RTA-OS

TriCore/GHS Port Guide

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 2008-2018 ETAS GmbH, Stuttgart.

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

Document: 10677-PG-5.0.8 EN-08-2018

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 **8**
- 1.1 About You 9
- 1.2 Document Conventions 9
- 1.3 References 10

2 Installing the RTA-OS Port Plug-in **11**
- 2.1 Preparing to Install 11
  - 2.1.1 Hardware Requirements 11
  - 2.1.2 Software Requirements 11
- 2.2 Installation 12
  - 2.2.1 Installation Directory 12
- 2.3 Licensing 13
  - 2.3.1 Installing the ETAS License Manager 13
  - 2.3.2 Licenses 14
  - 2.3.3 Installing a Concurrent License Server 15
  - 2.3.4 Using the ETAS License Manager 16
  - 2.3.5 Troubleshooting Licenses 19

3 Verifying your Installation **22**
- 3.1 Checking the Port 22
- 3.2 Running the Sample Applications 22

4 Port Characteristics **24**
- 4.1 Parameters of Implementation 24
- 4.2 Configuration Parameters 24
  - 4.2.1 Stack used for C-startup 24
  - 4.2.2 Stack used when idle 25
  - 4.2.3 Stack overheads for ISR activation 25
  - 4.2.4 Stack overheads for ECC tasks 25
  - 4.2.5 Stack overheads for ISR 26
  - 4.2.6 ORTI/Lauterbach 26
  - 4.2.7 ORTI/winIDEA 26
  - 4.2.8 ORTI Stack Fill 26
  - 4.2.9 Support winIDEA Analyzer 27
  - 4.2.10 ORTI/SMP 27
  - 4.2.11 CrossCore SRC0 27
  - 4.2.12 CrossCore SRC1 27
  - 4.2.13 CrossCore SRC2 28
  - 4.2.14 CrossCore SRC3 28
  - 4.2.15 CrossCore SRC4 28
  - 4.2.16 CrossCore SRC5 28
  - 4.2.17 Block default interrupt 28
  - 4.2.18 User Mode 29
  - 4.2.19 Trusted with protection PRS 29
  - 4.2.20 Guard supervisor access 29

4.2.21 Interrupt vector matches priority 30
4.2.22 OS Locks disable Cat1 30
4.2.23 Enable stack repositioning 30
4.2.24 Enhanced Isolation 31
4.2.25 Link Type 31
4.2.26 Small data threshold 31
4.2.27 Short enums 31
4.2.28 FP Instructions 32
4.2.29 Far jumps 32
4.2.30 Max Optimizations 32
4.2.31 Optimization Type 33
4.2.32 Customer Option Set 1 33
4.2.33 Customer Option Set 2 33
4.3 Generated Files 34

5 Port-Specific API **35**
5.1 API Calls 35
5.1.1 Os_GetTrapInfo 35
5.1.2 Os_InitializeInterruptTable 37
5.1.3 Os_InitializeServiceRequests 37
5.1.4 Os_InitializeTrapTable 39
5.1.5 Os_InitializeVectorTable 39
5.1.6 Os_StartCoreGate 40
5.2 Callbacks 41
5.2.1 Os_Cbk_StartCore 41
5.3 Macros 42
5.3.1 CAT1_ISR 42
5.3.2 CAT1_TRAP 42
5.3.3 OS_CORE_isrname 43
5.3.4 OS_INIT_srcname 43
5.3.5 OS_VEC_isrname 43
5.3.6 Os_DisableAllConfiguredInterrupts 43
5.3.7 Os_Disable_x 43
5.3.8 Os_EnableAllConfiguredInterrupts 44
5.3.9 Os_Enable_x 44
5.3.10 Os_IntChannel_x 44
5.4 Type Definitions 44
5.4.1 OsTrapInfoRefType 44
5.4.2 OsTrapInfoType 45
5.4.3 Os_StackSizeType 45
5.4.4 Os_StackTraceType 45
5.4.5 Os_StackValueType 45

6 Toolchain **46**
6.1 Compiler Versions 46
6.1.1 v2018.1.5 46
6.1.2 v2017.1.5 46
6.1.3 v2015.1.7 47
6.2 Options used to generate this guide 47
6.2.1 Compiler 47
6.2.2 Assembler 48
6.2.3 Librarian 49
6.2.4 Linker 49
6.2.5 Debugger 50

7 Hardware **52**
7.1 Supported Devices 52
7.2 Register Usage 53
7.2.1 Initialization 53
7.2.2 Modification 54
7.3 Interrupts 55
7.3.1 Interrupt Priority Levels 55
7.3.2 Allocation of ISRs to Interrupt Vectors 55
7.3.3 Vector Table 56
7.3.4 Writing Category 1 Interrupt Handlers 58
7.3.5 Writing Category 2 Interrupt Handlers 58
7.3.6 Default Interrupt 59
7.4 Memory Model 59
7.5 Processor Modes 59
7.6 Stack Handling 59

8 Performance **60**
8.1 Measurement Environment 60
8.2 RAM and ROM Usage for OS Objects 60
8.2.1 Single Core 61
8.2.2 Multi Core 61
8.3 Stack Usage 61
8.4 Library Module Sizes 62
8.4.1 Single Core 62
8.4.2 Multi Core 65
8.5 Execution Time 69
8.5.1 Context Switching Time 70

9 Finding Out More **72**

10 Contacting ETAS **73**
10.1 Technical Support 73
10.2 General Enquiries 73
10.2.1 ETAS Global Headquarters 73
10.2.2 ETAS Local Sales & Support Offices 73

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 TriCore/GHS port plug-in that customizes the RTA-OS development tools for the Infineon TriCore with the GHS 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 TriCore/GHS port and cover:

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

Chapters 4 to 8 provide reference information including:

- the number of OS objects supported;
- required and recommended toolchain parameters;
- how RTA-OS interacts with the TriCore, 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 TriCore/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 TriCore/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 TriCore/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 `TriCoreGHS_5.0.8`. This contains everything to do with the port plug-in.

Each version of the port installs in its own directory - the trailing `_5.0.8` 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:

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

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

RTA-OS TriCoreGHS 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`.

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;

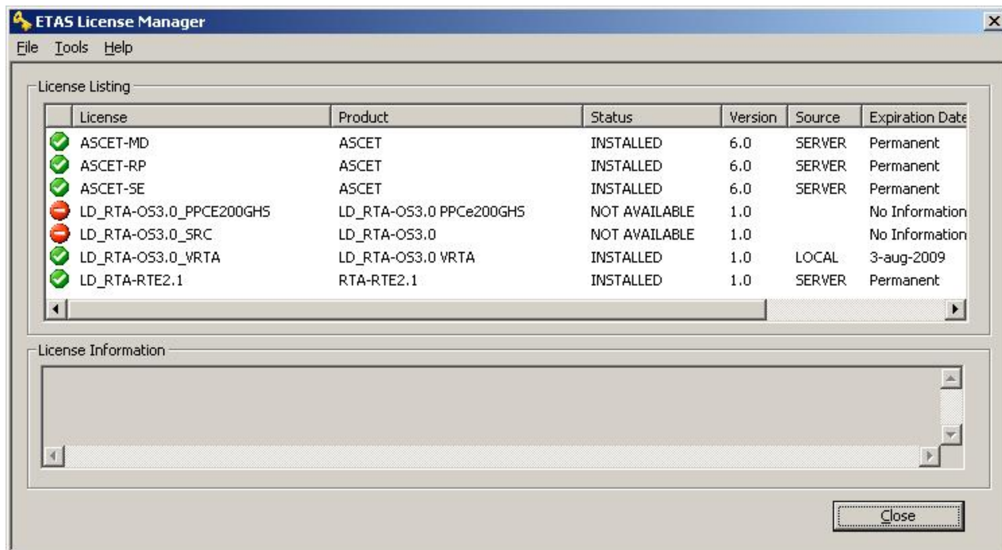


Figure 2.1: The ETAS License manager

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 10.

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 user-name 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.

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
```

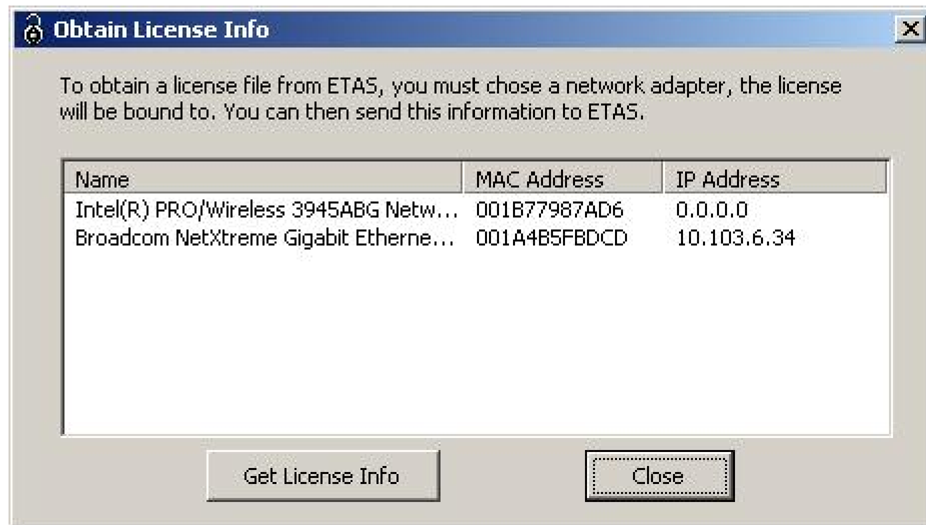


Figure 2.2: Obtaining License Information

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.

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.)

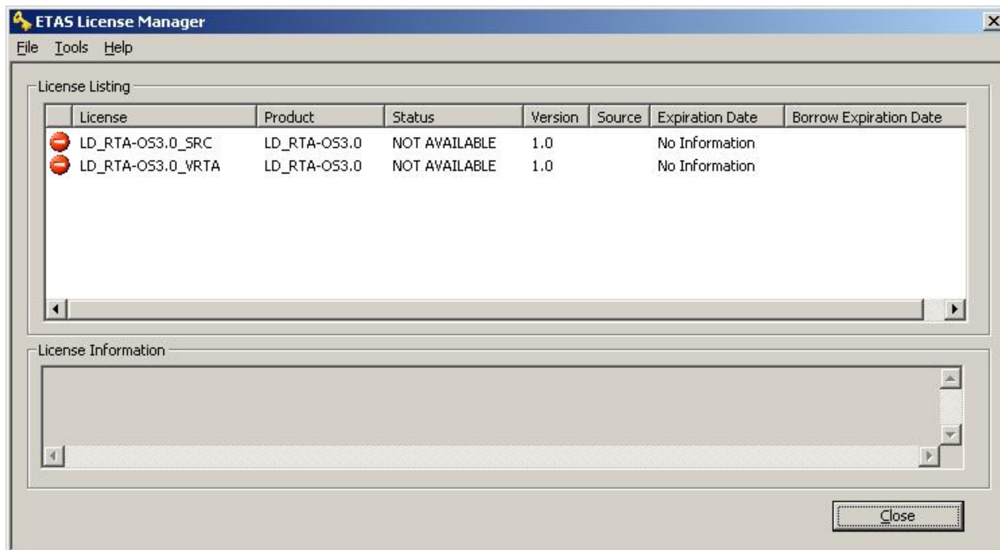


Figure 2.3: Unlicensed RTA-OS Installation

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

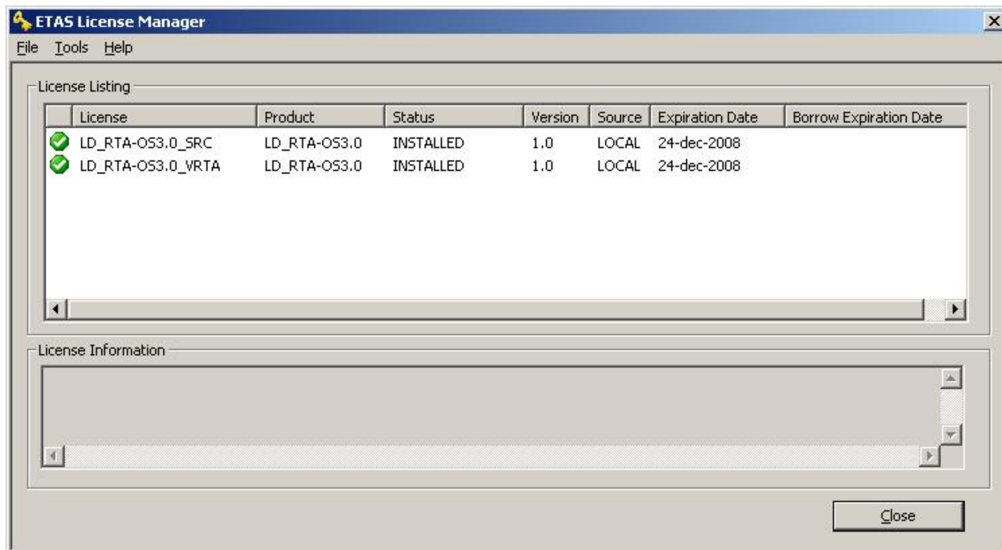


Figure 2.4: Licensed features for RTA-OS

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”

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`.

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.

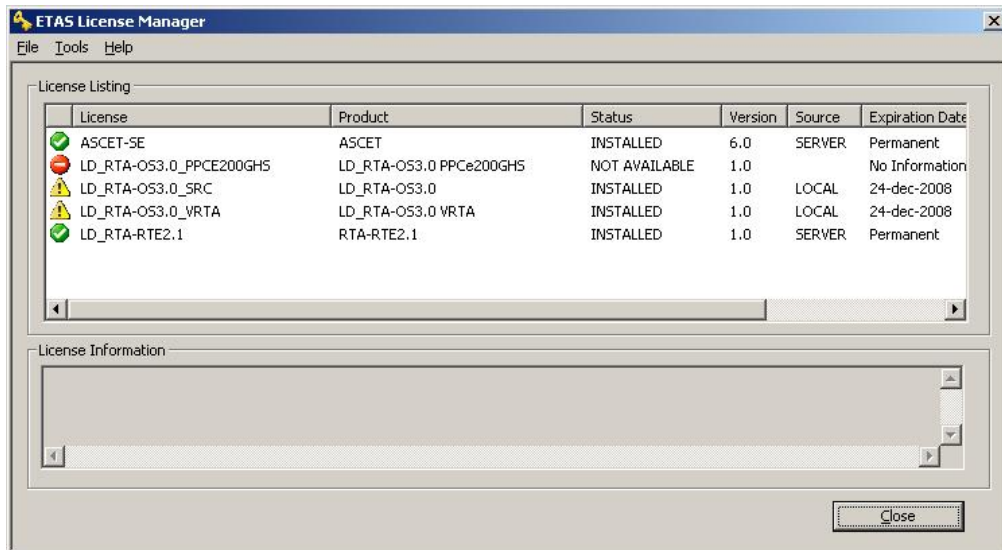


Figure 2.5: Licensed features that are due to expire

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 10.1. 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 TriCore/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 TriCore/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 TriCore/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 ORTI/Lauterbach

XML name Orti22Lauterbach

Description

Select ORTI generation for the Lauterbach debugger.

Settings

Value	Description
true	Generate Lauterbach ORTI
false	No Lauterbach ORTI (default)

4.2.7 ORTI/winIDEA

XML name Orti21winIDEA

Description

Select ORTI generation for winIDEA debugger.

Settings

Value	Description
true	Generate winIDEA ORTI
false	No winIDEA 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	Enable Stack Fill
false	Disable Stack Fill (default)

4.2.9 Support winIDEA Analyzer

XML name winIDEAAnalyzer

Description

Adds support for the winIDEA profiler to track ORTI items.

Settings

Value	Description
true	Support Analyzer
false	No support for Analyzer (default)

4.2.10 ORTI/SMP

XML name OrtiSMPProposal

Description

Emit ORTI according to the ORTI_SMP_Proposal_v4.pdf (multicore only).

Settings

Value	Description
true	Generate ORTI/SMP
false	Use RTA-OS legacy (default)

4.2.11 CrossCore SRC0

XML name CrossCoreSRC0

Description

Optionally specify the SRC assigned to the cross-core interrupt for core 0. e.g. SRC_GPSR02. A free register will be selected automatically if one is not specified. Multicore only.

4.2.12 CrossCore SRC1

XML name CrossCoreSRC1

Description

Optionally specify the SRC assigned to the cross-core interrupt for core 1. e.g. SRC_GPSR02. A free register will be selected automatically if one is not specified. Multicore only.

4.2.13 CrossCore SRC2

XML name CrossCoreSRC2

Description

Optionally specify the SRC assigned to the cross-core interrupt for core 2. e.g. SRC_GPSR02. A free register will be selected automatically if one is not specified. Multicore only.

4.2.14 CrossCore SRC3

XML name CrossCoreSRC3

Description

Optionally specify the SRC assigned to the cross-core interrupt for core 3. e.g. SRC_GPSR02. A free register will be selected automatically if one is not specified. Multicore only.

4.2.15 CrossCore SRC4

XML name CrossCoreSRC4

Description

Optionally specify the SRC assigned to the cross-core interrupt for core 4. e.g. SRC_GPSR02. A free register will be selected automatically if one is not specified. Multicore only.

4.2.16 CrossCore SRC5

XML name CrossCoreSRC5

Description

Optionally specify the SRC assigned to the cross-core interrupt for core 5. e.g. SRC_GPSR02. A free register will be selected automatically if one is not specified. Multicore only.

4.2.17 Block default interrupt

XML name block_default_interrupt

Description

This option is provided for compatibility reasons only. SRC registers for unused registers are left in their power-on disabled state which blocks spurious interrupts and has no impact on the generated code.

Settings

Value	Description
true	Ignored
false	Ignored (default)

4.2.18 User Mode

XML name UserMode

Description

Specify the PSW.IO user mode setting used for untrusted code. You may need to set up SYSCON register in User-1 mode.

Settings

Value	Description
User-0	PSW-IO 00: no peripheral access (default)
User-1	PSW-IO 01: regular peripheral access

4.2.19 Trusted with protection PRS

XML name TWPprs

Description

Specify the PSW.PRS setting used for trusted-with-protection code.

Settings

Value	Description
1	PSW-PRS 01
2	PSW-PRS 10 (default)
3	PSW-PRS 11

4.2.20 Guard supervisor access

XML name guard_supervisor_access

Description

This option adds extra security checks to the System Call trap handler to validate that the caller is the OS rather than some application code.

Settings

Value	Description
true	Extra checks
false	No checks (default)

4.2.21 Interrupt vector matches priority

XML name IPL_matches_vector

Description

RTA-OS will normally pack interrupts to minimize the size of the interrupt vector table. This reduces the memory size and reduces the interrupt entry time. Some customers prefer to use the interrupt priority to determine the interrupt's SRC.SRPN value.

Settings

Value	Description
true	Try to match SRPN and priority
false	Pack vectors (default)

4.2.22 OS Locks disable Cat1

XML name OSLockDisableAll

Description

Specify whether all interrupts are disabled while internal OS spinlocks are held. This may reduce cross-core blocking. It should normally be selected if OS option 'Add Spinlock APIs for CAT1 ISRs' is active. This does not affect spinlocks accessed using the GetSpinlock or TryToGetSpinlock APIs

Settings

Value	Description
true	Disable all interrupts
false	Do not disable interrupts (default)

4.2.23 Enable stack repositioning

XML name AlignUntrustedStacks

Description

Use to support realignment of the stack for untrusted code when there are MPU protection region granularity issues. Refer to the documentation for Os_Cbk_SetMemoryAccess

Settings

Value	Description
true	Support repositioning
false	Normal behavior (default)

4.2.24 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)

4.2.25 Link Type

XML name OsLinkerModel

Description

Select the type of map used in linker samples.

Settings

Value	Description
Standalone	Code in internal flash, data in internal RAM (default)
IntRAM	Code/data in internal RAM
ExtRAM	Code/data in external RAM

4.2.26 Small data threshold

XML name small_data_value

Description

Sets the value used for -sda=n when compiling. Defaults to 0. Refer to compiler documentation for full details.

4.2.27 Short enums

XML name short_enums

Description

Use the compiler option –short_enum to force enum types the smallest data type possible.

Settings

Value	Description
true	Short enums (default)
false	Standard enums

4.2.28 FP Instructions

XML name fsingle

Description

Use the compiler option -fsingle use floating point instructions where possible.

Settings

Value	Description
true	Use -fsingle (default)
false	Do not use -fsingle

4.2.29 Far jumps

XML name far_jumps

Description

Select far jumps if interrupt/trap handlers are located at an address that is more than '24-bits' away from the vector tables. Near jumps may be very slightly faster.

Settings

Value	Description
true	Far jumps (default)
false	Near jumps

4.2.30 Max Optimizations

XML name Omax

Description

Use the compiler option -Omax. Refer to compiler documentation for full details.

Settings

Value	Description
true	Use -Omax (default)
false	Do not use -Omax

4.2.31 Optimization Type

XML name opt_type

Description

Specify the optimization type. Refer to compiler documentation for full details.

Settings

Value	Description
speed	-Ospeed (default)
size	-Osize
general	-Ogeneral
none	-Onone

4.2.32 Customer Option Set 1

XML name option_set1

Description

Selects a different set of compiler options. Requested by a customer for a specific project and not supported elsewhere. The compiler options are: -fsingle, -Ospeed, -Omax, -sda=0, –short_enum, –no_commons, –unsigned_fields, -c99, –gnu_asm, -delete, -ignore_debug_references, -preprocess_assembly_files, -passsource, -g, -dwarf2, -prototype_errors, -Wundef, -Wimplicit-int, -diag_warning=2003.

Settings

Value	Description
true	Enables option set 1
false	Use standard options (default)

4.2.33 Customer Option Set 2

XML name option_set2

Description

Selects a different set of compiler options. Requested by a customer for a specific project and not supported elsewhere. Primarily for use with v2016 compiler onwards. Some options are not supported with the v2015.1.7 compiler, and there may be linker issues with the v2015.1.7 toolchain with pre 1.6 core architectures. The compiler options are: -cpu=tc1v16p, -fsingle, -farcallpatch, -g, -Ospeed, -Omax, -dwarf2, -nostartfiles, -delete, -ignore_debug_references, -lnk=-delete, -lnk=-ignore_debug_references, -split_data_sections_by_alignment, -individual_data_sections, -individual_pragma_data_sections, -individual_function_sections, -individual_pragma_function_sections, -individual_section_name_extra_dot, –prototype_errors, -Wundef, -Wimplicit-int, –diag_warning=2003, -globalcheck=normal, -map, -lnk=-mapfile_type=2, -Man, -Ml, -Mx, -Mu, -MD, -sda=0, –no_commons, -c99, –gnu_asm, -preprocess_assembly_files, -passsource, -discard_zero_initializers

Settings

Value	Description
true	Enables option set 2
false	Use standard options (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 TriCore/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_GetTrapInfo

Return information about the most recent unhandled trap.

Syntax

```
FUNC(StatusType, OS_CODE)Os_GetTrapInfo(
    OsTrapInfoRefType Info
)
```

Parameters

Name	Type	Mode	Description
info	OsTrapInfoRefType	in	Pointer to the OsTrapInfoType into which the information will be copied. OsTrapInfoType contains the trap class (.Class), identification number (.TIN) and return address (.ReturnAddress).

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ILLEGAL_ADDRESS	extended	Info is an address that is not legal for writing by the current OS-Application (only when there are untrusted OS-Applications).

Description

When an unhandled processor Trap is detected, RTA-OS records the trap class, identification number and return address. It stores this information independently for each core, and then calls the ProtectionHook (when configured).

You can call Os_GetTrapInfo() from within ProtectionHook to get a copy of the most recent trap information for the calling core.

You should only call Os_GetTrapInfo() when the StatusType passed to ProtectionHook is E_OS_PROTECTION_MEMORY or E_OS_PROTECTION_EXCEPTION.

Note that Os_GetTrapInfo() can only return the information for the most recent unhandled trap for the given core.

Example

```
FUNC(ProtectionReturnType, {memclass}) ProtectionHook(StatusType
    FatalError) {
  OsTrapInfoType trap_info;
  switch (FatalError) {
    case E_OS_PROTECTION_MEMORY:
      /* A memory protection error has been detected */
      Os_GetTrapInfo(&trap_info);
      return MyUnexpectedTrapHandler(trap_info.Class,
          trap_info.TIN, trap_info.ReturnAddress);
    case E_OS_PROTECTION_EXCEPTION:
      /* Trap occurred */
      Os_GetTrapInfo(&trap_info);
      return MyUnexpectedTrapHandler(trap_info.Class,
          trap_info.TIN, trap_info.ReturnAddress);
    ...
  }
  return PRO_SHUTDOWN;
}
```

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

ProtectionHook
OsTrapInfoType
OsTrapInfoRefType

5.1.2 Os_InitializeInterruptTable

Initialize the interrupt vector table.

Syntax

```
FUNC(void, OS_CODE) Os_InitializeInterruptTable(void)
```

Description

RTA-OS creates interrupt vector table(s) based upon the interrupts that are configured. In the TriCore, the BIV register must be set to the address of the appropriate table. It should be called before StartOS().

The interrupt table must be initialized by calling this for each AUTOSAR core in a multicore application.

You do not normally need to call Os_InitializeInterruptTable() directly because it gets called by Os_InitializeVectorTable().

You must ensure that the BIV register is in a state where it can be modified when you make this call. You will need to be running in Supervisor with ENDINIT protection off.

Example

```
Os_InitializeInterruptTable();
```

See Also

StartOS
Os_InitializeVectorTable

5.1.3 Os_InitializeServiceRequests

Initializes the TriCore Service Request Registers according to the application configuration.

Syntax

```
FUNC(void, OS_CODE) Os_InitializeServiceRequests(void)
```

Description

It is crucial that the initialization of the TriCore Service Request Registers is done in accordance with the interrupts and priorities declared in the application configuration.

This function should be called to set the correct SRC values. You do not normally need to call this explicitly because it is automatically called from Os_InitializeInterruptTable() when it is called from core 0.

Note that the hardware priority values allocated to each interrupt source are not the same as the logical interrupt priority levels (IPLs) that are assigned to an interrupt in the configuration. In a single-core system, the priorities are compressed to reduce the vector table size and improve response times. In multi-core systems, there are additional constraints that require priorities across cores to be aligned and the correct interrupt steering values to be set.

RTA-OS emits OS_INIT_<srcname> macros that contain the correct SRC values for each configured interrupt. If really necessary, you can use these to set the SRC values directly instead of calling this function.

You must ensure that the SRC registers are in a state where they can be modified when you make this call. You will need to be running in Supervisor with ENDINIT protection off.

Example

```
Os_InitializeServiceRequests();
StartOS();
```

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

Os_InitializeVectorTable
StartOS
Os_InitializeInterruptTable

5.1.4 Os_InitializeTrapTable

Initialize the trap vector table.

Syntax

```
FUNC(void, OS_CODE) Os_InitializeTrapTable(void)
```

Description

RTA-OS creates trap vector table(s) based upon the traps that are configured. In the TriCore, the BTV register must be set to the address of the appropriate table. It should be called before StartOS().

The trap table must be initialized by calling this for each AUTOSAR core in a multicore application.

You do not normally need to call Os_InitializeTrapTable() directly because it gets called by Os_InitializeVectorTable().

You must ensure that the BTV register is in a state where it can be modified when you make this call. You will need to be running in Supervisor with ENDINIT protection off.

Example

```
Os_InitializeTrapTable();
```

See Also

StartOS
Os_InitializeVectorTable

5.1.5 Os_InitializeVectorTable

Initialize the interrupt and trap vector tables.

Syntax

```
void Os_InitializeVectorTable(void)
```

Description

RTA-OS creates interrupt table(s) and trap vector table(s) based upon the interrupts and traps that are configured. In the TriCore, the BIV and BTV registers must be set to their start addresses.

In addition, the Service Request Control Registers must be set up correctly so that they match the configuration that is declared for the project. In particular, the TOS and SRPN values must be correct. Note that the SRPN value does not necessarily match the priority assigned to an interrupt.

Os_InitializeVectorTable() performs all of these initializations for you. It should be called before StartOS().

If you only want to initialize the interrupt system then call Os_InitializeInterruptTable() instead of Os_InitializeVectorTable.

If you only want to initialize the SRC registers then call Os_InitializeServiceRequests() instead of Os_InitializeVectorTable / Os_InitializeInterruptTable.

If you only want to initialize the trap system then call Os_InitializeTrapTable() instead of Os_InitializeVectorTable.

However it is recommended that you always use Os_InitializeVectorTable().

In a multicore application, each core must perform these initializations.

You must ensure that the BIV, BTV and SRC registers are in a state where they can be modified when you make these calls. You will need to be running in Supervisor with ENDINIT protection off.

Example

```
Os_InitializeVectorTable();
```

See Also

StartOS
Os_InitializeTrapTable
Os_InitializeVectorTable
Os_InitializeServiceRequests

5.1.6 Os_StartCoreGate

Control core startup.

Syntax

```
FUNC(void, OS_CODE) Os_StartCoreGate(void)
```

Description

In a multi-core AUTOSAR application it is necessary for the master core to control the start-up behavior of the slave cores. Ideally the slave cores should stay in reset until Os_Cbk_StartCore gets called to release them.

Sometimes this can not be enforced (for example a debugger may not support this). For this reason, the OS provides the Os_StartCoreGate() API that should be placed at the start of 'main'.

If a slave core is released too early, this API will cause it to spin waiting until its Os_Cbk_StartCore has been called.

In normal usage, the OS_MAIN macro hides the call to Os_StartCoreGate. If you choose not to use OS_MAIN, then you should call Os_StartCoreGate explicitly if slave cores cannot be held in reset.

Example

```
OS_MAIN() {
  /* The OS_MAIN macro implicitly calls Os_StartCoreGate */
  ...
}

or

int main(void) {
  Os_StartCoreGate();
  ...
}
```

See Also

Os_Cbk_StartCore

5.2 Callbacks

5.2.1 Os_Cbk_StartCore

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

Syntax

```
FUNC(StatusType, {memclass})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 multi-core application, the StartCore and 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 and causing it to enter OS_MAIN.

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.

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

Example

```
FUNC(StatusType, {memclass}) Os_Cbk_StartCore(uint16 CoreID) {
  SET_CORE_RSTVEC(CoreID);
  RELEASE_CORE(CoreID);
}
```

Required when

Required for non master cores that will be started.

See Also

StartCore
StartNonAutosarCore
StartOS

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.

Example

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

5.3.2 CAT1_TRAP

Macro that should be used to implement a trap handler. If you want to use your own trap handler instead of the OS supplied versions, you must declare it in the project configuration as if it were a category 1 ISR.

Example

```
CAT1_TRAP(MyTrapHandler) {...}
```

5.3.3 OS_CORE_isrname

This macro contains the core (0,1...) that the named interrupt runs on. This is only emitted for multicore applications

Example

```
if (OS_CORE_ID_MASTER == OS_CORE_timer_interrupt)...
```

5.3.4 OS_INIT_srcname

This macro contains initialization values for the named SRC register. This is only emitted for multicore applications

5.3.5 OS_VEC_isrname

This macro contains the vector number (1-255) that is assigned to the named interrupt

Example

```
MyVectors[OS_CORE_timer_interrupt][OS_VEC_timer_interrupt] =
    my_timer_interrupt_handler;
```

5.3.6 Os_DisableAllConfiguredInterrupts

The Os_DisableAllConfiguredInterrupts macro will disable all configured SRC interrupts by adjusting the SRC register settings. You will need to #include the file "Os_DisableInterrupts.h" if you want to use this macro. It may not be used by untrusted code.

Example

```
Os_DisableAllConfiguredInterrupts()
Os_Enable_Millisecond()
```

5.3.7 Os_Disable_x

The Os_Disable_x macro will disable the named interrupt by adjusting its SRC register settings. It is normally paired with a call to Os_Enable_x. The macro can be called using either the SRC name or the RTA-OS configured vector name. In the example, this is Os_Disable_STM_SRC0() and Os_Disable_Millisecond() respectively. You will need to #include the file "Os_DisableInterrupts.h" if you want to use these macros. They may not be used by untrusted code.

Example

```
Os_Disable_STM_SRC0()
Os_Disable_Millisecond()
```

5.3.8 Os_EnableAllConfiguredInterrupts

The Os_EnableAllConfiguredInterrupts macro will enable all configured SRC interrupts by adjusting the SRC register settings. You will need to #include the file "Os_DisableInterrupts.h" if you want to use this macro. It may not be used by untrusted code.

Example

```
Os_DisableAllConfiguredInterrupts()
...
Os_EnableAllConfiguredInterrupts()
```

5.3.9 Os_Enable_x

The Os_Enable_x macro will re-enable the named interrupt at the priority it was configured with by adjusting its SRC register settings. It is normally paired with a call to Os_Disable_x. The macro can be called using either the INTC vector name or the RTA-OS configured vector name. In the example, this is Os_Enable_STM_SRC0() and Os_Enable_Millisecond() respectively. You will need to #include the file "Os_DisableInterrupts.h" if you want to use these macros. They may not be used by untrusted code.

Example

```
Os_Enable_STM_SRC0()
Os_Enable_Millisecond()
```

5.3.10 Os_IntChannel_x

The Os_IntChannel_x macro returns the address of the SRC register that is associated with the named interrupt. You can use this, for example, to trigger the interrupt through software.

Example

```
  *Os_IntChannel_Millisecond = *Os_IntChannel_Millisecond +
     SRC_TRIGGER_BIT;
```

5.4 Type Definitions

5.4.1 OsTrapInfoRefType

A pointer to an object of OsTrapInfoType. OsTrapInfoType contains the trap class (.Class), identification number (.TIN) and return address (.ReturnAddress) describing a trap.

Example

```
OsTrapInfoRefType trap_info_ref = &trap_info;
Os_GetTrapInfo(trap_info_ref);
```

5.4.2 OsTrapInfoType

Structure used by the Os_GetTrapInfo() API to return information about unhandled traps.

5.4.3 Os_StackSizeType

A structure containing 'Os_StackTraceType sp' to represent a size (in bytes) on the regular stack (A10) and 'Os_StackTraceType ctx' to represent a size (in bytes) on the CSA list.

Example

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

5.4.4 Os_StackTraceType

An unsigned type used to represent values on the regular stack and the CSAs.

5.4.5 Os_StackValueType

A structure containing 'Os_StackTraceType sp' to represent the position of the regular stack (A10) and 'Os_StackTraceType ctx' to represent the position of the CSA list.

Example

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

6 Toolchain

This chapter contains important details about RTA-OS and the GHS 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 v2018.1.5

Release tests are performed on this version.

Tested on v2018.1.5 Release Date Thu Apr 19 20:12:43 PDT 2018

6.1.2 v2017.1.5

Release tests are performed on this version.

Tested on v2017.1.5 Release Date Mon Apr 10 22:26:30 PDT 2017

6.1.3 v2015.1.7

Release tests are performed on this version.

Tested on v2015.1.7 Release Date Mon Nov 02 12:26:56 PST 2015

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 cctri.exe
Version v2018.1.5 Release Date Thu Apr 19 23:03:11 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.

`-fsingle` Use floating-point instructions (configurable via target option 'FP Instructions')

`-Ospeed` Optimize for speed (configurable via target option 'Optimization Type')

`-Omax` Aggressive optimization (configurable via target option 'Max Optimizations')

`-sda=0` Set SDA threshold (configurable via target option 'Small data threshold')

`--short_enum` Store enumerations in the smallest possible type (configurable via target option 'Short enums')

`--no_commons` Uninitialized global variables are unique

`--unsigned_fields` Bitfields are unsigned

`-c99 -gcc` ISO C99. GNU Mode

`-preprocess_assembly_files` Assember files are preprocessed

`-g` Support debugging

-dwarf2 Dwarf2 debugging format

--prototype_errors Raise error if prototypes are missing

-Wundef Warn for undefined symbols in preprocessor expressions

-Wimplicit-int Warn if function return type is not declared

--diag_warning=2003 Compiler message 2003 is warning

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.

-cpu=tc1v16 Generate code for target processor (variant-specific)

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

6.2.2 Assembler

Name	cctri.exe
Version	v2018.1.5 Release Date Thu Apr 19 23:02:03 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 23:02:02 PDT 2018

6.2.4 Linker

Name elxr.exe
Version v2018.1.5 Release Date Thu Apr 19 23:02:01 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.

`-delete` Remove functions that are not referenced

`-ignore_debug_references` Ignores relocations from DWARF debug sections when using -delete

`-globalcheck=normal` Extra link-time warnings

-map Create map file

-mapfile_type=2 Detailed map format

-Man Alphabetic sort map file

-Ml Add locals to map file

-Mx Cross references in map file

-Mu Undefined symbols in map file

-keepmap Keep map file if link fails

-Manx Map file options Alpha, Numeric, X-reference

-Qn No linker ident string

T:\ghs\comp_201815\lib\tri16\libstartup.a Use startup library

T:\ghs\comp_201815\lib\tri16\libsys.a Use sys library (for startup)

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 the kernel

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 10654 or later

Notes

Supports .elf files and ORTI files.

Notes on using ORTI with the debugger

When ORTI information is enabled, extra code is added to the CAT1_ISR macro to support tracking of Category 1 interrupts by the debugger.

The 'ORTI Stack Fill' target option is provided to extend the ORTI support to allow evaluation of unused stack space. The ORTI information gets extended to include information about the base address, size and fill pattern for the A10 stack.

The stack information is read from constants that you must create and initialize with appropriate values. For the example linker file that ships with RTA-OS, you would use the following code (for core 0):

```
const uint32 OS_STACK0_BASE = (uint32)&__SP_BASE0;
const uint32 OS_STACK0_SIZE = (uint32)&__SP_LEN0;
```

Other cores follow the same pattern.

You must also specify the stack fill pattern in a 32 bit constant `OS_STACK_FILL`.

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

The stack must be initialized with this fill pattern before starting the OS. You can do this in the C start-up code 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: Infineon
Device: TriCore

The following variants of the TriCore are supported:

- Generic131 (Any 1.3.1 core)
- Generic16 (Any 1.6.0 core)
- Generic161 (Any 1.6.1 core)
- TC1387
- TC1724
- TC1728
- TC1736
- TC1767
- TC1784
- TC1793
- TC1797
- TC1798
- TC21x
- TC22x
- TC23x
- TC23xADAS
- TC265D (A-Step)
- TC26x (A-Step)
- TC26xB (B-Step)
- TC27x (B-Step)
- TC27xA (A-Step)
- TC27xB (B-Step)

- TC27xC (C-Step)
- TC27xD (D-Step)
- TC298TP (A-Step)
- TC299TP (A-Step)
- TC29x (A-Step)
- TC29xB (B-Step)
- TC35x
- TC37x
- TC38x
- TC39x (A-Step)
- TC39xB (B-Step)

If you require support for a variant of TriCore 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
BIV	The Base Interrupt Vector has to be set to the start of CPU Interrupt vector table. This is done by calling Os_InitializeVectorTable() (or Os_InitializeInterruptTable()). This must be done for each OS core in a multicore application.
BTV	The Base Trap Vector has to be set to the start of the CPU Trap vector table. This is done by calling Os_InitializeVectorTable() (or Os_InitializeTrapTable()). This must be done for each OS core in a multicore application.
FCX,LCX	The Free Context List must be initialized to a contiguous block of context save areas (CSAs). Each block must link to its immediate neighbor such that FCX gets smaller as CSAs are allocated. (This is the default behavior of the compiler startup code.)
PSW	IO must be set to Supervisor Mode and IS must be set to 1.
SRR / SRC	The Service request registers for each interrupt source must be initialized correctly. This is done by calling Os_InitializeServiceRequests() (or Os_InitializeVectorTable(), which calls it for you). Note that the hardware priority values allocated to each interrupt source are not the same as the logical interrupt priority levels (IPLs) that are assigned to an interrupt in the configuration. In a single-core system, the priorities are compressed to reduce the vector table size and improve response times. In multi-core systems, there are additional constraints that require priorities across cores to be aligned and the correct interrupt steering values to be set. If you have to, you can use and the correct interrupt steering values to be set. If you have to, you can use the OS_INIT_srcname macros to set the values directly. You can override this behavior by using the 'Interrupt vector matches priority' target option.

7.2.2 Modification

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

Register	Notes
BIV	The Base Interrupt Vector.
BTV	The Base Trap Vector.
FCX	The Free CSA List Head Pointer.
Interrupt Control Registers	This includes SRC priority and TOS fields.
LCX	The Free CSA List Limit Pointer.
PCX	The Previous CSA List Head Pointer.
PCXI	The Previous Context Information Register.
PSW	After StartOS(), only the User Status bits may be written to.

7.3 Interrupts

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

7.3.1 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 TriCore:

IPL	ICR	Description
0	IE=1, CCPN=0	User (task) level
1-255	IE=1, CCPN=1-255	Category 1 and 2 level
256	-	Traps

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

7.3.2 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 TriCore. A ✓ indicates that the mapping is permitted and a ✗ indicates that it is not permitted:

Address	Category 1	Category 2
A named SRC register	✓	✓
A named trap	✓	✗

7.3.3 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
.text.Os_interrupt_handlers	Contains Os_InterruptVectorTable[n], where [n] is 0, 1 or 2 in multicore targets. The table must be aligned such that it fits within a memory range where its upper 20 address bits are the same. You should call the function Os_InitializeVectorTable before StartOS() to set register BIV to the start of the table. It should be called for each AUTOSAR core in a multicore application. When you tell the OS not to generate the interrupt vectors, it will put its interrupt handler code in this section. The code that you supply to handle interrupts can then jump to this code to implement the default OS behavior. (See the description of user generated vectors).
.text.Os_trap_handlers	Contains the Trap vector table. You should call the function Os_InitializeVectorTable before StartOS() to set register BTV to the start of the table. It should be called for each AUTOSAR core in a multicore application. If you choose to reassign BTV to point to a different set of traps, be aware you will not be able to use untrusted OS Applications because the OS expects to use the system call trap to switch modes. In addition, the OS will not be able to detect memory access violations. When you tell the OS not to generate the trap vectors, it will put its trap handler code in this section. The code that you supply to handle traps can then jump to this code to implement the default OS behavior. (See the description of user generated vectors).

When ’Suppress Vector Table Generation’ is configured to TRUE, no vector tables get generated. You are responsible for providing the vector tables and initializing the BIV/BTV registers. RTA-OS still provides the interrupt and trap handler code for you to bind to your handlers, but it is not linked to the vector tables. Note that this is the same code that would normally be placed directly in the interrupt/trap tables, so must be entered with the same conditions that were in effect when the vector was taken. In particular, the stack must be the same because the handler code expects to perform the return from interrupt/trap. The handler code uses the bisr instruction to ensure that the interrupts run at the correct priority. In the simplest case your code will simply jump to the appropriate interrupt or trap handler. There is a naming convention that helps you to do this:

Interrupt handler naming:

Each interrupt handler is given 2 names by which it can be accessed: Os_Interrupt_nnn and Os_Interrupt_<name>. ’nnn’ represents the vector number 001 to 255. <name> is the name of your ISR. You can choose which label to use. (In a multicore application the first of these becomes Os_Interrupt_c_nnn, where c is the core number 0,1..) It is critically important that the handlers get associated with the correct vector. You may find the macros OS_VEC_<name> and OS_CORE_<name> that are in Os_Cfg.h helpful if you want to auto-generate the vector tables.

Trap handler naming:

Each trap handler has a name appropriate to its responsibility. The names are Os_memory_trap, Os_protection_trap, Os_instruction_trap, Os_context_trap, Os_bus_trap, Os_assert_trap, Os_syscall_trap and Os_nmi_trap.

Cat1 ISR Implementation:

The CAT1_ISR macro must be used to implement Category 1 ISRs. It ensures that the interrupt runs at the correct priority and saves / restores the correct registers.

Trap Implementation:

The CAT1_TRAP macro must be used to implement Category 1 Traps. It ensures that the trap runs at the correct priority and saves / restores the correct registers.

Multicore Issues:

Each core that is running the AUTOSAR OS needs to use a software interrupt for cross-core communication. RTA-OS will choose unallocated SRC registers

for this purpose, or you can configure specific registers. Macros in Os_Cfg.h can be used to determine which registers are being used.

7.3.4 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 */
}
```

You can configure your own trap handlers (declared as Category 1 ISRs) that will override the OS-provided handlers. However the system call trap should only be overridden if you do not have untrusted code.

You should use the CAT1_TRAP macro to implement your handlers because the OS just jumps to your handler and it must therefore be implemented as a trap, not as an interrupt.

Alternatively, if you name your trap handler b_(name) then RTA-OS will branch directly to b_(name) without any modification to the CSAs or registers. You are entirely responsible for the trap handing code in this case.

If you do provide your own handler, you can still jump to the default OS handler code for the trap, using the naming rules described for supplying your own interrupt vector table.

7.3.5 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.

7.3.6 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 7.3.4 for further details).

7.4 Memory Model

The following memory models are supported:

Model	Description
far data	Default: nothing allocated to near data
near data	Target configuration parameters can be used to allocate data small data areas.

7.5 Processor Modes

RTA-OS can run in the following processor modes:

Mode	Notes
Supervisor	All OS and "trusted" code runs in supervisor mode.
User	All "untrusted" code runs in user mode.

Trusted code, including the OS, runs with PSW.PRS = 00 (Protection register set 0) and PSW.IO = 10. Untrusted code runs with PSW.PRS = 01 (Protection register set 1) and PSW.IO = 00 or 01.

7.6 Stack Handling

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

RTA-OS manages both the locals stack (via register A10) and the CSA list. CSAs are used in such a way that they behave as if they were a normal stack, so worst-case stack usage can be calculated for the CSA area in the usual way.

8 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 TriCore/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

8.1 Measurement Environment

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

Device	TC27xC on TC2x5 V1.0
CPU Clock Speed	80.0MHz
Stopwatch Speed	80.0MHz
Code	Internal RAM
Data	Internal RAM

8.2 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.

8.2.1 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	70
OS-Application	0	0
PeripheralArea	0	0
Resource	8	4
ScheduleTable	16	12
Task	20	0

8.2.2 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	4	12
Cat 2 ISR	12	0
Core Overheads (each OS core)	0	68
Core Overheads (each processor core)	20	28
Counter	28	4
CounterCallback	4	0
ExpiryPoint	3.5	0
OS Overheads (max)	0	6
OS-Application	2.66	0
PeripheralArea	0	0
Resource	12	4
ScheduleTable	16	12
Task	32	0

8.3 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.

8.4 Library Module Sizes

8.4.1 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	.bss	.data	.rodata	.text	.text.Os	.text.Os_interrupt_handlers	.text.Os_trap_handlers
ActivateTask				128			
AdvanceCounter				32			
CallTrustedFunction				32			
CancelAlarm				96			
ChainTask				128			
CheckISRMemoryAccess				32			
CheckObjectAccess				96			
CheckObjectOwnership				96			
CheckTaskMemoryAccess				32			
ClearEvent				32			
ControlIdle	8			96			
DisableAllInterrupts	8			64			
DispatchTask				224			

Library Module	.bss	.data	.rodata	.text	.text.Os	.text.Os_interrupt_handlers	.text.Os_trap_handlers
ElapsedTime				224			
EnableAllInterrupts				64			
GetActiveApplicationMode				32			
GetAlarm				160			
GetAlarmBase				64			
GetApplicationID				64			
GetCounterValue				64			
GetCurrentApplicationID				64			
GetElapsedCounterValue				64			
GetEvent				32			
GetExecutionTime				32			
GetISRID				32			
GetIsrMaxExecutionTime				32			
GetIsrMaxStackUsage				64			
GetResource				96			
GetScheduleTableStatus				64			
GetStackSize				32			
GetStackUsage				64			
GetStackValue				32			
GetTaskID				32			
GetTaskMaxExecutionTime				32			
GetTaskMaxStackUsage				64			
GetTaskState				64			
GetVersionInfo				32			
Idle				32			
InShutdown				32			
IncrementCounter				32			
InterruptSource			4	288			
ModifyPeripheral				192			
NextScheduleTable				128			
Os_Cfg	559		776	320			
Os_Cfg_Counters			728	11712			
Os_Cfg_KL				64			

Library Module	.bss	.data	.rodata	.text	.text.Os	.text.Os_interrupt_handlers	.text.Os_trap_handlers
Os_GetCurrentIMask				32			
Os_GetCurrentTPL				32			
Os_GetTrapInfo				32			
Os_Interrupts						74	
Os_SrcInit				32			
Os_Stack				64			
Os_StartCores		16		224			
Os_TrapInit				32			
Os_TrapSupport	8			32			
Os_Traps							256
Os_VectorInit				32			
Os_Wrapper				160			
Os_longjmp_ext					60		
Os_setjmp					184		
ProtectionSupport				64			
ReadPeripheral				192			
ReleaseResource				96			
ResetIsrMaxExecutionTime				32			
ResetIsrMaxStackUsage				32			
ResetTaskMaxExecutionTime				32			
ResetTaskMaxStackUsage				32			
ResumeAllInterrupts				64			
ResumeOSInterrupts				64			
Schedule				96			
SetAbsAlarm				96			
SetEvent				32			
SetRelAlarm				160			
SetScheduleTableAsync				64			
ShutdownOS				96			
StackOverrunHook				32			
StartOS				160			
StartScheduleTableAbs				128			
StartScheduleTableRel				128			

Library Module	.bss	.data	.rodata	.text	.text.Os	.text.Os_interrupt_handlers	.text.Os_trap_handlers
StartScheduleTableSynchron				64			
StopScheduleTable				96			
SuspendAllInterrupts	8			64			
SuspendOSInterrupts	8			96			
SyncScheduleTable				64			
SyncScheduleTableRel				64			
TerminateTask				32			
ValidateCounter				64			
ValidateISR				32			
ValidateResource				64			
ValidateScheduleTable				64			
ValidateTask				64			
WaitEvent				32			
WritePeripheral				192			

8.4.2 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	.bss	.data	.rodata	.text	.text.Os	.text.Os_interrupt_handlers	.text.Os_trap_handlers
ActivateTask				256			
AdvanceCounter				32			
CallTrustedFunction				32			
CancelAlarm				160			
ChainTask				192			
CheckISRMemoryAccess				32			
CheckObjectAccess				192			
CheckObjectOwnership				128			
CheckTaskMemoryAccess				32			
ClearEvent				32			
ControlIdle	16			96			
CrossCore				64			
DisableAllInterrupts				64			
DispatchTask				448			
ElapsedTime				224			
EnableAllInterrupts				64			
GetActiveApplicationMode				32			
GetAlarm				160			
GetAlarmBase				64			
GetApplicationID				64			
GetCounterValue				64			
GetCurrentApplicationID				64			
GetElapsedCounterValue				64			
GetEvent				32			
GetExecutionTime				32			
GetISRID				32			
GetIsrMaxExecutionTime				32			
GetIsrMaxStackUsage				64			
GetNumberOfActivatedCores				32			
GetResource				96			
GetScheduleTableStatus				128			
GetSpinlock				32			

Library Module	.bss	.data	.rodata	.text	.text.Os	.text.Os_interrupt_handlers	.text.Os_trap_handlers
GetStackSize				32			
GetStackUsage				64			
GetStackValue				64			
GetTaskID				32			
GetTaskMaxExecutionTime				32			
GetTaskMaxStackUsage				64			
GetTaskState				96			
GetVersionInfo				32			
Idle				32			
InShutdown				32			
IncrementCounter				32			
InterruptSource			4	384			
ModifyPeripheral				192			
NextScheduleTable				192			
Os_Cfg	710		1308	448			
Os_Cfg_Counters			888	13984			
Os_Cfg_KL				96			
Os_CrossCore			16	256			
Os_GetCurrentIMask				32			
Os_GetCurrentTPL				96			
Os_GetTrapInfo				32			
Os_Interrupts						170	
Os_ScheduleQ				64			
Os_Spinlock				32			
Os_SrcInit				64			
Os_Stack				64			
Os_StartCores		16		224			
Os_TrapInit				32			
Os_TrapSupport				32			
Os_Traps							256
Os_VectorInit				64			
Os_Wrapper				160			
Os_longjmp_ext					60		

Library Module	.bss	.data	.rodata	.text	.text.Os	.text.Os_interrupt_handlers	.text.Os_trap_handlers
Os_setjmp					184		
ProtectionSupport				64			
ReadPeripheral				192			
ReleaseResource				96			
ReleaseSpinlock				32			
ResetIsrMaxExecutionTime				32			
ResetIsrMaxStackUsage				32			
ResetTaskMaxExecutionTime				32			
ResetTaskMaxStackUsage				32			
ResumeAllInterrupts				64			
ResumeOSInterrupts				64			
Schedule				96			
SetAbsAlarm				160			
SetEvent				32			
SetRelAlarm				256			
SetScheduleTableAsync				64			
ShutdownAllCores				64			
ShutdownOS				128			
StackOverrunHook				32			
StartCore				64			
StartNonAutosarCore				64			
StartOS				320			
StartScheduleTableAbs				192			
StartScheduleTableRel				192			
StartScheduleTableSynchron				64			
StopScheduleTable				160			
SuspendAllInterrupts				64			
SuspendOSInterrupts				96			
SyncScheduleTable				64			
SyncScheduleTableRel				64			
TerminateTask				64			
TryToGetSpinlock				32			
ValidateCounter				64			

Library Module	.bss	.data	.rodata	.text	.text.Os	.text.Os_interrupt_handlers	.text.Os_trap_handlers
ValidateISR				32			
ValidateResource				64			
ValidateScheduleTable				64			
ValidateTask				96			
WaitEvent				32			
WritePeripheral				192			

8.5 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 8.5.1 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.

8.5.1 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()`).

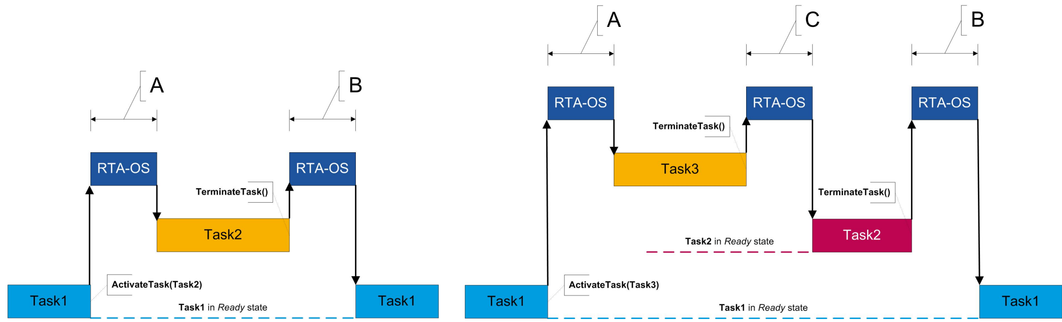
Interrupt latency is the time between an interrupt request being recognized by the target hardware and the execution of the first instruction of the user provided handler function:

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.

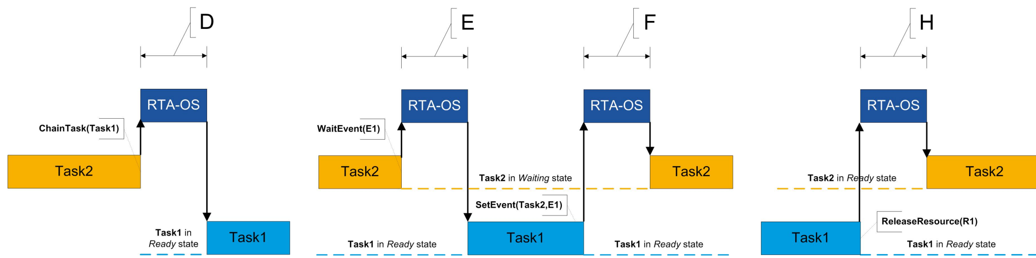
Figure 8.1 shows the measured context switch times for RTA-OS.

Switch	Key	CPU Cycles	Actual Time
Task activation	A	244	3.05us
Task termination with resume	B	122	1.53us
Task termination with switch to new task	C	162	2.02us
Chaining a task	D	294	3.67us
Waiting for an event resulting in transition to the WAITING state	E	1182	14.8us
Setting an event results in task switch	F	1446	18.1us
Non-preemptive task offers a preemption point (co-operative scheduling)	G	232	2.9us
Releasing a resource results in a task switch	H	212	2.65us
Entering a Category 2 ISR	I	92	1.15us
Exiting a Category 2 ISR and resuming the interrupted task	J	94	1.18us
Exiting a Category 2 ISR and switching to a new task	K	206	2.58us
Entering a Category 1 ISR	L	24	300ns



(a) Task activated. Termination resumes preempted task.

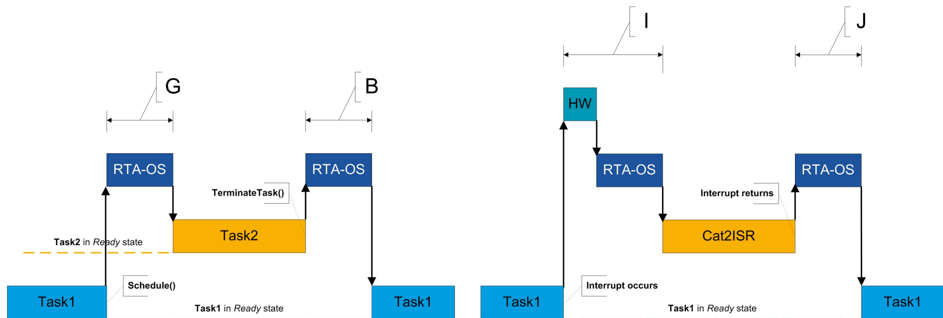
(b) Task activated. Termination switches into new task.



(c) Task chained.

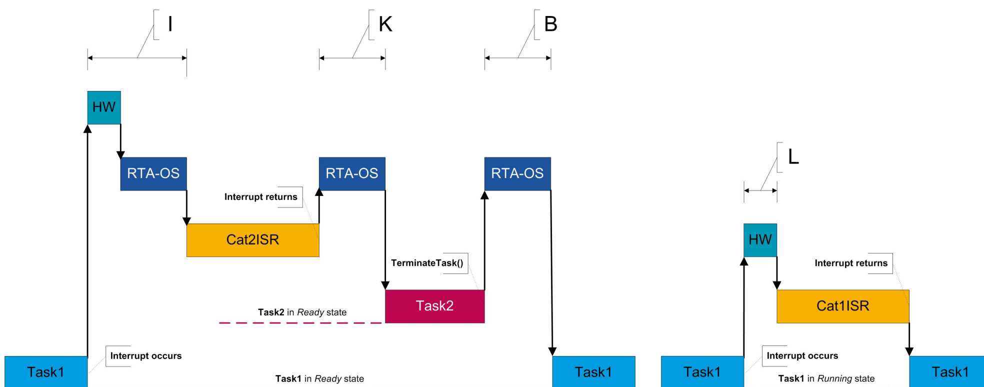
(d) Task waits. Task is resumed when event set.

(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 8.1: Context Switching

9 Finding Out More

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

***TriCore/GHS Release Note*.** This document provides information about the TriCore/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.

10 Contacting ETAS

10.1 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 10.2.2).

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

10.2 General Enquiries

10.2.1 ETAS Global Headquarters

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

10.2.2 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, 48
AUTOSAR OS includes
  Os.h, 34
  Os_Cfg.h, 34
  Os_MemMap.h, 34

C
CAT1_ISR, 42
CAT1_TRAP, 42
Compiler, 47
Compiler (v2015.1.7), 47
Compiler (v2017.1.5), 46
Compiler (v2018.1.5), 46
Compiler Versions, 46
Configuration
  Port-Specific Parameters, 24

D
Debugger, 50

E
ETAS License Manager, 13
  Installation, 13

F
Files, 34

H
Hardware
  Requirements, 11

I
Installation, 11
  Default Directory, 12
  Verification, 22
Interrupts, 55
  Category 1, 58
  Category 2, 58
  Default, 59
IPL, 55

L
Librarian, 49
Library
  Name of, 34
License, 13
  Borrowing, 18
  Concurrent, 15
  Grace Mode, 14
  Installation, 17
  Machine-named, 15
  Status, 18
  Troubleshooting, 19
  User-named, 15
Linker, 49

M
Memory Model, 59

O
Options, 47
Os_Cbk_StartCore, 41
OS_CORE_isrname, 43
Os_Disable_x, 43
Os_DisableAllConfiguredInterrupts, 43
Os_Enable_x, 44
Os_EnableAllConfiguredInterrupts, 44
Os_GetTrapInfo, 35
OS_INIT_srcname, 43
Os_InitializeInterruptTable, 37
Os_InitializeServiceRequests, 37
Os_InitializeTrapTable, 39
Os_InitializeVectorTable, 39
Os_IntChannel_x, 44
Os_StackSizeType, 45
Os_StackTraceType, 45
Os_StackValueType, 45
Os_StartCoreGate, 40
OS_VEC_isrname, 43
OsTrapInfoRefType, 44
OsTrapInfoType, 45

P
Parameters of Implementation, 24
Performance, 60
  Context Switching Times, 70

Library Module Sizes, 62
RAM and ROM, 60
Stack Usage, 61
Processor Modes, 59
Supervisor, 59
User, 59

R

Registers
BIV, 54, 55
BTV, 54, 55
FCX, 55
FCX,LCX, 54
Initialization, 53
Interrupt Control Registers, 55
LCX, 55
Non-modifiable, 54
PCX, 55
PCXI, 55
PSW, 54, 55
SRR / SRC, 54

S

Software
Requirements, 11
Stack, 59

T

Target, 52
Variants, 53
Toolchain, 46

V

Variants, 53
Vector Table
Base Address, 56