RTA-OS

ZynqUSA53/ARM Port Guide

Copyright

Document: 10721-PG-5.0.1 EN-10-2018

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 **7**
- 1.1 About You . . . 8
- 1.2 Document Conventions . . . 8
- 1.3 References . . . 9

2 Installing the RTA-OS Port Plug-in **10**
- 2.1 Preparing to Install . . . 10
  - 2.1.1 Hardware Requirements . . . 10
  - 2.1.2 Software Requirements . . . 10
- 2.2 Installation . . . 11
  - 2.2.1 Installation Directory . . . 11
- 2.3 Licensing . . . 12
  - 2.3.1 Installing the ETAS License Manager . . . 12
  - 2.3.2 Licenses . . . 13
  - 2.3.3 Installing a Concurrent License Server . . . 14
  - 2.3.4 Using the ETAS License Manager . . . 15
  - 2.3.5 Troubleshooting Licenses . . . 18

3 Verifying your Installation **21**
- 3.1 Checking the Port . . . 21
- 3.2 Running the Sample Applications . . . 21

4 Port Characteristics **23**
- 4.1 Parameters of Implementation . . . 23
- 4.2 Configuration Parameters . . . 23
  - 4.2.1 Stack used for C-startup . . . 23
  - 4.2.2 Stack used when idle . . . 24
  - 4.2.3 Stack overheads for ISR activation . . . 24
  - 4.2.4 Stack overheads for ECC tasks . . . 24
  - 4.2.5 Stack overheads for ISR . . . 25
  - 4.2.6 ORTI/Lauterbach . . . 25
  - 4.2.7 ORTI Stack Fill . . . 25
  - 4.2.8 Enable stack repositioning . . . 25
  - 4.2.9 Enable untrusted stack check . . . 26
  - 4.2.10 CrossCore SGI0 . . . 26
  - 4.2.11 CrossCore SGI1 . . . 26
  - 4.2.12 CrossCore SGI2 . . . 27
  - 4.2.13 CrossCore SGI3 . . . 27
  - 4.2.14 Set floating-point mode . . . 27
  - 4.2.15 Block default interrupt . . . 28
  - 4.2.16 GetAbortStack always . . . 28
  - 4.2.17 Set interrupt priority range . . . 28
  - 4.2.18 Read CoreID from GIC . . . 29
- 4.3 Generated Files . . . 29

5 Port-Specific API **31**
5.1 API Calls . . . 31
5.1.1 Os_InitializeGICGroup . . . 31
5.1.2 Os_InitializeVectorTable . . . 31
5.2 Callbacks . . . 32
5.2.1 Os_Cbk_GetAbortStack . . . 32
5.2.2 Os_Cbk_StartCore . . . 33
5.2.3 Os_Cbk_StopCore . . . 35
5.3 Macros . . . 36
5.3.1 CAT1_ISR . . . 36
5.3.2 Os_Clear_x . . . 37
5.3.3 Os_DisableAllConfiguredInterrupts_CPUx . . . 37
5.3.4 Os_Disable_x . . . 37
5.3.5 Os_EnableAllConfiguredInterrupts_CPUx . . . 38
5.3.6 Os_Enable_x . . . 38
5.3.7 Os_IntChannel_x . . . 38
5.3.8 Os_Set_Edge_Triggered_x . . . 39
5.3.9 Os_Set_Level_Sensitive_x . . . 39
5.4 Type Definitions . . . 39
5.4.1 Os_StackSizeType . . . 39
5.4.2 Os_StackValueType . . . 40

6 Toolchain **41**
6.1 Compiler Versions . . . 41
6.1.1 ARM DS-5 Ultimate Edition: ARM Compiler 6.6 . . . 41
6.2 Options used to generate this guide . . . 42
6.2.1 Compiler . . . 42
6.2.2 Assembler . . . 43
6.2.3 Librarian . . . 44
6.2.4 Linker . . . 44
6.2.5 Debugger . . . 45

7 Hardware **47**
7.1 Supported Devices 47
7.2 Register Usage 47
7.2.1 Initialization 47
7.2.2 Modification 48
7.3 Interrupts 49
7.3.1 Interrupt Priority Levels 49
7.3.2 Allocation of ISRs to Interrupt Vectors 50
7.3.3 Vector Table 51
7.3.4 Writing Category 1 Interrupt Handlers 52
7.3.5 Writing Category 2 Interrupt Handlers 52
7.3.6 Default Interrupt 53
7.4 Memory Model 53
7.5 Processor Modes 53
7.6 Stack Handling 54
7.7 Processor state when calling StartOS() 54

8 Performance **55**
8.1 Measurement Environment 55
8.2 RAM and ROM Usage for OS Objects 55
8.2.1 Single Core 56
8.2.2 Multi Core 56
8.3 Stack Usage 56
8.4 Library Module Sizes 57
8.4.1 Single Core 57
8.4.2 Multi Core 59
8.5 Execution Time 62
8.5.1 Context Switching Time 63

9 Finding Out More **66**

10 Contacting ETAS **67**
10.1 Technical Support 67
10.2 General Enquiries 67
10.2.1 ETAS Global Headquarters 67
10.2.2 ETAS Local Sales & Support Offices 67

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 ZynqUSA53/ARM port plug-in that customizes the RTA-OS development tools for the Xilinx Zynq UltraScale+ Cortex-A53 with the ARM_DS_5_V6 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 ZynqUSA53/ARM port and cover:

- how to install the ZynqUSA53/ARM port plug-in;
- how to configure ZynqUSA53/ARM-specific attributes;
- how to build an example application to check that the ZynqUSA53/ARM 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 Zynq UltraScale+ Cortex-A53, 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 ZynqUSA53/ARM 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 ZynqUSA53/ARM 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 ZynqUSA53/ARM 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 `ZynqUSA53ARM_5.0.1`. This contains everything to do with the port plug-in.

Each version of the port installs in its own directory - the trailing `_5.0.1` 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:

ZynqUSA53ARM.dll - the port plug-in that is used by **rtaosgen** and **rtaoscfg**.

RTA-OS ZynqUSA53ARM Port Guide.pdf - the documentation for the port (the document you are reading now).

RTA-OS ZynqUSA53ARM 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;

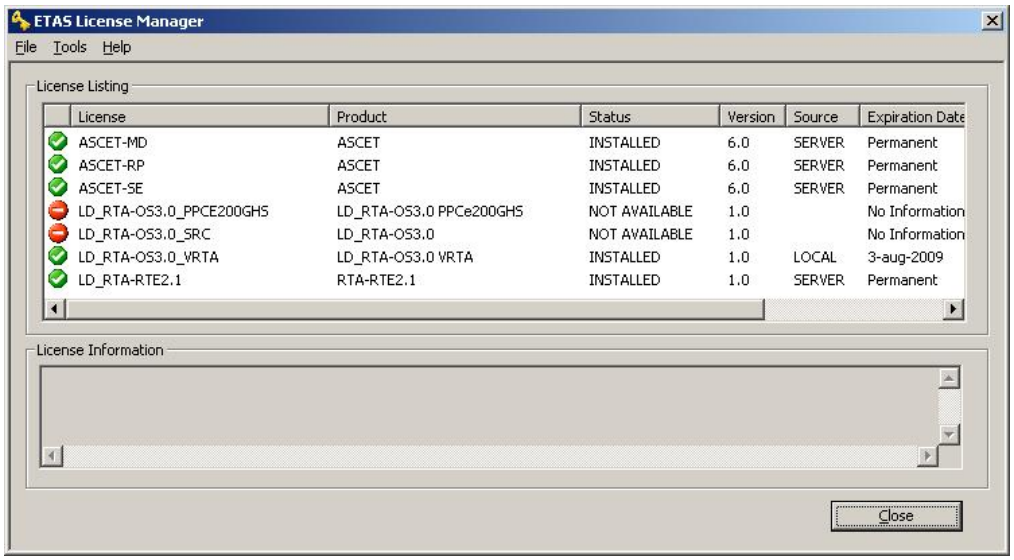


Figure 2.1: The ETAS License manager

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.)

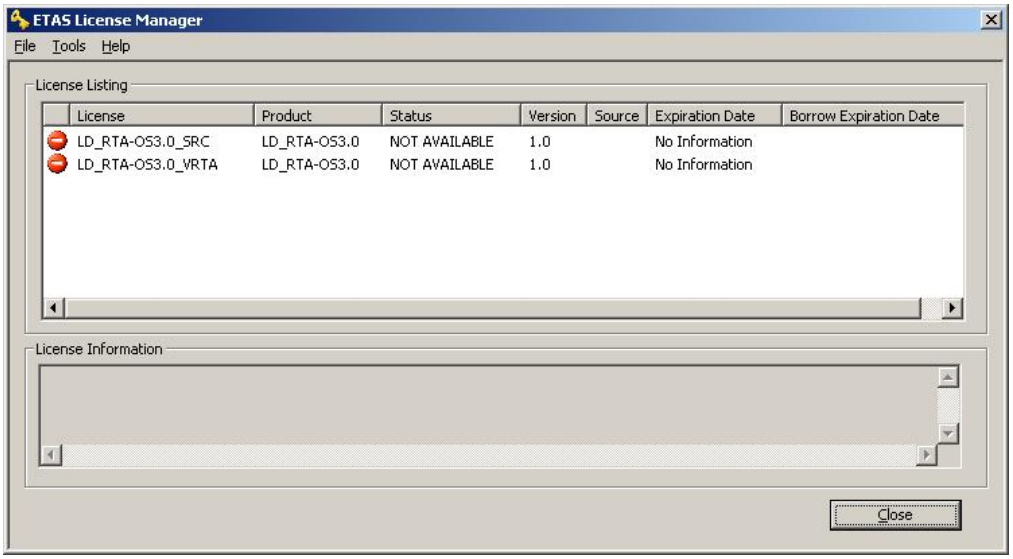


Figure 2.3: Unlicensed RTA-OS Installation

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

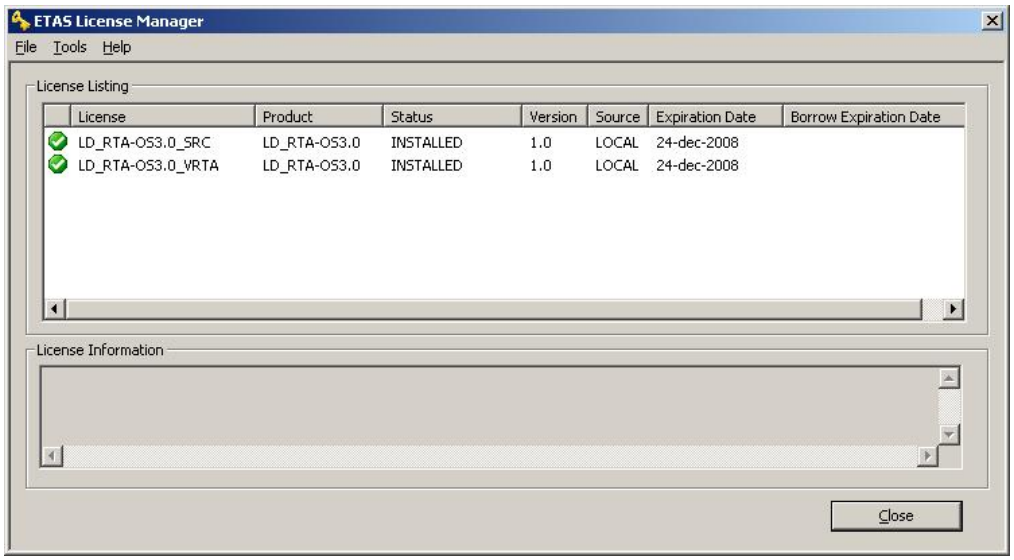


Figure 2.4: Licensed features for RTA-OS

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.

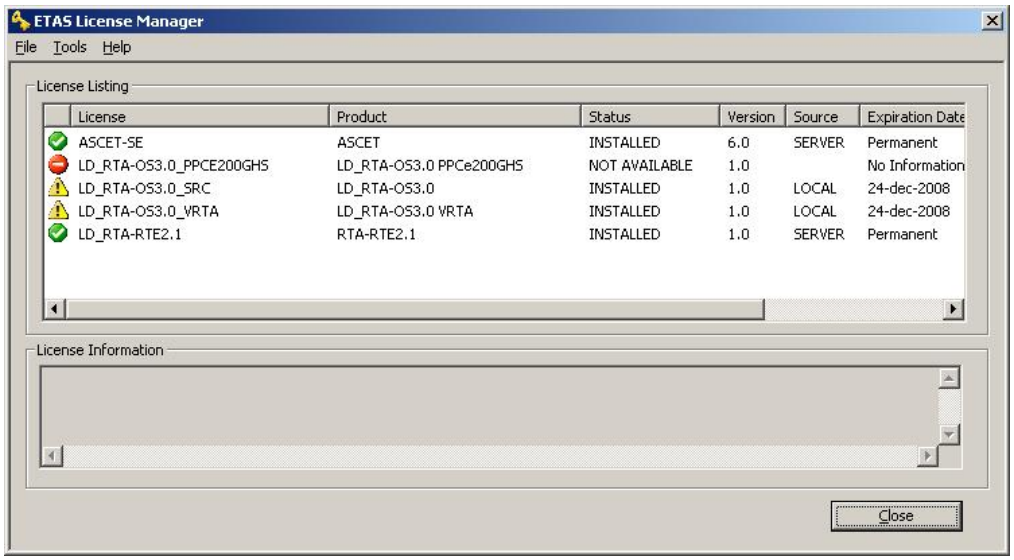


Figure 2.5: Licensed features that are due to expire

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 ZynqUSA53/ARM 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 ZynqUSA53/ARM 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 ZynqUSA53/ARM 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 ORTI
false	No ORTI (default)

4.2.7 ORTI Stack Fill

XML name OrtiStackFill

Description

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

Settings

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

4.2.8 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)

4.2.9 Enable untrusted stack check

XML name DistrustStacks

Description

Extra code can be placed in interrupt handlers to detect when untrusted code has an illegal stack pointer value. Also exception handlers run on a private stack (Refer to the documentation for Os_Cbk_GetAbortStack). This has a small performance overhead, so is made optional. When enabled 32 bytes of EL1 stack are required for each core.

Settings

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

4.2.10 CrossCore SGI0

XML name CrossCoreSGI0

Description

Optionally specify the SGI used for cross-core interrupts for core 0. A free SGI will be selected automatically if one is not specified. Used in multicore applications only.

Settings

Value	Description
CPU0 GIC0	Lowest SGI for CPU0
CPU0 GIC15	Highest SGI for CPU0 (all values between are valid)

4.2.11 CrossCore SGI1

XML name CrossCoreSGI1

Description

Optionally specify the SGI used for cross-core interrupts for core 1. A free SGI will be selected automatically if one is not specified. Used in multicore applications only.

Settings

Value	Description
CPU1 GIC0	Lowest SGI for CPU1
CPU1 GIC15	Highest SGI for CPU1 (all values between are valid)

4.2.12 CrossCore SGI2

XML name CrossCoreSGI2

Description

Optionally specify the SGI used for cross-core interrupts for core 2. A free SGI will be selected automatically if one is not specified. Used in multicore applications only.

Settings

Value	Description
CPU2 GIC0	Lowest SGI for CPU2
CPU2 GIC15	Highest SGI for CPU2 (all values between are valid)

4.2.13 CrossCore SGI3

XML name CrossCoreSGI3

Description

Optionally specify the SGI used for cross-core interrupts for core 3. A free SGI will be selected automatically if one is not specified. Used in multicore applications only.

Settings

Value	Description
CPU3 GIC0	Lowest SGI for CPU3
CPU3 GIC15	Highest SGI for CPU3 (all values between are valid)

4.2.14 Set floating-point mode

XML name FloatingPointMode

Description

Enable or disable hardware floating-point instructions. Used in the compiler command line option -mcpu to select the optional architectural feature for floating-point mode extensions to the instruction set.

Settings

Value	Description
nofp	Code does not use floating-point (default)
fp	Floating-point instructions generated

4.2.15 Block default interrupt

XML name block_default_interrupt

Description

Where a default interrupt is specified, it will normally execute if an unexpected (i.e. unused) interrupt triggers. This option changes this behavior by lowering the priority assigned to unused interrupt sources. When selected the default interrupt handler will be blocked by higher priority code.

Settings

Value	Description
true	Block the default interrupt
false	Allow the default interrupt handler to run if an unexpected interrupt fires (default)

4.2.16 GetAbortStack always

XML name always_call_GetAbortStack

Description

When an unexpected interrupt/exception or memory protection violation occurs always use the Os_Cbk_GetAbortStack() callback to set up a safe area of memory to use as a stack executing the ProtectionHook (please refer to the documentation for Os_Cbk_GetAbortStack).

Settings

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

4.2.17 Set interrupt priority range

XML name InterruptPriorityRange

Description

Select the range of priorities used by the Generic Interrupt Controller (GIC) for Software Generated Interrupts (SGIs), Private Peripheral Interrupts (PPIs), or Shared Peripheral Interrupts (SPIs). If applications are run in the non-secure state then the GIC only supports 16 priority levels and this option should be set accordingly.

Settings

Value	Description
16	4 bit GIC interrupt priority values
32	5 bit GIC interrupt priority values (default)

4.2.18 Read CoreID from GIC

XML name read_CoreID_from_GIC

Description

In untrusted code the RTA-OS by default uses an SVC call to identify the current CPU core from the MPIDR_EL1 register. Selecting this option causes RTA-OS to to identify the CPU core from the memory mapped register GICD_ITARGETSR0 instead. This improves the performance of untrusted code by removing the need for an expensive synchronous exception. Since this isn't necessary for trusted code the option is ignored if no untrusted code is present. If selected the MMU must allow untrusted code read access for the GIC registers.

Settings

Value	Description
true	Read from GIC
false	Read from MPIDR_EL1 (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 ZynqUSA53/ARM 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_InitializeGICGroup

Initialize the GIC GICD_IGROUPRx registers.

Syntax

```
void Os_InitializeGICGroup(void)
```

Description

Os_InitializeGICGroup() is available when the GIC will be used in the non-secure state. All GIC interrupts are configured as Group 1 (non-secure) interrupts

Os_InitializeGICGroup() should be called when the CPU is in the secure state before StartOS(). If the application is to be run in the secure state then Os_InitializeGICGroup() need not be called. In multi-core applications it should be called by all cores. It should be called even if 'Suppress Vector Table Generation' is set to TRUE. Os_InitializeVectorTable() should also be called to configure the other GIC registers.

Example

```
Os_InitializeGICGroup();
```

See Also

Os_InitializeVectorTable
StartOS

5.1.2 Os_InitializeVectorTable

Initialize the GIC GICD_IPRIORITYRx, GICD_ISENABLERx/GICD_ICENABLERx, OS_GICD_ITARGETSRx, and VBAR_EL1 registers.

Syntax

```
void Os_InitializeVectorTable(void)
```

Description

Os_InitializeVectorTable() initializes the GIC GICD_IPRIORITYRx, GICD_ISENABLERx/GICD_ICENABLERx, and OS_GICD_ITARGETSRx according to the requirements of the project configuration. The GIC priority is set to block Category 2 interrupts and allow Category 1 interrupts. FIQ and IRQ interrupts are also enabled

When the CPU is run in the secure state the GICD_IGROUPx registers are also initialized.

Os_InitializeVectorTable() should be called before StartOS(). In multi-core applications it should be called by all cores. It should be called even if 'Suppress Vector Table Generation' is set to TRUE.

Example

```
Os_InitializeVectorTable();
```

See Also

Os_InitializeGICGroup
StartOS

5.2 Callbacks

5.2.1 Os_Cbk_GetAbortStack

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

Syntax

```
FUNC(void *,OS_APPL_CODE) Os_Cbk_GetAbortStack(void)
```

Return Values

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

Description

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

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

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

A default implementation of Os_Cbk_GetAbortStack is supplied in the RTA-OS library that will place the abort stack at the starting stack location of the untrusted code.

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

Example

```
FUNC(void *,OS_APPL_CODE) Os_Cbk_GetAbortStack(void) {
  /* 64-bit alignment is needed for EABI. */
  static long long abortstack[40U] __attribute__((aligned (16)));
  return &abortstack[40U];
}
```

Required when

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

5.2.2 Os_Cbk_StartCore

Callback routine used to start a non-master core on a multi-core 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 or StartNonAutosarCore OS APIs have to be called prior to StartOS for each core that is to run. For this target port, these APIs make a call to Os_Cbk_StartCore() which is responsible for starting the specified core.

RTA-OS provides a default implementation of Os_Cbk_StartCore() that will be appropriate for most normal situations. To support multi-core applications the caches and SCU must be enabled in all cores otherwise data coherency cannot be maintained between cores. Os_Cbk_StartCore() does not get called for core 0,

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)
{
  StatusType ret = E_OS_ID;
  volatile uint32 *Os_CoreStartPtr = (volatile uint32
      *)&Os_StartCoreFlags;

  /* If not the primary core. */
  if (CoreID != 0U) {
    /* Issue a SEV instruction to signal an event to all PEs to
       wake-up a
     * secondary core. Events can be triggered by the debugger
       and semi-hosting
     * so additionally use a variable to signal that RTA-OS is
       waking-up a
     * specific core. This is reset once the core is awake. */
    *(Os_CoreStartPtr + CoreID) = 0xA0U;
    __asm volatile("dsb SY" : : : "memory", "cc");
    __asm volatile("sev" : : : "memory", "cc");

    /* Wait for the core to wake up by monitoring the variable
       value */
    while (*(Os_CoreStartPtr + CoreID) == 0xA0U) {
      __asm volatile("dsb SY" : : : "memory", "cc");
      __asm volatile("nop" : : : "memory", "cc");
```

```
        __asm volatile("nop" : : : "memory", "cc");
        __asm volatile("nop" : : : "memory", "cc");
        __asm volatile("nop" : : : "memory", "cc");
        __asm volatile("nop" : : : "memory", "cc");
        __asm volatile("nop" : : : "memory", "cc");
      }

      ret = E_OK;
    }

    return ret;
  }
```

Required when

Required for non-master cores that will be started.

See Also

StartCore
StartNonAutosarCore
StartOS

5.2.3 Os_Cbk_StopCore

Callback routine used to stop a non-master core on a multi-core variant.

Syntax

```
FUNC(StatusType, {memclass})Os_Cbk_StopCore(
    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

Each non-master (secondary) core should call Os_Cbk_StopCore() before entering OS_MAIN(). This will cause the core to pause. When the primary master core calls Os_Cbk_StartCore() the stopped core will be released and will continue execution. Os_Cbk_StopCore() should be used before the primary master core uses the StartCore() or StartOS() API calls.

RTA-OS provides a default implementation of Os_Cbk_StopCore() that will be appropriate for most normal situations. To support multi-core applications

the caches and SCU must be enabled in all cores otherwise data coherency is not maintained between cores. Os_Cbk_StopCore() should not get called for core 0 as this is the primary master core.

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

Example

```
FUNC(StatusType, {memclass}) Os_Cbk_StopCore(uint16 CoreID)
{
  volatile uint32 *Os_CoreStartPtr = (volatile uint32
      *)&Os_StartCoreFlags;

  /* If an expected core, shutdown until the value in
      Os_StartCoreFlags[] holds
   * a value set by Os_Cbk_StartCore() for this core */
  while (*(Os_CoreStartPtr + CoreID) != 0xA0U) {
    /* Issue a WFE instruction to put the secondary core into a
        low power state.
     * Note starting other cores, debug and semihosting events
         can cause a SEV
     * to wake-up the core. Check if it is valid to wake-up this
         core. */
    __asm volatile("wfe" : : : "memory", "cc")
  }

  /* Reset the variable to let core 0 know that the core has
      started. */
  *(Os_CoreStartPtr + CoreID) = 0x0U;
  __asm volatile("dsb SY" : : : "memory", "cc");

  return E_OK;
}
```

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 should only be used on Category 1 ISRs that are attached to the Generic

Interrupt Controller (GIC) not the Cortex CPU exceptions (See the later section on "Writing Category 1 Interrupt Handlers" for more information). This macro exists to help make your code portable between targets.

Example

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

5.3.2 Os_Clear_x

Use of the Os_Clear_x macro will clear the interrupt request bit of the GICD_ICPEND register for the named interrupt channel. The macro can be called using either the GIC channel number or the RTA-OS configured vector name. In the example, this is Os_Clear_GIC48() and Os_Clear_Millisecond() respectively. To use the Os_Clear_x macro the file Os_ConfigInterrupts.h must be included through the use of #include. The macro is provided so the interrupt channel can be cleared without corrupting the interrupt priority value configured by calling Os_InitializeVectorTable(). It may not be used by untrusted code.

Example

```
Os_Clear_GIC48()
Os_Clear_Millisecond()
```

5.3.3 Os_DisableAllConfiguredInterrupts_CPUx

The Os_DisableAllConfiguredInterrupts_CPUx macro will disable all configured GIC interrupt channels on CPUx (where x = 0 to 3). In the example, this is CPU0. To use the Os_DisableAllConfiguredInterrupts_CPUx macro the file Os_ConfigInterrupts.h must be included through the use of #include. The macro is provided so the interrupt channels can be disabled without corrupting the interrupt priority values configured by calling Os_InitializeVectorTable(). It may not be used by untrusted code.

Example

```
Os_DisableAllConfiguredInterrupts_CPU0()
...
Os_EnableAllConfiguredInterrupts_CPU0()
```

5.3.4 Os_Disable_x

Use of the Os_Disable_x macro will disable the named interrupt channel. The macro can be called using either the GIC channel number or the RTA-OS configured vector name. In the example, this is Os_Disable_GIC48() and Os_Disable_Millisecond() respectively. To use the Os_Disable_x macro the file Os_ConfigInterrupts.h must be included through the use of #include. The macro is provided so the interrupt channel can be masked without corrupting

the interrupt priority value configured by calling Os_InitializeVectorTable(). It may not be used by untrusted code.

Example

```
Os_Disable_GIC48()
Os_Disable_Millisecond()
```

5.3.5 Os_EnableAllConfiguredInterrupts_CPUx

The Os_EnableAllConfiguredInterrupts_CPUx macro will enable all configured EI interrupt channels on CPUx (where x = 0 to 3). In the example, this is CPU0. To use the Os_EnableAllConfiguredInterrupts_CPUx macro the file Os_ConfigInterrupts.h must be included through the use of #include. The macro is provided so the interrupt channels can be enabled without corrupting the interrupt priority values configured by calling Os_InitializeVectorTable(). It may not be used by untrusted code.

Example

```
Os_DisableAllConfiguredInterrupts_CPU0()
...
Os_EnableAllConfiguredInterrupts_CPU0()
```

5.3.6 Os_Enable_x

Use of the Os_Enable_x macro will enable the named interrupt channel. The macro can be called using either the GIC channel number or the RTA-OS configured vector name. In the example, this is Os_Enable_GIC48() and Os_Enable_Millisecond() respectively. To use the Os_Enable_x macro the file Os_ConfigInterrupts.h must be included through the use of #include. The macro is provided so the interrupt channel can be enabled without corrupting the interrupt priority value configured by calling Os_InitializeVectorTable(). It may not be used by untrusted code.

Example

```
Os_Enable_GIC48()
Os_Enable_Millisecond()
```

5.3.7 Os_IntChannel_x

The Os_IntChannel_x macro can be used to get the vector number associated with the named GIC interrupt (0, 1, 2...). The macro can be called using either the GIC vector name or the RTA-OS configured vector name. In the example, this is Os_IntChannel_Parity_Core_0 and Os_IntChannel_Millisecond respectively. To use the Os_IntChannel_x macro the file Os_ConfigInterrupts.h must be included through the use of #include. On a multi-core CPU GIC interrupts on different cores may have the same vector number. This is because the

GIC maintains a copy of interrupts 0 to 31 for each CPU core. Those outside this range share a single copy between all CPU cores.

Example

```
trigger_interrupt(Os_IntChannel_TPU);
trigger_interrupt(Os_IntChannel_Millisecond);
```

5.3.8 Os_Set_Edge_Triggered_x

Use of the Os_Set_Edge_Triggered_x macro will configure the named GIC interrupt channel as Edge-triggered. The macro can be called using either the channel name or the RTA-OS configured vector name. In the example, this is Os_Set_Edge_Triggered_GIC32() and Os_Set_Edge_Triggered_Millisecond() respectively. Only GIC channels 32 and above can be modified; the other channels have fixed settings. To use the Os_Set_Edge_Triggered_x macro the file Os_ConfigInterrupts.h must be included through the use of #include. It may not be used by untrusted code.

Example

```
Os_Set_Edge_Triggered_GIC32()
Os_Set_Edge_Triggered_Millisecond()
```

5.3.9 Os_Set_Level_Sensitive_x

Use of the Os_Set_Level_Sensitive_x macro will configure the named GIC interrupt channel as Level-sensitive. The macro can be called using either the channel name or the RTA-OS configured vector name. In the example, this is Os_Set_Level_Sensitive_GIC32() and Os_Set_Level_Sensitive_Millisecond() respectively. Only GIC channels 32 and above can be modified; the other channels have fixed settings. To use the Os_Set_Level_Sensitive_x macro the file Os_ConfigInterrupts.h must be included through the use of #include. It may not be used by untrusted code.

Example

```
Os_Set_Level_Sensitive_GIC32()
Os_Set_Level_Sensitive_Millisecond()
```

5.4 Type Definitions

5.4.1 Os_StackSizeType

An unsigned value representing an amount of stack in bytes.

Example

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

5.4.2 Os_StackValueType

An unsigned value representing the position of the stack pointer (USR/SYS mode).

Example

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

6 Toolchain

This chapter contains important details about RTA-OS and the ARM_DS_5_V6 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 ARM DS-5 Ultimate Edition: ARM Compiler 6.6

Ensure that armclang.exe is on the path and that the appropriate environment variables have been set.

Tested on ARM Compiler 6.6

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 armclang.exe
Version Component: ARM Compiler 6.6

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.

`--target=aarch64-arm-none-eabi` Generates A64 instructions for AArch64 state

`-fno-vectorize` Disables generation of Advanced SIMD vector instructions

`-mcpu=cortex-a53+nocrc+nocrypto+nofp+nosimd` Target the Cortex-A53 architecture and additional features including the floating-point mode (value set by target option)

`-mno-unaligned-access` Disable unaligned access to data

`-Ofast` Set fast optimization level

`-std=gnu11` 2011 C standard code with GNU extensions

`-fms-extensions` Enable the compiler extensions for section use pragmas

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.

`-fshort-enums` Set the size of an enumeration type to the smallest data type

`-fvectorize` Generate Advanced SIMD vector instructions

`-mbig-endian` Generate big-endian code

`-munaligned-access` Enable unaligned access to data

`-std=x` Other C standard code apart from gnu11

`--target=arm-arm-none-eab` Generates A32/T32 instructions for AArch32 state

- Any other options that conflict with kernel options

6.2.2 Assembler

Name armclang.exe
Version Component: ARM Compiler 6.6

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 armar.exe
Version Component: ARM Compiler 6.6

6.2.4 Linker

Name armlink.exe
Version Component: ARM Compiler 6.6

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.

`--info=totals,sizes,unused` Specify map file contents

`--datacompressor=off` Disable RW data compression

`--noremove` Do not remove unused input sections

`--xref` Output cross reference information to the map file

`--map` Output memory map to the map file

`--symbols` Output symbol table to the map file

`--verbose` Output detailed information to the map file

`--entry=reset_handler` Specify the application entry point

Mandatory Options for Application Code

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

- The same options as for compilation

Forbidden Options for Application Code

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

- Any options that conflict with kernel options

6.2.5 Debugger

Name Lauterbach TRACE32
Version Build 92037 or later

Notes on using ORTI with the Lauterbach debugger

When ORTI information for the Trace32 debugger is enabled entry and exit times for Category 1 interrupts are increased by a few cycles to support tracking of Category 1 interrupts by the debugger.

ORTI Stack Fill with the Lauterbach debugger

The 'ORTI Stack Fill' target option is provided to extend the ORTI support to allow evaluation of unused stack space. The Task.Stack.View command can then be used in the Trace32 debugger. The following must also be added to an application to ensure correct operation (as demonstrated in the sample applications):

The linker file must create labels holding the start address and stack size for each stack (one per core). The labels automatically generated by the linker can be used. For a single core system (i.e. core 0 only) the labels are:

```
extern const uint64 Image$$ARM_LIB_STACKHEAP$$ZI$$Base;
extern const uint64 Image$$ARM_LIB_STACKHEAP$$ZI$$Length;
OS_STACK0_BASE = (uint64)&Image$$ARM_LIB_STACKHEAP$$ZI$$Base;
OS_STACK0_SIZE = (uint64)&Image$$ARM_LIB_STACKHEAP$$ZI$$Length;
```

where ARM_LIB_STACKHEAP is the section containing the Core 0 stack.

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

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

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

7 Hardware

7.1 Supported Devices

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

Name: Xilinx
Device: Zynq UltraScale+ Cortex-A53

The following variants of the Zynq UltraScale+ Cortex-A53 are supported:

- GenericZynqUSA53

If you require support for a variant of Zynq UltraScale+ Cortex-A53 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
GICD_IGROUPRx	The GIC group registers have to be configured to match the secure state that the application runs in. These registers can only be modified in the secure state. If running in the secure state then these registers can be configured by calling Os_InitializeVectorTable(). If running in the non-secure state then Os_InitializeGICGroup() should be called before entering the non-secure state.
GICD_IPRIORITYRx	The GIC priorities have to be set to match the values declared in the configuration. This can be done by calling Os_InitializeVectorTable().
GICD_ISENABLERx/GICD_ICENABLERx	The GIC mask registers have to be set to match the declared ISRs in the configuration. This can be done by calling Os_InitializeVectorTable().
GICD_ITARGETSRx	The GIC processor targets have to be set to match the values declared in the configuration. This can be done by calling Os_InitializeVectorTable().
PSTATE	The PSTATE must select a privileged mode (i.e. EL1) before calling Os_InitializeVectorTable().
SP	The stack must be allocated and SP initialized to only use SP_EL0 before calling Os_InitializeVectorTable().
VBAR_EL1	The VBAR_EL1 must be set to the address of the EL1 vector table. This can be done by calling Os_InitializeVectorTable().

7.2.2 Modification

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

Register	Notes
GIC	User code may not modify the GIC directly.
PSTATE	User code may not change the exception level, execution state or exception masking bits.
SP	User code may not change the SP_EL0 stack pointer other than as a result of normal program flow.

RTA-OS operates all code with the PSTATE.A bit enabled.

7.3 Interrupts

This section explains the implementation of RTA-OS's interrupt model on the Zynq UltraScale+ Cortex-A53.

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 Zynq UltraScale+ Cortex-A53:

IPL	GICC_PMR	Description
		When 32 GIC Interrupt Priority Levels are in use
0	0xF8	User (task) level. No interrupts are masked.
1	0xF0	Maskable Category 1 and 2 interrupts routed through IRQ.
...	...	Maskable Category 1 and 2 interrupts routed through IRQ.
31	0x00	Maskable Category 1 and 2 interrupts routed through IRQ or Spurious GIC interrupt handler
32	n/a	Cortex-A5x CPU FIQ Category 1 exceptions.
33	n/a	Cortex-A5x CPU Synchronous and SError Category 1 exceptions.
		When 16 GIC Interrupt Priority Levels are in use
0	0xF0	User (task) level. No interrupts are masked.
1	0xE0	Maskable Category 1 and 2 interrupts routed through IRQ.
...	...	Maskable Category 1 and 2 interrupts routed through IRQ.
15	0x00	Maskable Category 1 and 2 interrupts routed through IRQ or Spurious GIC interrupt handler
32	n/a	Cortex-A5x CPU FIQ Category 1 exceptions.
33	n/a	Cortex-A5x CPU Synchronous and SError Category 1 exceptions.

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 Zynq UltraScale+ Cortex-A53. A ✓ indicates that the mapping is permitted and a ✗ indicates that it is not permitted:

Address	Category 1	Category 2
CPU EL1 Trap0 exception handler (Synchronous Current EL with SP0)	✓	✗
CPU EL1 Trap1 exception handler (IRQ/vIRQ Current EL with SP0)	✓	✗
CPU EL1 Trap2 exception handler (FIQ/vFIQ Current EL with SP0)	✓	✗
CPU EL1 Trap3 exception handlers (SError/vSError Current EL with SP0)	✓	✗
CPU EL1 Trap4 exception handlers (Synchronous Current EL with SPx)	✓	✗
CPU EL1 Trap5 exception handlers (IRQ/vIRQ Current EL with SPx)	✓	✗
CPU EL1 Trap6 exception handlers (FIQ/vFIQ Current EL with SPx)	✓	✗
CPU EL1 Trap7 exception handlers (SError/vSError Current EL with SPx)	✓	✗
CPU EL1 Trap8 exception handlers (Synchronous Lower EL using AArch64)	✓	✗
CPU EL1 Trap9 exception handlers (IRQ/vIRQ Lower EL using AArch64)	✓	✗
CPU EL1 Trap10 exception handlers (FIQ/vFIQ Lower EL using AArch64)	✓	✗
CPU EL1 Trap11 exception handlers (SError/vSError Lower EL using AArch64)	✓	✗
CPU EL1 Trap12 exception handlers (Synchronous Lower EL using AArch32)	✓	✗
CPU EL1 Trap13 exception handlers (IRQ/vIRQ Lower EL using AArch32)	✓	✗
CPU EL1 Trap14 exception handlers (FIQ/vFIQ Lower EL using AArch32)	✓	✗
CPU EL1 Trap15 exception handlers (SError/vSError Lower EL using AArch32)	✓	✗
CPU EL1 Trap16 Spurious (FIQ or IRQ) GIC interrupt handler	✓	✗
GIC interrupt handlers	✓	✓

RTA-OS requires that on entry to exceptions the Cortex CPU switches to use the EL_SP0 stack.

7.3.3 Vector Table

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

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

Section	Valid Addresses
Os_ExceptionVectors	Should either be located in accordance with the EL1 Cortex Vector Base Address Register VBAR_EL1. The first entry is the Current EL with SP0 Synchronous exception handler.

The RTA-OS generated vector table does not include the reset vector. This should be added for an application and be located at the address contained within the the RVBAR_Elx register.

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

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 */
}
```

Cortex-A53 CPU exception handlers can be configured as Category 1 ISRs. As there is no compiler support to write the entry code in C these should not use the the CAT1_ISR macro.

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
Standard	The standard AArch64 EABI memory model is used.

Apart from some small code sections RTA-OS uses the default compiler memory sections unless modified by the AUTOSAR MemMap.h overrides. The non-default code sections all use the prefix 'Os_' (i.e. Os_primitives).

7.5 Processor Modes

RTA-OS can run in the following processor modes:

Mode	Notes
Trusted	All trusted code runs at Exception level 1 (EL1).
Untrusted	All untrusted code runs at Exception level 0 (EL0).

RTA-OS uses the Synchronous handler to transfer between Untrusted and Trusted code in applications containing untrusted objects (i.e. ISRs, tasks and functions). This functionality must be supported if a user provided SVC handler is used in such applications. RTA-OS does not support use of the AArch32 instruction set in application code.

7.6 Stack Handling

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

RTA-OS manages the SP_EL0 stack (via register SP). No other stacks are used.

If there are Category 1 ISRs attached to the Cortex CPU exception handlers (i.e. Synchronous, FIQ, SError) then care must be taken that either the SP_EL1 stack has been initialized or that the exception handler transfers back to the SP_EL0 before using any stack.

If the 'Enable untrusted stack check' target option is selected then a small amount of SP_EL1 stack is required for each core (32 bytes per core).

7.7 Processor state when calling StartOS()

At StartOS() the following conditions should be true:

- The CPU must be operating in trusted EL1 mode.
- The CPU must be using the SP_EL0 stack.
- FIQ and IRQ interrupts must be enabled (see Os_InitializeVectorTable()).
- The GIC must be enabled and the GIC group configured (see Os_InitializeVectorTable() and Os_InitializeGICGroup()).

If any of these conditions are not met then ShutdownOS() will be called.

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 ZynqUSA53/ARM 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	GenericZynqUSA53 on
CPU Clock Speed	1199.988037MHz
Stopwatch Speed	1199.988037MHz

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	16	0
Counter	32	4
CounterCallback	8	0
ExpiryPoint	3.5	0
OS Overheads (max)	0	81
OS-Application	0	0
PeripheralArea	0	0
Resource	16	4
ScheduleTable	24	24
Task	32	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	8	12
Cat 2 ISR	24	0
Core Overheads (each OS core)	0	80
Core Overheads (each processor core)	40	25
Counter	48	4
CounterCallback	8	0
ExpiryPoint	3.5	0
OS Overheads (max)	0	8
OS-Application	8	0
PeripheralArea	0	0
Resource	24	4
ScheduleTable	32	24
Task	48	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	Code	RO Data	RW Data	ZI Data
ActivateTask.o	172	0	0	0
AdvanceCounter.o	8	0	0	0
CallTrustedFunction.o	48	0	0	0
CancelAlarm.o	128	0	0	0
ChainTask.o	164	0	0	0
CheckISRMemoryAccess.o	80	0	0	0
CheckObjectAccess.o	152	48	0	0
CheckObjectOwnership.o	132	48	0	0
CheckTaskMemoryAccess.o	80	0	0	0
ClearEvent.o	48	0	0	0
ControlIdle.o	76	0	0	4
DisableAllInterrupts.o	64	0	0	8
DispatchTask.o	300	0	0	0
ElapsedTime.o	256	0	0	0
EnableAllInterrupts.o	52	0	0	0
GetActiveApplicationMode.o	12	0	0	0
GetAlarm.o	224	0	0	0
GetAlarmBase.o	56	0	0	0
GetApplicationID.o	60	0	0	0
GetCounterValue.o	60	0	0	0

Library Module	Code	RO Data	RW Data	ZI Data
GetCurrentApplicationID.o	60	0	0	0
GetElapsedCounterValue.o	88	0	0	0
GetEvent.o	48	0	0	0
GetExecutionTime.o	48	0	0	0
GetISRID.o	12	0	0	0
GetIsrMaxExecutionTime.o	48	0	0	0
GetIsrMaxStackUsage.o	48	0	0	0
GetResource.o	96	0	0	0
GetScheduleTableStatus.o	60	0	0	0
GetStackSize.o	8	0	0	0
GetStackUsage.o	48	0	0	0
GetStackValue.o	20	0	0	0
GetTaskID.o	24	0	0	0
GetTaskMaxExecutionTime.o	48	0	0	0
GetTaskMaxStackUsage.o	48	0	0	0
GetTaskState.o	56	0	0	0
GetVersionInfo.o	48	0	0	0
Idle.o	8	0	0	0
InShutdown.o	4	0	0	0
IncrementCounter.o	24	0	0	0
InterruptSource.o	304	2	0	0
ModifyPeripheral.o	192	0	0	0
NextScheduleTable.o	176	0	0	0
Os_Cfg.o	268	1296	0	765
Os_Cfg_Counters.o	5268	1112	0	0
Os_Cfg_KL.o	64	0	0	0
Os_ExceptionVectors.o	2144	0	0	0
Os_GICSupport.o	44	0	0	0
Os_GetAbortStack.o	16	0	0	320
Os_GetCurrentIMask.o	16	0	0	0
Os_GetCurrentTPL.o	40	0	0	0
Os_IRQConst.o	16	297	0	0
Os_IRQHandler.o	156	0	0	0
Os_StartCores.o	152	0	0	4
Os_Wrapper.o	152	0	0	0
Os_setjmp.o	80	0	0	0
Os_tgt.o	44	0	0	0
Os_vec_init.o	264	376	0	0
ProtectionSupport.o	64	0	0	0
ReadPeripheral.o	168	0	0	0

Library Module	Code	RO Data	RW Data	ZI Data
ReleaseResource.o	112	0	0	0
ResetIsrMaxExecutionTime.o	48	0	0	0
ResetIsrMaxStackUsage.o	48	0	0	0
ResetTaskMaxExecutionTime.o	48	0	0	0
ResetTaskMaxStackUsage.o	48	0	0	0
ResumeAllInterrupts.o	52	0	0	0
ResumeOSInterrupts.o	52	0	0	0
Schedule.o	140	0	0	0
SetAbsAlarm.o	160	0	0	0
SetEvent.o	48	0	0	0
SetRelAlarm.o	224	0	0	0
SetScheduleTableAsync.o	92	0	0	0
ShutdownOS.o	92	0	0	0
StackOverrunHook.o	8	0	0	0
StartOS.o	240	0	0	0
StartScheduleTableAbs.o	176	0	0	0
StartScheduleTableRel.o	160	0	0	0
StartScheduleTableSynchron.o	92	0	0	0
StopScheduleTable.o	116	0	0	0
SuspendAllInterrupts.o	64	0	0	8
SuspendOSInterrupts.o	88	0	0	8
SyncScheduleTable.o	96	0	0	0
SyncScheduleTableRel.o	96	0	0	0
TerminateTask.o	28	0	0	0
ValidateCounter.o	60	0	0	0
ValidateISR.o	20	0	0	0
ValidateResource.o	60	0	0	0
ValidateScheduleTable.o	84	0	0	0
ValidateTask.o	84	0	0	0
WaitEvent.o	48	0	0	0
WritePeripheral.o	156	0	0	0

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	Code	RO Data	RW Data	ZI Data
ActivateTask.o	348	0	0	0
AdvanceCounter.o	8	0	0	0
CallTrustedFunction.o	48	0	0	0
CancelAlarm.o	240	0	0	0
ChainTask.o	260	0	0	0
CheckISRMemoryAccess.o	80	0	0	0
CheckObjectAccess.o	232	48	0	0
CheckObjectOwnership.o	292	48	0	0
CheckTaskMemoryAccess.o	80	0	0	0
ClearEvent.o	48	0	0	0
ControlIdle.o	88	0	0	8
CrossCore.o	60	0	0	0
DisableAllInterrupts.o	84	0	0	0
DispatchTask.o	684	0	0	0
ElapsedTime.o	256	0	0	0
EnableAllInterrupts.o	72	0	0	0
GetActiveApplicationMode.o	12	0	0	0
GetAlarm.o	232	0	0	0
GetAlarmBase.o	68	0	0	0
GetApplicationID.o	92	0	0	0
GetCounterValue.o	60	0	0	0
GetCurrentApplicationID.o	92	0	0	0
GetElapsedCounterValue.o	88	0	0	0
GetEvent.o	48	0	0	0
GetExecutionTime.o	48	0	0	0
GetISRID.o	36	0	0	0
GetIsrMaxExecutionTime.o	48	0	0	0
GetIsrMaxStackUsage.o	48	0	0	0
GetNumberOfActivatedCores.o	44	0	0	0
GetResource.o	120	0	0	0
GetScheduleTableStatus.o	176	0	0	0
GetSpinlock.o	8	0	0	0
GetStackSize.o	8	0	0	0
GetStackUsage.o	48	0	0	0
GetStackValue.o	44	0	0	0
GetTaskID.o	48	0	0	0
GetTaskMaxExecutionTime.o	48	0	0	0
GetTaskMaxStackUsage.o	48	0	0	0
GetTaskState.o	100	0	0	0
GetVersionInfo.o	48	0	0	0

Library Module	Code	RO Data	RW Data	ZI Data
Idle.o	8	0	0	0
InShutdown.o	4	0	0	0
IncrementCounter.o	24	0	0	0
InterruptSource.o	316	2	0	0
ModifyPeripheral.o	192	0	0	0
NextScheduleTable.o	288	0	0	0
Os_Cfg.o	372	2096	0	923
Os_Cfg_Counters.o	8700	1592	0	0
Os_Cfg_KL.o	124	0	0	0
Os_CrossCore.o	240	0	0	0
Os_ExceptionVectors.o	2144	0	0	0
Os_GICSupport.o	44	0	0	0
Os_GetAbortStack.o	32	0	0	640
Os_GetCurrentIMask.o	16	0	0	0
Os_GetCurrentTPL.o	128	0	0	0
Os_IRQConst.o	16	601	0	0
Os_IRQHandler.o	188	0	0	0
Os_ScheduleQ.o	68	0	0	0
Os_StartCores.o	152	0	0	8
Os_Wrapper.o	184	0	0	0
Os_setjmp.o	80	0	0	0
Os_tgt.o	44	0	0	0
Os_vec_init.o	336	412	0	0
ProtectionSupport.o	64	0	0	0
ReadPeripheral.o	168	0	0	0
ReleaseResource.o	144	0	0	0
ReleaseSpinlock.o	8	0	0	0
ResetIsrMaxExecutionTime.o	48	0	0	0
ResetIsrMaxStackUsage.o	48	0	0	0
ResetTaskMaxExecutionTime.o	48	0	0	0
ResetTaskMaxStackUsage.o	48	0	0	0
ResumeAllInterrupts.o	72	0	0	0
ResumeOSInterrupts.o	72	0	0	0
Schedule.o	168	0	0	0
SetAbsAlarm.o	276	0	0	0
SetEvent.o	48	0	0	0
SetRelAlarm.o	344	0	0	0
SetScheduleTableAsync.o	92	0	0	0
ShutdownAllCores.o	100	0	0	0
ShutdownOS.o	136	0	0	0

Library Module	Code	RO Data	RW Data	ZI Data
StackOverrunHook.o	8	0	0	0
StartCore.o	76	0	0	0
StartNonAutosarCore.o	76	0	0	0
StartOS.o	944	0	0	0
StartScheduleTableAbs.o	284	0	0	0
StartScheduleTableRel.o	268	0	0	0
StartScheduleTableSynchron.o	92	0	0	0
StopScheduleTable.o	228	0	0	0
SuspendAllInterrupts.o	84	0	0	0
SuspendOSInterrupts.o	108	0	0	0
SyncScheduleTable.o	96	0	0	0
SyncScheduleTableRel.o	96	0	0	0
TerminateTask.o	48	0	0	0
TryToGetSpinlock.o	16	0	0	0
ValidateCounter.o	84	0	0	0
ValidateISR.o	20	0	0	0
ValidateResource.o	84	0	0	0
ValidateScheduleTable.o	60	0	0	0
ValidateTask.o	160	0	0	0
WaitEvent.o	48	0	0	0
WritePeripheral.o	156	0	0	0

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()`).

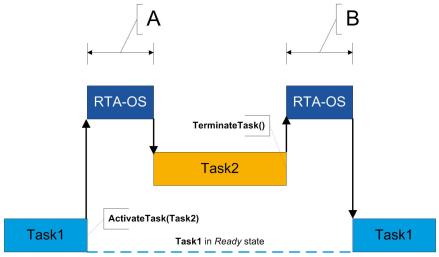
Interrupt latency is the time between an interrupt request being recognized by the target hardware and the execution of the first instruction of the user provided handler function:

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

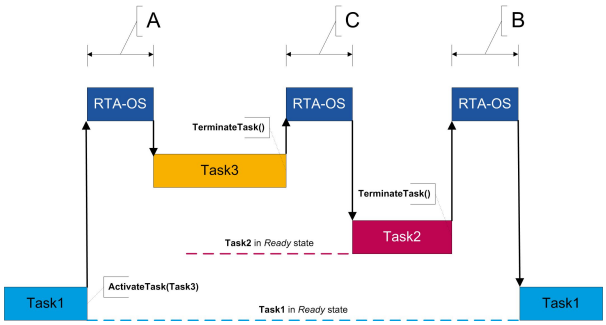
For Category 2 ISRs this is the time required for the hardware to recognize the interrupt plus the time required by RTA-OS to set-up the context in which the ISR runs.

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

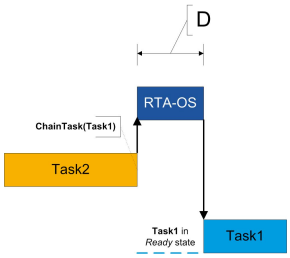
Switch	Key	CPU Cycles	Actual Time
Task activation	A	691	576ns
Task termination with resume	B	475	396ns
Task termination with switch to new task	C	475	396ns
Chaining a task	D	936	780ns
Waiting for an event resulting in transition to the WAITING state	E	1101	918ns
Setting an event results in task switch	F	1382	1.15us
Non-preemptive task offers a preemption point (co-operative scheduling)	G	691	576ns
Releasing a resource results in a task switch	H	705	588ns
Entering a Category 2 ISR	I	944	787ns
Exiting a Category 2 ISR and resuming the interrupted task	J	734	612ns
Exiting a Category 2 ISR and switching to a new task	K	727	606ns
Entering a Category 1 ISR	L	908	757ns



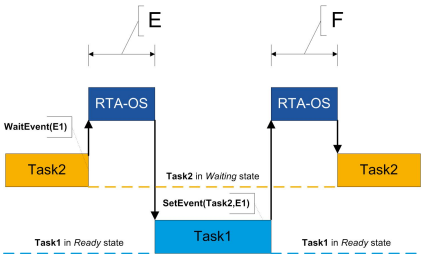
(a) Task activated. Termination resumes preempted task.



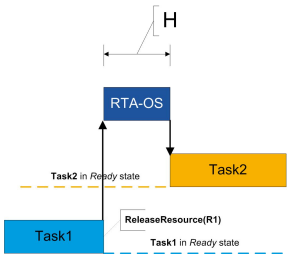
(b) Task activated. Termination switches into new task.



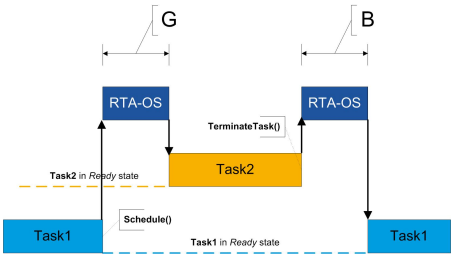
(c) Task chained.



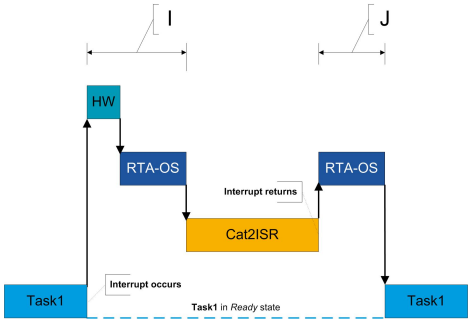
(d) Task waits. Task is resumed when event set.



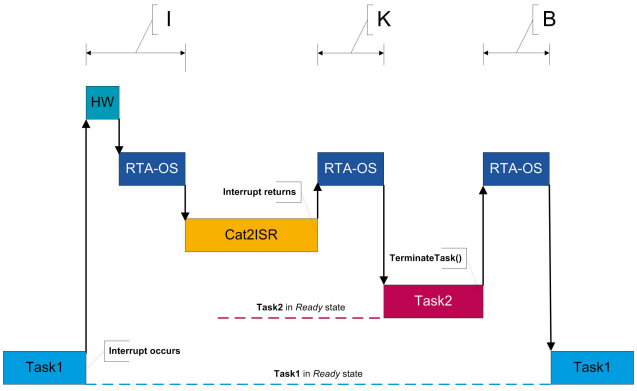
(e) Task switch when resource is released.



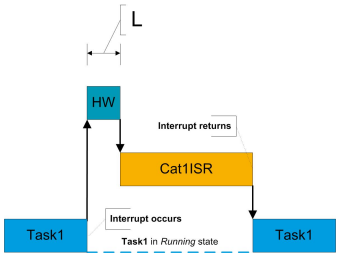
(f) Request for scheduling made by non-preemptive task.



(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 ZynqUSA53/ARM-specific parts of RTA-OS can be found in the following manuals:

***ZynqUSA53/ARM Release Note*.** This document provides information about the ZynqUSA53/ARM 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, 43
AUTOSAR OS includes
    Os.h, 30
    Os_Cfg.h, 30
    Os_MemMap.h, 30

C

CAT1_ISR, 36
Compiler, 42
Compiler (ARM DS-5 Ultimate Edition: ARM Compiler 6.6), 41
Compiler Versions, 41
Configuration
    Port-Specific Parameters, 23

D

Debugger, 45

E

ETAS License Manager, 12
    Installation, 12

F

Files, 29

H

Hardware
    Requirements, 10

I

Installation, 10
    Default Directory, 11
    Verification, 21
Interrupts, 49
    Category 1, 52
    Category 2, 52
    Default, 53
IPL, 49

L

Librarian, 44
Library
    Name of, 30
License, 12
    Borrowing, 17
    Concurrent, 14
    Grace Mode, 13
    Installation, 16
    Machine-named, 14
    Status, 17
    Troubleshooting, 18
    User-named, 14
Linker, 44

M

Memory Model, 53

O

Options, 42
Os_Cbk_GetAbortStack, 32
Os_Cbk_StartCore, 33
Os_Cbk_StopCore, 35
Os_Clear_x, 37
Os_Disable_x, 37
Os_DisableAllConfiguredInterrupts_CPUx, 37
Os_Enable_x, 38
Os_EnableAllConfiguredInterrupts_CPUx, 38
Os_InitializeGICGroup, 31
Os_InitializeVectorTable, 31
Os_IntChannel_x, 38
Os_Set_Edge_Triggered_x, 39
Os_Set_Level_Sensitive_x, 39
Os_StackSizeType, 39
Os_StackValueType, 40

P

Parameters of Implementation, 23
Performance, 55
    Context Switching Times, 63
    Library Module Sizes, 57
    RAM and ROM, 55
    Stack Usage, 56
Processor Modes, 53
    Trusted, 53
    Untrusted, 53

Processor state when calling StartOS(), 54

R

Registers
- GIC, 49
- GICD_IGROUPRx, 48
- GICD_IPRIORITYRx, 48
- GICD_ISENABLERx/GICD_ICENABLERx, 48
- GICD_ITARGETSRx, 48
- Initialization, 47
- Non-modifiable, 48
- PSTATE, 48, 49
- SP, 48, 49
- VBAR_EL1, 48

S

Software
- Requirements, 10

Stack, 54

T

Target, 47
- Variants, 47

Toolchain, 41

V

Variants, 47

Vector Table
- Base Address, 52