enhanced Isolation

For details on the ’Enhanced Isolation’ extensions to RTA-OS and how these are used in an application please refer to ’RTA-OS RH850x2GHS Enhanced Isolation.pdf’.

9 Performance

This chapter provides detailed information on the functionality, performance and memory demands of the RTA-OS kernel. RTA-OS is highly scalable. As a result, different figures will be obtained when your application uses different sets of features. The figures presented in this chapter are representative for the RH850x2/GHS port based on the following configuration:

- There are 32 tasks in the system
- Standard build is used
- Stack monitoring is disabled
- Time monitoring is disabled
- There are no calls to any hooks
- Tasks have unique priorities
- Tasks are not queued (i.e. tasks are BCC1 or ECC1)
- All tasks terminate/wait in their entry function
- Tasks and ISRs do not save any auxiliary registers (for example, floating point registers)
- Resources are shared by tasks only
- The generation of the resource `RES_SCHEDULER` is disabled

9.1 Measurement Environment

The following hardware environment was used to take the measurements in this chapter:

Device	E2GM on Renesas RH850/E2GM 373 Pin E2GM (R7F702Z05)
CPU Clock Speed	400.0MHz
Stopwatch Speed	400.0MHz

9.2 RAM and ROM Usage for OS Objects

Each OS object requires some ROM and/or RAM. The OS objects are generated by **rtaosgen** and placed in the RTA-OS library. In the main:

- `Os_Cfg_Counters` includes data for counters, alarms and schedule tables.
- `Os_Cfg` contains the data for most other OS objects.

9.2.1 Single Core

The following table gives the ROM and/or RAM requirements (in bytes) for each OS object in a simple single-core configuration. Note that object sizes will vary depending on the project configuration and compiler packing issues.

Object	ROM	RAM
Alarm	2	12
Cat 2 ISR	8	0
Counter	20	4
CounterCallback	4	0
ExpiryPoint	3.5	0
OS Overheads (max)	0	69
OS-Application	0	0
PeripheralArea	0	0
Resource	8	4
ScheduleTable	16	12
Task	20	0

9.2.2 Multi Core

The following table gives the ROM and/or RAM requirements (in bytes) for each OS object in a simple multi-core configuration. Note that object sizes will vary depending on the project configuration and compiler packing issues.

Object	ROM	RAM
Alarm	8	12
Cat 2 ISR	16	0
Core Overheads (each OS core)	0	60
Core Overheads (each processor core)	20	25
Counter	32	4
CounterCallback	4	0
ExpiryPoint	3.5	0
OS Overheads (max)	0	9
OS-Application	8	0
PeripheralArea	0	0
Resource	16	4
ScheduleTable	20	12
Task	36	0

9.3 Stack Usage

The amount of stack used by each Task/ISR in RTA-OS is equal to the stack used in the Task/ISR body plus the context saved by RTA-OS. The size of the run-time context saved by RTA-OS depends on the Task/ISR type and the exact system configuration. The only reliable way to get the correct value for Task/ISR stack usage is to call the `Os_GetStackUsage()` API function.

Note that because RTA-OS uses a single-stack architecture, the run-time contexts of all tasks reside on the same stack and are recovered when the task terminates. As a result, run-time contexts of mutually exclusive tasks (for example, those that share an internal resource) are effectively overlaid. This means that the worst case stack usage can be significantly less than the sum of the worst cases of each object on the system. The RTA-OS tools automatically calculate the total worst case stack usage for you and present this as part of the configuration report.

9.4 Library Module Sizes

9.4.1 Single Core

The RTA-OS kernel is demand linked. This means that each API call is placed into a separately linkable module. The following table lists the section sizes for each API module (in bytes) for the simple single-core configuration in standard status.

Library Module	.Os_EI_vect	.Os_intvect	.Os_primitives	.bss	.rodata	.rosdata	.sbss	.text
ActivateTask								100
AdvanceCounter								4
CallTrustedFunction								26
CancelAlarm								68
ChainTask								86
CheckISRMemoryAccess								44
CheckObjectAccess								116
CheckObjectOwnership								112
CheckTaskMemoryAccess								44
ClearEvent								18
ControlIdle							4	32
DisableAllInterrupts							8	38
DispatchTask								182
ElapsedTime								98
EnableAllInterrupts								32
GetActiveApplicationMode								8
GetAlarm								134
GetAlarmBase								54
GetApplicationID								36
GetCounterValue								28
GetCurrentApplicationID								36
GetElapsedCounterValue								56
GetEvent								18
GetExecutionTime								18
GetISRID								8
GetIsrMaxExecutionTime								18

Library Module	.Os_EI_vect	.Os_intvect	.Os_primitives	.bss	.rodata	.rosdata	.sbss	.text
GetIsrMaxStackUsage								18
GetResource								56
GetScheduleTableStatus								28
GetStackSize								6
GetStackUsage								18
GetStackValue								14
GetTaskID								14
GetTaskMaxExecutionTime								18
GetTaskMaxStackUsage								18
GetTaskState								36
GetVersionInfo								24
Idle								4
InShutdown								2
IncrementCounter								18
InterruptSource						4		138
ModifyPeripheral								84
NextScheduleTable								106
Os_CacheCoreID								10
Os_Cfg				528	768	8	33	214
Os_Cfg_Counters					472			4178
Os_Cfg_KL								54
Os_FE_wrapper			190					
Os_GetCurrentIMask								6
Os_GetCurrentTPL								34
Os_StartCores							8	42
Os_TstAndSet								8
Os_Vectors	516	224	48					
Os_Wrapper								84
Os_abort			12					
Os_mid_wrapper			190					
Os_setjmp			120					
Os_vec_init								114
ProtectionSupport								28
ReadPeripheral								78
ReleaseResource								62
ResetIsrMaxExecutionTime								18
ResetIsrMaxStackUsage								18
ResetTaskMaxExecutionTime								18
ResetTaskMaxStackUsage								18
ResumeAllInterrupts								32

Library Module	.Os_EI_vect	.Os_intvect	.Os_primitives	.bss	.rodata	.rosdata	.sbss	.text
ResumeOSInterrupts								32
Schedule								80
SetAbsAlarm								76
SetEvent								18
SetRelAlarm								130
SetScheduleTableAsync								42
ShutdownOS								44
StackOverrunHook								6
StartOS								176
StartScheduleTableAbs								102
StartScheduleTableRel								92
StartScheduleTableSynchron								42
StopScheduleTable								60
SuspendAllInterrupts							8	38
SuspendOSInterrupts							8	48
SyncScheduleTable								42
SyncScheduleTableRel								42
TerminateTask								20
ValidateCounter								38
ValidateISR								14
ValidateResource								32
ValidateScheduleTable								32
ValidateTask								32
WaitEvent								18
WritePeripheral								66

9.4.2 Multi Core

The RTA-OS kernel is demand linked. This means that each API call is placed into a separately linkable module. The following table lists the section sizes for each API module (in bytes) for the simple multi-core configuration in standard status.

Library Module	.Os_EI_vect	.Os_intvect	.Os_primitives	.bss	.rodata	.rosdata	.sbss	.text
ActivateTask								216
AdvanceCounter								4

Library Module	.Os_EI_vect	.Os_intvect	.Os_primitives	.bss	.rodata	.rosdata	.sbss	.text
CallTrustedFunction								26
CancelAlarm								100
ChainTask								152
CheckISRMemoryAccess								46
CheckObjectAccess								192
CheckObjectOwnership								148
CheckTaskMemoryAccess								46
ClearEvent								18
ControlIdle							8	46
CrossCore								42
DisableAllInterrupts								42
DispatchTask								352
ElapsedTime								98
EnableAllInterrupts								38
GetActiveApplicationMode								8
GetAlarm								132
GetAlarmBase								52
GetApplicationID								52
GetCounterValue								28
GetCurrentApplicationID								54
GetElapsedCounterValue								56
GetEvent								18
GetExecutionTime								18
GetISRID								22
GetIsrMaxExecutionTime								18
GetIsrMaxStackUsage								18
GetNumberOfActivatedCores								24
GetResource								64
GetScheduleTableStatus								58
GetSpinlock								4
GetStackSize								6
GetStackUsage								18
GetStackValue								58
GetTaskID								28
GetTaskMaxExecutionTime								18
GetTaskMaxStackUsage								18
GetTaskState								68
GetVersionInfo								24
Idle								4
InShutdown								2

Library Module	.Os_EI_vect	.Os_intvect	.Os_primitives	.bss	.rodata	.rosdata	.sbss	.text
IncrementCounter								18
InterruptSource						4		138
ModifyPeripheral								84
NextScheduleTable								140
Os_CacheCoreID								10
Os_Cfg				680	1524		11	274
Os_Cfg_Counters					632			5238
Os_Cfg_KL								82
Os_CrossCoreISR								92
Os_FE_wrapper			190					
Os_GetCurrentIMask								6
Os_GetCurrentTPL								80
Os_RaiseCrossCoreISR								14
Os_ScheduleQ								50
Os_StartCores							8	68
Os_TstAndSet								62
Os_Vectors	516	224	64					
Os_Wrapper								96
Os_abort			12					
Os_mid_wrapper			190					
Os_setjmp			120					
Os_vec_init								158
ProtectionSupport								28
ReadPeripheral								78
ReleaseResource								78
ReleaseSpinlock								4
ResetIsrMaxExecutionTime								18
ResetIsrMaxStackUsage								18
ResetTaskMaxExecutionTime								18
ResetTaskMaxStackUsage								18
ResumeAllInterrupts								38
ResumeOSInterrupts								38
Schedule								88
SetAbsAlarm								112
SetEvent								18
SetRelAlarm								164
SetScheduleTableAsync								42
ShutdownAllCores								60
ShutdownOS								74
StackOverrunHook								6

Library Module	.Os_EI_vect	.Os_intvect	.Os_primitives	.bss	.rodata	.rosdata	.sbss	.text
StartCore								54
StartNonAutosarCore								54
StartOS								310
StartScheduleTableAbs								140
StartScheduleTableRel								124
StartScheduleTableSynchron								42
StopScheduleTable								92
SuspendAllInterrupts								42
SuspendOSInterrupts								52
SyncScheduleTable								42
SyncScheduleTableRel								42
TerminateTask								30
TryToGetSpinlock								10
ValidateCounter								32
ValidateISR								14
ValidateResource								32
ValidateScheduleTable								38
ValidateTask								60
WaitEvent								18
WritePeripheral								66

9.5 Execution Time

The following tables give the execution times in CPU cycles, i.e. in terms of ticks of the processor's program counter. These figures will normally be independent of the frequency at which you clock the CPU. To convert between CPU cycles and SI time units the following formula can be used:

Time in microseconds = Time in cycles / CPU Clock rate in MHz

For example, an operation that takes 50 CPU cycles would be:

- at 20MHz = 50/20 = 2.5μs
- at 80MHz = 50/80 = 0.625μs
- at 150MHz = 50/150 = 0.333μs

While every effort is made to measure execution times using a stopwatch running at the same rate as the CPU clock, this is not always possible on the target hardware. If the stopwatch runs slower than the CPU clock, then when RTA-OS reads the stopwatch,

there is a possibility that the time read is less than the actual amount of time that has elapsed due to the difference in resolution between the CPU clock and the stopwatch (the *User Guide* provides further details on the issue of uncertainty in execution time measurement).

The figures presented in Section 9.5.1 have an uncertainty of 0 CPU cycle(s).

Values are given for single-core operation only. Timings for cross-core activations, though interesting, are variable because of the nature of multi-core operation. Minimum values cannot be given, because timings are dependent on the activity on the core that receives the activation.

9.5.1 Context Switching Time

Task switching time is the time between the last instruction of the previous task and the first instruction of the next task. The switching time differs depending on the switching contexts (e.g. an `ActivateTask()` versus a `ChainTask()`).

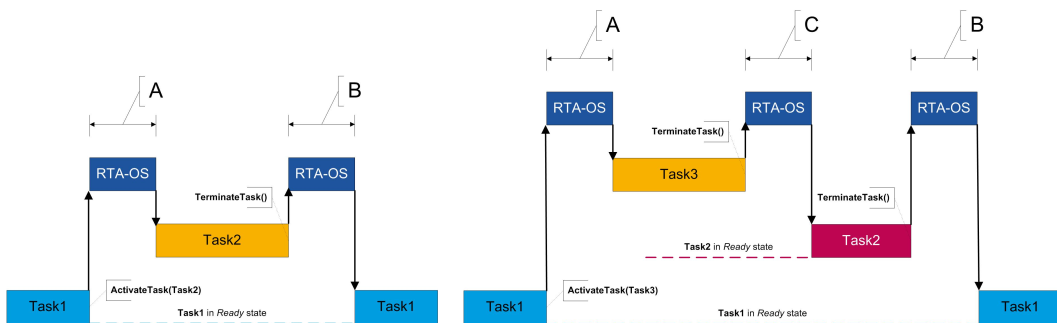
Interrupt latency is the time between an interrupt request being recognized by the target hardware and the execution of the first instruction of the user provided handler function:

For Category 1 ISRs this is the time required for the hardware to recognize the interrupt.

For Category 2 ISRs this is the time required for the hardware to recognize the interrupt plus the time required by RTA-OS to set-up the context in which the ISR runs.

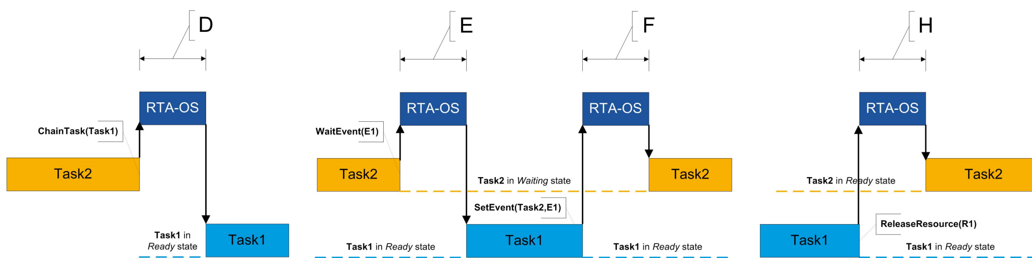
Figure 9.1 shows the measured context switch times for RTA-OS.

Switch	Key	CPU Cycles	Actual Time
Task activation	A	93	232ns
Task termination with resume	B	62	155ns
Task termination with switch to new task	C	98	245ns
Chaining a task	D	134	335ns
Waiting for an event resulting in transition to the WAITING state	E	237	592ns
Setting an event results in task switch	F	310	775ns
Non-preemptive task offers a preemption point (co-operative scheduling)	G	88	220ns
Releasing a resource results in a task switch	H	87	218ns
Entering a Category 2 ISR	I	83	208ns
Exiting a Category 2 ISR and resuming the interrupted task	J	94	235ns
Exiting a Category 2 ISR and switching to a new task	K	96	240ns
Entering a Category 1 ISR	L	51	128ns



(a) Task activated. Termination resumes preempted task.

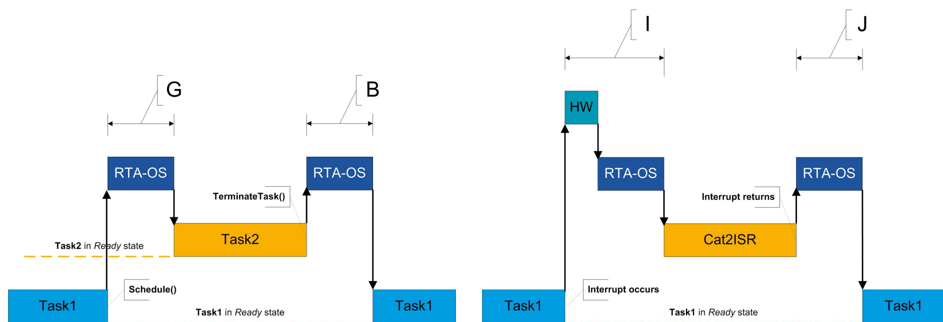
(b) Task activated. Termination switches into new task.



(c) Task chained.

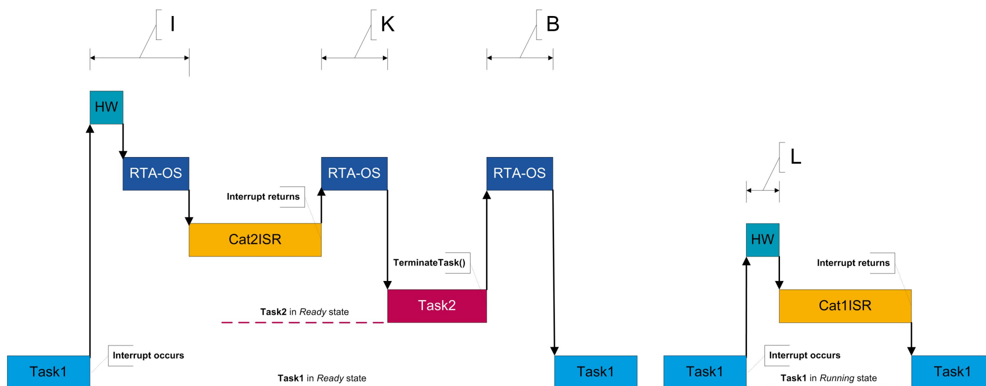
(d) Task waits. Task is resumed when event set.

(e) Task switch when resource is released.



(f) Request for scheduling made by non-preemptive task.

(g) Category 2 interrupt entry. Interrupted task resumed on exit.



(h) Category 2 interrupt entry. Switch to new task on exit.

(i) Category 1 interrupt entry.

Figure 9.1: Context Switching

10 Finding Out More

Additional information about RH850x2/GHS-specific parts of RTA-OS can be found in the following manuals:

***RH850x2/GHS Release Note*.** This document provides information about the RH850x2/GHS port plug-in release, including a list of changes from previous releases and a list of known limitations.

Information about the port-independent parts of RTA-OS can be found in the following manuals, which can be found in the RTA-OS installation (typically in the Documents folder):

***Getting Started Guide*.** This document explains how to install RTA-OS tools and describes the underlying principles of the operating system

***Reference Guide*.** This guide provides a complete reference to the API, programming conventions and tool operation for RTA-OS.

***User Guide*.** This guide shows you how to use RTA-OS to build real-time applications.

11 Contacting ETAS

11.1 Technical Support

Technical support is available to all users with a valid support contract. If you do not have a valid support contract, please contact your regional sales office (see Section 11.2.2).

The best way to get technical support is by email. Any problems or questions about the use of the product should be sent to:

rta.hotline.uk@etas.com

If you prefer to discuss your problem with the technical support team, you call the support hotline on:

+44 (0)1904 562624.

The hotline is available during normal office hours (0900-1730 GMT/BST).

In either case, it is helpful if you can provide technical support with the following information:

- Your support contract number
- Your `.xml`, `.arxml`, `.rtaos` and/or `.stc` files
- The command line which caused the error
- The version of the ETAS tools you are using
- The version of the compiler tool chain you are using
- The error message you received (if any)
- The file `Diagnostic.dmp` if it was generated

11.2 General Enquiries

11.2.1 ETAS Global Headquarters

ETAS GmbH		
Borsigstrasse 24	Phone:	+49 711 3423-0
70469 Stuttgart	Fax:	+49 711 3423-2106
Germany	WWW:	www.etas.com

11.2.2 ETAS Local Sales & Support Offices

Contact details for your local sales office and local technical support team (where available) can be found on the ETAS web site:

ETAS subsidiaries	www.etas.com/en/contact.php
ETAS technical support	www.etas.com/en/hotlines.php

Index

A
Assembler, 41
AUTOSAR OS includes
  Os.h, 29
  Os_Cfg.h, 29
  Os_MemMap.h, 29

C
CAT1_ISR, 36
Compiler, 39
Compiler (Green Hills v2018.1.5), 39
Compiler Versions, 39
Configuration
  Port-Specific Parameters, 21
Core ID Caching, 48

D
Debugger, 43

E
Enhanced Isolation, 54
ETAS License Manager, 11
  Installation, 11

F
Files, 29

H
Hardware
  Requirements, 9

I
Installation, 9
  Default Directory, 10
  Verification, 19
Interrupt Priority Level Extension, 50
Interrupts, 48
  Category 1, 52
  Category 2, 52
  Default, 53
IPL, 48

L
Librarian, 42
Library
  Name of, 29
License, 11
  Borrowing, 15
  Concurrent, 12
  Grace Mode, 12
  Installation, 15
  Machine-named, 12
  Status, 15
  Troubleshooting, 16
  User-named, 12
Linker, 42

M
Memory Model, 53
Multicore, 47

O
Options, 39
Os_AwaitStartup, 30
Os_CacheCoreID, 31
Os_Cbk_GetAbortStack, 33
Os_Cbk_StartCore, 34
Os_Clear_x, 36
Os_Disable_x, 37
Os_DisableAllConfiguredInterrupts_Corex, 36
Os_Enable_x, 37
Os_EnableAllConfiguredInterrupts_Corex, 37
Os_InitializeVectorTable, 32
Os_IntChannel_x, 38
Os_StackSizeType, 38
Os_StackValueType, 38

P
Parameters of Implementation, 21
Performance, 55
  Context Switching Times, 63
  Library Module Sizes, 57
  RAM and ROM, 55
  Stack Usage, 56
Processor Modes, 53
  Supervisor, 53
  User, 53
ProtectionHook, 49, 50

R
Register Banks, 50

Registers
  CTPC, 46, 47
  EBASE, 46
  EIBDn, 46
  EICn, 46
  Initialization, 45
  INTBP, 46
  INTCFG, 47
  INTCFG.EPL, 46
  INTCFG.ISPC, 46
  ISPR, 47
  Non-modifiable, 46
  PLMR, 47
  PSW, 47
  PSW.EBV, 46
  PSW.EP, 46
  PSW.ID, 46
  PSW.NP, 46
  PSW.UM, 46
  R2, 47
  R4, 46
  R5, 46
  SP, 46, 47

S

Software
  Requirements, 9
Stack, 53
StartOS, 47

T

Target, 45
  Variants, 45
Toolchain, 39

U

Using Raw Exception Handlers, 52

V

Variants, 45
Vector Table
  Base Address, 50