
RTA-OS3.0

User 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 ETAS GmbH, Stuttgart.

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

Document: 10384-UG-1.0.0

Contents

1	Welcome to RTA-OS3.0!	13
1.1	Related Documents	13
1.2	About You	13
1.3	Document Conventions	14
1.4	References	14
2	Introduction	15
2.1	Features of the RTA-OS3.0 Kernel	17
	2.1.1 OSEK	17
	2.1.2 AUTOSAR	20
	2.1.3 Unique RTA-OS3.0 Features	22
2.2	Summary	23
3	Development Process	24
3.1	Configuration	24
	3.1.1 OS Configuration	25
	3.1.2 RTA-TRACE Configuration	28
	3.1.3 Build	28
	3.1.4 Project Files	29
	3.1.5 Error Checking	31
	3.1.6 Generating Reports	32
3.2	Library Generation	32
	3.2.1 Preparing the Tool Chain	34
	3.2.2 Understanding AUTOSAR Dependencies	34
	3.2.3 Running rtaosgen	37
	3.2.4 Building the library	37

	3.2.5	Generated Files	38
3.3		Integration	39
	3.3.1	Accessing the OS in your Source Code	39
	3.3.2	Implementing Tasks and ISRs	40
	3.3.3	Starting the OS	40
	3.3.4	Interacting with the RTA-OS3.0	41
	3.3.5	Compiling and Linking	41
3.4		Memory Images and Linker Files	41
	3.4.1	Sections	41
	3.4.2	The Linker Control File	43
3.5		Summary	44
4		Tasks	45
4.1		Scheduling	45
4.2		Basic and Extended Tasks	47
	4.2.1	Task States	48
	4.2.2	Task Priorities	50
	4.2.3	Queued Task Activation	51
4.3		Conformance Classes	51
4.4		Maximizing Performance and Minimizing Memory	52
4.5		Task Configuration	53
	4.5.1	Scheduling Policy	55
	4.5.2	Queued Activation	55
	4.5.3	Auto-starting Tasks	56
4.6		Stack Management	57
	4.6.1	Working with Extended Tasks	58

4.6.2	Specifying Stack Allocation	60
4.6.3	Optimizing the Extended Task context save	63
4.6.4	Additional Stack Information	64
4.6.5	Handling Stack Overrun	64
4.7	Implementing Tasks	66
4.8	Activating Tasks	67
4.8.1	Direct Activation	68
4.8.2	Indirect Activation	69
4.9	Controlling Task Execution Ordering	69
4.9.1	Direct Activation Chains	69
4.9.2	Using Priority Levels	70
4.10	Co-operative Scheduling in RTA-OS3.0	72
4.10.1	Optimizing out the Schedule() API	72
4.11	Terminating Tasks	73
4.11.1	Optimizing Termination in RTA-OS3.0	73
4.12	The Idle Mechanism	74
4.13	Pre and Post Task Hooks	76
4.14	Saving Hardware Registers across Preemption	78
4.15	Summary	81
5	Interrupts	83
5.1	Single-Level and Multi-Level Platforms	83
5.2	Interrupt Service Routines	83
5.3	Category 1 and Category 2 Interrupts	84
5.3.1	Category 1 Interrupts	84
5.3.2	Category 2 Interrupts	84

5.4	Interrupt Priorities	84
5.4.1	User Level	87
5.4.2	OS Level	87
5.5	Interrupt Configuration	88
5.5.1	Vector Table Generation	89
5.6	Implementing Interrupt Handlers	90
5.6.1	Category 1 Interrupt Handlers	90
5.6.2	Category 2 Interrupt Handlers	91
5.6.3	Dismissing Interrupts	91
5.6.4	Writing Efficient Interrupt Handlers	92
5.7	Enabling and Disabling Interrupts	93
5.8	Saving Register Sets	94
5.9	The Default Interrupt	94
5.10	Summary	96
6	Resources	98
6.1	Resource Configuration	100
6.2	Resources on Interrupt Level	100
6.3	Using Resources	101
6.3.1	Nesting Resource Calls	102
6.4	Linked Resources	103
6.5	Internal Resources	105
6.6	Using Resources to Minimize Stack Usage	108
6.6.1	Internal Resources	109
6.6.2	Standard Resources	109
6.7	The Scheduler as a Resource	110

6.8	Choosing a Preemption Control Mechanism	110
6.9	Avoiding Race Conditions	112
6.10	Summary	113
7	Events	114
7.1	Configuring Events	114
7.1.1	Defining Waiting Tasks	115
7.2	Waiting on Events	115
7.2.1	Single Events	117
7.2.2	Multiple Events	117
7.2.3	Deadlock with Extended Tasks	118
7.3	Setting Events	119
7.3.1	Setting Events with an Alarm	120
7.3.2	Setting Events with a Schedule Table Expiry Point	120
7.4	Clearing Events	120
7.5	Simulating Extended Tasks with Basic Tasks	121
7.6	Summary	122
8	Counters	123
8.1	Configuring Counters	123
8.2	Counter Drivers	124
8.2.1	Software Counter Drivers	125
8.2.2	Hardware Counter Drivers	129
8.3	Accessing Counter Attributes at Runtime	132
8.3.1	Special Counter Names	133
8.4	Reading Counter Values	133
8.5	Tick to Time Conversions	134

8.6	Summary	136
9	Alarms	137
9.1	Configuring Alarms	137
9.1.1	Activating a Task	138
9.1.2	Setting an Event	138
9.1.3	Alarm Callbacks	138
9.1.4	Incrementing a Counter	140
9.2	Setting Alarms	142
9.2.1	Absolute Alarms	142
9.2.2	Relative Alarms	145
9.3	Auto-starting Alarms	146
9.4	Canceling Alarms	147
9.5	Working out when an Alarm will occur	148
9.6	Non-cyclic (aperiodic) Alarms	149
9.7	Summary	150
10	Schedule Tables	151
10.1	Configuring a Schedule Table	152
10.1.1	Configuring Expiry Points	153
10.2	Starting Schedule Tables	154
10.2.1	Absolute Start	154
10.2.2	Relative Start	156
10.3	Stopping Schedule Tables	157
10.4	Switching Schedule Tables	157
10.5	Schedule Table Status	159
10.6	Summary	159

11	Writing Hardware Counter Drivers	160
11.1	The Hardware Counter Driver Model	160
11.1.1	Interrupt Service Routine	161
11.1.2	Callbacks	162
11.2	Using Output Compare Hardware	163
11.2.1	Callbacks	164
11.2.2	Interrupt Handlers	168
11.2.3	Handling a Hardware modulus not equal to TickType	172
11.3	Free Running Counter and Interval Timer	177
11.3.1	Callbacks	178
11.3.2	ISR	179
11.4	Using Match on Zero Down Counters	180
11.4.1	Callbacks	181
11.4.2	Interrupt Handler	182
11.5	Software Counters Driven by an Interval Timer	184
11.6	Summary	184
12	Startup and Shutdown	185
12.1	From System Reset to StartOS ()	185
12.1.1	Power-on or Reset	185
12.1.2	C Language Start-up Code	186
12.1.3	Running main ()	187
12.2	Starting RTA-OS3.0	189
12.2.1	Startup Hook	190
12.2.2	Application Modes	191
12.3	Shutting Down RTA-OS3.0	196

12.3.1	Shutdown Hook	196
12.4	Restarting RTA-OS3.0	196
12.5	Summary	198
13	Error Handling	199
13.1	Centralized Error Handling - the ErrorHandler()	200
13.1.1	Configuring Advanced Error Logging	202
13.1.2	Working out which Task is Running	204
13.1.3	Working out which ISR is Running	204
13.1.4	Generating a Skeleton ErrorHandler()	205
13.2	Inline Error Handling	205
13.3	Conditional Inclusion of Error Checking Code	206
13.4	Summary	206
14	Measuring and Monitoring Stack Usage	207
14.1	Stack Monitoring	207
14.1.1	Setting Defaults	208
14.1.2	Configuring Stack Allocation per Task/ISR	208
14.2	Using the Os_Cbk_StackOverrunHook()	210
14.3	Measuring Stack Usage	212
14.3.1	Marking the Worst Case for Function Calls	213
14.4	Summary	216
15	Measuring and Monitoring Execution Time	217
15.1	Enabling Timing Measurement	217
15.1.1	Providing a Stopwatch	217
15.1.2	Scaling the Stopwatch	219
15.2	Automatic Measurement of Task and ISR Execution Times	219

15.3	Manual Time Measurement	221
15.4	Imprecise Computation	222
15.5	Monitoring Execution Times against Budgets	223
15.6	Summary	225
16	Using an ORTI-Compatible Debugger	226
16.1	Development Process	226
16.2	Intrusiveness	227
16.3	Validity	228
16.4	Interactions	228
16.5	Summary	228
17	RTA-TRACE Integration	230
17.1	Basic Configuration	230
17.2	Controlling RTA-TRACE	232
	17.2.1 Controlling with Objects are Traced	233
17.3	User-Defined Trace Objects	236
	17.3.1 Tracepoints	236
	17.3.2 Task Tracepoints	237
	17.3.3 Intervals	237
	17.3.4 Controlling which User-Defined Objects are Traced	239
	17.3.5 Format Strings	241
17.4	ECU Links	244
	17.4.1 Debugger Links	244
	17.4.2 Serial Links	245
17.5	Summary	251
18	Contacting ETAS	252

18.1	Technical Support	252
18.2	General Enquiries	253

1 Welcome to RTA-OS3.0!

This user guide tells you how to use RTA-OS3.0 to build AUTOSAR OS-based applications and is structured as follows:

Chapter 2 introduces you to RTA-OS3.0, covering what tools are provided, which standards are supported by the kernel and gives a brief overview of kernel features.

Chapter 3 takes you through the stages of development with RTA-OS3.0, including how to use the tools supplied to configure and build a kernel library and how to integrate it with your application.

Chapters 4-12 explain in detail how to configure the OS for each major class of OS object and how to use the kernel APIs that manipulate those objects at runtime.

Chapters 13-17 explain what to do when things go wrong. They cover how to detect erroneous use of the kernel API during development, how to check for stack overruns and timing faults and how to integrate with external debugging and profiling tools to get additional insight into how the OS is behaving at runtime..

1.1 Related Documents

A complete technical reference to RTA-OS3.0 can be found in the *RTA-OS3.0 Reference Guide* in the same directory as this user guide. Specific technical details about the implementation of RTA-OS3.0 for your choice of compiler and target hardware (what we call a “port”) is contained in the *RTA-OS3.0 Target/Compiler Port Guide*. There is one *RTA-OS3.0 Target/Compiler Port Guide* for each installed RTA-OS3.0 port that can be found in the root of the port’s installation directory.

1.2 About You

You are a trained embedded systems developer who wants to build real-time applications using a pre-emptive 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 tool chain. Elementary knowledge about your target microcontroller, such as the start address, memory layout, location of peripherals as so on, is essential.

You should also be familiar with common use of the Microsoft Windows®2000, Windows®XP or Windows®Vista operating systems, including installing software, selecting menu items, clicking buttons, navigating files and directories.

1.3 Document Conventions

The following conventions are used in this guide:

Choose **File > Open**.

Click **OK**.

Press <Enter>.

The “Open file” dialog box appears

Activate(Task1)

See Section **1.3**.

ETAS



Menu options are printed in **bold, blue** characters.

Button labels are printed in **bold** characters

Key commands are enclosed in angle brackets.

The names of program windows, dialog boxes, fields, etc. are enclosed in double quotes.

Program code, header file names, C type names, C functions and RTA-OS3.0. Component API call names all appear in the courier typeface.

Hyperlinks through the document are shown in **red letters**.

Functionality that is provided in RTA-OS3.0 but it may not be portable to another AUTOSAR OS implementation is marked with the ETAS logo.

Caution! Notes like this contain important instructions that you must follow carefully in order for things to work correctly.

1.4 References

OSEK is a European automotive industry standards effort to produce open systems interfaces for vehicle electronics. 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 Introduction

RTA-OS3.0 is a statically configurable, preemptive, real-time operating system (RTOS) for use in high-performance, resource-constrained applications. RTA-OS3.0 is a full implementation of the open-standard AUTOSAR OS R3.0 (Scalability Class 1) and includes functionality that is fully compliant to Version 2.2.3 of the OSEK/VDX OS Standard.

The RTA-OS3.0 kernel has been designed to be:

high performance - the kernel is very small and very fast. The memory footprint of the kernel and its run-time performance are class leading, making RTA-OS3.0 particularly suitable for systems manufactured in large quantities, where it is necessary to meet very tight constraints on hardware costs and where any final product must function correctly.

RTA-OS3.0 provides a number of unique optimizations that contribute to reductions in unit cost of systems. The kernel uses a single-stack architecture for all types of task. This provides significant RAM savings over a traditional stack-per-task model. Furthermore, careful application design can exploit the single-stack architecture to offer significant stack RAM savings.

The offline tools analyze your OS configuration and use this information to build the smallest and fastest kernel possible. Code that you are not going to use is excluded from the kernel, to avoid wasting execution time and memory space.

real-time - conventional RTOS designs normally have unpredictable overheads, usually dependent on the number of tasks and the state of the system at each point in time. This makes it difficult to guarantee real-time predictability - no matter how “fast” the kernel is. In RTA-OS3.0 the kernel is fast and all runtime overheads, such as switching to and from tasks, handling interrupts and waking up tasks, have low worst-case bounds and little (or even no) variability within execution times. In many cases, context switching happens in constant execution time. This means RTA-OS3.0 can be used for the development of hard real-time systems where responses must be made within specific timing deadlines. Meeting hard deadlines involves calculating the worst-case response time of each task and Interrupt Service Routine (ISR) and ensuring that everything runs on time, every time. RTA-OS3.0 is a *real* RTOS because it meets the assumptions of fixed priority schedulability analysis.

portable - RTA-OS3.0 is available for a wide variety of microcontroller/compiler combinations (called a ‘port’). All ports share the same common

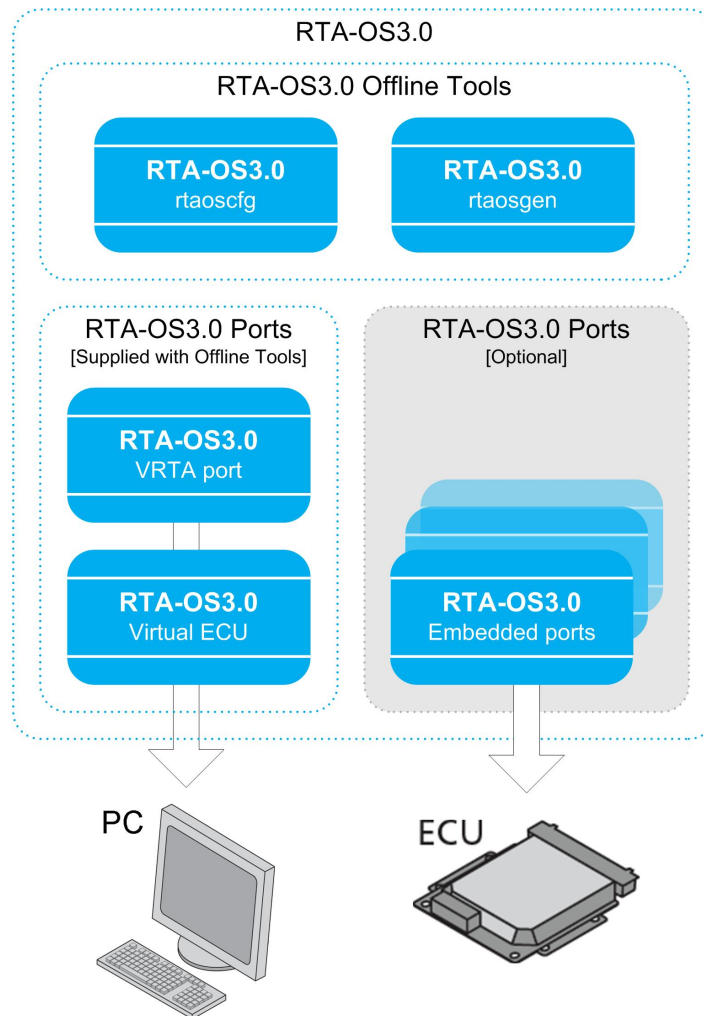


Figure 2.1: RTA-OS3.0 Product Architecture

RTA-OS3.0 code, which comprises about 97% of the total kernel functionality. The kernel is written in ANSI C that is MISRA-C 2004 compliant. A MISRA report for RTA-OS3.0 can be generated by the offline tools.

RTA-OS3.0 does not impose on hardware, where possible. Generally, there is no need to 'hand over' control of hardware, such as the cache, watchdog timers and I/O ports. As a result of this, hardware can be used freely, allowing 'legacy software' to be integrated into the system.

The RTA-OS3.0 product architecture is shown in Figure 2.1 and consists of:

- **rtaoscfg** a graphical configuration tool that reads and writes configurations in the AUTOSAR XML configuration language.

- **rtaosgen** a command-line tool for generating a RTA-OS3.0 kernel library from your input configuration.
- Port plug-ins, one for each target/compiler combination on which you use RTA-OS3.0. You can install multiple ports at the same time and switch between them with the offline tooling. You can also install multiple versions of the same port concurrently, allowing you to easily manage projects that use legacy compilers and/or microcontrollers.
- VRTA, a special port plug-in that provides the functionality of RTA-OS3.0 on a standard Windows PC. This allows you to design and test application behavior without needing real target hardware. VRTA comes with a development kit that allows you to build Virtual ECUs that can simulate interrupts, I/O etc.

2.1 Features of the RTA-OS3.0 Kernel

RTA-OS3.0 builds on the proven technology of earlier ETAS operating systems which, to date, have been used in over 350 million ECUs worldwide. The kernel provides an implementation of the AUTOSAR OS R3.0 open standard, a standard which subsumes features from the earlier OSEK OS standard¹. The kernel also provides a number of additional features that are unique to RTA-OS3.0. The following sections provide a short introduction to the standards and their features.

2.1.1 OSEK

OSEK is a European automotive industry standards effort to produce open systems interfaces for vehicle electronics. The full name of the project is OSEK/VDX. OSEK is an acronym formed from a phrase in German, which translates as “Open Systems and Corresponding Interfaces for Automotive Electronics”. VDX is based on a French standard (Vehicle Distributed eXecutive), which has now been merged with OSEK. OSEK/VDX is referred to as OSEK in this guide.

The goals of OSEK are to support portability and reusability of software components across a number of projects. This will allow vendors to specialize in “Automotive Intellectual Property”, whereby a vendor can develop a purely-software solution and run software in any OSEK-compliant ECU.

To reach this goal, however, detailed specifications of the interfaces to each non application-specific component are required. OSEK standards, therefore, include an Application Programming Interface (API) that abstracts away from

¹For the sake of brevity, the term “AUTOSAR OS” is used throughout this document to refer to the combined AUTOSAR and OSEK OS standard.

the specific details of the underlying hardware and the configuration of the in-vehicle networks.

For further information see <http://www.osek-vdx.org>.

OSEK OS

OSEK OS is the most mature and most widely used of the OSEK standards. OSEK OS has been adopted in all types of automotive ECUs, from power train, through chassis and body to multi-media devices.

The most recent version of OSEK OS is 2.2.3, the third minor revision of the 2.2 standard originally introduced in September 2001. This version of OSEK OS is also part of the ISO17356 standard.

OSEK OS is entirely statically defined using an offline configuration language called OIL (OSEK Implementation Language). All objects are known at system generation time. This means that implementations can be extremely small and efficient.

OSEK OS provides the following OS features:

Tasks are the main building block of OSEK OS systems. Unlike some other OS's, tasks in OSEK are not required to be "self-scheduling" - it is not necessary to place the body of the task inside an infinite loop². There are four types of task in OSEK OS:

1. Basic tasks with unique priority and non-queued activation. These are the simplest form of task and ideally suited for hard real-time systems. Tasks are activated and must run and terminate before they can be activated again. This type of task cannot suspend itself mid-way through execution to wait for an event. In RTA-OS3.0 these are called **BCC1** tasks because they correspond to OSEK OS's BCC1 conformance class (see Section 4.3 for more details about OSEK's Conformance Classes).
2. Basic tasks with shared priority and queued activation. These tasks can share priorities with other tasks in the system and do not need to terminate before being activated again. The OS queues pending task activations and runs the next activation when the current one has terminated. Like BCC1 tasks, this type of task cannot suspend itself mid-way through execution to wait for an event. In RTA-OS3.0 these are called **BCC2** tasks because they correspond to OSEK OS's BCC2 conformance class.
3. Extended tasks with unique priority. An extended task is allowed to wait for events during execution (i.e. the task can "self suspend").

²Though you can do this for the class of tasks called "extended tasks".

However, activations cannot be queued and the tasks must have unique priorities. In RTA-OS3.0 these are called **ECC1** tasks because they correspond to OSEK OS's ECC1 conformance class.

4. Extended tasks with shared priority. These are like ECC1 tasks but can share priorities with other tasks in the system. In this regard they are similar to BCC2 tasks. However, unlike BCC2 tasks, extended tasks cannot have queued activations. In RTA-OS3.0 these tasks are called **ECC2** tasks.

You can mix all of these task types together in the same system.

Scheduling Tasks can be scheduled either preemptively or non-preemptively and co-operative schedulers can be constructed easily.

Interrupts allow for the interaction of the OS with asynchronous external triggers. There are two types of interrupt in OSEK OS:

1. Category 1 interrupts are not handled by the OS
2. Category 2 interrupts are handled by, and can interact with, the OS

Resources are simple binary semaphores that allow you to provide mutual exclusion over critical sections shared between tasks and interrupts. Resources are managed by the OS using the priority ceiling protocol which guarantees freedom from deadlock and minimizes priority inversion at runtime³.

Counters and alarms are used to provide periodic (and aperiodic) scheduling of tasks. Counters, as the name suggests, count the occurrence of (domain specific) events and register values as 'ticks'. Alarms can be set to expire at run-time configurable count values, either at absolute count value or relative to the 'tick' value of the counter when the alarm is set.

Debugging Support is provided natively in the OS through the use of build levels. The OS also provides two build levels:

1. Standard is "lean and mean" and provides minimum error handling.
2. Extended is the "debugging" build that provides extensive error detection facilities to check if you are using the OS correctly.

³Priority inversion is the situation where a low priority task is running in preference to a higher priority task. With priority ceiling protocol this situation can occur at most once each time a higher priority task is activated (and it is always at the start of execution) and is called the blocking time for the higher priority task. The blocking time is bounded by the longest time any single task shares data with the higher priority object - there is no cumulative blocking due to the interaction of lower priority tasks.

Debugging is also provided through the OSEK ORTI (OSEK Run-Time Interface) standard. This provides a common way for OS implementations, like RTA-OS3,0, to export symbol details to third-party debuggers so that the debugger can display information about the internal state of the OS at runtime (e.g. which task is running, which tasks are ready to run etc.).

2.1.2 AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) is an open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers and tool developers worldwide.

AUTOSAR provides specifications for “Basic Software Modules” (BSW) such as operating systems, communication drivers, memory drivers and other microcontroller abstractions. The AUTOSAR standard also defines a component-based architectures model. This model defines a “Virtual Function Bus” (VFB) that defines an abstraction for communication application software components (SW-Cs). The VFB allows SW-Cs to be independent of the underlying hardware, making them portable between different ECUs and reusable across multiple automotive projects. The VFB abstraction is encapsulated by the AUTOSAR Run-Time Environment (RTE). The RTE provides the “glue” between SW-Cs and the BSW.

For further information see <http://www.autosar.org>.

AUTOSAR OS

AUTOSAR OS is an extension to the OSEK OS specification. An AUTOSAR OS includes all the features of OSEK OS and adds some new functionality which is divided into four “Scalability Classes” as follows:

Scalability Class 1 includes OSEK OS plus:

Schedule Tables provide an easier and alternative way to program repeating activities than OSEK OS’s alarms. Each schedule table can be managed as a single unit and you can switch between tables at runtime, allowing you to build ‘modal’ systems easily.

Software Counter Interface standardizes the interaction between the OS and counters (in OSEK this was vendor specific).

Stack Monitoring provides additional debugging support so that you can trap stack faults.

Scalability Class 2 includes Scalability Class 1 plus:

Schedule Table Synchronization to a global time source (though this is trivially possible in Scalability Class 1).

Timing Protection provides protection against tasks and interrupts executing for too long or too often. The protection scheme allows you to constrain at runtime those aspects of system timing that control whether your system meets its deadlines or not.

Scalability Class 3 includes Scalability Class 1 plus:

Memory Protection allows you to partition your system into **OS-Applications**. OS-Applications can be configured to be “trusted”, i.e. they run in what is typically called “supervisor mode”, or “non-trusted”, i.e. they run in what is typically called “user mode”. Memory access constraints can be programmed for non-trusted OS-Applications and the OS manages the target microcontroller’s memory management features at runtime to provide protection.

API Call Protection allows you to grant or deny access to the OS API for configured tasks/ISRs. For example you can forbid a task in one OS-Application from activating tasks in another OS-Application. API call protection also provides a mechanism for extending the API by adding “trusted functions” and granting or denying access to these functions as you would for the OS API.

Scalability Class 4 includes Scalability Classes 2 and 3



RTA-OS3.0 1.0.0 does not support the AUTOSAR OS R3.0 features from Scalability Classes 2, 3 and 4.

As AUTOSAR OS is based on OSEK OS, it is backwards compatible to existing OSEK OS-based applications - i.e. applications written for OSEK OS will largely run on AUTOSAR OS without modification. However, the AUTOSAR OS standard also clarifies some of the ambiguities in the OSEK OS specification that arise when the behavior of OSEK OS is *undefined* or *vendor specific* because these represent a barrier to portability. Users who are migrating from an OSEK OS and relied on a particular implementation of an OSEK OS feature should be aware that AUTOSAR OS defines the required OSEK OS behavior in the following cases:

OSEK OS	AUTOSAR OS
Behavior of relative alarms started at an offset of zero is undefined	Relative alarms cannot be started at a relative time of zero
The StartOS() API call may or may not return depending on the vendor implementation	StartOS() must not return
The behavior of ShutdownOS() is not defined if the ShutdownHook() returns.	ShutdownOS() disables all interrupts and enters an infinite loop.

The *RTA-OS3.0 Reference Guide* provides an API call compatibility listing between OSEK OS and AUTOSAR OS R3.0.

AUTOSAR OS replaces OSEK's OIL configuration format with an XML-based configuration language. AUTOSAR XML adopts the same configuration objects and concepts found in OIL, but uses a different syntax.

2.1.3 Unique RTA-OS3.0 Features

RTA-OS3.0 is much more than an AUTOSAR OS. The kernel is designed to support software engineers building and integrating real-time systems.

ETAS *RTA-OS3.0-specific features are not guaranteed to be portable to other implementations of OSEK OS or AUTOSAR OS.*

The additional features include:

Time Monitoring to measure the execution time of tasks and Category 2 ISRs at runtime and optionally check times against pre-configured budgets.

Enhanced Stack Monitoring provides additional possibilities to help you debug stack problems

RTA-TRACE Integration provides automatic instrumentation of the OS kernel to support the ETAS RTA-TRACE real-time OS profiling and visualization tool so you can view exactly what the OS is doing in real-time.

User control of hardware so there is no need to 'hand over' control of hardware, such as peripheral timers, the cache and I/O ports etc. to the OS. All hardware interaction occurs through RTA-OS3.0's well-defined hardware interface.

Predictable run-time overheads all runtime overheads such as switching to and from tasks, handling interrupts and waking up tasks, have low worst-case bounds and little variability within execution times.

Graphical offline configuration editor supports AUTOSAR XML configuration of the OS.

Easy integration into your build process as RTA-OS3.0 code generation requires just one command-line tool that can be driven from any build environment.

Highly scalable kernel architecture using offline tools that automatically to optimize the kernel for your application.

2.2 Summary

- RTA-OS3.0 is a pre-emptive RTOS for embedded systems
- The kernel provides the features specified in the AUTOSAR OS R3.0 (SC1) standard, including support for the legacy OSEK OS.
- RTA-OS3.0 provides additional features that make it easier to integrate AUTOSAR OS into your build process.

3 Development Process

This chapter provides a short overview of how to use RTA-OS3.0 in your applications. The process involves the following steps:

1. Configure the features of the OS you want to use
2. Generate a customized RTA-OS3.0 kernel library
3. Write application code that uses the OS
4. Compile your application code and linking with the RTA-OS3.0 library
5. Run your application on your target

The following sections cover each of these steps.

3.1 Configuration

RTA-OS3.0 is statically configured, which means that every task and interrupt you need must be declared at configuration time, together with any critical sections, synchronization points, counters etc.

All configuration is held in XML files that conform to the AUTOSAR standard. This means that you are free to use other 3rd party tools to provide the XML configuration files for the OS. If you look at one of these files in the examples provided by the target plug-ins then you will realize that the configuration language is not that easy to read.

To help you with configuration, RTA-OS3.0 includes **rtaoscfg**, a graphical configuration editor for configuring your RTA-OS3.0 application. **rtaoscfg** accepts any AUTOSAR XML file as input and allows you to edit the OS-specific parts of configuration. If the input file contains both OS and non-OS specific configuration then only the OS configuration will be modified.



The nature of XML parsers means that the ordering of configuration objects in the files may not be preserved when edited with different tools.

rtaoscfg has five main areas panes as shown in Figure 3.1.

1. Menu/Toolbar
2. Project Navigator.
See an overview of the top-level objects in your configuration, switch between configuration workspaces and manage the files in the project.
3. Configuration Workspace.
This is where you do most of your configuration. RTA-OS3.0 provides the following workspaces:

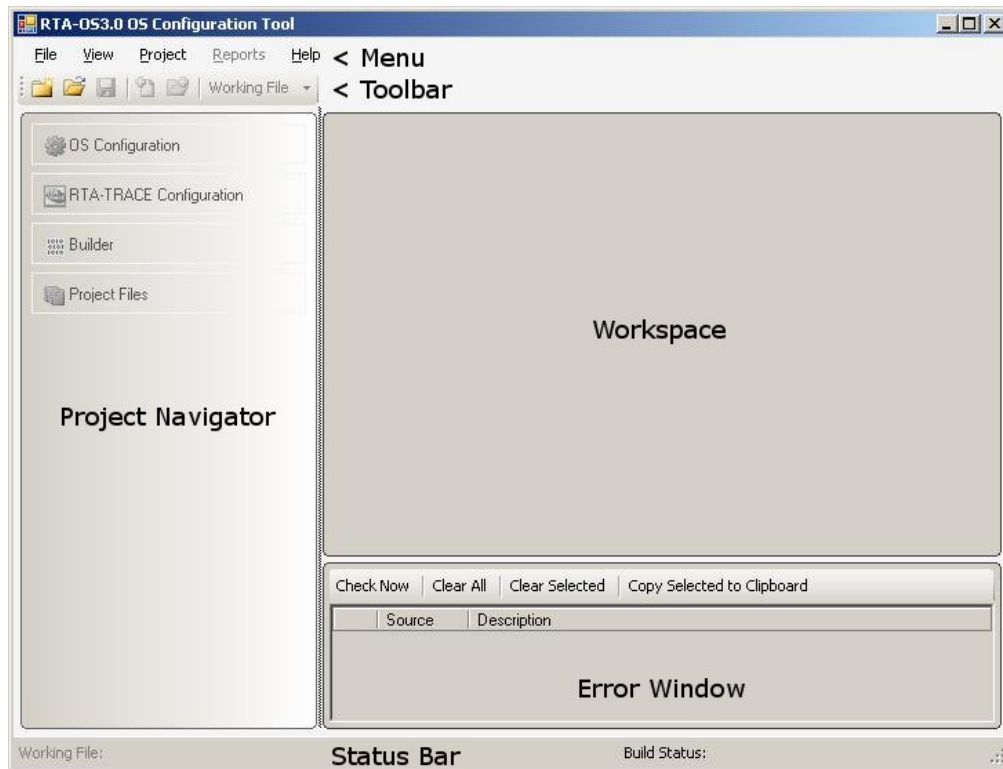


Figure 3.1: The **rtaoscfg** configuration tool

- (a) OS Configuration
- (b) RTA-TRACE Configuration
- (c) Builder
- (d) Project Files

4. Error Viewer.

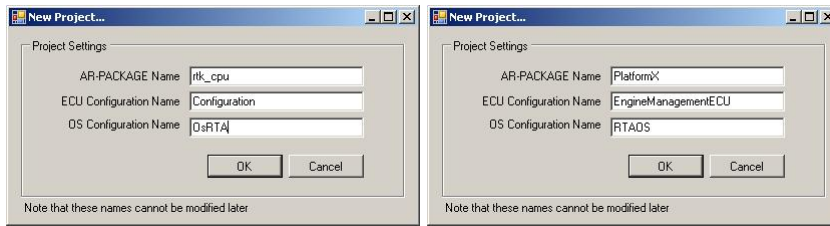
This displays a list of errors in the currently loaded configuration.

5. Status Bar

3.1.1 OS Configuration

The OS configuration navigator displays the logical structure of your OS configuration in the left hand window, ordered by OS object. You can expand each object to see the instances you have created. Clicking on an instance of an object displays the configuration panel in the right hand window. Individual items are configured in the workspace on the right hand side of the navigator.

To create a new configuration, select **File → New project** from the menu or use the keyboard shortcut <Ctrl+N>. Each new configuration requires you to



(a) Default Settings

(b) Customized Values

Figure 3.2: Defining project-wide settings

specify the “administrative” parts of an AUTOSAR XML configuration. This is required because parts of the OS configuration need to reference other parts (for example, tasks need to reference which resources they use) and these references are formed as an absolute path to an item in the AUTOSAR XML configuration. The items required are:

AR-PACKAGE Name defines the name of the AUTOSAR package. All AUTOSAR configuration items live in an AR-PACKAGE and a system may contain multiple packages. The OS configuration for a single ECU must live in a single package - it is not possible to split an OS configuration over multiple packages.

ECU Configuration Name defines the name of the ECU-CONFIGURATION of which this OS configuration will be a part. An ECU-CONFIGURATION contains all the configuration elements for all of the basic software for one ECU.

OS Configuration Name defines the name of the OS configuration MODULE-CONFIGURATION. This is the name that will be used to refer to the OS from the ECU-CONFIGURATION.

Figure 3.2 shows the default settings for a new project and example of how these might be customized for a particular project, in this case the ECU “EngineManagementECU” for the vehicle platform “PlatformX”.



*Project settings are set once at the creation of a new project. It is not possible to change them later using **rtaoscfg**.*

Clicking on a ‘root’ object like “Tasks”, “ISRs”, “Resources” etc. in the project navigator takes you to the configuration workspace for that type of object. Clicking on individual objects takes you to the configuration workspace for the selected object. Figure 3.3 shows an OS configuration where configuration for TaskB has been selected.

OS configuration is split into three main parts:

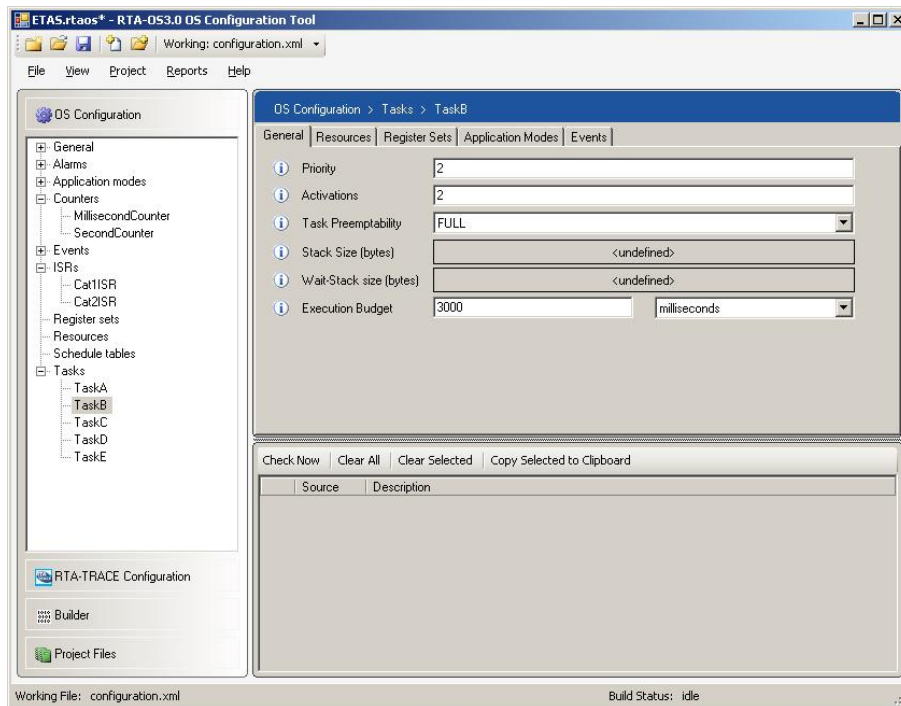


Figure 3.3: OS configuration in **rtaoscfg**

1. System-wide configuration for target-neutral general settings such as the level is debugging information you want to record, whether you monitor the stack at runtime, which hooks (callbacks) you are going to use, etc.
2. Target-specific settings including:
 - the target device you are using. You can use **rtaoscfg** to configure any target device for which you have installed a licensed RTA-OS3.0 port.
 - the variant of the target device if the port supports multiple variants of the target.
 - the version of the target to use if you have more than one version of a specific RTA-OS3.0 port installed
 - “Target Specific” aspects of configuration.
3. Object configuration for each of the OS objects you want to use. This includes task and interrupt configuration, resources, events, alarms, counters and schedule tables.

Specific aspects of configuration are covered in the later chapters of this user guide.

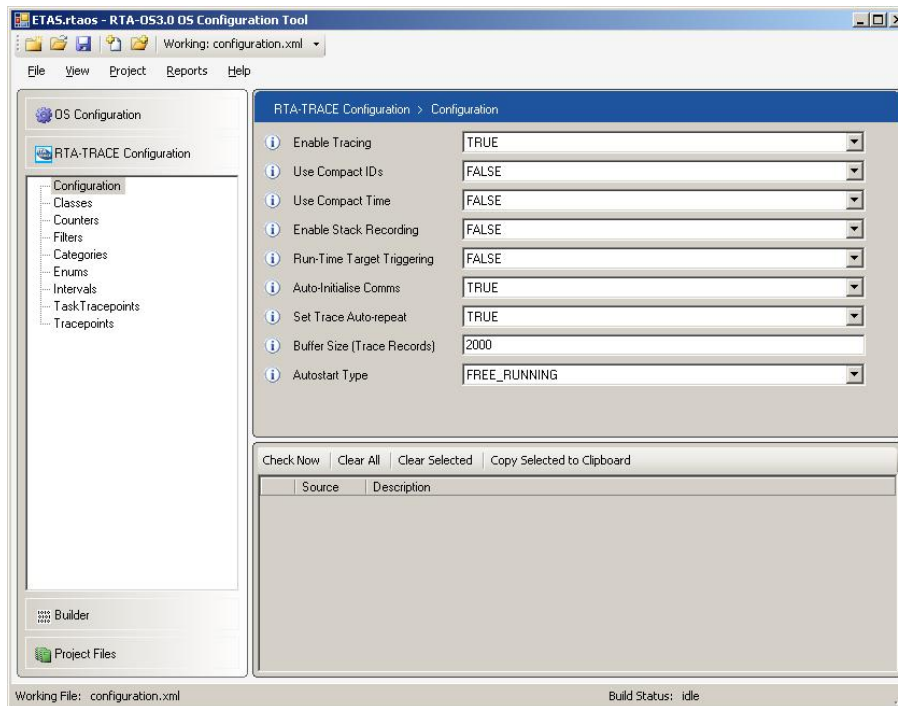


Figure 3.4: RTA-TRACE configuration in **rtaoscfg**

3.1.2 RTA-TRACE Configuration

The RTA-TRACE configuration editor, shown in Figure 3.4, allows you to configure RTA-OS3.0 to support ETAS' RTA-TRACE runtime profile and monitoring tool¹. Like the OS configuration, an overview of the RTA-TRACE configuration is displayed in the project navigator and configuration windows are shown in the workspace. You can expand each object to see the instances you have created. Clicking on an instance of an object displays the configuration panel in the right hand window.

Configuring RTA-TRACE tells RTA-OS3.0 to include all the necessary OS instrumentation that allows RTA-TRACE to gather runtime data for your application. There is no harm configuring RTA-TRACE instrumentation if you do not have the RTA-TRACE to view the trace data, but this will make your RTA-OS3.0 configuration larger and slower that it would be without instrumentation.

3.1.3 Build

The RTA-OS3.0 library is built using the command-line **rtaosgen** tool. If you prefer to configure and build within the same tool then you can do so from the build workspace shown Figure 3.5. The builder allows you to configure

¹RTA-TRACE is not supplied with RTA-OS3.0. For further information about RTA-TRACE contact your ETAS sales office.

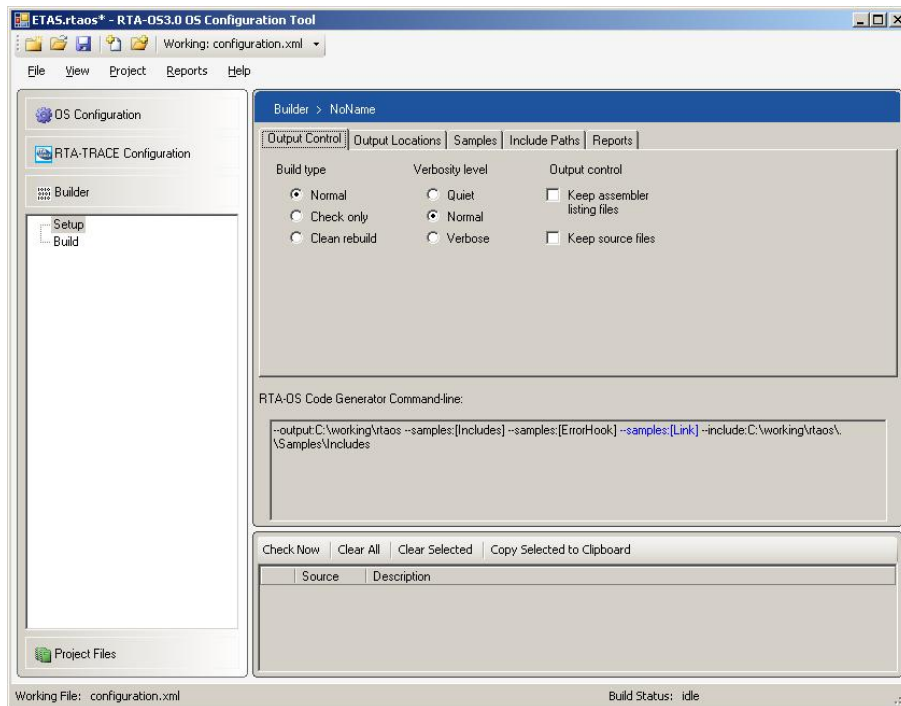


Figure 3.5: Running **rtaosgen** from **rtaoscfg**

the build process and run **rtaosgen** from inside **rtaoscfg**. Further details on **rtaosgen** are in Section 3.2.

3.1.4 Project Files

When you are working on an RTA-OS3.0 configuration, **rtaoscfg** calls the working configuration a project. A project is simply one or more XML files that define your OS configuration. By default, a project contains a single AUTOSAR XML file.

The AUTOSAR XML language allows you to partition your OS configuration across multiple files according to the demands of your build and/or version control process. This is essential if you are working with other tools that generate fragments of OS configuration that need to be integrated into your main configuration. This is also useful if you want to maintain a ‘core’ configuration and then have multiple customizations of that core for different target hardware.

The project viewer allows you to manage these complex projects. You can add/remove XML files to/from the project with ease. When you are working with a multi-file XML project, **rtaoscfg** internally merges all configuration data, allowing you to work with multiple configuration files simultaneously as if they are a single OS configuration. That way you can see the entire

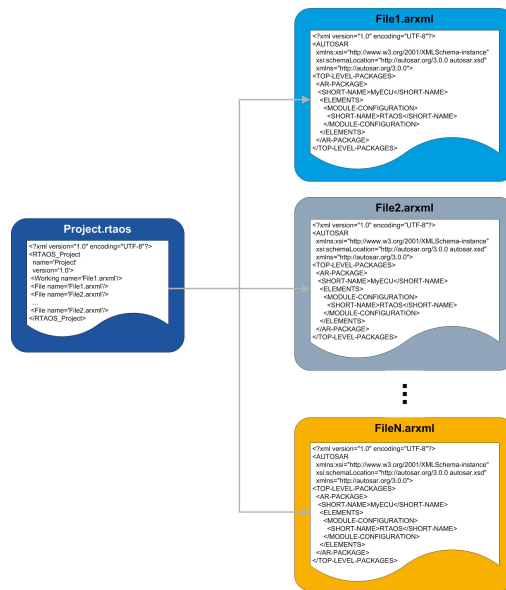


Figure 3.6: Relationship between Project File and AUTOSAR XML Files

model and check for consistency even though configuration data is physically separated.

When the project is saved, **rtaosgen** remembers which parts of configuration came from which XML files so that when you save your configuration each element of configuration data is written back to the correct file.

RTA-OS3.0 also writes out a project file with a `.rtaos` extension. A project file is a special RTA-OS3.0 file that lists all the AUTOSAR XML files in your project. Figure 3.6 shows the basic concept.

ETAS Project files are specific to RTA-OS3.0

The Working File

When you create a new OS object (e.g. a task) or an attribute (e.g. a task's priority) in your configuration, **rtaoscfg** writes it to the *working file*. The project viewer shows you the current working file and allows you to switch the working file to be any one in your project.

If you create an object or attribute and realize that you added it to the wrong file, then you can simply move it to another file in your project by deleting it from the current working file, changing to a new working file and then re-creating the object or attribute.

If you work with many XML file fragments as part of your OS configuration then it would be tiresome to open each in turn when you want to make modi-

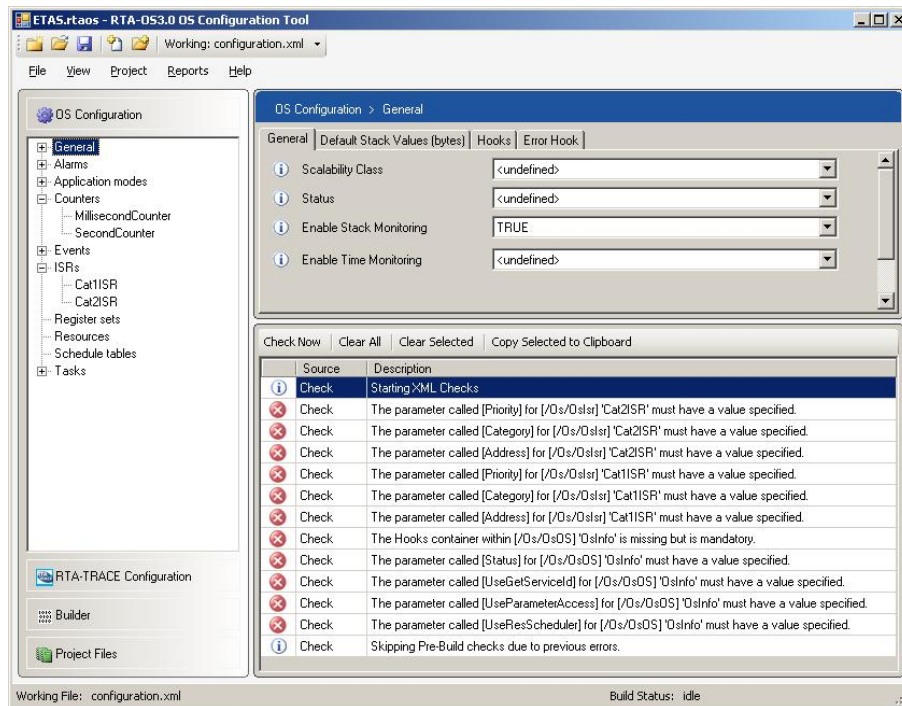


Figure 3.7: Viewing configuration errors

fications. **rtaosgen** allows you to open the project file itself. This loads every XML file referenced by the project file automatically.

3.1.5 Error Checking

When you add or remove configuration items **rtaoscfg** reports configuration errors in the Error Viewer.

Three types of errors are reported:

Information tells you summary information about the configuration, such as how many objects of a type you have configured.

Warnings tell you that your configuration might not behave as expected because you might be missing something in your configuration.

Errors tell you that parts of your configuration are incorrect.

You can check your configuration for consistency by clicking on the **Check Now** button. Any errors that are found are reported in the error window. Figure 3.7 shows how errors are reported .

3.1.6 Generating Reports

You can generate reports about your configuration from the menu/toolbar. Reports present summary information about your configuration that can be used for Quality Assurance audits, internal communication between departments, etc. The reports provided include:

Configuration Summary - an overview of the OS configuration.

OS API Reference - a customized reference guide for the configured OS. This may include documentation for target-specific features such as additional API calls and types that are not part of the standard *RTA-OS3.0 Reference Guide*.

Stack Usage - worst-case stack usage for the configured OS. This requires that stack allocations have been provided for each tasks and ISR.

MISRA Deviations - provides the MISRA-C 2004² deviations for the configured OS. This includes which deviations apply, why the deviation has been made and how many times it occurs.

All reports are generated in HTML and displayed in your default web-browser.

Generation of reports is actually done by **rtaosgen** so you can produce these at the same time as your build process runs, for example you may generate the MISRA Deviation report to provide evidence for your QA process. Build-time generation of reports allows the format for the report to be selected. Reports are provided as plain text, XML and HTML.

A full list of the reports available for your target can be obtained using the following command line:

```
rtaosgen --target:YourTarget --report:?
```

3.2 Library Generation

Before you can use RTA-OS3.0 in your application, you need to generate an RTA-OS3.0 kernel library and associated header files. **rtaosgen** generates a customized RTA-OS3.0 kernel library that is optimized for your OS configuration by:

- Analyzing your XML configuration and automatically optimizing the RTA-OS3.0 kernel so that it contains only those features that you will use. This makes RTA-OS3.0 as small and efficient as possible.

²Motor Industry Standards and Reliability Association

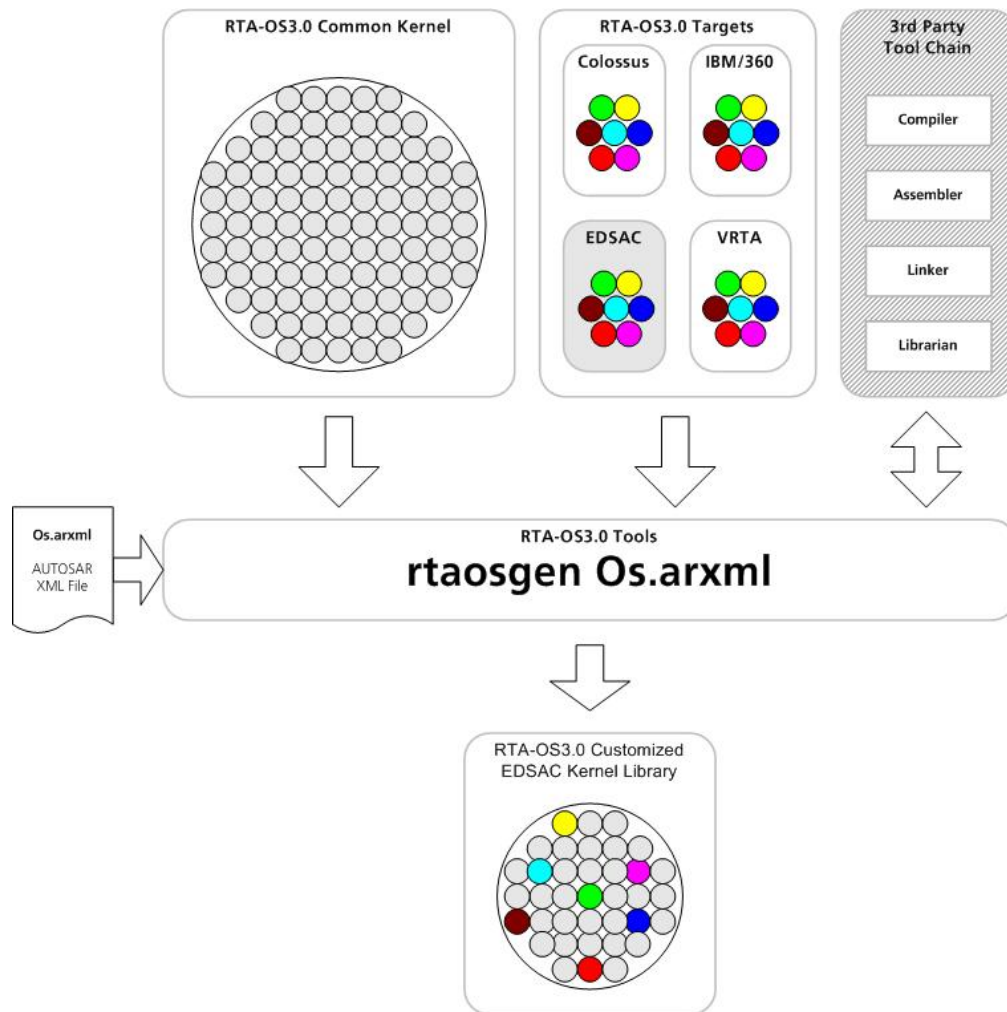


Figure 3.8: Building an RTA-OS3.0 Kernel Library

- Customizing the optimized kernel to your chosen target using information provided by the target plug-in.
- Building a kernel library using the same 3rd party tool chain that you are using for your application. This guarantees tool chain compatibility between RTA-OS3.0 and your own code.

The tool integrates the core OS kernel, enables optional kernel features you have selected and customizes this with the information about the target supplied by the port plug-in. Figure 3.8 illustrates the basic process by showing how a kernel would be generated for the EDSAC target.

3.2.1 Preparing the Tool Chain

To generate a kernel library, **rtaosgen** needs access to the compiler tool chain for your target. **rtaosgen** knows how to run the compiler, assembler and librarian for your target and what options to use. You need only be concerned with two things:

1. Your compiler tool chain must be accessible on your PATH.
2. Your compiler tool chain must be compatible with RTA-OS3.0.

You can find out if your compiler is on your PATH by opening a Windows Command Prompt using **Start → Run** and running **cmd**.

Typing `C:\>set` at the command prompt will list every Windows environment variable. You should see your compiler's executable directory on the path. If you do not, then you can add your compiler to the path by typing:

```
C:\> set PATH=PATH;<Path to your compiler executable>
```

To check whether you are using a compatible version of the compiler tool chain you should consult the *RTA-OS3.0 Target/Compiler Port Guide* for your port which will tell you which version (or versions) are compatible.

3.2.2 Understanding AUTOSAR Dependencies

RTA-OS3.0 is an AUTOSAR basic software module³ and as such it must conform to the AUTOSAR basic software module build concept. If you know how this works already then skip ahead to Section 3.2.4.

In AUTOSAR, all basic software modules provide a single include file called `<BSW_Short_name>.h`. For the OS this is `Os.h`. Each of these header files has dependencies on two other AUTOSAR include files:

Std_Types.h provides all the portable (i.e. target hardware invariant type definitions for AUTOSAR). `Std_Types.h` includes a further two AUTOSAR header files:

Platform_Types.h defines the AUTOSAR standard types (`uint8`, `uint16`, `boolean`, `float32` etc.) for the target hardware.

Compiler.h defines a set of macros that abstract compiler addressing models (e.g. banked and non-banked access, near and far pointer access, etc.). These macros are used internally by basic software

³One of many - there are other modules for communication stacks (CAN, LIN, FlexRay etc.), non-volatile memory handling, peripherals drivers, etc. that are available from third-parties.

modules to mark functions, data and pointers according to the mode by which they can be addressed.

Each basic software module defines a series of macros defining the classes and their mapping onto a specific compiler's primitives in a file called `Compiler_Cfg.h`. The system integrator must merge the `Compiler_Cfg.h` files from all basic software modules to create a "master" `Compiler_Cfg.h` *before the system is compiled*.

In RTA-OS3.0, the OS module's `Compiler_Cfg.h` is called `Os_Compiler_Cfg.h` so it can be **#included** in `Compiler_Cfg.h`. You can view what virtual sections are defined by RTA-OS3.0 by looking at the contents of `Os_Compiler_Cfg.h`.

You should take particular note of the RTA-OS3.0 section called `OS_APPL_CODE`. This should be used to place all application code that is required by the kernel, i.e. all your hooks and callback routines. Code can be placed in this section using the directive `FUNC(<typename>, OS_APPL_CODE)`. For example the following code shows how to place the `ErrorHook()` into `OS_APPL_CODE`:

```
FUNC(void, OS_APPL_CODE) ErrorHook(StatusType Error){
    /* Handle error */
}
```

The RTA-OS3.0 documentation always uses this form when defining callbacks and hook routines.

MemMap.h defines how data and code is mapped to memory sections and uses the compiler's primitives for placing code and data into different types of memory section according to the following process:

1. each basic software module defines a series of section names using macros (a minimum set of which are defined by the AUTOSAR standard). For the OS, all section name macros start with `OS_`. These names are the ones you can find in `Os_Compiler_Cfg.h`.
2. the vendor of the basic software module uses these macros to place code in the virtual sections during implementation, for example

```
#define OS_START_SEC_CODE
#include "MemMap.h"
/* Some OS code here */
#define OS_STOP_SEC_CODE
#include "MemMap.h"
```

3. the system integrator develops a `MemMap.h` file that maps the basic software's virtual section names onto system-wide section names and from there onto primitives of the compiler for section placement, for example:

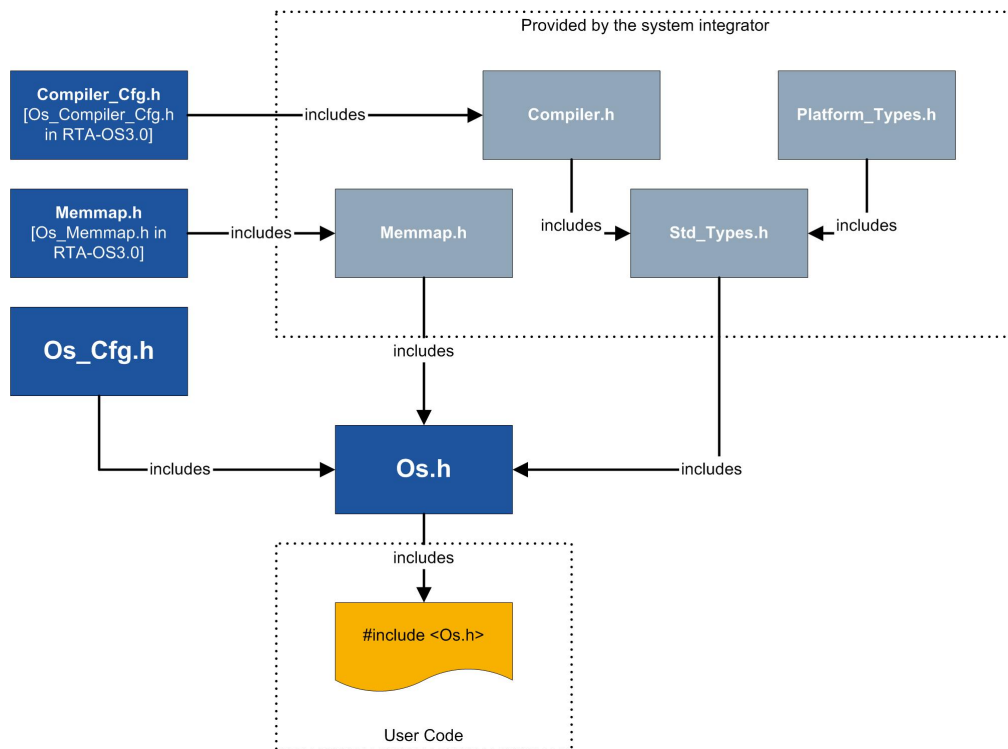


Figure 3.9: AUTOSAR Header File Hierarchy

```

/* Map OS code into the section containing all BSW
   code */
#ifdef OS_START_SEC_CODE
    #undef OS_START_SEC_CODE
    #define START_SECTION_BSW_CODE
#endif
...
/* Name the system section with a compiler
   primitive */
#ifdef START_SECTION_BSW_CODE
    #pragma section code "bsw_code_section"
#endif
  
```

As with `Compiler_Cfg.h`, each basic software module must also provide a `MemMap.h` that the system integrator can merge together to create a “master” `MemMap.h` before the system is compiled.

The include hierarchy is shown in Figure 3.9

To build an RTA-OS3.0 library it follows that all the standard AUTOSAR header files are required as inputs to the build process and these are outside the

scope of the OS. However, RTA-OS3.0 can generate *sample* versions of the AUTOSAR standard header files if required.



*You must enhance or replace the sample AUTOSAR standard header files generated by **rtaosgen** for use in production software.*

3.2.3 Running **rtaosgen**

rtaosgen is a command line tool. You can invoke it from the Windows command prompt, from a make script, Ant script, in fact from anywhere where you can call a Windows application. The **rtaosgen** tool can be run from the **rtaoscfg** Builder if you prefer to use a graphical environment.

rtaosgen takes one or more configuration files as input. Configuration files can be:

- AUTOSAR XML
- RTA-OS3.0 project files
- A mixture of both

Like **rtaoscfg**, **rtaosgen** will merge the contents of all files passed on the command line into an in-memory OS configuration before generating the kernel.

3.2.4 Building the library

To build an RTA-OS3.0 library it follows that all the standard AUTOSAR header files are required as inputs to the build process. You must include the path to the location of the AUTOSAR standard headers files when invoking **rtaosgen**. For example, to build a library for the “Hello World” example application for an RTA-OS3.0 target you can type:

```
C:\>rtaosgen --include:PathToAutosarHeaderFiles HelloWorld.  
rtaos
```

If you do not have access to AUTOSAR include files (for example, if you are using RTA-OS3.0 outside of a full AUTOSAR system), then **rtaosgen** can generate them automatically for you.

```
C:\>rtaosgen --samples:[Includes] --include:Samples\Includes  
HelloWorld.rtaos
```

Note that **rtaosgen** does not force you to use a specific extension - you can use any extension you like.

rtaosgen generates four classes of messages during execution:

Information. These messages tell you useful things about the configuration, for example how many tasks you have configured. **rtaosgen** will generate output files.

Warning. These messages warn you that your configuration will result in an OS that might not behave as you expect. Rtaosgen will generate output files.

Error. These messages tell you that there is something wrong with your configuration. **rtaosgen** will stop processing your configuration at a convenient point and no output files will be generated.

Fatal. You will get at most one fatal message. It tells you that there is something fundamentally wrong with either your configuration or rtaosgen. **rtaosgen** stops immediately.

You can do other things from the command line like change the output directory for generated files, suppress messages, etc. For more details, see the *RTA-OS3.0 Reference Guide*.

Building from **rtaoscfg**

It is also possible to build RTA-OS3.0 from within **rtaoscfg** in the “Builder” workspace. This allows you to run **rtaosgen** with command line you specify **rtaoscfg**. If you specify a command line, then it will be saved in the RTA-OS3.0 project file.

The same tool, **rtaosgen** is used to build the kernel irrespective of whether it is called directly from the command line or internally from **rtaoscfg**.

Note that if you build from within **rtaoscfg** then you still need to ensure that your compiler tool chain is on your Windows path.

Any settings that you configure in the “Builder” are stored in your RTA-OS3.0 project file.

3.2.5 Generated Files

When **rtaosgen** runs and terminates without reporting any errors or fatal messages then it will have generated the following files:

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-OS3.0 to merge with the system-wide MemMap.h file.
RTAOS.<lib>	The RTA-OS3.0 library for your application. The extension <lib> depends on your target.
RTAOS.<lib>.sig	A signature file for the library for your application. The extension <lib> depends on your target.

Generating sample code

The **rtaosgen** code generator can generate sample code that can be used as a basis for your application. You have already seen one case of this in Section 3.2.4 when `--samples[Includes]` was used to generate sample AUTOSAR standard header files.

The set of samples provided is port-dependant, but you can get a list of provided samples using the following command line:

```
C:\>rtaosgen --target:YourTarget --samples:?
```

Most ports will provide samples that show how to write AUTOSAR OS hook functions like the `ErrorHook()`, `StartupHook()` etc. For example, to generate a default `ErrorHook()` you could use the following command line:

```
C:\>rtaosgen --samples:[ErrorHook] --include:
    PathToAutosarHeaderFiles HelloWorld.rtaos
```

3.3 Integration

3.3.1 Accessing the OS in your Source Code

To access RTA-OS3.0 in your source code you simply include **#include** `<Os.h>` in every C compilation unit (i.e. every C source code file) where you need to access RTA-OS3.0. The header file is protected against multiple-inclusion. RTA-OS3.0 does not place any restrictions on how you organize your source code - you can put all of your code into a single source file or place each task and interrupt implementation into its own source file as you (or your configuration control process) requires.

3.3.2 Implementing Tasks and ISRs

Tasks

For each task that you declare at configuration time you must provide an implementation of the task. Each task needs to be marked using the TASK(x) macro. Tasks typically have the following structure:

```
#include <Os.h>
TASK(MyTask){
    /* Do something */
    TerminateTask();
}
```

Category 2 ISRs

Each Category 2 ISR that you declare needs to be implemented. This is also marked, this time by ISR(x):

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



A Category 2 ISR handler does not need a return from interrupt call - RTA-OS3.0 does this automatically. Depending on the behavior of interrupt sources on your target hardware, you may need to clear the interrupt pending flag. Please consult the hardware documentation provided by your silicon vendor for further details.

Category 1 ISRs

Each Category 1 ISR that you declared also needs to be implemented. Your compiler will use a special convention for marking a C function as an interrupt. RTA-OS3.0 provides a macro that expands to the correct directive for your compiler. Your Category 1 handler will therefore look something like this:

```
CAT1_ISR(MyCat1ISR) {
    /* Do something */
}
```

3.3.3 Starting the OS

RTA-OS3.0 does not take control of your hardware so you need to start the OS manually using the StartOS() API call, usually in your main() program. RTA-OS3.0 provides a macro called OS_MAIN() which expands to the correct type of main() definition for your compiler toolchain⁴.

⁴On many compilers this will be **void** main(**void**), but there are compilers that insist upon the main() program returning an integer or other (non **void**) type.


```

#include <Os.h>
OS_MAIN() {
    /* Initialize target hardware */
    /* Do any mode management, pre-OS functions etc. */
    StartOS();
    /* Call does not return so you never reach here */
}

```

3.3.4 Interacting with the RTA-OS3.0

You interact with RTA-OS3.0 by making kernel API calls. You can find a complete list of calls in the *RTA-OS3.0 Reference Guide*.

3.3.5 Compiling and Linking

When you compile your code you must make sure that `Os.h` and `Os_Cfg.h` are reachable on your compiler include path. When you link your application you must link against `RTAOS.<lib>`, and the library must be on your linker's library path.

3.4 Memory Images and Linker Files

When you build your application, the various pieces of code, data, ROM and RAM that were placed into the sections defined in `MemMap.h` need to be located at the right place in memory. This is typically done by your linker⁵ which resolves references made by user-supplied code to the RTA-OS3.0 library, binds together the relevant object modules and allocates the resultant code and data to addresses in memory before producing an image that can be loaded onto the target.

The linker needs to know what parts of the program to place in which types of memory, where the ROM and RAM are on the microcontroller, and how map the parts of the program to the correct sort of memory.

3.4.1 Sections

Code and data output by compilers and assemblers is typically organized into "sections". Some sections will contain just code, some code and data and some will contain data only. You might see a piece of assembler that says something like that shown in Code Example 3.1.

```

.section CODE
.public MYPROC
mov     r1, FRED

```

⁵An historical note: Technically this job is that of the locator which locates sections into memory by mapping virtual to physical addresses and these tools used to be called linker/locators. In modern times the locator part has dropped out of common usage and the tools are commonly referred to as 'linkers'.

```

add    r1, r1
ret
.end CODE
.section DATA
.public FRED
.word 100, 200, 300, 400
.end DATA
.section BSS
.public WORKSPACE
.space 200
.end BSS

```

Code Example 3.1: Example Assembler Output Showing Sections

This means that the code for MYPROC should be assembled and the object code should assume that it will be located in a section of memory called CODE whose location we will specify later in the linker control file. Similarly, the data labeled FRED will be placed in a section called DATA, and a space of 200 bytes labeled WORKSPACE will be allocated in section BSS.

C compilers typically output your code into a section called code or text, constants that must go into ROM in a section called something like **const**, and variables into data. There will usually be more - consult the reference manual for your toolchain for more details on what the sections are called and familiarize yourself with where they need to go.

Under AUTOSAR, your MemMap.h will define the actual names of the sections that need to be located. For example, so far we have yet to map these onto addresses in “real” memory. We must therefore look at how these sections are mapped into a memory image.

“Near” and “Far” space

On some processors there exist regions of memory space that can be addressed economically (typically with shorter, smaller instructions that have simpler effective-address calculations), are located on-chip rather than off-chip, or that are fabricated in a technology such that they are more cycle-efficient to access. RTA-OS3.0 terms this memory “near” space and on these processors places some key data in these areas. On such platforms you will be supplied with information on where you must locate “near” space in ROM and/or RAM, and told in the *RTA-OS3.0 Target/Compiler Port Guide* what data is placed in it. “Far” space refers to the whole of memory.

Program and Data Space on Harvard Architectures

Most of the discussion about memory so far has assumed the conventional “von Neumann” architecture, in which data and code occupy one address

space with ROM and RAM located at different offsets inside this. Some processors (typically very small microcontrollers like PICs, or high-performance Digital Signal Processors) adopt a “Harvard” architecture, in which there are distinct address spaces for code and data (there are some performance advantages to this that offset the programming disadvantages). On a Harvard-architecture processor, RTA-OS3.0 may use data space (typically RAM) to store data that would normally be ROM constants on a von Neumann architecture processor, and the startup code will typically contain code to fetch a copy of the constant data into data space. If you are using a Harvard architecture processor, the *RTA-OS3.0 Target/Compiler Port Guide* will contain information on any use of RAM used to store copies of constants.

3.4.2 The Linker Control File

The linker control file governs the placement of code, data and reserved space in the image that is downloaded to the target microcontroller. Linker files vary considerably between platforms and targets, but typically include at least the following:

- declarations of where ROM and RAM are located on chip - these may vary across different variants in a CPU family.
- Lists of sections that can be placed into each memory space
- Initialization of the stack pointer, reset address, interrupt vectors etc.

Code Example 3.2 shows a hypothetical linker control file:

```
ONCHIPRAM start 0x0000 {
    Section .stack size 0x200 align 16 # system stack
    Section .sdata align 16           # small data
    Section bsw_near align 16         # near data
}

def __SP = start stack                # initialize stack ptr

RAM start 0x4000 {
    Section .data align 16             # compiler data
    Section .bss align 16             # compiler BSS
    Section bsw_zero_init align 16    # Basic Software zeroed
    RAM
    Section bsw_startup_init align 16 # Basic Software
    initialized RAM
    Section swc_startup_init align 16 # Application initialized
    RAM
}
```

```

ROM start 0x8000 {
    Section .text                # compiler code
    Section .const               # compiler constants
    Section swc_data align 16    # Application static data
    Section swc_init align 16    # Application initial data
    Section bsw_init align 16    # Basic Software initial
    data
}

VECTBL start 0xFF00 {
    Section 0sVectorTable        # RTA-OS's vector table
}

def __RESET = __main           # reset to __main

```

Code Example 3.2: A Linker Control File

The file above defines four separate parts of memory - ONCHIPRAM, RAM, ROM, and VECTBL. Into each section are placed the appropriate data, as described by the comments.

The example applications supplied with RTA-OS3.0 embedded ports will contain a fully-commented linker control file; consult this and the *RTA-OS3.0 Target/Compiler Port Guide* for details of how to locate the sections correctly for your target platform.

3.5 Summary

- There are 5 steps to integrate RTA-OS3.0 with your application:
 1. Configure the features of the OS you want to use
 2. Generate a customized RTA-OS3.0 kernel library
 3. Write application code that uses the OS
 4. Compile your application code and linking with the RTA-OS3.0 library
 5. Run your application on your target
- There are two offline tools: **rtaoscfg** to configure RTA-OS3.0 and **rtaosgen** to generate and build the kernel library
- RTA-OS3.0 is an AUTOSAR Basic Software module and has dependencies to AUTOSAR header files. These can be generated by **rtaosgen** if required.
- Linking and locating of RTA-OS3.0 depends on the content of the MemMap.h file with which **rtaosgen** builds the kernel library.

4 Tasks

A system that has to perform a number of different activities at the same time is known as concurrent. These activities may have some software part, so the programs that provide them must execute concurrently. The programs will have to cooperate whenever necessary, for example, when they need to share data.

Each concurrent activity in a real-time system is represented by a task. The majority of the application code exists within tasks. If you have a number of tasks that must be executed at the same time, you will need to provide a means to allow concurrency. One way for you to do this is to have a separate processor for each task. You could use a parallel computer, but this solution is too expensive for many applications.

A much more cost effective way for you to achieve concurrent behavior is to run one task at a time on a single processor. You can then switch between tasks, so that they appear to be executing at the same time.

4.1 Scheduling

RTA-OS3.0 provides a scheduler that switches between tasks according to a fixed priority which is assigned at configuration time. A priority is just a reflection of the relative urgency of tasks. There are many schemes that you can use to assign priorities to tasks, but common ones you may have heard of are:

Deadline Monotonic Assignment where you allocate higher priorities to tasks with shorter deadlines.

Rate Monotonic Assignment where you allocate higher priorities to tasks that you need to run the most frequently.

However you choose to assign priorities, the sequence in which your tasks execute is determined by a scheduling policy. The scheduling policy determines when tasks actually run.

AUTOSAR OS supports two scheduling policies:

1. Preemptive Scheduling.

The fixed priority preemptive scheduling algorithm is simple: run the highest priority task that is ready to run. If a task is running and a higher priority task becomes ready to run, then the higher priority task preempts the running task. This is called a task switch. When the higher priority task has finished then the preempted task resumes.

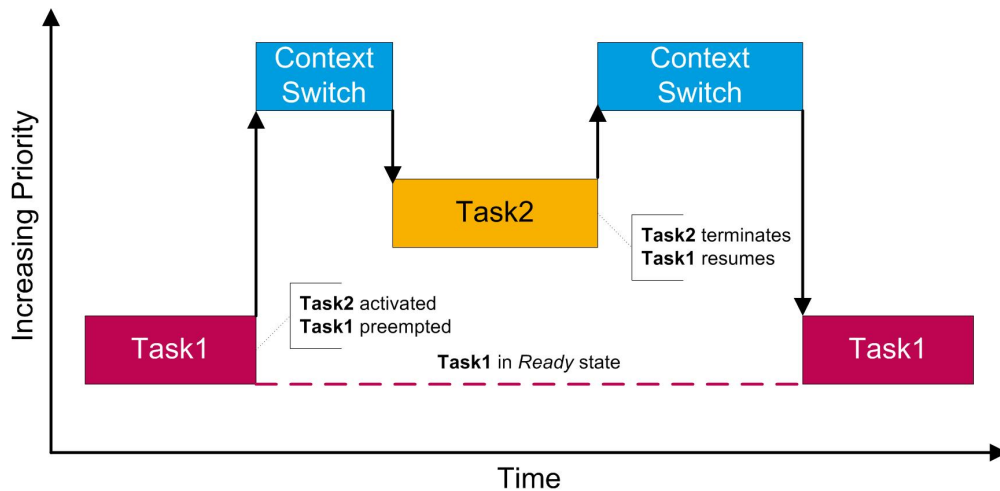


Figure 4.1: Preemptive scheduling of tasks

For a system where all tasks need to meet their deadlines at runtime, preemptive scheduling is the most efficient scheduling policy and will guarantee the shortest time between a task being activated (made ready to run) and terminating. This time is called the response time for the task. Preemptively scheduled systems need to consider the effect of preemption on shared data and may need to introduce mechanisms for concurrency control (see Chapter 6).

2. Non-Preemptive scheduling.

The OS runs the highest priority task that is ready to run, as with preemptive scheduling. However, unlike preemptive scheduling, if a higher priority task becomes ready, then it remains ready to run until the running task terminates - it does not preempt. What this means is that a non-preemptive task that starts running will always run to completion and then terminate.

Non-preemptive scheduling results in a less responsive system than preemptive scheduling (i.e. tasks will usually have longer response times), but the system does not need to worry about concurrency problems that arise for accessing shared data because the scheduling model doesn't allow concurrent access to shared data.

Actually, AUTOSAR OS provides support for a third type of scheduling called cooperative scheduling because it allows a non-preemptive task to tell the OS when it could be preempted. The reason we said AUTOSAR OS supports 2 policies is that there is only configuration for two - the third you have to build yourself.

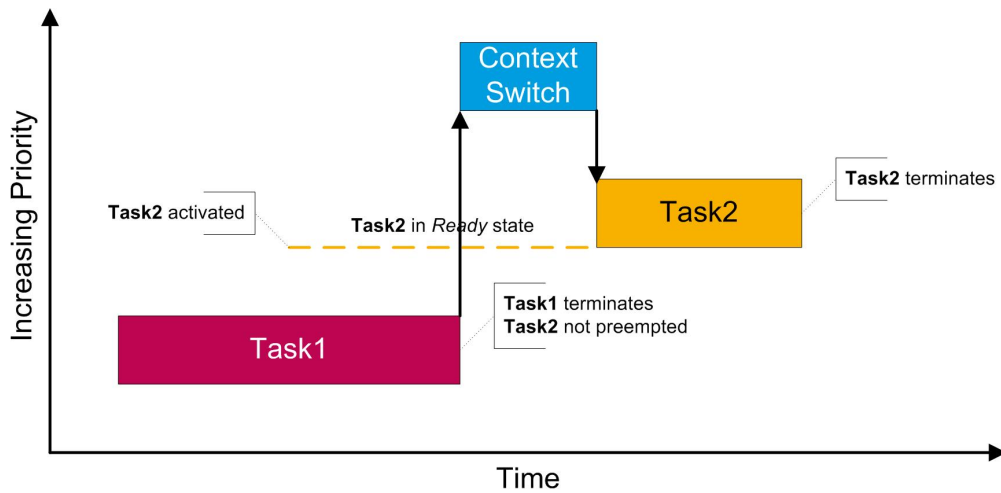


Figure 4.2: Non-preemptive scheduling of tasks

3. Cooperative scheduling.

The OS runs the highest priority task that is ready to run. If a higher priority task becomes ready, then it remains ready to run until either: the running task terminates (just like non-preemptive scheduling); or the running task makes a `Schedule()` API call to tell the OS it can be preempted. When the `Schedule()` call is made then the higher priority task preempts the running task and a task switch is said to have occurred (just like preemptive scheduling). When the higher priority task has finished then the preempted task resumes.

With careful design, the co-operative model provide can provide systems that, while not as responsive as fully preemptive systems, do not suffer the lack of responsiveness found with non-preemptive scheduling.

With all these types of scheduling it is important to realize that any task, whether preemptive or not, can be interrupted (preempted) by an interrupt service routine. Chapter 5 provides more information about how RTA-OS3.0 deals with interrupts.

4.2 Basic and Extended Tasks

RTA-OS3.0 OS supports two types of task:

1. Basic tasks.

Basic tasks start, execute and terminate (this is often called a single-shot tasking model). A basic task only releases the processor if it terminates, or if it is preempted by a higher priority task. This behavior

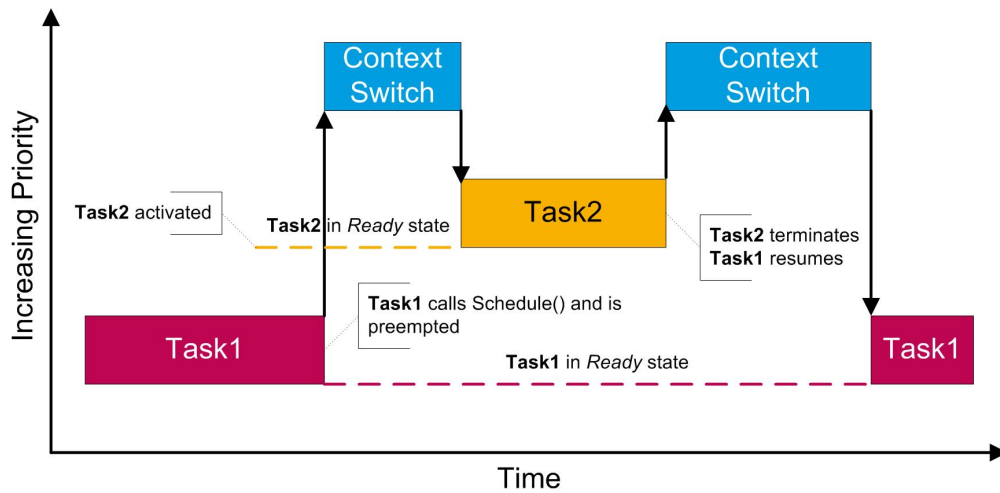


Figure 4.3: Co-operative scheduling of tasks

makes them highly suitable for embedded control functionality. Basic tasks are fast and efficient.

2. Extended tasks.

Extended tasks start, execute, wait for events and (optionally) terminate. The ability for an extended task to voluntarily suspend itself during execution provides a way for the task to have synchronization points. This feature makes extended tasks more suitable for functionality requiring mid-execution synchronization (for example, waiting for user interaction) than basic tasks.

4.2.1 Task States

Basic tasks operate on a 3-state model. A basic task can exist in the following states:

1. Suspended.
2. Ready.
3. Running.

Extended tasks can have an extra state which they enter when waiting for events:

4. Waiting.

Figure 4.4 shows the 3 and 4 state task models.

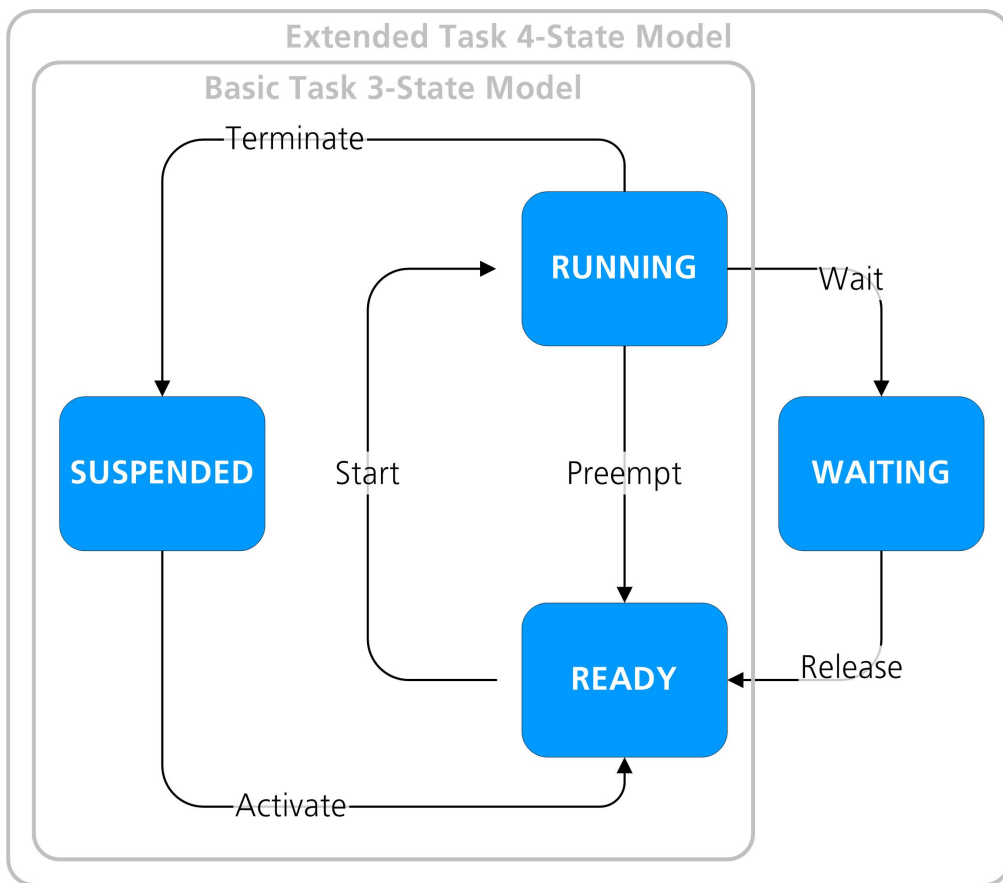


Figure 4.4: Task State Model

The default state for all tasks is suspended. A task is moved into the ready state by the process of activation. It is important to understand that activation does not cause the task to run - it makes it ready to run. Activation can happen a number of ways, for example by calling the `ActivateTask()` API in your code or as a result of some trigger, such as the expiry of an alarm (see Chapter 9) or a schedule table expiry point (see Chapter 10).

When a task becomes the highest priority task in the system, RTA-OS3.0 moves the task into the running state and starts task execution at the first statement in the task. This is often referred to as dispatching the task. A task may be preempted during execution by other higher priority tasks that become ready.

If a higher priority task becomes ready to run, the currently executing task is preempted and is moved from the running state into the ready state. This means that only one task can be in the running state at any one time.

A task returns to the suspended state by terminating. A task can be made ready again later and the whole process can repeat.

Basic and extended tasks behave identically with respect to the ready, running and suspended states. Extended tasks, however, can also enter the waiting state. An extended task moves from the running to the waiting state when it voluntarily suspends itself by waiting on an event.


An event is simply an OS object that is used to provide an indicator for a system event. Examples of events include data becoming ready for use or sensor values being read. You can find out more about events in Chapter 7.

When an extended task enters the waiting state, then the OS will dispatch the highest priority task that is ready to run. When an event is set, the task is moved from the waiting to the ready state. Note that extended tasks return to the ready state and not the running state. This is because, during the time that the extended task was in the waiting state, some other higher priority task may have been activate and then dispatched.


4.2.2 Task Priorities

AUTOSAR OS allows tasks to share priorities. When tasks have the same priority, each task at the shared priority will run in mutual exclusion from each other. This means that if one task is running, then its execution will be serialized with all other tasks that share the same priority.

When tasks share priorities they are released from the ready state in first-in, first-out (FIFO) order.

 *When shared priorities and queued task activation are used together, RTA-OS3.0 maintains an internal queue at the priority level. You should avoid this type of configuration if you want a fast and efficient OS.*

If you need to serialize the execution of a set of tasks, then this is best achieved using unique priorities and AUTOSAR OS's internal resources (see Section 6.5) rather than sharing task priorities. Using internal resources guarantees serialization, just like sharing priorities, and the uniqueness of task priorities means that when multiple tasks become ready at the same time the OS has a statically defined dispatch ordering

 *Sharing priorities between tasks is bad real-time programming practice because it prevents you from performing schedulability analysis on your system. This is because, in the general case, sharing priorities makes the release point for a task (i.e. the point from where a response time is measured) computationally impossible to calculate. If it is impossible to work out when the release occurs then it is impossible to decide if the task will meet its deadline!*

4.2.3 Queued Task Activation

Under most circumstances you will only activate a task when it is in the suspended state. In fact AUTOSAR OS treats the activation of a task while it is in the ready, running or waiting states as an error case.

However, there are some situations where you may need to implement a system where the same task must be activated a number of times but the shortest time between successive activations can be less than the time needed to run the task. For example, you might be unpacking CAN bus frames in a task and need to handle transient bursting of frames on the network.

This means you need to queue task activations at run time. AUTOSAR OS allows you to queue the activation of basic tasks to help you build this kind of application. Like other things in AUTOSAR OS the size of the task queue is statically configured. You must specify the maximum number of activations that can be pending for the task.

If the queue is already full when you try and activate the task then this will be handled as an error and the activation will be ignored.

Of course, you might have tasks that share priorities and use queued activation. In this case, tasks are queued in FIFO order in a queue with a length equal to the sum of the queue lengths for each task that shares the same priority. However, each task can only use up to its own number of entries.

4.3 Conformance Classes

You know now that tasks can:

- Be basic or extended
- Can share priorities
- Can queue activations.

However, AUTOSAR OS places some restrictions on what kind of features be used together. These are called Conformance Classes and are used to group task features for ease of understanding, enable partial implementations of the standard and provided scalability for different classes of application.

AUTOSAR OS has four conformance classes:

BCC1 - Basic tasks, unique priority and no queued activation.

BCC2 - Basic tasks, shared priorities and/or queued activation.

ECC1 - Extended tasks, unique priority and no queued activation. An ECC1 task is like a BCC1 task, but it can wait on events.

ECC2 - Extended tasks, shared priorities and no queued activation. Note that, unlike BCC2 tasks, ECC2 tasks cannot queue activations.

The following table gives a quick summary of the types tasks that can be used in different classes of AUTOSAR OS system:

System Class	Basic Tasks	Extended Tasks	Shared Task Priorities	Queued Task Activation
BCC1	✓	X	X	X
BCC2	✓	X	✓	✓
ECC1	✓	✓	X	X
ECC2	✓	✓	✓	✓ ¹

Each conformance class requires more resources - a system that is BCC1 will be much faster and smaller than a system which is ECC2. You do not need to be concerned about which conformance class to use - RTA-OS3.0 supports all conformance classes and will calculate the conformance class from your OS configuration.

4.4 Maximizing Performance and Minimizing Memory

RTA-OS3.0 is designed to be very aggressive at minimizing code and data usage on the target application. It will analyze the characteristics of the application and generate a system containing only the features that are required.

¹But only for basic tasks within the ECC2 system. Activations of extended tasks cannot be queued.

Your choice of task characteristics has a major influence on the final application size and speed. There is “no such thing as a free lunch”, so as you add tasks to your application that use more advanced types of tasks, the system will inevitably become slightly larger and slower.

A system with one or more BCC2 tasks has a greater overhead than one with only BCC1 tasks. A system without shared priorities, even if multiple activations are allowed, will be more efficient than one with shared priorities.

A system with ECC1 tasks has an even greater overhead still and a system with one or more ECC2 tasks has the largest overhead of all.

To make RTA-OS3.0 as efficient as possible you should use basic tasks only and not share priorities.

4.5 Task Configuration

Unlike other real-time operating systems that you might have seen, the tasks in AUTOSAR OS (and, therefore, RTA-OS3.0) are defined statically. This technique is used because it saves RAM and execution time.

Tasks cannot be created or destroyed dynamically. Most of the information about a task can be calculated offline, allowing it to be stored in ROM.

The maximum number of tasks supported by RTA-OS3.0 depends upon your port and you should consult the *RTA-OS3.0 Target/Compiler Port Guide* for further details. For all ports, RTA-OS3.0 can provide a highly optimized system if you limit your number of tasks to the native word size of your microcontroller.

Device Type	Maximum	Optimal
8-bit	256	16 or fewer
16-bit	256	16 or fewer
32-bit	1024	32 or fewer

When you configure your task properties, you will most likely use the `rtaoscfg` configuration tool. Figure 4.5 shows the task configuration entry.

An AUTOSAR task has 5 attributes:

Name. The name is used to refer to, or provide a handle to, C code that you will write to implement the task functionality.

Priority. The priority is used by the scheduler to determine when the task runs. Priorities cannot be changed dynamically. Zero is the lowest possible task priority in RTA-OS3.0. Higher task priorities are represented by larger integers. Tasks can share priorities, but if you are building a real-time system, then you should not do this because it cannot be

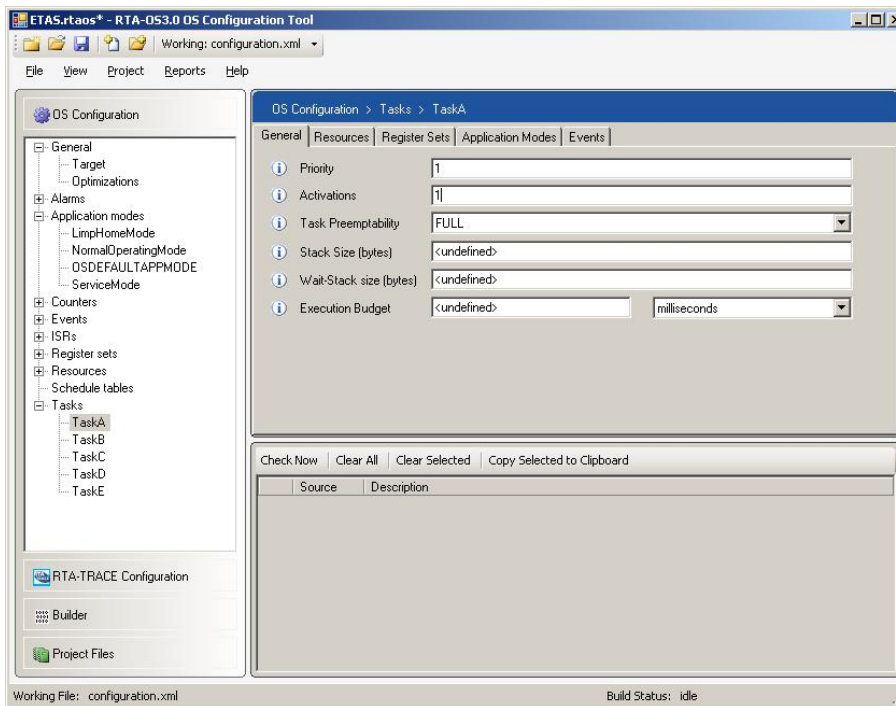


Figure 4.5: Task Configuration

analyzed.

Scheduling. A task can run fully preemptively or non-preemptively. In general, fully preemptive scheduling should be selected over non-preemptive scheduling for best application performance.

Activations. The maximum number of task activations that can be queued in the ready state. For a BCC1, ECC1 and ECC2 tasks the number of activations is always one. This means that these types of task can only be activated if they are in the suspended state. Any attempt to activate a such a task when it is not suspended will result in an error. A value greater than one indicates that the OS will queue activations (for example to smooth out transient peak loads in your application).

Autostart. This controls whether the task is started automatically when you start the OS.



The number of tasks that can be defined is fixed for each target (it is usually 256 or 1024, depending on the target processor). The RTA-OS3.0 Target/Compiler Port Guide for your target will contain further information.

4.5.1 Scheduling Policy

A fully preemptable task can be preempted by a task of higher priority. That means that when a higher priority task is made ready to run, it will run in preference.

You can prevent a task from being preempted by declaring it to be non-preemptable at configuration time. Tasks that are declared as non-preemptive cannot be preempted by other tasks. When a non-preemptive task moves to the running state it will run to completion and then terminate (unless it makes a `Schedule()` call, as explained in Section 4.10). Making tasks non-preemptive therefore means that if a lower priority task is started before a higher priority task, then the higher priority task will be prevented from executing for the time that the lower priority task runs. This is called blocking. Systems that use non-preemptive tasks will, in general, be less responsive than systems that run preemptively.

Even if a task is non-preemptive, it can still be interrupted by ISRs.

You will often find that it is unnecessary to use non-preemptable tasks because there are other, more suitable methods, which you can use to achieve the same effect. If you use these other techniques, it will usually result in a more responsive system. You will find out more about these techniques later, but they include:

- Using standard resources to serialize access to data or devices.
- Using internal resources to specify exactly which other tasks cannot cause preemption.

4.5.2 Queued Activation

Under most circumstances you will only activate a task when it is in the suspended state. However, you may need to implement a system where the same task must be activated a number of times and where the shortest time between successive activations is less than the time needed to run the task.

If this happens you will be activating the task while it is in the ready state or the running state. This means that activations will be lost.

To prevent loss of activations, you must specify the maximum number of multiple activations required for the task.



In accordance with the AUTOSAR OS standard, this feature is only available for basic tasks. You cannot specify multiple activations for extended tasks.

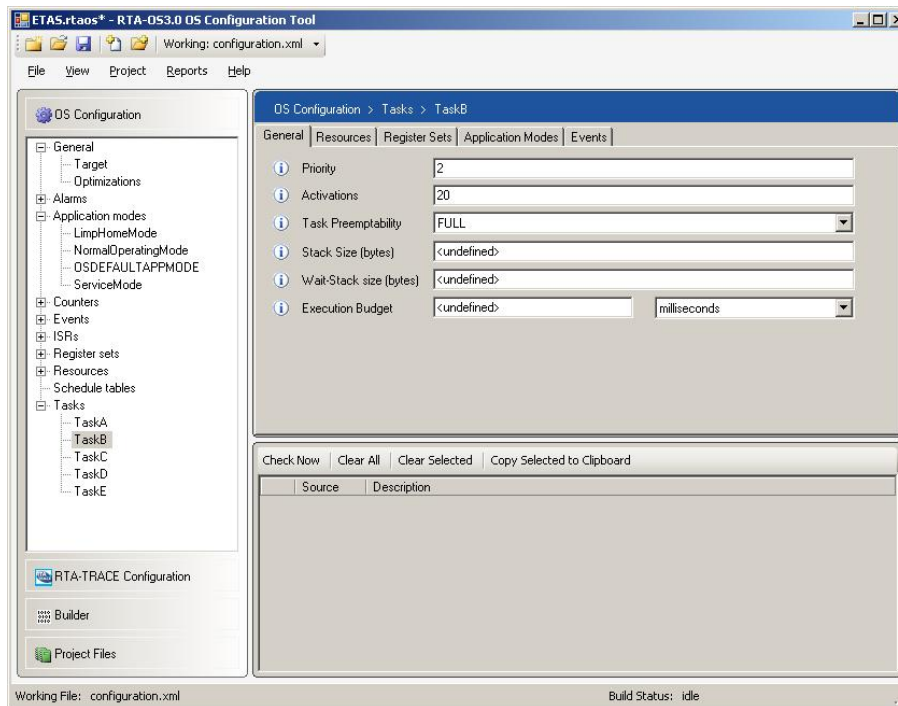


Figure 4.6: Specifying the number of queued activations

You will use `rtaoscfg` to specify the maximum number of simultaneous task activations. Figure 4.6 shows that, for the task in this example, the maximum number of activations has been set to ten.

When multiple activations are specified, RTA-OS3.0 automatically identifies that the task is BCC2. When you build your application, RTA-OS3.0 will calculate the maximum size of the multiple activation queue needed for each BCC2 task.

When BCC2 tasks share priorities, RTA-OS3.0 uses a FIFO queue to hold pending activations. If a BCC2 task has a unique priority in your AUTOSAR OS application then RTA-OS3.0 automatically optimizes the queuing strategy to counted activation. Counted activation is significantly more efficient than FIFO activation and should be used wherever possible.

4.5.3 Auto-starting Tasks

Tasks can be auto-started, which means that when the operating system starts, they are activated automatically during `StartOS()`.

For basic tasks, which start, run and then terminate, auto-starting a task will make it run exactly once before it will return to the suspended state (from where it can be activated again). Auto-starting is mainly useful for starting extended tasks that wait on events because it removes the need to write

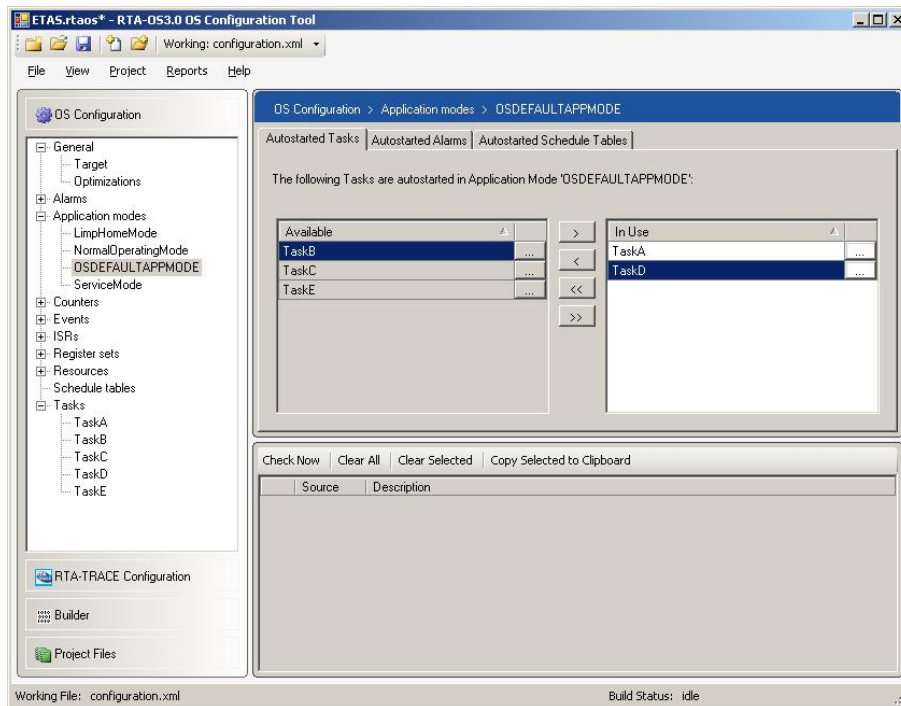


Figure 4.7: Configuring auto-started tasks

code to activate the tasks.

rtaoscfg can be used to specify that a task is only auto-activated in specific application modes, choose the application mode in question and select the tasks that you want to auto activate.

In Figure 4.7, TaskA and TaskD are auto-started in the OSDEFAULTAPPMODE application mode but tasks TaskB, TaskC and TaskE are not.

4.6 Stack Management

RTA-OS3.0 uses a single-stack model which means that all tasks and ISRs run on a single stack. The single stack is simply the C stack for the application.

As a task runs, its stack usage grows and shrinks as normal. When a task is preempted, the higher priority task's stack usage continues on the same stack (just like a standard function call). When a task terminates, the stack space it was using is reclaimed and then re-used for the next highest priority task to run (again, just as it would be for a standard function call). Figure 4.8 shows how the single stack behaves as tasks are stated, preempted and terminate.

In the single stack model, the stack size is proportional to the number of priority levels in the system, not the number of tasks/ISRs. This means that

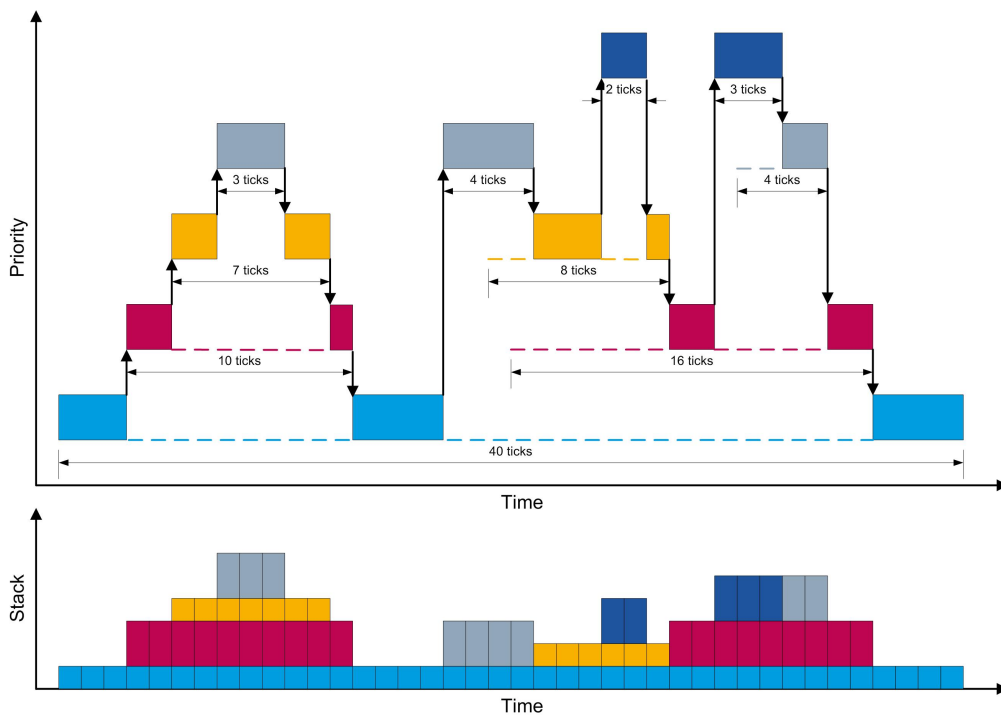


Figure 4.8: Single-stack behavior

tasks which share priorities, either directly, or by sharing internal resources, or through being configured as non-preemptive, can never be on the stack at the same time. The same is true of ISRs that share priorities in hardware. This means that you can trade system responsiveness, i.e. how long it takes for a task or ISR to complete, for stack space by simple changes to configuration.

Figure 4.9 shows the execution of the same task set, with the same arrival pattern as Figure 4.8 but this time the tasks are scheduled non-preemptively. You can see that the response times for the higher priority tasks are much longer than when they were preemptively scheduled but the overall stack consumption is much lower.

The single stack model also significantly simplifies the allocation of stack space at link time as you need only allocate a single memory section for the entire system stack, in exactly the same way as if you were not using an OS at all.

4.6.1 Working with Extended Tasks

RTA-OS3.0 uniquely extends the single stack model to provide support for extended tasks without any impact on the performance of basic tasks.

In RTA-OS3.0, the lifecycle of an extended task is as follows:

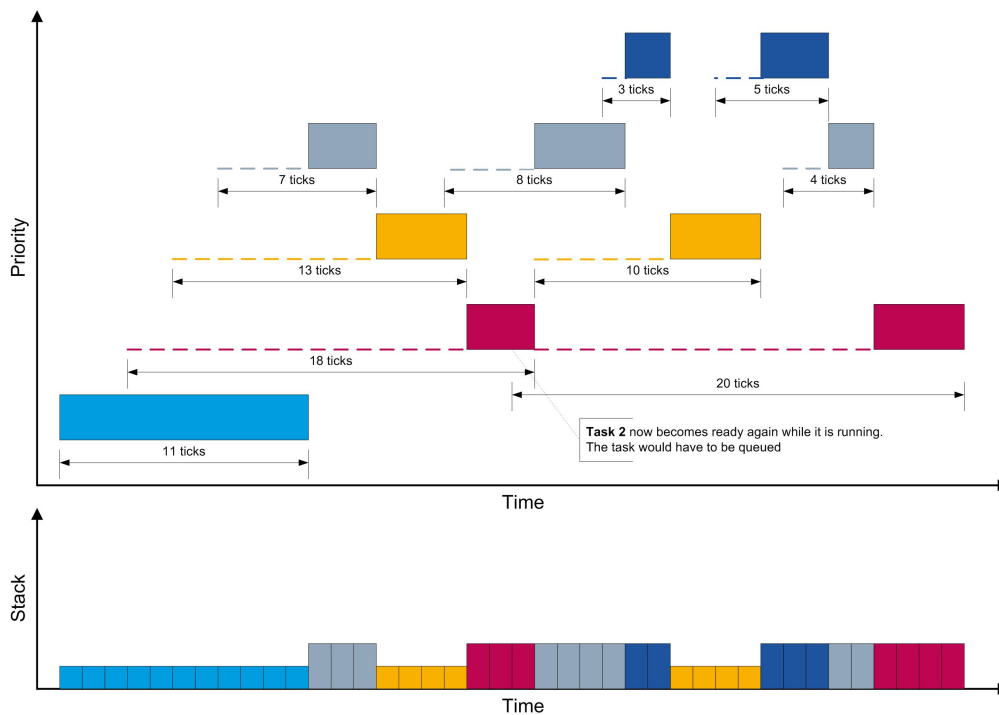


Figure 4.9: Single-stack behavior with non-preemptive tasks

Suspended → Ready The task is added to the ready queue.

Ready → Running The task is dispatched but, unlike a basic task where the context is placed in the top of the stack, the context is placed in the stack space at the pre-calculated worst case preemption depth of all lower priority tasks.

Running → Ready The extended task is preempted. If the preempting task is a basic task, then it is dispatched on the top of the stack as normal. If the preempting task is an extended task, then it is dispatched at the pre-calculated worst case preemption depth of all lower priority tasks.

Running → Waiting The task's "Wait Event Stack" context, comprising the OS context, local data, stack frames for function calls, etc, is saved to an internal OS buffer

Waiting → Ready The task is added to the ready queue.

Running → Suspended The task's "Wait Event Stack" context is copied from the internal OS buffer back onto the stack at the pre-calculated worst case preemption depth of all lower priority tasks.

This process allows the additional cost of managing extended tasks to apply only to extended task themselves. Basic tasks in system including extended

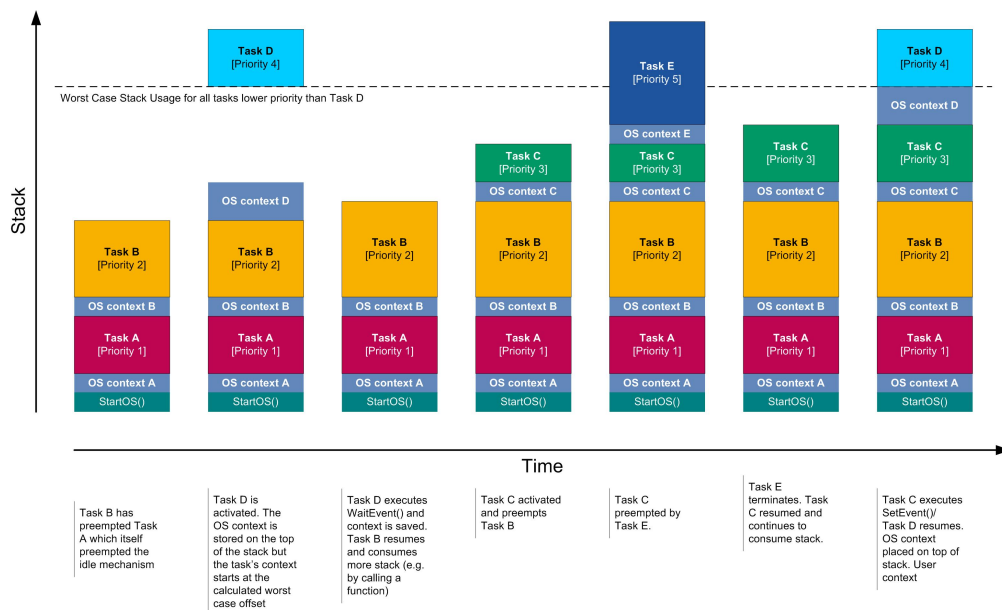


Figure 4.10: Single-stack management with Extended Tasks

tasks have the same performance as they would have in a basic task only system.

The key parts of this lifecycle are the dispatch/resume at the worst case preemption depth and the copy on and off the stack. The dispatch at the worst case preemption point guarantees that whenever an extended task resumes after waiting, it can resume with its local variables at exactly the same location in memory. It is guaranteed that every possible preemption pattern of lower priority tasks will never exceed the dispatch point of the extended task. The dispatch-wait-resume cycle for an extended task D is illustrated in Figure 4.10.

The copy off and on allows the extended tasks stack context to be restored. This is necessary because higher priority tasks and/or ISRs may occur while the extended task is waiting. These may consume stack space greater than the worst case preemption point (remember that the worst case point is for lower priority objects only), thereby overwriting the context of the extended task. However, fixed priority preemptive scheduling guarantees that no higher priority task can be ready to run at the point the extended task is resumed (it could not be resumed if this was the case).

4.6.2 Specifying Stack Allocation

In systems that contain only basic tasks it is not necessary to tell RTA-OS3.0 any stack allocation unless you are doing stack monitoring (see Section 14.1). You simply need to allocate a stack section large enough for your application

in your linker/locator. This is one of the benefits of the single stack architecture.

For applications that use extended tasks, you allocate your linker section as before, but you must also tell RTA-OS3.0 the stack allocation for every task in your configuration that is lower priority than the highest priority extended task, even if they are basic tasks. RTA-OS3.0 uses the stack allocation information to calculate the worst case preemption point for each extended task off-line.

The stack allocation you specify is the entire stack used for the task and includes:

- the OS context
- space for local variables in the task body
- the space required for any functions called from the task body (and their locals)

You can use RTA-OS3.0's stack measurement feature to obtain accurate values for the stack allocation. See Section 14.1 for further details.



RTA-OS3.0 only uses the stack information you provide to calculate the worst case preemption point. RTA-OS3.0 does not reserve any stack space. You must still specify the stack application stack space in the same way you would do for a normal application.

Figure 4.11 shows how stack allocation is configured.

While RTA-OS3.0 uses a single-stack model, on some ports this does not necessarily mean that just one *physical* stack is used. It may be the case that either the compiler or the hardware forces data onto different stacks automatically. For example, some devices place interrupts on to a dedicated interrupt stack.

Even with multiple physical stacks, RTA-OS3.0 still provides the benefits of the single-stack architecture - the stack space required on every physical stack can be overlaid when tasks and/or ISRs share a priority level. However, for the stack allocation to work correctly you will need to specify the space needed on each stack. RTA-OS3.0 will automatically ask you for multiple stack values if you configure a target for which this information is required. Figure 4.12 shows a dialogue box from such a configuration where there are two stacks: 'Supervisor' and 'Context'.

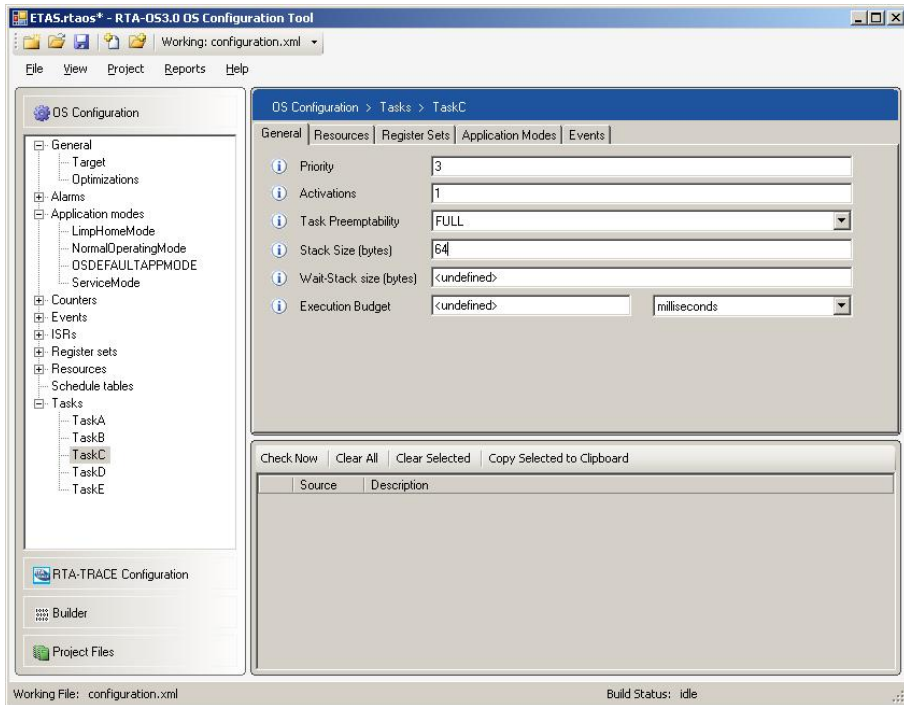


Figure 4.11: Stack Allocation Configuration

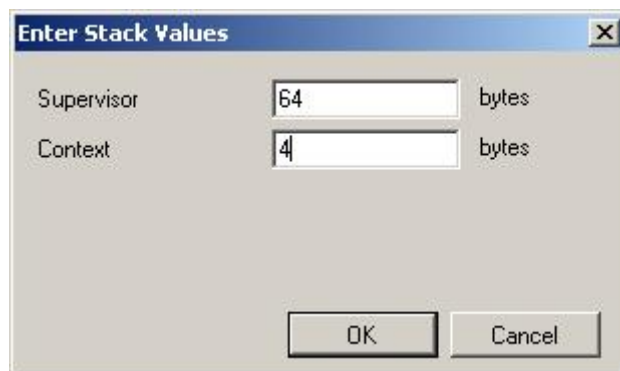


Figure 4.12: Stack Allocation Configuration for multiple stacks

4.6.3 Optimizing the Extended Task context save

Recall from Section 4.6.1 that each time an extended task enters the waiting state, RTA-OS3.0 saves the task's "Wait Event Stack" context and that the context is restored when the task re-enters the running state.

RTA-OS3.0 saves the "Wait Event Stack" context in an internal buffer. By default, RTA-OS3.0 allocates a buffer equal to the worst case stack allocation you specify for the task. Assuming that your stack allocation is correct, this should always be enough to hold the worst case stack usage when you call `WaitEvent()`.

This sounds expensive because it appears that RTA-OS3.0 needs to allocate twice the RAM you would expect for each extended task: once on the stack and once for the task's save/restore buffer! However, RTA-OS3.0 needs to save the context *only* when `WaitEvent()` is called. This means that you can significantly optimize the RAM size required by RTA-OS3.0 when using extended tasks by allocating only enough buffer space to save the worst case "Wait Event Stack" context, rather than the absolute worst case space required by the task.

Typically, most applications that use extended tasks only call `WaitEvent()` from the task's entry function where only a small amount of local data is on the stack so this optimization can be applied in most extended task systems.

You can control exactly how many bytes of stack are saved by RTA-OS3.0 by specifying the worst case stack depth at the point you call `WaitEvent()` as shown in Figure 4.13.



If you leave the `WaitEvent()` Stack allocation as 'undefined' then RTA-OS3.0 will default to use the number of bytes you specified for the stack allocation.

Using Default Values

While you should set a stack value for each task for memory efficiency, RTA-OS3.0 allows you to set a global default value that is used by all tasks. This can be found in [General → Default Stack Values](#).

If a Stack Allocation is not configured for a task, then RTA-OS3.0 will use the default value for:

- Calculating the worst case stack offset
- Configuring the `WaitEvent()` save/restore area
- Stack Monitoring (when configured)

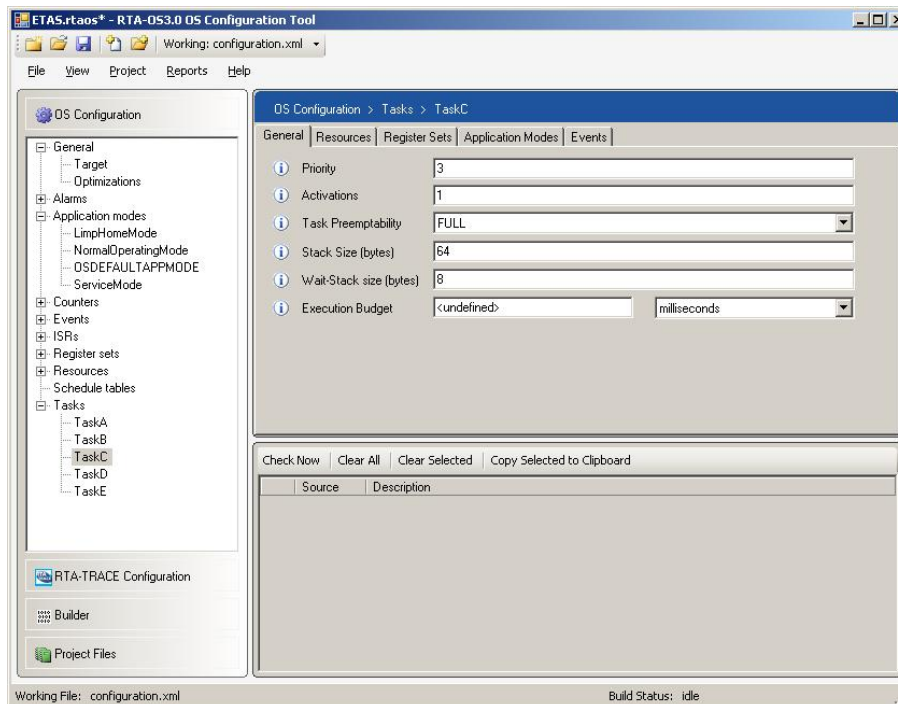


Figure 4.13: Specifying a WaitEvent () Stack allocation

The specification of a task/ISR-specific stack allocation overrides the default value.

4.6.4 Additional Stack Information

The calculated worst case dispatch points are relative to the base address of the stack at the point the OS is started. These offsets are stored as ROM data in the extended task control blocks and are added to the base stack pointer at runtime.

RTA-OS3.0 typically needs to know various port-specific stack measurements. The exact details of how this is done are port-specific. You should consult the *RTA-OS3.0 Target/Compiler Port Guide* for your port for additional guidance.

4.6.5 Handling Stack Overrun

If the stack allocation figures you provided to RTA-OS3.0 are wrong (i.e. they are too small) then this is a potential source of errors at runtime. There are three things that can go wrong:

1. the extended task cannot start because the current value of the stack pointer is higher than the calculated worst case dispatch point when RTA-OS3.0 tries to dispatch an extended task. This means one (or more)

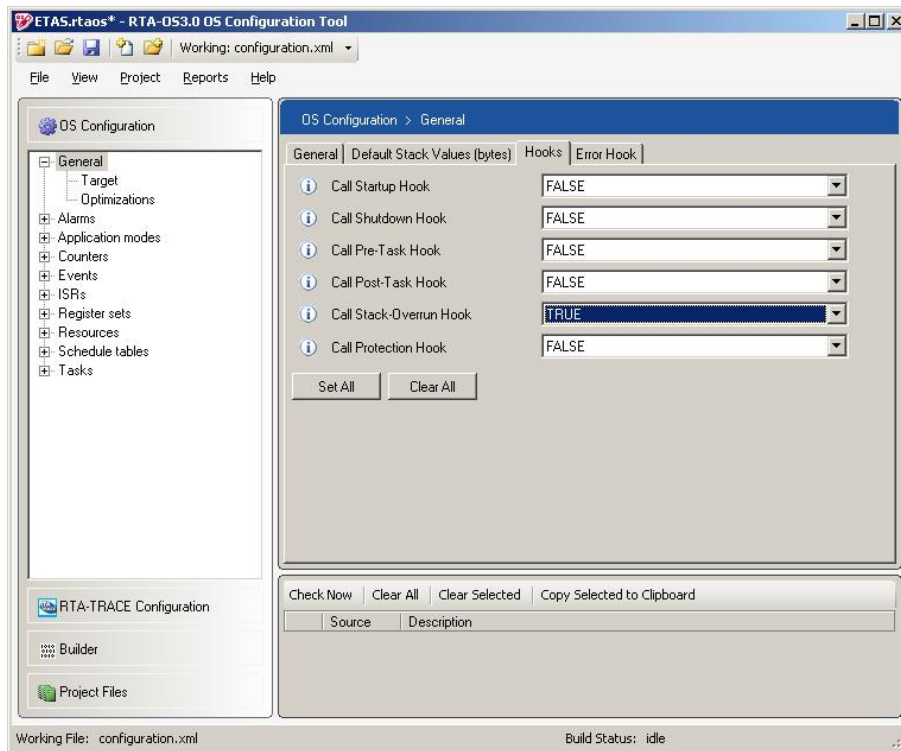


Figure 4.14: Enabling the `Os_Cbk_StackOverrunHook()`

of the lower priority tasks that are on the stack has consumed too much space (Stack monitoring, described in Section 14.1, can be used to identify which task is at fault).

2. the extended task cannot resume from the waiting state because the stack pointer is higher it should be. This may occur when `SetEvent()` has been called for an event on which the extended task was waiting and the extended task is now the highest priority task in the system.
3. the extended task cannot enter the waiting state because the current amount of stack the task is using is greater than the size of the 'WaitEvent()' stack' that was configured.

When RTA-OS3.0 detects a problem with extended task stack management it will call `ShutdownOS()` with the error code `E_OS_STACKFAULT`.

If you want to debug the problem then you can enable the stack fault hook as shown in Figure 4.14.

When configured, RTA-OS3.0 will call the user-provided callback `Os_Cbk_StackOverrunHook()` instead of `ShutdownOS()` when a stack fault occurs. The callback is passed two parameters:

1. Overrun tells you the number of bytes of the overrun
2. Reason tells you what caused the overrun

For an extended task system without stack monitoring enabled, the overrun can be either:

- OS_ECC_START - the extended task could not start (or resume from waiting) because the current stack pointer exceeds the worst case dispatch point calculated at build time. The cause of this fault is that one (or maybe more) of the lower priority tasks has exceeded the configured stack allocation. To fix this problem you need to identify which task is in error. Chapter 14 explains how to do this using RTA-OS3.0's stack monitoring feature.
- OS_ECC_WAIT - the extended task could not enter the waiting state because the amount of stack space it has consumed exceeds the configured WaitEvent() stack size. To fix this problem, you should increase the WaitEvent() stack size by at least the number of bytes indicated by the Overrun parameter.

Code Example 4.1 shows a simple example.

```
#ifdef OS_STACKOVERRUNHOOK
FUNC(void, OS_APPL_CODE) Os_Cbk_StackOverrunHook(
    Os_StackSizeType Overrun, Os_StackOverrunType Reason) {
{
    /* Identify problem */
    for(;;) {
        /* Do not return! */
    }
}
#endif /* OS_STACKOVERRUNHOOK */
```

Code Example 4.1: Minimum recommended Os_Cbk_StackOverrunHook()

4.7 Implementing Tasks

Tasks are similar to C functions that implement some form of system functionality when they are called by RTA-OS3.0.



You do not need to provide any C function prototypes for task entry functions. These are provided though the Os.h header file generated by RTA-OS3.0.

When a task starts running, execution begins at the task entry function. The task entry function is written using the C syntax in Code Example 4.2.

```
TASK(task_identifier)
{
    /* Your code */
}
```

Code Example 4.2: A Task Entry Function

Remember that basic tasks are single-shot. This means that they execute from their fixed task entry point and terminate when completed.

Code Example 4.3 shows the code for a basic task called BCC_Task.

```
#include <Os.h>
TASK(BCC_Task) {
    do_something();
    /* Task must finish with TerminateTask() or equivalent. */
    TerminateTask();
}
```

Code Example 4.3: A Basic Task

Now, compare the example in Code Example 4.3 with Code Example 4.4. Code Example 4.4 shows that extended tasks need not necessarily terminate and can remain in a loop waiting for events.

```
#include <Os.h>
TASK(ECC_Task) {
    InitializeTheTask();
    while (WaitEvent(SomeEvent)==E_OK) {
        do_something();
        ClearEvent(SomeEvent);
    }
    /* Task never terminates. */
}
```

Code Example 4.4: Extended Task Waiting for Events

4.8 Activating Tasks

A task can only run after it has been activated. Activation either moves a task from the suspended state into the ready state or it adds another entry to the queue of ready tasks (if the task supports multiple activation). The task will run once for each of the activations. It is an error to exceed the activation count and your application will generate E_OS_LIMIT errors when this happens (even in the Standard build status).

Tasks can be activated from both tasks and (Category 2) ISRs.

Activating a task does not cause the task to begin executing immediately, it just makes it ready to run. However, RTA-OS3.0 needs to check whether the activated task has a higher priority than the currently running task and, if it does, cause a context switch to occur so the new task can preempt the currently running task.

When you activate a task RTA-OS3.0 from another task, the exact behavior depends upon the relative task priorities. If the activated task has higher priority than the currently running task, then the newly activated task will preempt the current task. Otherwise, the task will remain on the ready queue until it becomes the highest priority ready task.

In a well-designed real-time system, it is unusual for a task to activate a higher priority task. Normally ISRs capture system triggers and then activate the tasks to do any associated processing. In turn, these tasks may activate lower priority tasks to implement trigger responses that have longer deadlines.

Observing this fact leads to one of the major optimizations in RTA-OS3.0. If you specify that your tasks never activate higher priority tasks, RTA-OS3.0 can eliminate a large amount of internal code because a test for context switch on each activation is never required because it cannot happen. This is configured by selecting the “Disable Upwards Activation” optimization.

This is similar to the behavior when activating a task from an ISR. All ISRs have a priority that is strictly higher priority than the highest task priority. When a task is activated from an ISR it can never enter the running state immediately so it is never necessary to check for a context switch. Such a check is only necessary when leaving the ISR.

4.8.1 Direct Activation

Tasks can be activated in a number of different ways. The basic mechanism for task activation is the `ActivateTask()` API call, which directly activates a task. The `ActivateTask(TaskID)` call places the named task into the ready state. The `ChainTask(TaskID)` call terminates the calling task (see Section 4.11) and places the named task into the ready state.

API Call	Description
<code>ActivateTask()</code>	A task or ISR can make this call to activate the task directly.
<code>ChainTask()</code>	A task can make this call to terminate the currently running task and to activate the task indicated.

4.8.2 Indirect Activation

Besides directly activating tasks, it is possible to use other AUTOSAR OS mechanisms to indirectly activate a task. These methods are described in more detail in later chapters of this user guide.

Activation by an Alarm. For each alarm in the system, you can specify a task that is activated each time the alarm expires.

Activation by a Schedule Table. For each schedule table in the system, you can specify a task that is activated on one or more expiry points on the table.

4.9 Controlling Task Execution Ordering

In many cases, you will need to constrain the execution order of specific tasks. This is particularly true in data flow based designs where one task needs to perform some calculation before another task uses the calculated value. If the execution order is not constrained, a race condition may occur and the application behavior will be unpredictable. Task execution ordering can be controlled in the following ways:

- Direct activation chains (see Section [4.9.1](#)).
- Priority levels (see Section [4.9.2](#)).
- Non-preemptable tasks (see Section [2](#)).

4.9.1 Direct Activation Chains

When you use direct activation chains to control the execution order, tasks make `ActivateTask()` calls on the task(s) that must execute following the task making the call.

Consider the following; there are three tasks Task1, Task2 and Task3 that must execute in the order Task1, then Task2, then Task3. Code Example [4.5](#) shows example task bodies.

```
#include <Os.h>
TASK(Task1) {
    /* Task1 functionality. */
    ActivateTask(Task2);
    TerminateTask();
}

TASK(Task2) {
```

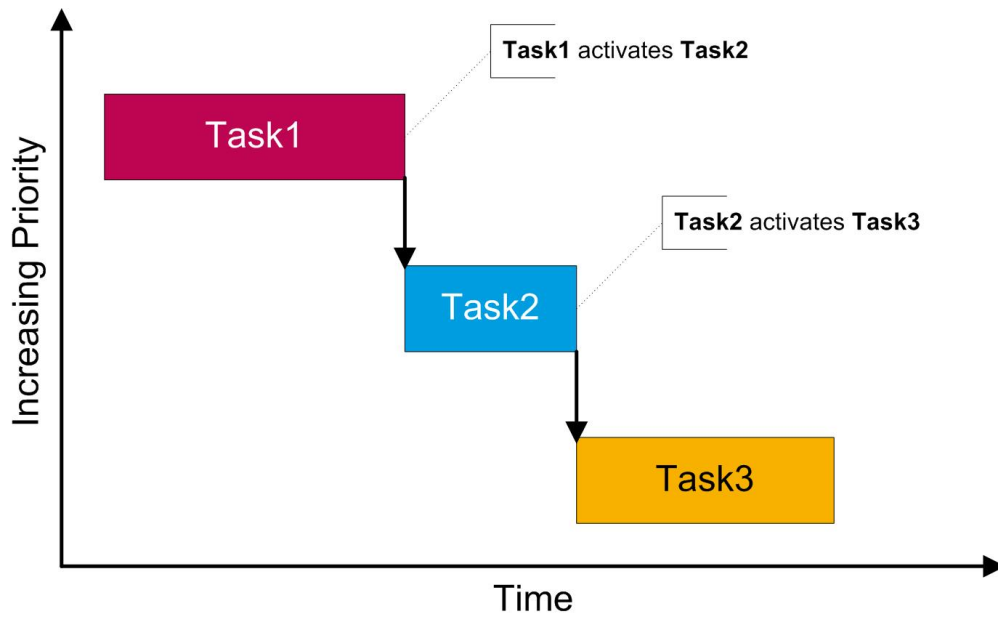


Figure 4.15: Direction activation to control task execution order

```

/* Task2 functionality. */
ActivateTask(Task3);
TerminateTask();
}

TASK(Task3) {
/* Task3 functionality. */
TerminateTask();
}

```

Code Example 4.5: Using Direct Activation Chains

Figure 4.15 shows how these tasks would execute assuming that Task1 has the highest priority and Task 3 has the lowest priority.

4.9.2 Using Priority Levels

The priority level approach to constraining task execution ordering can be used to exploit the nature of the preemptive scheduling policy to control activation order.

Recall from Section 4.1 that, under fixed priority preemptive scheduling, the scheduler always runs the highest priority task. If a number of tasks are released onto the ready queue, they will execute in priority order. This means that you can use task priorities to control execution order.

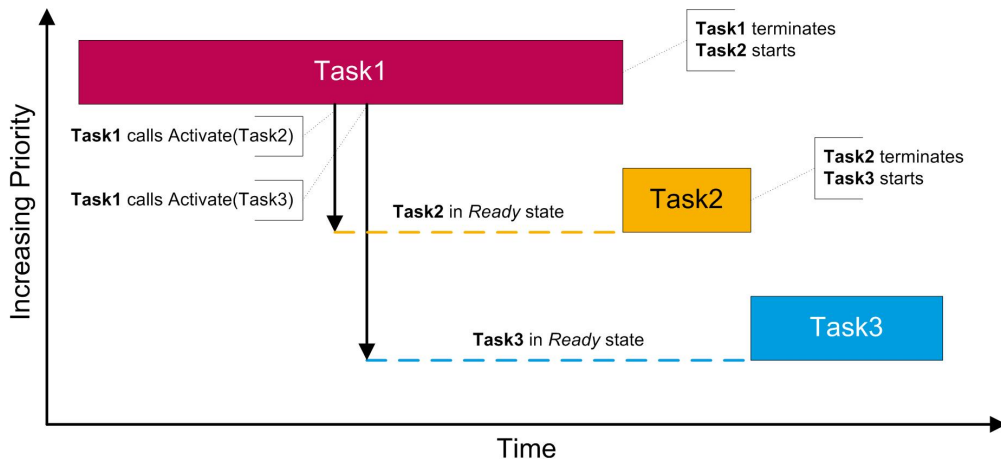


Figure 4.16: Using priority to control task execution order

Following on from our previous example, in Code Example 4.5, let's assume that Task1 has the highest priority and Task3 has the lowest priority. This means that the task bodies can be rewritten to exploit priority level controlled activation. This can be seen in Code Example 4.6.

```
#include <Os.h>
TASK(Task1) {
    /* Task1 functionality. */
    ActivateTask(Task2); /* Runs when Task1 terminates. */
    /* More Task1 functionality. */
    ActivateTask(Task3); /* Runs when Task2 terminates. */
    TerminateTask();
}

TASK(Task2) {
    /* Task2 functionality. */
    TerminateTask();
}

TASK(Task3) {
    /* Task3 functionality. */
    TerminateTask();
}
```

Code Example 4.6: Using Priority Level Controlled Activation

```
/* Task1 functionality. */
```

Figure 4.16 shows how these tasks would execute.

4.10 Co-operative Scheduling in RTA-OS3.0

When a task is running non-preemptively, it prevents any task (including those of higher priority) from executing. Sometimes, however, it is useful for non-preemptive tasks to offer explicit places where rescheduling can take place. This is more efficient than simply running non-preemptively because higher priority tasks can have shorter response times to system stimuli. A system where tasks run non-preemptively and offer points for rescheduling is known as a co-operatively scheduled system.

The `Schedule()` API call can be used to momentarily remove the preemption constraints imposed by both the non-preemptive tasks and the tasks using internal resources.

When `Schedule()` is called, any ready tasks that have a higher priority than the calling task are allowed to run. `Schedule()` does not return until all higher priority tasks have terminated.

In the following code example, the non-preemptive task `Cooperative` includes a series of function calls. Once started, each function runs to completion without preemption, but the task itself can be preempted between each function call.

```
#include "Cooperative.h"
TASK(Cooperative){
    Function1();
    Schedule();/* Allow preemption */
    Function2();
    Schedule();/* Allow preemption */
    Function3();
    Schedule();/* Allow preemption */
    Function4();
    TerminateTask();
}
```

Figure 4.17 shows how two tasks, `Task1` and `Task2`, which are co-operative would interact. The white sections represent non-preemptable sections of code.

4.10.1 Optimizing out the `Schedule()` API

`Schedule()` is of no use in a fully preemptive system. If you do not intend to use it, you can disallow calls to `Schedule()` in `rtaoscfg` using the "Optimizations, RTA-OS, Disallow `Schedule()`". If you disallow calls to `Schedule()` then you will see that the worst-case stack requirement for the system is reduced.

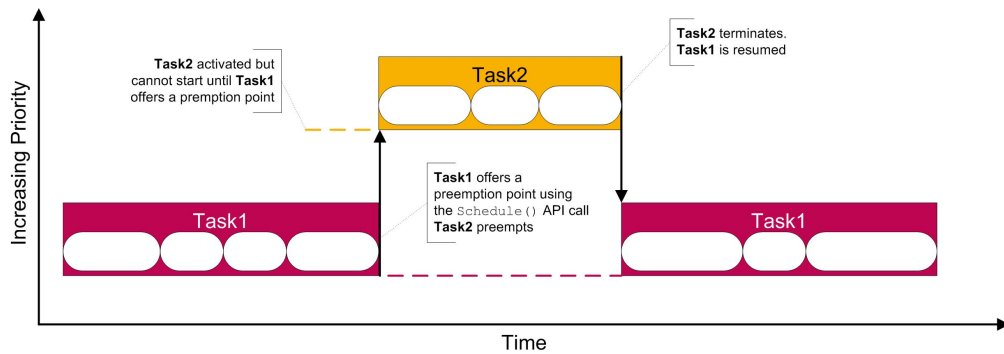


Figure 4.17: Co-operative tasks

4.11 Terminating Tasks

Tasks that terminate in AUTOSAR OS must make an API call to tell the OS that this is happening. The AUTOSAR OS standard defines two API calls for task termination. One of these must be used to terminate any task. These API calls are:

- `TerminateTask()`
- `ChainTask(TaskID)`

When a task has finished, it must make one of these API calls. This ensures that RTA-OS3.0 can correctly schedule the next task that is ready to run.

`TerminateTask()` forces the calling task into the suspended state. RTA-OS3.0 will then run the next highest priority task in the ready state.

`ChainTask(TaskID)` terminates the calling task and activates the task `TaskID`. The API is therefore like executing a `TerminateTask()` followed immediately by `ActivateTask(TaskID)`. Chaining a task places the named task into the ready state.

4.11.1 Optimizing Termination in RTA-OS3.0

The AUTOSAR OS standard allows task termination API calls to be called by a task at any point, including within a deeply nested set of function calls.

In Code Example 4.7, the task entry function makes nested calls to other functions.

```

/* Include Header file generated by \RTAOS */
#include <Os.h>

void Function1(void) {

```

```

    ...
    Function2();
    ...
}

void Function2(void) {
    if (SomeCondition) {
        TerminateTask();
    }
}

TASK(Task1) {
    /* Make a nested function call. */
    Function1();
    /* Terminate the task in the entry function*/
    TerminateTask();
}

```

Code Example 4.7: Terminating a Task

Code Example 4.7 shows that when Task1 runs, it calls Function1(). Function1() then calls Function2(). Function2() contains codes that can terminate the calling task (in this example, this is Task1).

The example is valid in AUTOSAR OS but is bad programming practice - equivalent to the use of **goto**. At runtime, RTA-OS3.0 must store information that allows it to clear the stack when the task terminates somewhere other than the entry function. This is normally done using a `setjmp/longjmp` pair.

However, one of the key benefits of the a single-stack architecture is that a task which terminates in its entry function can simply return - `TerminateTask()` does not need to do anything. If all your tasks either do not terminate or only terminate in their entry function, then the context that RTA-OS3.0 saves to allow a return from anywhere does not need to be stored.

RTA-OS3.0 allows you to exploit good application design using the fast termination optimization (**Optimizations → Fast Terminate**). You can enable this optimization when all tasks that execute the `TerminateTask()` or `ChainTask()` APIs only do so in their entry function. The optimization tells RTA-OS3.0 not to generate code to save unnecessary context and, as a result, save stack space.

4.12 The Idle Mechanism

Any preemptive operating system must have something to do when there are no tasks or ISRs to run. In AUTOSAR OS this is achieved by an idle mechanism.

In RTA-OS3.0 the OS will sit in a busy wait loop doing nothing when there are no tasks or ISRs to run.

However, you can override the default behavior by providing your own implementation of the idle mechanism by declaring a callback called `Os_Cbk_Idle`.

The `Os_Cbk_Idle` behaves in the same way as a task except that:

- it cannot be activated
- it cannot be terminated
- it cannot wait for events
- it cannot be chained
- it cannot use internal resources

The `Os_Cbk_Idle` has the lowest priority of any task in the system, so it runs only when there are no tasks (or ISRs) that are ready to run. The idle mechanism therefore gives you an “extra task” that is almost entirely free from system overheads.

Code Example 4.8 shows an implementation of `Os_Cbk_Idle` that is used to control RTA-TRACE (see Chapter 17).

```
#include <Os.h>
FUNC(boolean, OS_APPL_CODE) Os_Cbk_Idle() {
    #ifdef OS_TRACE
        CheckTraceOutput();
        UploadTraceData();
    #endif /* OS_TRACE */
    return TRUE;
}

OS_MAIN()
{
    /* System hardware initialization. */
    StartOS(OSDEFAULTAPPMODE);
    /* The call never returns */
}
```

Code Example 4.8: An Idle Mechanism

`Os_Cbk_Idle` returns a boolean on exit that tells RTA-OS3.0 whether or not to call `Os_Cbk_Idle` again. When TRUE is returned then RTA-OS3.0 immediately calls `Os_Cbk_Idle` again. When FALSE is returned then RTA-OS3.0 stops

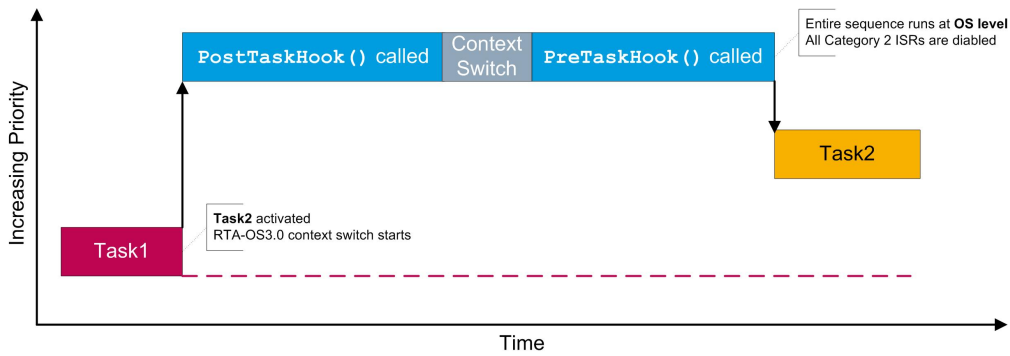


Figure 4.18: The PreTaskHook() and PostTaskHook() Relative to Task Pre-emption

calling `Os_Cbk_Idle` and enters the default behavior of sitting in a busy wait loop.

4.13 Pre and Post Task Hooks

Suppose that you need to execute some code before each task starts and/or after each task ends, for example to profile a trace of execution. You can do this using the PreTask and PostTask hooks provided by AUTOSAR OS.

The PreTask Hook is called by RTA-OS3.0 whenever a task moves into the running state. This means that the PreTask Hook will also be called whenever a task is resumed after preemption.

The PostTask Hook is called by RTA-OS3.0 whenever a task moves out of the running state. The PostTask Hook will be called when the task terminates and each time a task is preempted.

Figure 4.18 shows where the PreTask and PostTask Hooks are called relative to task preemption.

Both of these hooks are only called when configured. Figure 4.19 shows how to enable the hooks.

Code Example 4.9 shows how the hooks should appear in your code.

```

FUNC(void, OS_APPL_CODE) PreTaskHook(void) {
    /* PreTask hook code. */
}

FUNC(void, OS_APPL_CODE) PostTaskHook(void) {
    /* PostTask hook code. */
}

```

Code Example 4.9: The PreTaskHook and PostTaskHook

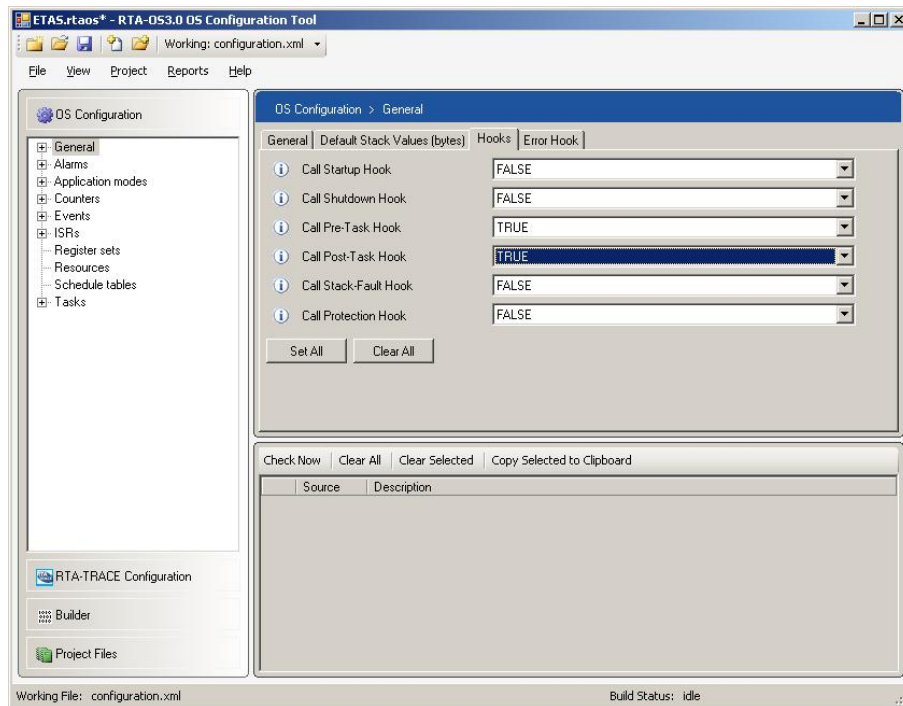


Figure 4.19: Enabling the PreTaskHook() and PostTaskHook()

The PreTask and PostTask Hooks are called on entry and exit of tasks and for each preemption/resumption. This means that it is possible to use these hooks to log an execution trace of your application. Since the same PreTask and PostTask Hooks must be used for all of the tasks in the application, it is necessary to use the GetTaskID() API call to work out which task has been or will be running when the hook routine is entered.

RTA-OS3.0 defines a set of macros that are only defined if the corresponding hook is enabled. These macros are called:

- OS_PRETASKHOOK
- OS_POSTTASKHOOK

This allows you write code where the hooks can be conditionally compiled as shown in Code Example 4.10.

```
#ifndef OS_PRETASKHOOK
FUNC(void, OS_APPL_CODE) PreTaskHook (void)
{
    /* Your code */
}
```

```
#endif /* OS_PRETASKHOOK */
```

Code Example 4.10: Conditional Compilation of PreTaskHook

4.14 Saving Hardware Registers across Preemption

RTA-OS3.0 saves as little context as necessary on a context switch - only the context for the correct operation of the OS is saved. However, you may find that you need to save and restore additional application-dependant context at runtime. For example, you may have tasks that use floating-point registers and therefore will require the floating-point context for your microcontroller to be saved across context switches.

You could choose to implement this by hand using the PreTask and PostTask hooks and an application-managed stack. However, it becomes difficult to optimize this type of implementation without making it fragile to changes in OS configuration. You can either:

- always save the context on every switch into a task and then restore on every switch out.

This model means you might be making unnecessary saves and restores (for example, saving a register set when switching into a task that doesn't use it); or

- calculate the saves required offline and then write a more complex pair of hooks that use GetTaskID() / GetISID() to work out if a save/restore is needed.

This model is fragile because changes to the configuration, for example adding new tasks/ISRs or modifying priorities, will mean that re-work is necessary.

To avoid these issues, RTA-OS3.0 provides a simple general-purpose mechanism for saving user-specific context together with the OS context. RTA-OS3.0 is able to exploit its knowledge of the priority space to calculate exactly which tasks need to save register sets at runtime so that unnecessary saves are optimized away automatically, saving both the time and stack required for the context switch. For example:

- if you only have one task or ISR that saves a given register set then no save or restore is needed.
- if multiple tasks use the same register set but cannot execute at the same time (because they are non-preemptable, share an internal resource or share priority) then RTA-OS3.0 does not need to save the register set.

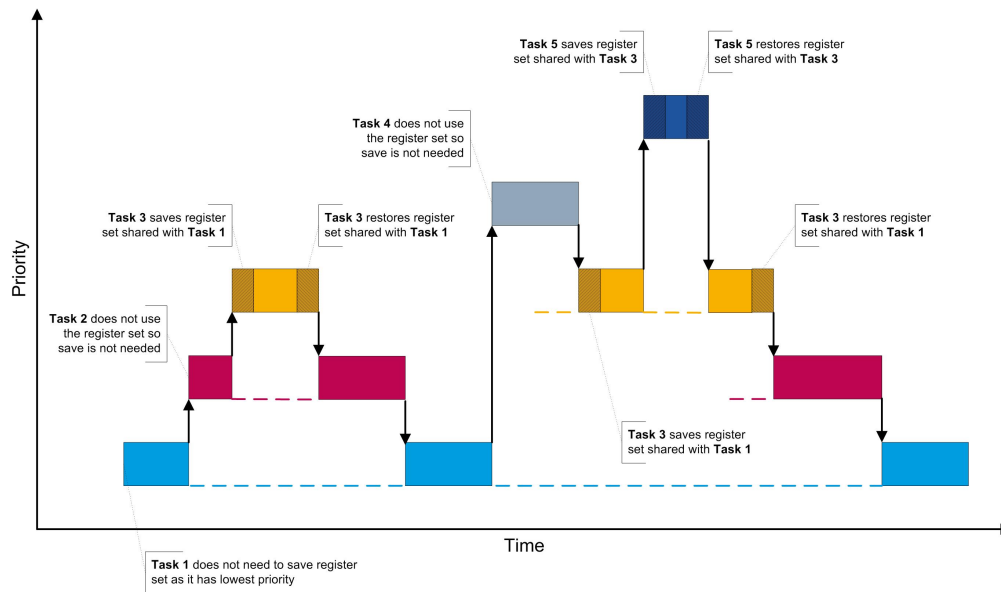


Figure 4.20: Register saving in action

- a context switch into the lowest priority task that uses a register set does not need to do a save because it can be guaranteed that no other task can be using the set (because the lowest priority task could not be running if a higher priority task was using the register set).
- similarly, a context switch from the highest priority task that uses a register set does not need to do a save because no higher priority task uses the register set and therefore cannot corrupt the context.

Figure 4.20 shows a register set that is shared by tasks 1, 3 and 5. You can see that when a save is not needed (when switching into a task that does not use the register set) then no context save is made.

Each register set you need to save needs to be declared to RTA-OS3.0 at configuration time. **rtaosgen** uses the declaration to define two callback functions that you must provide to save and restore the register set. Figure 4.21 shows the definition of three register sets.

Each task that uses a register set needs to declare this at runtime so that **rtaosgen** can calculate the maximum number of sets that need to be saved. Figure 4.22 shows one how this is done for a task.

RTA-OS3.0 does not know how or where to save and restore the register sets you declare - it just knows how many saves are necessary and when to save and restore them. For each register set you define, RTA-OS3.0 generates a macro called `OS_REGSET_<RegisterSetName>_SIZE` that defines the worst-

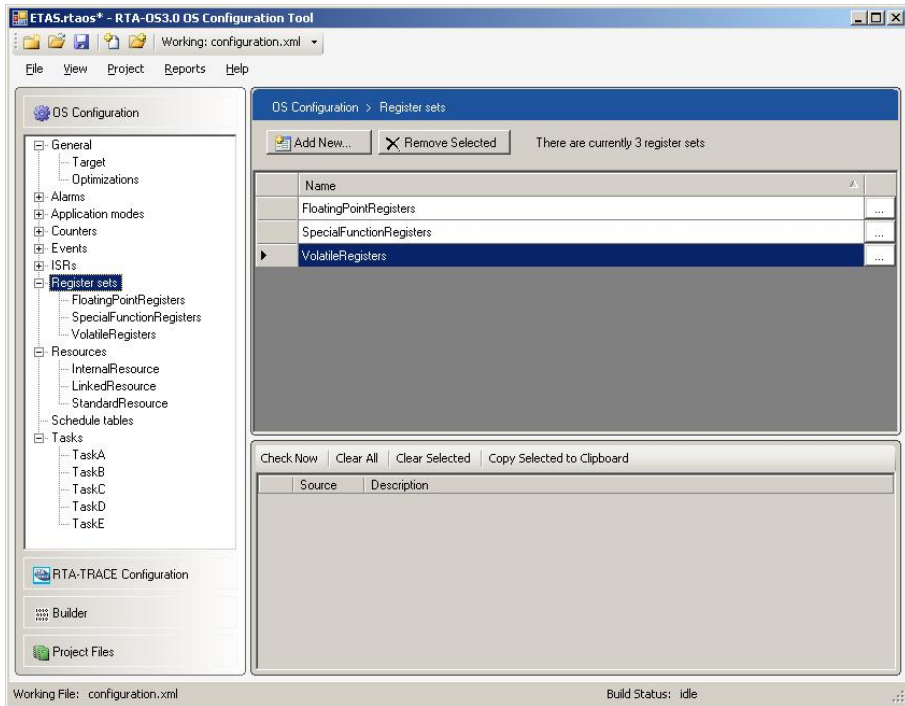


Figure 4.21: Register Set Definition

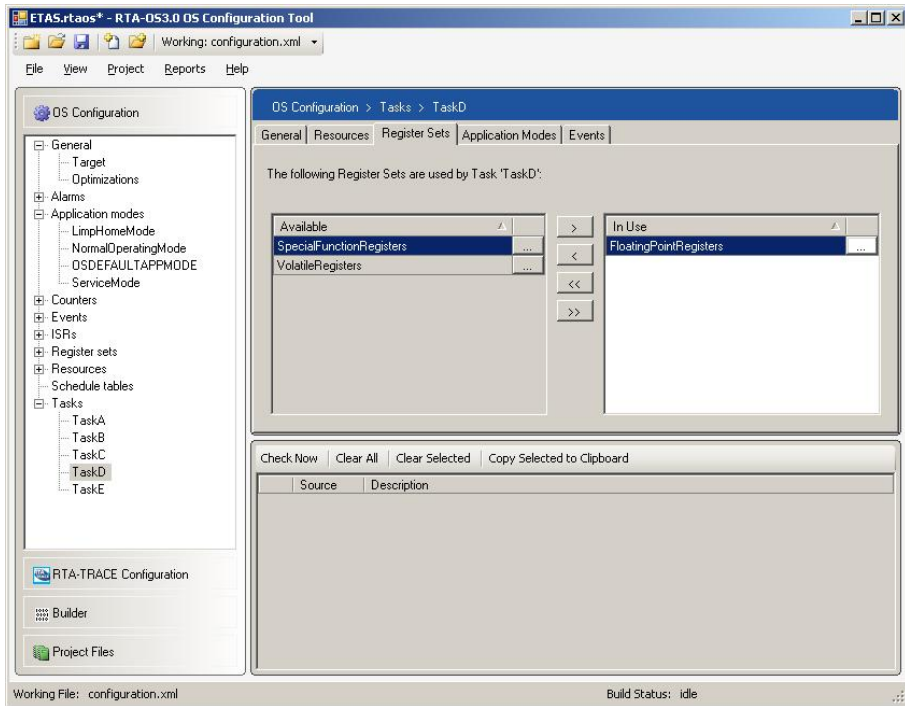


Figure 4.22: Using a register set in a task

case number of register set saves required. You should use this in your application code to define an array of size `OS_REGSET_<RegisterSetName>_SIZE` where each element of the array holds the saved register set.

You will also need to provide callback functions for the save and restore operations:

- `Os_Cbk_RegSetSave_<Name>(Os_RegSetDepthType Depth)` is called by RTA-OS3.0 whenever it is necessary to save a register set.
- `Os_Cbk_RegSetRestore_<NAME>(Os_RegSetDepthType Depth)` is called by RTA-OS3.0 whenever it is necessary to restore a register set.

Both of the callbacks are passed a `Depth` value that indicates the register set to save or restore. Code Example 4.11 shows how the callbacks might should appear in your code.

```
typedef volatile uint32 RegType;

#define VOLATILEREGISTER (*(RegType*)(0xFECAFECA))

uint32 VolatileRegisterSaveArea[OS_REGSET_VolatileRegister_SIZE
    ];

FUNC(void, OS_APPL_CODE) Os_Cbk_RegSetSave_VolatileRegister(
    Os_RegSetDepthType Depth) {
    RegisterSaveStack[Depth] = VOLATILEREGISTER;
}

FUNC(void, OS_APPL_CODE) Os_Cbk_RegSetRestore_VolatileRegister(
    Os_RegSetDepthType Depth) {
    VOLATILEREGISTER = RegisterSaveStack[Depth];
}
```

Code Example 4.11: Register Set Save And Restore

4.15 Summary

- A task is a concurrent activity.
- There are two classes of tasks: basic and extended.
- Tasks can share priorities, though it is recommended that you do not do this.

- Tasks are scheduled according to priority. When a higher priority task is made ready to run it will preempt lower priority tasks but it will not preempt any task that has been configured as non-preemptive.
- Tasks exist in states: ready, running, suspended or waiting (however, only extended tasks can enter the waiting state).
- If a task terminates, it must call `TerminateTask()` or `ChainTask(TaskID)` to do so.
- Systems where all tasks that terminate do so in their entry functions can use the “fast termination” optimization to minimize stack usage and context switching time.
- Tasks can only be activated when they are in the suspended state unless you specify multiple activations.
- The PreTask and PostTask Hooks allow you to execute code before your task starts and after it ends. This can be used to profile your application at run-time.

5 Interrupts

Interrupts provide the interface between your application and the things that happen in the real-world. You could, for example, use an interrupt to capture a button being pressed, to mark the passing of time or to capture some other stimulus.

When an interrupt occurs, the processor usually looks at a predefined location in memory called a vector. A vector usually contains the address of the associated interrupt handler. Your processor documentation and the *RTA-OS3.0 Target/Compiler Port Guide* for your target will give you further information on this. The block of memory that contains all the vectors in your application is known as the vector table.

5.1 Single-Level and Multi-Level Platforms

Target processors are categorized according to the number of interrupt priority levels that are supported. You should make sure that you fully understand the interrupt mechanism on your target hardware.

There are two different types of target:

Single-level. On single-level platforms there is a single interrupt priority. If an interrupt is being handled, all other pending interrupts must wait until current processing has finished.

Multi-level. On multi-level platforms there are multiple interrupt levels. If an interrupt is being handled, it can be preempted by any interrupt of higher priority. This is sometimes called a “nested” interrupt model.

5.2 Interrupt Service Routines

AUTOSAR operating systems capture interrupts using Interrupt Service Routines (ISRs). ISRs are similar to tasks; however, ISRs differ because:

- They cannot be activated by RTA-OS3.0 API calls.
- They cannot make `TerminateTask()` and `ChainTask()` API calls.
- They start executing from their entry point at the associated interrupt priority level.
- Only a subset of the RTA-OS3.0 API calls can be made.

The *RTA-OS3.0 Reference Guide* tells you the permitted calling context for every API call. You can refer to this to see whether or not you can use an API call in an ISR.

5.3 Category 1 and Category 2 Interrupts

AUTOSAR operating systems classify interrupts into two categories called Category 1 and Category 2. The category indicates whether or not the OS is involved with handling the interrupt.

5.3.1 Category 1 Interrupts

Category 1 interrupts do not interact with RTA-OS3.0. They should always be the highest priority interrupts in your application. It is up to you to configure the hardware correctly, to write the handler and to return from the interrupt.

You can find out more about Category 1 interrupt handlers in Section [5.6.1](#).

The handler executes at or above the priority level of RTA-OS3.0. However, you can make RTA-OS3.0 API calls for enabling/disabling and resuming/suspending interrupts.

5.3.2 Category 2 Interrupts

With Category 2 interrupts, the interrupt vector points to internal RTA-OS3.0 code. When the interrupt is raised, RTA-OS3.0 executes the internal code and then calls the handler that you have supplied.

The handler is provided as an ISR bound to the interrupt (which you can think of as a very high priority task). Execution starts at the specified entry point of the ISR and continues until the entry function returns. When the entry function returns, RTA-OS3.0 executes another small section of internal code and then returns from the interrupt.

Figure [5.1](#) shows the state diagram for a Category 2 interrupt handler.

Figure [5.2](#) shows how the internal RTA-OS3.0 code wrappers can be visualized.

5.4 Interrupt Priorities

Interrupts execute at an interrupt priority level (IPL). RTA-OS3.0 standardizes IPLs across all target microcontrollers, with IPL 0 indicating user level, where all tasks execute, and an IPL of 1 or more indicating interrupt level. It is important that you do not confuse IPLs with task priorities. An IPL of 1 is higher than the highest task priority used in your application.

The IPL is a processor-independent description of the interrupt priority on your target hardware. The *RTA-OS3.0 Target/Compiler Port Guide* for your port will tell you more about how IPLs are mapped onto target hardware interrupt priorities.

On a single-level platform there are two IPLs, 0 and 1. IPL 0 means that the

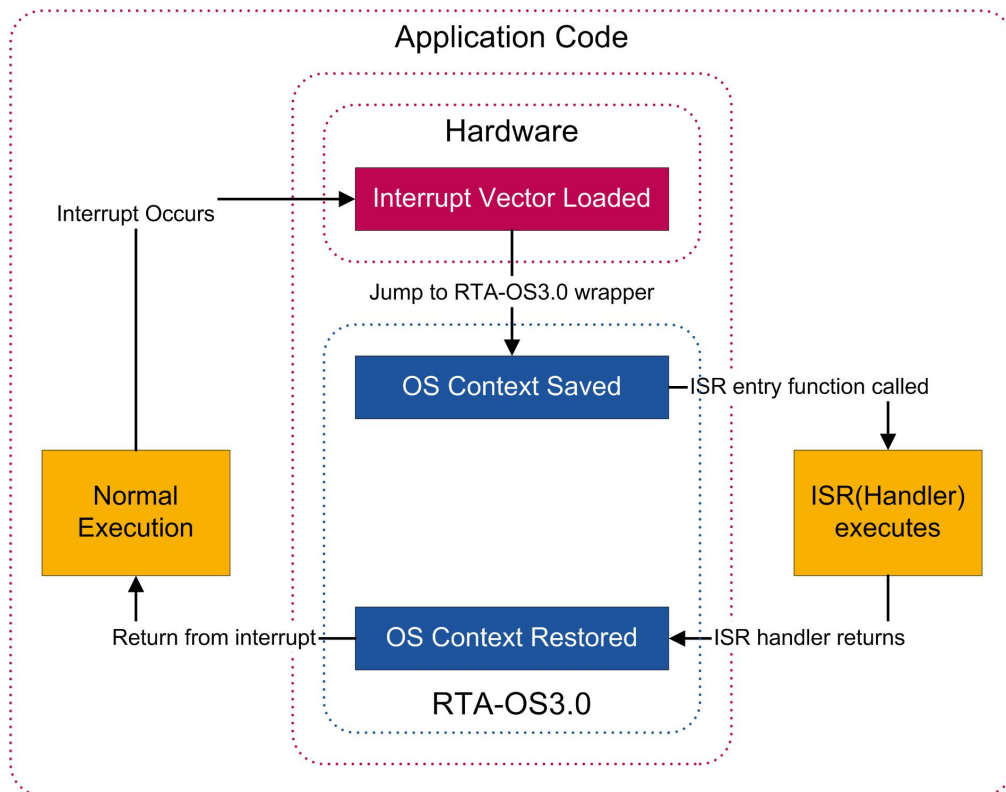


Figure 5.1: Category 2 Interrupt Handling State Diagram

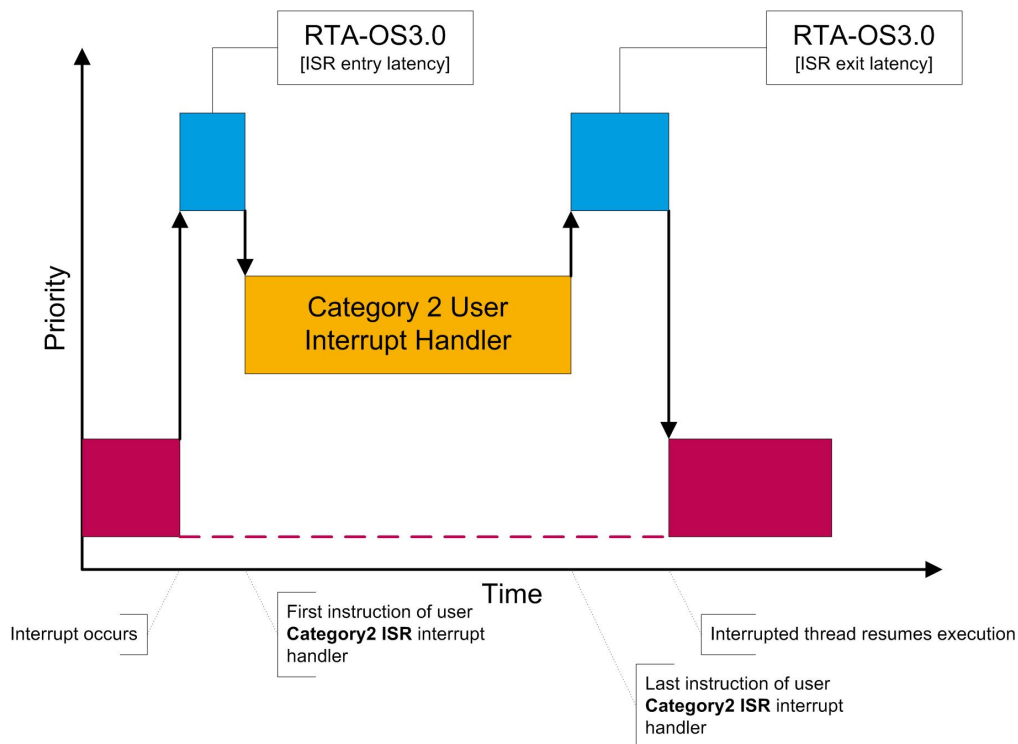


Figure 5.2: Visualizing RTA-OS3.0 Category 2 Wrappers

target is not interrupted and tasks run in priority order. IPL 1 means that the target is servicing an interrupt. As there is only one non-zero IPL, all interrupts, both Category1 and Category 2, run at the same priority. This means that all interrupts are serialized.

On multi-level platforms, higher priority interrupts can preempt lower priority interrupts and, therefore, the ISR handlers can be nested. So, for example, a higher priority ISR can interrupt the execution of a low priority ISR. However, an ISR can never be preempted by a task.

A Category 1 ISR must never be interrupted by a Category 2 ISR. This is because it is possible for a Category 2 ISR to activate a task and the OS therefore needs to check for a context switch when leaving the ISR - this is what the OS is doing in the second part of the “wrapper” function shown in Figure 5.2. As ISRs can nest on a multi-level platform, this check must happen as each interrupt exit. Now, if a Category 1 ISR could be preempted by a Category 2 ISR, on exit from the Category 1 ISR no checking for a context switch would occur and the originally preempted task would resume instead of the activated higher priority task. This is priority inversion and can cause unknown side-effects in your system.

This issue means that all Category 2 ISRs must have an IPL that is no higher

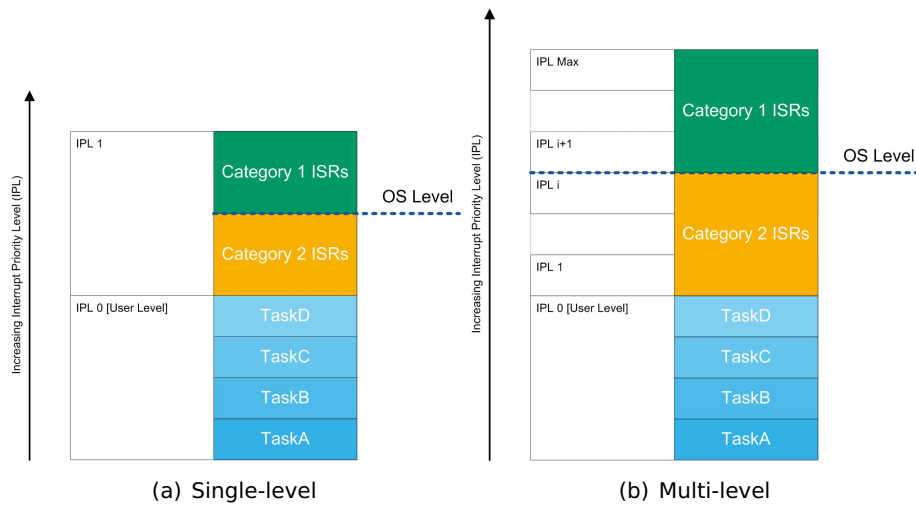


Figure 5.3: Interrupt Priority Hierarchies

than the lowest priority Category 1 ISR. The RTA-OS3.0 automatically checks this at build time and will generate an error if this is the case.

The interrupt priority hierarchies for single and multi-level platforms are shown in Figure 5.3.

5.4.1 User Level

User level is the lowest interrupt priority level that allows all interrupts to be handled. All tasks start executing at user level from their entry point.

A task will sometimes need to run above user level, for example it may need to access data shared with an ISR. While the data is being accessed it must prevent the interrupt being serviced. The simplest way to do this is for the task to disable interrupts while the data is being accessed. This is discussed in Section 5.7. An alternative mechanism is to use AUTOSAR OS's resource mechanism. This is discussed in Chapter 6.

An ISR may preempt a task even when the task is running with interrupt priority level above user level. It can only do this, however, if the ISR has a higher interrupt priority level than the current level.

5.4.2 OS Level

The priority of the highest priority Category 2 ISR defines OS level. If execution occurs at OS level, or higher, then no other Category 2 interrupts can occur.

RTA-OS3.0 uses OS level to guard against concurrent access to internal OS data structures. Any RTA-OS3.0 API that manipulates the internal state of the

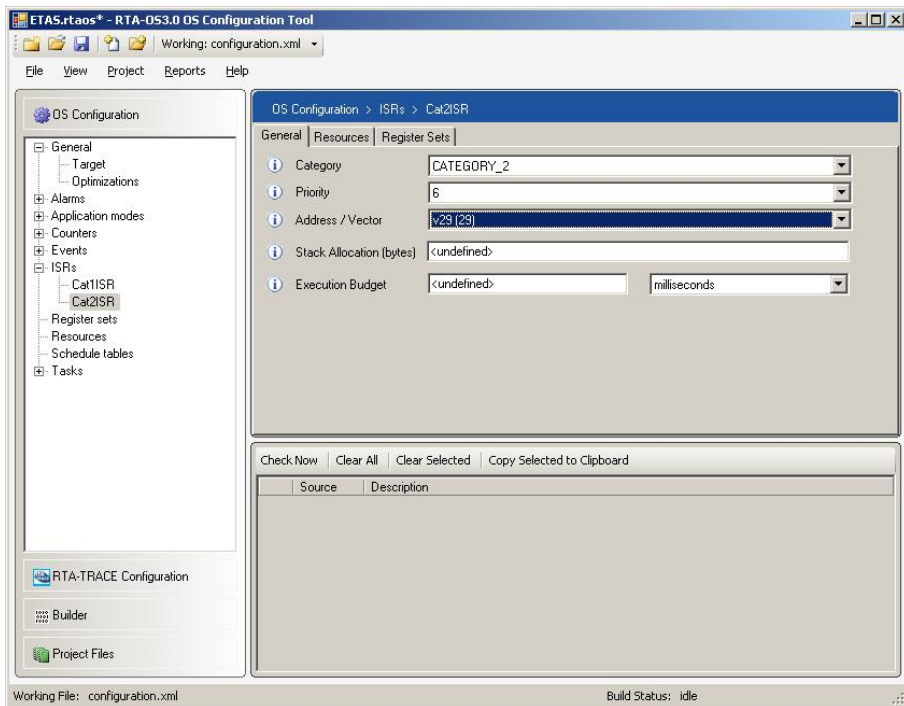


Figure 5.4: Configuring an Interrupt using in **rtaoscfg**

OS will perform some (if not all) of its execution time at OS level. OS hooks (for example the Error Hook, PreTask and PostTaskHook and OS callbacks also run at OS level. If a task executes at OS level, then no RTA-OS3.0 operations will take place (except for calls made by the task).

5.5 Interrupt Configuration

In RTA-OS3.0, interrupts are configured statically using **rtaoscfg**. Figure 5.4 shows how an interrupt has been constructed.

At the simplest level, an interrupt has the following attributes:

An interrupt name. The name is used to refer to C code that you will write to implement the handler functionality (you will learn how to do this in Section 5.6).

An interrupt category. This is either Category 1 if the handler does not need to execute RTA-OS3.0 API calls and Category 2 otherwise.

An interrupt priority. The priority is used by the scheduler to determine when the interrupt runs (in a similar way to a task priority being used for tasks). Priority is a microcontroller specific parameter so an RTA-OS3.0 target must be selected before you can set a priority. Note that some targets only support a single interrupt priority.

On microcontrollers where the IPL is user-programmable then it is your responsibility to ensure that the programmed priority level of an interrupting device matches the level you have configured for RTA-OS3.0. RTA-OS3.0 is not able to do this for you as this must occur before the OS is started because there may be Category 1 ISRs that need to execute. RTA-OS3.0 may generate appropriate configuration data for you to use. You should consult your RTA-OS3.0 Target/Compiler Port Guide for specific instructions.



An interrupt vector. RTA-OS3.0 uses the specified vector to generate the vector table entry for the interrupt. Like the interrupt priority, interrupt vector configuration is microcontroller specific so a target must be selected before the interrupt vector can be configured.

In RTA-OS3.0 it is possible to swap between different targets, for example allow you to quickly migrate one OS configuration to a new microcontroller. When the target is changed, all target-specific configuration is removed, including the interrupt priority and interrupt vector settings. New configuration, appropriate to the new target, will need to be provided.



5.5.1 Vector Table Generation

In most cases, RTA-OS3.0 can generate the vector table automatically¹. **rtaosgen** will create a vector table with the correct vectors pointing to the internal wrapper code and place this in the generated library.

If you want to write your own vector table then you must make sure that RTA-OS3.0 does not generate a vector table. You can prevent a vector table being generated by disabling vector table generation (**Target → Disable Vector Table Generation**) as shown in Figure 5.5.

When you write your own vector table you will need to make sure that all interrupt vectors that are associated with Category 2 ISRs branch to the RTA-OS3.0 interrupt wrapper that sets up the context in which the ISR executes.



You must not branch directly to your interrupt handler implementation. Doing so will bypass RTA-OS3.0 and any interaction you try to make with the kernel in the context of the handler is likely to result in unrecoverable corruption of the kernel state.

Typically your own vector table will need to branch to labels of the form `Os_Wrapper_VECTOR` where `VECTOR` is the hexadecimal address of the vector. However, the exact details are port-specific. You should consult the *RTA-OS3.0 Target/Compiler Port Guide* for your port to obtain specific details of how to provide your own vector table.

¹It may be the case that the compiler for your port generates the vector table. You should consult the *RTA-OS3.0 Target/Compiler Port Guide* for your port to obtain specific details.

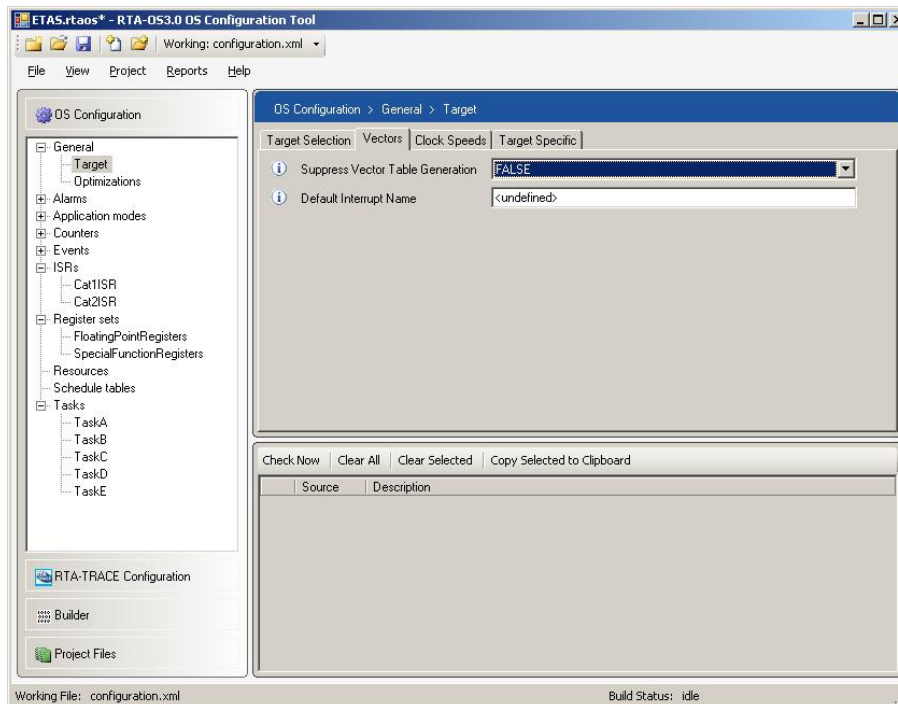


Figure 5.5: Preventing RTA-OS3.0 from Automatically Generating a Vector Table

5.6 Implementing Interrupt Handlers

You will now learn about interrupt handlers for Category 1 and Category 2 interrupts.

5.6.1 Category 1 Interrupt Handlers

The format for writing a Category 1 ISR is non-portable. The compiler for the microcontroller typically defines a compiler-specific extension to ANSI C that allows a function to be marked as an interrupt. Some compilers, however, cannot do this. When this happens you will need to write an assembly language handler.

You must make sure that the name of a Category 1 ISR entry function is the same as the name that you specified for the ISR during configuration.

For Category 1 ISRs, there is usually a compiler-specific keyword (sometimes called a “pragma” or a “directive”) that has to be used when defining entry functions. RTA-OS3.0 provides a macro called `CAT1_ISR` that expands to the correct directive for your compiler toolchain which you should use to mark your function as a Category 1 ISR.

An entry function for a Category 1 ISR is shown in Code Example 5.1.

```

CAT1_ISR(Interrupt1) {
    /* Handler body. */
    /* Return from interrupt. */
}

```

Code Example 5.1: Entry Function for a Category 1 ISR

5.6.2 Category 2 Interrupt Handlers

You saw earlier that Category 2 interrupts are handled under the control of RTA-OS3.0. A Category 2 ISR is similar to a task. It has an entry function that is called by RTA-OS3.0 when the interrupt handler needs to run. A Category 2 interrupt handler is written using the C syntax in Code Example 5.2.

```

#include <Os.h>
ISR(isr_identfier){
    /* Handler body. */
}

```

Code Example 5.2: Entry Function for a Category 2 ISR

You do not need to provide any C function prototypes for Category 2 ISR entry functions. These are provided in the `Os.h` header file that is generated by [rtaosgen](#).



You must not place a “return from interrupt” command in your Category 2 ISR. Returning from the interrupt is handled by RTA-OS3.0.

5.6.3 Dismissing Interrupts

When the hardware detects an interrupt, it will typically set a *pending bit* which tells the interrupt controller that an interrupt has occurred. The interrupt controller will then branch to the handler through the interrupt vector table.

The handling of the pending bit is target dependent but there are two basic models:

1. the pending bit is cleared automatically after the interrupt is handled (i.e. when the branch to the interrupt handler occurs). When the handler exits it will be automatically re-triggered if an interrupt has become pending while the current interrupt was being handled;
2. the pending bit must be cleared manually by user code in the interrupt handler. The body of the interrupt handler, whether Category 1 or Category 2, will need to include the code to clear the pending bit and signal to the hardware that the interrupt has been handled.

If you need to clear the pending bit, it is good practice to do this immediately on entry to the handler because this minimizes the time between the pending bit being set by a second instance of the interrupt occurring and then subsequently cleared. This helps to prevent issues where the interrupt becomes pending multiple times but this cannot be recognized by the hardware. Code example 5.3 shows how the recommended structure of a Category 2 ISR handler.

```
#include <Os.h>
ISR(Interrupt1) {
    /* Dismiss the interrupt where required */
    /* Rest of the handler */
}
```

Code Example 5.3: Dismissing the interrupt

You will need to consult your hardware reference manual to find out what you need to do on your target hardware.

5.6.4 Writing Efficient Interrupt Handlers

Each interrupt handler you write will block all interrupts of equal or lower priority for the time that it takes your code to execute. When you write an interrupt handler it is good practice to make the handler as short as possible. A long running handler will add additional latency to the servicing of lower priority interrupts.

By minimizing the execution time of your interrupt handlers you can maximize overall system responsiveness.

If you need to execute a long-running piece of code in response to the interrupt occurring, then you can put that code into a task and then activate the task from a Category 2 ISR. Code Example 5.4 and Code Example 5.5 show how these techniques differ.

With Category 2 handlers you can move the required functionality to a task, a simply use the interrupt handler to activate the task and then terminate.

```
#include <Os.h>
ISR(InefficientHandler) {
    /* Long handler code. */
}
```

Code Example 5.4: Inefficient interrupt handler

```
#include <Os.h>
ISR(EfficientHandler) {
    ActivateTask(Task1);
}
```

```

}

TASK(Task1) {
    /* Long handler code. */
    TerminateTask();
}

```

Code Example 5.5: More efficient interrupt handler

5.7 Enabling and Disabling Interrupts

Interrupts will only occur if they are enabled. By default, RTA-OS3.0 ensures that all interrupts are enabled when `StartOS()` returns.



AUTOSAR OS uses the term `Disable` to mean masking interrupts and `Enable` to mean unmasking interrupts. The `enable` and `disable` API calls do not therefore enable or disable the interrupt source; they simply prevent the processor from recognizing the interrupt (usually by modifying the processor's interrupt mask).

You will often need to disable interrupts for a short amount of time to prevent interrupts occurring in a critical section of code in either tasks or ISRs. A critical section is a sequence of statements that accesses shared data.

You can enable and disable interrupts using a number of different API calls:

- `DisableAllInterrupts()` and `EnableAllInterrupts()`
 Disable and enable all interrupts that can be disabled on the hardware (usually all those interrupts that can be masked).
 These calls cannot be nested.
- `SuspendAllInterrupts()` and `ResumeAllInterrupts()`
 Suspend and resume all interrupts that can be disabled on the hardware (usually all those interrupts that can be masked).
 These calls can be nested.
- `SuspendOSInterrupts()` and `ResumeOSInterrupts()`
 Suspend and resume all Category 2 interrupts on the hardware.
 These calls can be nested.



You must make sure that there are never more 'Resume' calls than 'Suspend' calls. If there are, it can cause serious errors and the behavior is undefined. Subsequent 'Suspend' calls may not work. This will result in unprotected critical sections.

Code Example 5.6 shows you how the interrupt control API calls are used and nested correctly.

```

#include <Os.h>
TASK(Task1) {
    DisableAllInterrupts();
        /* First critical section */
        /* Nesting not allowed */
    EnableAllInterrupts();
    SuspendOSInterrupts();
        /* Second critical section */
        /* Nesting allowed. */
    SuspendAllInterrupts();
        /* Third critical section */
        /* Nested inside second */
    ResumeAllInterrupts();
    ResumeOSInterrupts();
    TerminateTask();
}

```

Code Example 5.6: Nesting Interrupt Control API Calls

In the case of Category 1 ISRs, you must make sure that no RTA-OS3.0 API calls are made (except for other Suspend/Resume calls) for the entire time that the interrupts are disabled.

If a Category 2 ISR raises the interrupt level above OS level by calling `DisableAllInterrupts()` then it may not make any other RTA-OS3.0 API calls, except for the `EnableAllInterrupts()` call to restore the interrupt priority. When executing an ISR, you are not allowed to lower the interrupt priority level below the initial level.

5.8 Saving Register Sets

Recall from Section 4.14 that RTA-OS3.0 provides a mechanism for saving register sets across context switches and that `rtaosgen` can optimize the amount of saving that is required to improve runtime performance.

The same mechanism can also be used by Category 2 ISRs by simply selecting which ISRs use the configured register set as shown in Figure 5.6.

5.9 The Default Interrupt

If you are using RTA-OS3.0 to generate a vector table, then you may want to fill unused vector locations with a default interrupt.

Figure 5.7 shows how the default interrupt is defined.



The default interrupt is not supported by all ports.

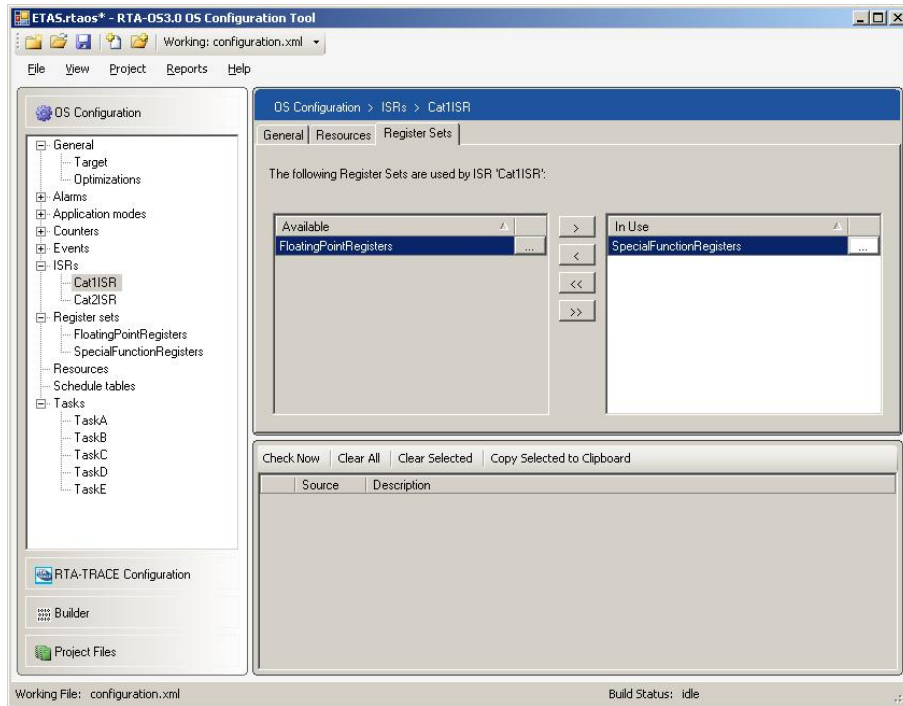


Figure 5.6: Using a register set in a Category 2 ISR

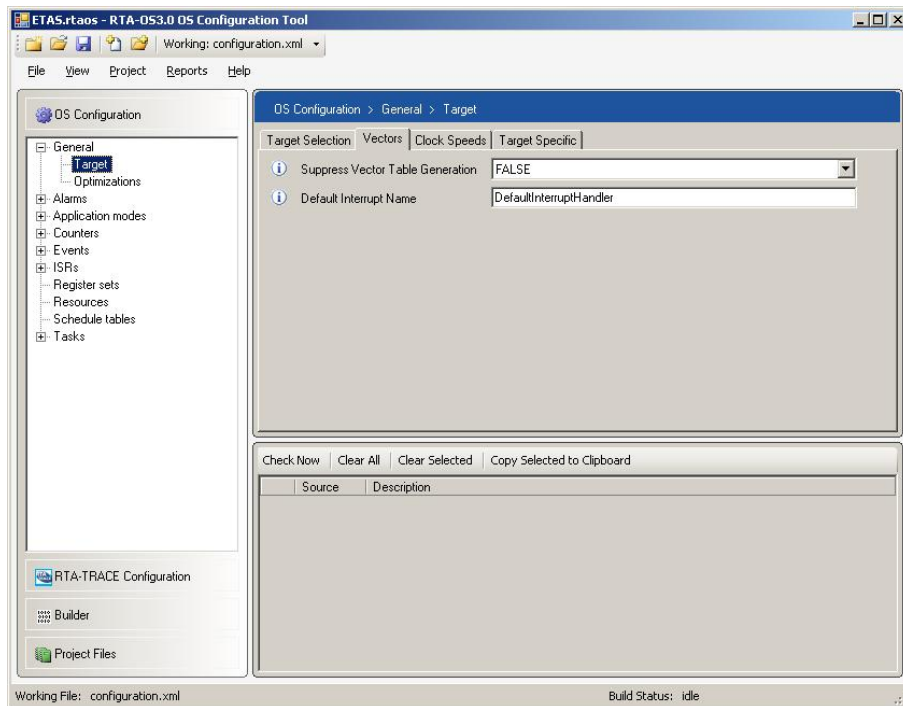


Figure 5.7: Placing a Default Interrupt in the Vector Table

The name allocated to the default interrupt at configuration time is the name that must be used in your application code when you write the handler. Code Example 5.7 shows a default handler that would work with the configuration shown in Figure 5.7.

The default interrupt is slightly different to other interrupts. It is used to fill every location in the vector table for which you have not defined an interrupt. This feature has been provided as a debugging aid and as a means of providing a “fail-stop” in the event of erroneous generation of interrupts in production systems. If you actually want to attach interrupt handlers to vectors to do useful work, you should explicitly create them as ISRs.

There are limitations on the use of the default interrupt handler. It cannot make any OS calls, and system behavior is undefined if it ever returns.



Do not make any RTA-OS3.0 API calls from the default interrupt and you must not return from the handler.

The default interrupt is implemented like an OSEK Category 1 interrupt and must therefore be marked as an interrupt with the CAT1_ISR macro. The last statement in your default interrupt handler should be an infinite loop. Code Example 5.7 shows how this can be done.

```
CAT1_ISR(DefaultInterruptHandler) {
    /* invoke target-specific code to lock interrupts */
    asm('di'); /* or whatever on your platform */
    for (;;) {
        /* Loop forever */
    }
    /* Do NOT return from default handler. */
}
```

Code Example 5.7: The Default Interrupt Handler

5.10 Summary

- RTA-OS3.0 supports two categories of interrupts: Category 1 and Category 2.
- Category 1 ISRs are normal embedded system interrupts that bypass RTA-OS3.0. As a result they cannot interact with the OS and are forbidden from making (most) RTA-OS3.0 API calls. They should be marked using the CAT1_ISR macro.
- Category 2 ISRs are OS managed interrupts that run in a wrapper provided by RTA-OS3.0. These interrupts can make RTA-OS3.0 API calls. They must be marked using the ISR macro.

- All interrupts run at an Interrupt Priority Level (IPL) which is always strictly higher than the highest task priority.
- IPLs standardize the interrupt priority model across all hardware devices - higher IPLs mean higher priority.
- RTA-OS3.0 can generate an interrupt vector table or you can choose to write your own. When generating a vector table, RTA-OS3.0 can plug unused locations with a user-configured default interrupt.

6 Resources

Access to hardware or data that needs to be shared between tasks and ISRs can be unreliable and unsafe. This is because task or ISR preemption can occur whilst a lower priority task or ISR is part way through updating the shared data. This situation is known as a race condition and is extremely difficult to test for.

A sequence of statements that accesses shared data is known as a critical section. To provide safe access to code and data referenced in the critical section you need to enforce mutual exclusion. In other words, you must make sure that no other task or Category 2 ISR in the system is able to preempt the executing task during the critical section.

In Chapter 4 you saw that you can declare tasks to be non-preemptive and that this prevents problems with mutual exclusion. However, this method is 'brute-force' because it prevents preemption problems by preventing preemption - rather like preventing car accidents by getting rid of cars!

The OS provide alternative mutual exclusion mechanisms based on resources. A resource is just a binary semaphore. When a task or Category 2 ISR gets a resource, no other task or ISR can get the resource. This prevents any other task or ISR entering the same critical section at the same time. When the critical section is finished, the task or ISR releases the resource and the critical section can be entered by another task/ISR.

When a high priority task is being prevented from executing by a lower priority task this is called priority inversion because the higher priority task takes longer to complete its execution than the lower priority task. The lower priority task appears to be running in precedence to the higher priority task, contrary to what would be expected from their actual priority assignment. The high priority task is said to be blocked by the low priority task.

Binary semaphores in traditional operating systems often get a bad name because priority inversion can introduce unbounded blocking in the system. For example, if the low priority task is preventing the high priority task from executing but is itself preempted by a medium priority task that does not need access to the shared resource then the high priority task will be blocked by execution of the medium priority task as well. As the low priority task might be preempted multiple times while it holds the shared resource, the blocking suffered by the high priority task can be unbounded, posing a significant problem if you need to determine the longest time it takes a task to respond¹. In extreme cases, tasks can reach a state called 'deadlock' where each task is waiting to enter a critical section that is being used by some other task.

¹Because the response time of the task depends on a factor that you cannot calculate.

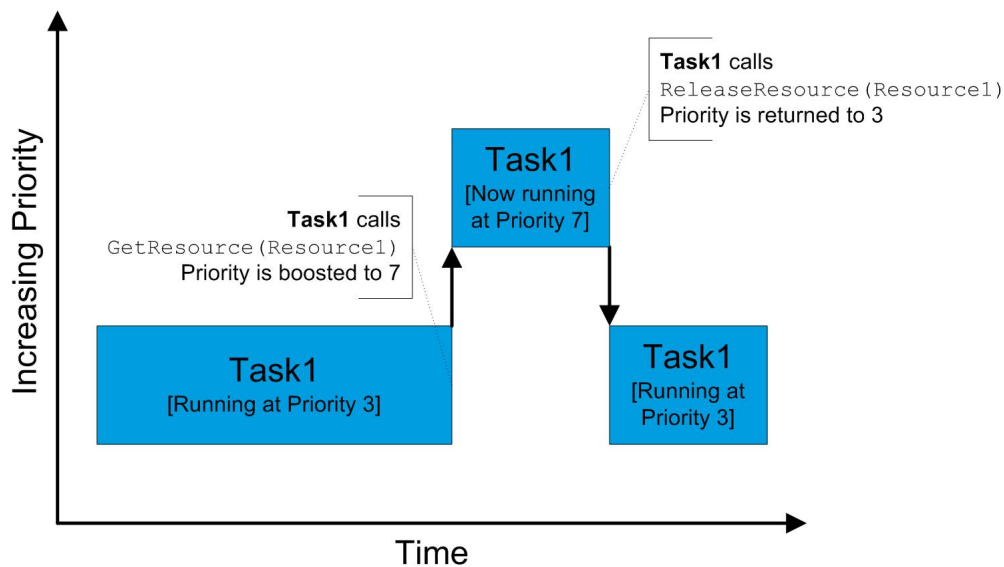


Figure 6.1: Raising to ceiling priority

In AUTOSAR OS, the problems typically associated with priority inversion and deadlock are avoided because resources are locked according to a locking protocol. This locking protocol is called priority ceiling protocol, in particular a version called immediate inheritance priority ceiling protocol (or alternatively stack resource protocol).

Priority ceiling protocol uses the concept of a ceiling priority. Each resource in the system is allocated a ceiling priority that is equal to the highest priority of any task or ISR that needs access to the resource. When a task or ISR gets a resource, the running priority of the task/ISR is increased to the ceiling priority of the resource (if and only if this is higher than the task/ISR's current running priority). When the resource is released, the priority of the task or reverts to the priority immediately prior to the task or ISR making the call. This is shown in Figure 6.1.

Immediate inheritance priority ceiling protocol provides two major benefits:

1. Priority inversion is minimized.

Each time a high priority task or ISR becomes ready, its execution can only be delayed at most once by a single lower priority task or ISR that already holds the a resource. This means there is no cumulative blocking so it is possible to place an upper bound on the blocking that a task suffers - the maximum blocking time is the longest time that a lower priority task/ISR holds the shared resource. Furthermore, this blocking always occurs at the start of execution. A consequence of this is that a resource is always free at the point it needs to be locked. There is no

need in AUTOSAR OS to wait for a resource to be released.

2. It is guaranteed to be deadlock free.

A task or ISR must be executing in order to make the lock. This can be proved by contradiction. Assume that a task (or ISR) tries to get a resource. If another task or ISR already had the resource then, because that task or ISR must be running at the ceiling priority, the task making the request not be executing (it would not be the highest priority task or ISR in the system) and, therefore, could not be attempting to lock the resource.

6.1 Resource Configuration

At the most basic level, resources only need to be named and assigned a type. There are three types of resource in AUTOSAR OS:

1. Standard resources are normal OS semaphores. Configuring a standard resource creates a resource with the specified name.
2. Linked resources allow you to alias a standard (or another linked) resource so that nested locking of the same resource is possible. These are discussed in more detail in Section 6.4.
3. Internal resources are resources that are locked automatically on entry to a task and released automatically on termination. These are discussed in more detail in Section 6.5.

Figure 6.2 shows how a standard resource is configured in the `rtaoscfg`.

RTA-OS3.0 needs to know which tasks and ISRs use which resources. It can then calculate the ceiling priorities used by the priority ceiling protocol.

Additional resource usage information for each task or ISR can be configured during task or ISR configuration.

Figure 6.2 shows that a resource called Resource1 has been declared. When you refer to this resource in your program you must use the same name.

6.2 Resources on Interrupt Level

Resources that are shared between tasks and interrupts are optional in OSEK. This optional feature is supported by RTA-OS3.0.

RTA-OS3.0 will automatically identify the resources that are combined resources, so you don't need to do any special configuration.

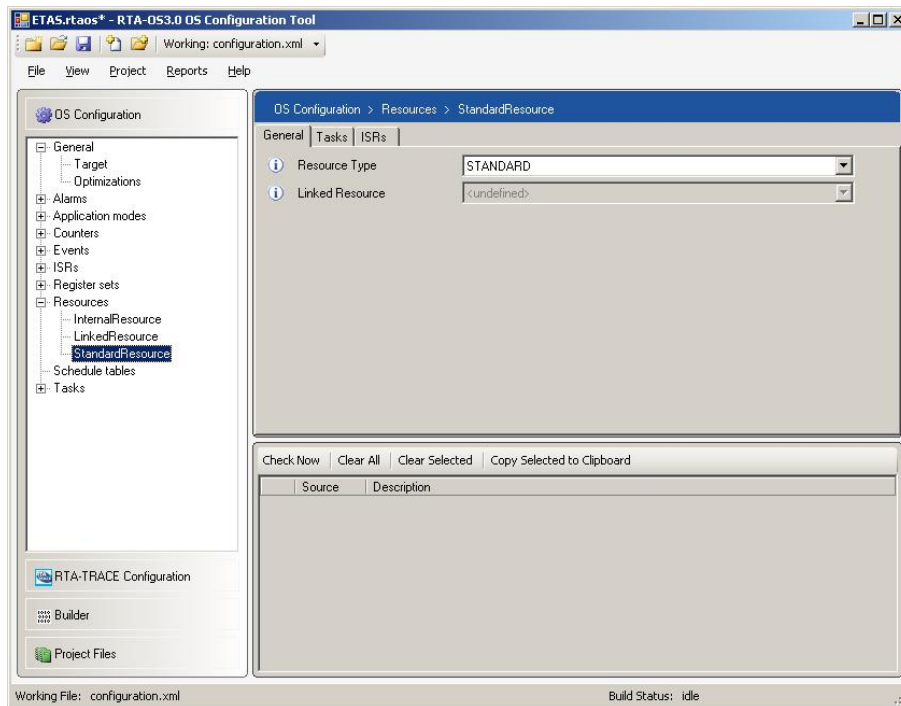


Figure 6.2: Configuring Resources using the **rtaoscfg**

When a task gets a resource shared with an ISR, RTA-OS3.0 will mask all interrupts with interrupt priority less than or equal to the highest priority interrupt that shares the resource.

This is simply an extension of priority ceiling protocol.

Sharing resources between tasks and ISRs means provides greater control over interrupt masking than the Enable/Disable and Suspend/Resume API calls because they make it possible to mask a subset of interrupts up to a particular priority level. Resources on interrupt level are therefore especially useful when using an RTA-OS3.0 port that supports nested interrupts.

6.3 Using Resources

You can get a resource using the `GetResource()` API call. You can then release a resource using the `ReleaseResource()` call. A task or ISR must not terminate until it has released all resources that it locked.

A task or ISR can only use the resources that you specify during RTA-OS3.0 configuration. Code Example 6.1 shows you how resources are used in Task1.

```
#include <Os.h>
TASK(Task1) {
    ...
}
```

```

    GetResource(Resource1);
    /* Critical section. */
    ReleaseResource(Resource1);
    ...
    TerminateTask();
}

```

Code Example 6.1: Using Resources

Calls to `GetResource()` and `ReleaseResource()` must be matched. You cannot get a resource that is already locked. You cannot release a resource you have not already locked.

When a `GetResource()` is made, it boosts the priority of the calling task or ISR to the ceiling priority of the resource. The resource's ceiling priority is the highest priority of any task or ISR that shares the resource and is automatically calculated by RTA-OS3.0. If any task with a priority less than the ceiling priority is made ready to run, then it is prevented from executing (it is **blocked**) until the priority of the running task returns to normal.

Figure 6.3 shows this effect with the following configuration:

Task	Priority	Locks Resource R1	Locks ResourceR2
3	High	✓	✗
2	Medium	✗	✓
1	Low	✓	✓

The first activation of Task 2 is blocked because Task 1 has locked R1. The second activation of Task 2 is also blocked, but this time because Task 1 has locked R2. The first activation of Task 3 is similarly blocked because of Task 1 holding R1. When Task 1 releases R1, the OS runs the highest priority ready task which is Task 3. On termination of Task 3, Task 2 executes and finally, when Task 2 terminates and Task 1 resumes.

6.3.1 Nesting Resource Calls

You can get more than one resource concurrently, but the API calls must be strictly nested. Let's look at two examples; one showing incorrectly nested calls and the other showing the API calls nested correctly. Code Example 6.2 shows `Resource1` and `Resource2` being released in the wrong order.

```

GetResource(Resource1);
    GetResource(Resource2);
ReleaseResource(Resource1); /* Illegal! */
    /* You must release Resource2 before Resource1 */
    ReleaseResource(Resource2);

```

Code Example 6.2: Illegal Nesting of Resource Calls

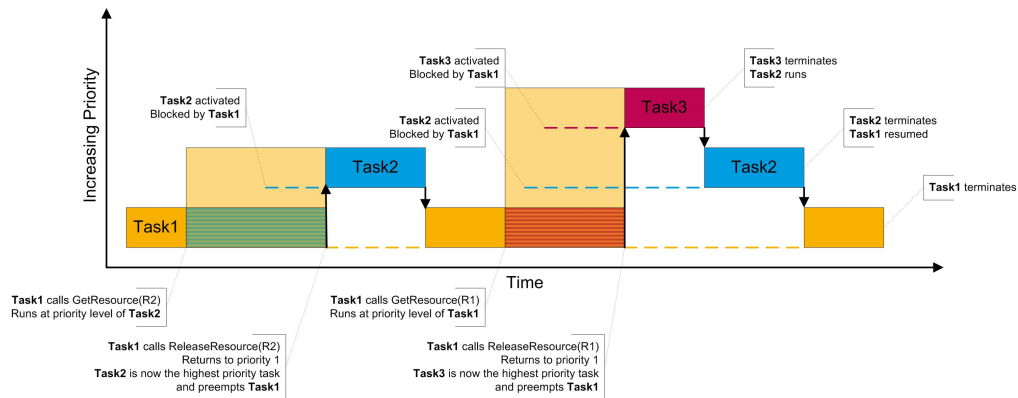


Figure 6.3: Execution of tasks with resource locks

A correctly nested example is shown in Code Example 6.3. All of the resources are held and then released in the correct order.

```

GetResource(Resource1);
  GetResource(Resource2);
    GetResource(Resource3);
      ReleaseResource(Resource3);
    ReleaseResource(Resource2);
  ReleaseResource(Resource1);

```

Code Example 6.3: Correctly Nested Resource Calls

6.4 Linked Resources

In AUTOSAR OS, GetResource() API calls for the same resource cannot be nested. However sometimes, there are cases where you may need make nested resource locks.

Your application may, for instance, use a function shared amongst a number of tasks. What happens if the shared function needs to get a resource used by one of the tasks, but not by the others? Have a look at Code Example 6.4.

```

#include <Os.h>
void SomeFunction(void) {
  GetResource(Resource1);    /* !!! Not allowed if caller is
    Task1 !!! */
  ...
  ReleaseResource(Resource1); /* !!! Not allowed if caller is
    Task1 !!! */
}

```

```

TASK(Task1) {
  GetResource(Resource1);
}

```

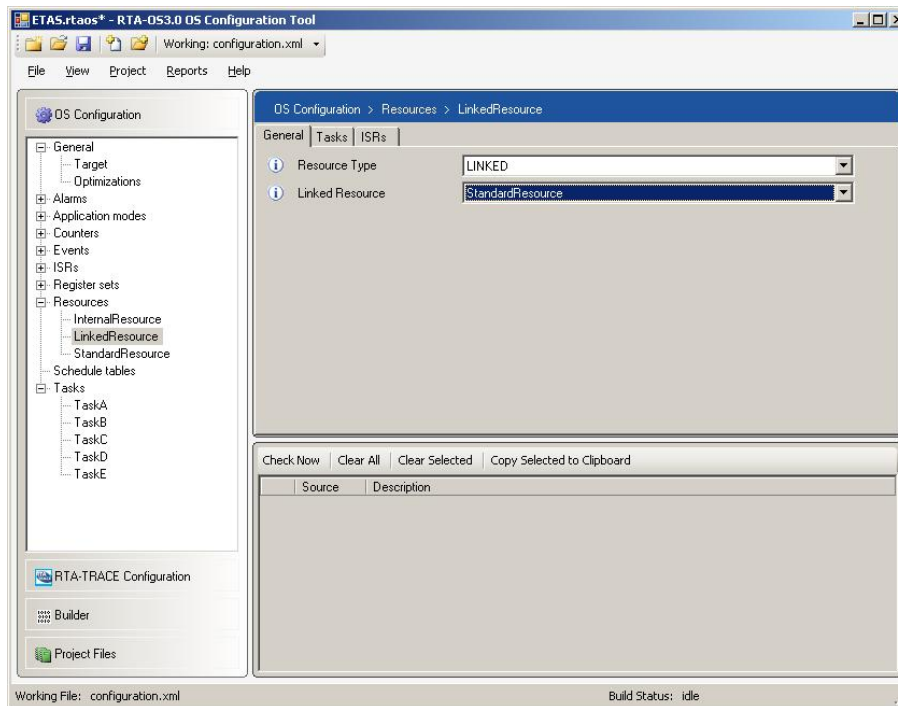


Figure 6.4: Configuring a Linked Resource

```

    /* Critical section. */
    SomeFunction();
    ReleaseResource(Resource1);
}

TASK(Task2) {
    SomeFunction();
}

```

Code Example 6.4: Illegal locking of previously locked resource

In these cases, the nesting of a (potentially) held resource must use linked resources. A linked resource is an alias for an existing resource and protects the same, shared, object.

Figure 6.4 shows how linked resources are declared using **rtaoscfg**.

With the linked resource, Code Example 6.4 would be re-written as shown in Code Example 6.5.

```

#include <Os.h>
void SomeFunction(void) {
    GetResource(LinkedToResource1); /* Okay */
    ...
}

```



```

    ReleaseResource(LinkedToResource1); /* Okay */
}

TASK(Task1) {
    GetResource(Resource1);
    /* Critical section. */
    SomeFunction();
    ReleaseResource(Resource1);
}

TASK(Task2) {
    SomeFunction();
}

```

Code Example 6.5: Using Linked Resources

Linked resources are held and released using the same API calls for standard resources (these are explained in Section 6.3). You can also create linked resources to existing linked resources.

6.5 Internal Resources

If a set of tasks share data very closely, then it may be too expensive, in terms of runtime cost, to use standard resources to guard each access to each item of data. You may not even be able to identify all the places where resources need to be held.

You can prevent concurrent access to shared data by using internal resources. Internal resources are resources that are allocated for the lifecycle of a task.

Internal resources are configured offline using **rtaoscfg**. Unlike normal resources, however, you cannot get and release them. Conceptually, RTA-OS3.0 locks the internal resource immediately before starting the task and releases the resource immediately after the task terminates.

In AUTOSAR OS R3.0 internal resources are only available to tasks. However, there is no reason why internal resources cannot be shared by Category 1 and 2 ISRs as well. RTA-OS3.0 provides an extension to AUTOSAR OS R3.0 that allows ISRs to use internal resources. When the a task locks an internal resource that is shared with an ISR, then the task executes at the IPL of the interrupt and all interrupts of equal or lower priority will be blocked for the duration of the task.

The implementation of internal resources in RTA-OS3.0 does not incur a runtime cost when the task enters the running state because **rtaosgen** calculates the priority at which the task will run offline and simple dispatches the task at this priority. The set of tasks that share an internal resource is stati-

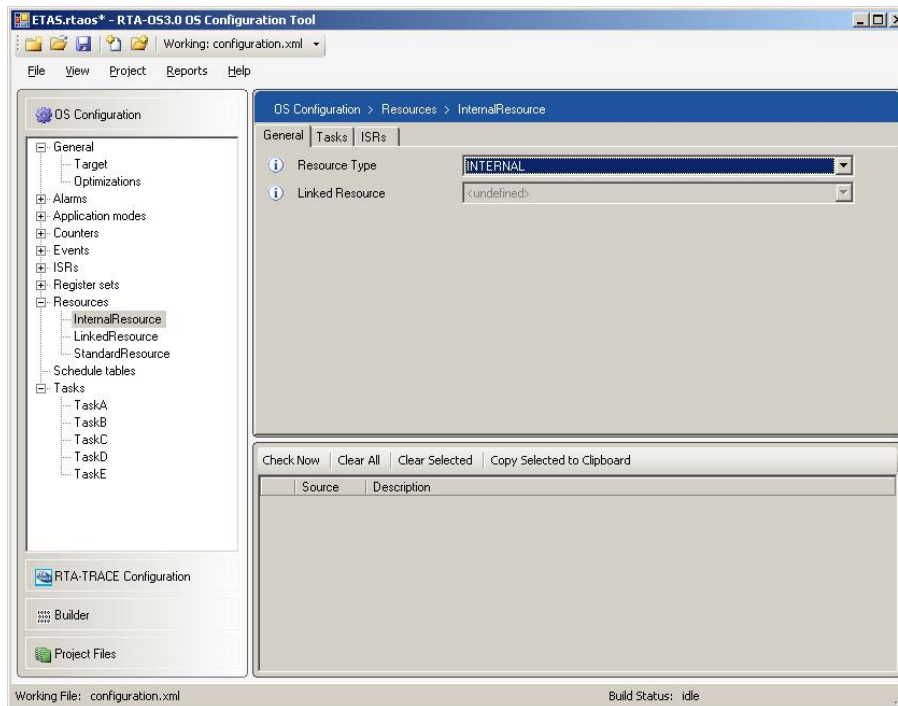


Figure 6.5: Declaring an Internal Resource using **rtaoscfg**

cally defined at configuration time using **rtaoscfg**.

Figure 6.5 shows the declaration of an internal resource, called `IntResource1`, which is shared between two tasks called `t1` and `t3`.

If a task uses an internal resource, RTA-OS3.0 will automatically get the internal resource before calling the task's entry function. The resource will then be automatically released after the task terminates, makes a `Schedule()` or a `WaitEvent()` call.

During task execution, all other tasks sharing the internal resource will be prevented from running until the internal resource is released. Figure 6.6 shows the execution of three tasks that share the same internal resource.

It is important to note that the OS makes a scheduling decision based on the normal (base) priority of the ready tasks when a task that holds an internal resource terminates. If a task is running and multiple tasks that share the same internal resource have become active then, on termination of the running task, the highest priority ready tasks is selected to run and then is dispatched at the ceiling priority of the internal resource.

When tasks share internal resources, preemption is still possible by all higher priority tasks that do not share the internal resource. However, any tasks

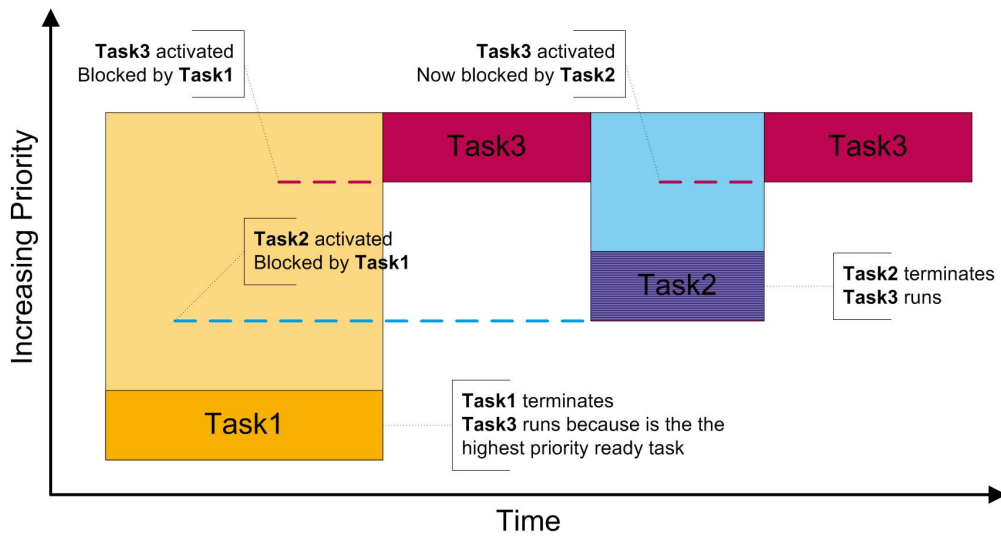


Figure 6.6: Execution with internal resources

with a priority lower than the ceiling priority of the internal resource, including those that do not share the internal resource, will be blocked if a task sharing the internal resource is executing. You can see an illustration of this in Figure 6.7 where Task1 with Priority 1 shares an internal resource with a Task that has Priority 3.

Figure 6.6 shows that initially Task 1 is running at priority 3 because it shares an internal resource with a task of priority 3. While Task 1 is running, Task 2 becomes ready to run. Task 2 is lower priority than the active priority of Task1 so it cannot preempt. When Task4 is activated, it can preempt Task1 because its priority is 4 i.e. it is higher priority than the active priority of task 1. Task 2 can only run when Task 1 terminates.

From this behavior it should be clear that a task which locks an internal resource will prevent *any* task with a higher priority than itself but lower priority than the ceiling priority of the internal resource from running *for the entire duration of the task*. When a lower priority task prevents a higher priority task from executing this is called blocking.

Tasks that share an internal resource run non-preemptively with respect to each other. Once a task in the set sharing the internal resource gets access to the CPU, it will run without being preempted by any other task in the set. The consequence of this is that it may take longer for higher priority tasks to get access to the CPU than would be the case in a fully preemptive system.

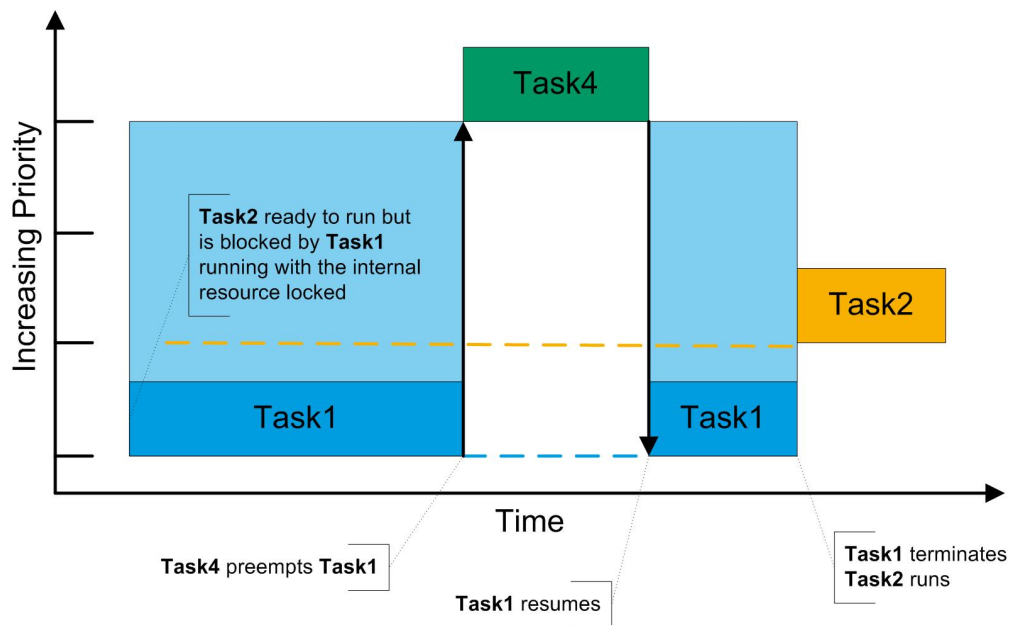


Figure 6.7: Internal resources blocking tasks that do not share the resource

6.6 Using Resources to Minimize Stack Usage

The primary role of resources in an application is to provide mutual exclusion over critical sections. However, the single-stack model of RTA-OS3.0 means that resources have a useful secondary role - minimizing stack usage. Recall that tasks which share resources do not preempt each other. In the single-stack model used by RTA-OS3.0 this means that their stack usage is effectively overlaid.

It is possible to exploit this feature to trade off time in the system against stack usage. The following sections describe how simple modifications to an application can reduce stack usage. All of these modifications will introduce additional blocking factors into the system.

The impact of these blocking factors depends on the system. Recall that the priority ceiling protocol ensures that a task or ISR is blocked at most once during execution. The worst-case blocking time is the maximum time that any lower priority task or ISR can hold the same resource.

This means that if the additional blocking factors are less than or equal to the current worst-case blocking suffered by a task/ISR, then there will be no impact on response times and the reduced stack usage will be free. If the additional blocking factors are longer than current worst-case blocking then response times will be longer. Providing that response times remain inside the required deadlines for tasks/ISRs, the system will still behave correctly.

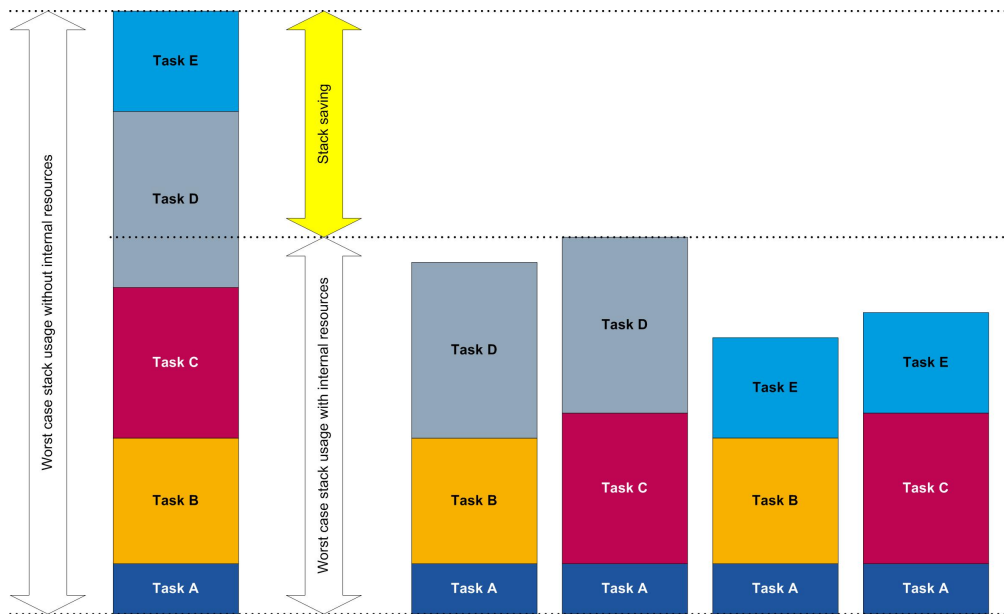


Figure 6.8: Saving Stack Space Using Internal Resources

6.6.1 Internal Resources

Given a set of tasks that share an internal resource, the worst case stack used by RTA-OS3.0 is equal to the maximum stack space required by the task that uses the most stack. In conventional operating systems, the maximum stack space would be equal to the sum of the task's stacks, not their maximum.

If you need to minimize stack space then you can exploit this benefit of RTA-OS3.0's single-stack architecture by sharing internal resources between tasks which consume lots of stack. The first stack in Figure 6.8 shows the worst-case stack consumption for 5 preemptive tasks, A, B, C, D and E. By sharing an internal resource between tasks B and C, and between tasks D and E a significant saving of stack space can be made. The other four stacks in Figure 6.8 show the cases that can now occur - the worst case is A preempted by the worst of B or C preempted by the worst of D and E. You can see from the figure that A preempted by C preempted by D gives the worst case and that this is significantly less stack than when internal resources were not used.

6.6.2 Standard Resources

If a task calls a function that uses a lot of stack then you could consider locking a resource around the function call and sharing the resource with the tasks of higher priority. The tasks do not need to lock the resource in code or call the function - the sharing is simply to force the execution of the task to run at a higher priority. This will prevent higher priority tasks preempting

the task while it is using lots of stack and will therefore reduce the total stack requirement.

Disabling interrupts around the function call has a similar effect - effectively overlaying the function call's stack usage with the ISRs that are temporarily masked.

6.7 The Scheduler as a Resource

A task can hold the scheduler if it has a critical section that must be executed without preemption from any other task in the system (recall that the scheduler is used to perform task switching). A predefined resource called `RES_SCHEDULER` is available to all tasks for this purpose. `RES_SCHEDULER` is a convenient way for tasks to share data without you needing to declare a resource that is shared between all tasks manually.

When a task gets `RES_SCHEDULER`, all other tasks will be prevented from preempting until the task has released `RES_SCHEDULER`. This effectively means that the task becomes non-preemptive for the time that `RES_SCHEDULER` is held. This is better than making the entire task non-preemptive, particularly when a task only needs to prevent preemption for a short part of its total execution time.

You must specify whether your application uses `RES_SCHEDULER` or not. This is configured in **General → Optimizations**. If you configure `RES_SCHEDULER` then RTA-OS3.0 will automatically generate a standard resource called `RES_SCHEDULER` and share it between every task in your configuration. As `RES_SCHEDULER` behaves like a standard resource, you can create linked resources that link to `RES_SCHEDULER` as shown in Figure 6.9.

Using `RES_SCHEDULER` can improve response times of low priority tasks that might otherwise suffer multiple preemptions by other tasks in the application, but at the cost of longer response times for higher priority tasks.

If you have no need to use `RES_SCHEDULER` in your application then you can save ROM and RAM space by disabling its generation as shown in Figure 6.10.

6.8 Choosing a Preemption Control Mechanism

If code that does not require locks appears between a pair of `GetResource()` and `ReleaseResource()` calls, the system responsiveness can potentially be reduced.

With this in mind, when you use resources in your application, you should place `GetResource()` calls as closely as possible around the section of code you are protecting with the resource.

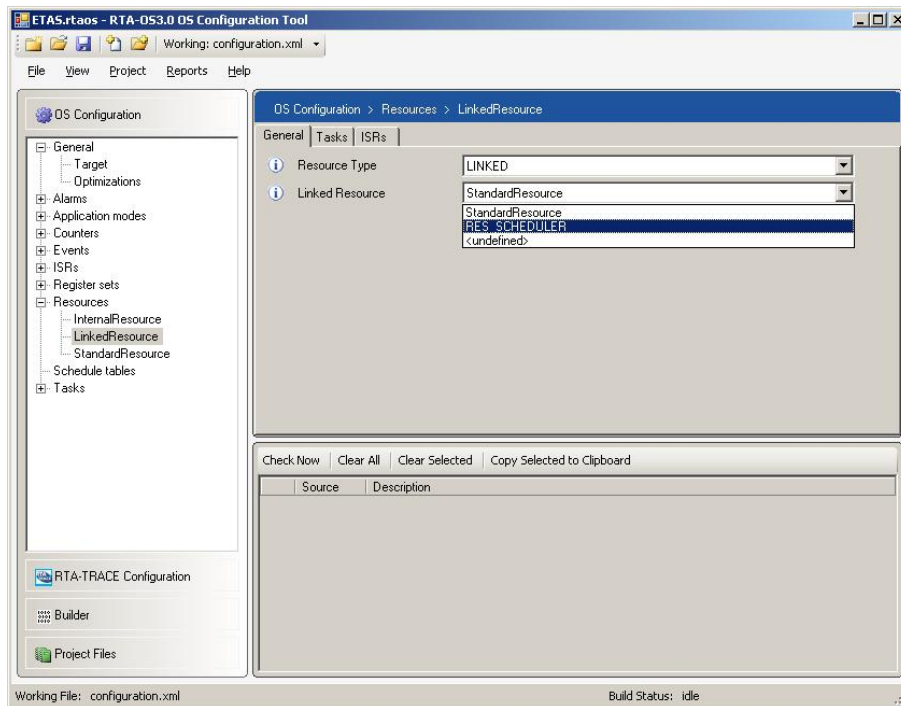


Figure 6.9: Linking to RES_SCHEDULER

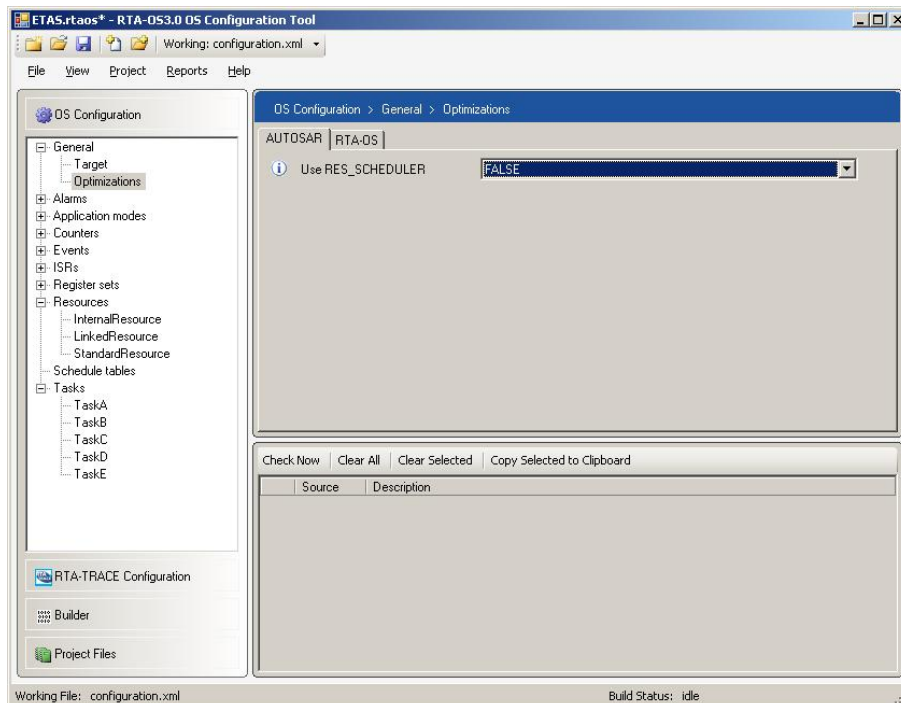


Figure 6.10: Disabling RES_SCHEDULER

However, there is an exception to this rule. This exception occurs when you have a short running task or ISR that makes many `GetResource()` and `ReleaseResource()` calls to the same resource. The cost of the API calls may then make up a significant part of the overall task execution time and, therefore, potentially the response time.

You may find that placing the entire task or ISR body between `GetResource()` and `ReleaseResource()` calls actually shortens the worst-case response time.

You should avoid using non-preemptive tasks and getting `RES_SCHEDULER` wherever possible. System responsiveness and schedulability is improved when resources are held for the minimum amount of time and when this affects the smallest number of tasks.

6.9 Avoiding Race Conditions

The AUTOSAR OS standard specifies that resources must be released before a `TerminateTask()` call is made. In some circumstances, this can introduce a race condition into your application. This can cause task activations to be missed (you learnt about race conditions at the beginning of this chapter).

Code Example 6.6 shows the type of system where race conditions can become a problem. Assume that two BCC1 tasks exchange data over a bounded buffer.

```
#include <Os.h>
TASK(Write)
    /* Highest priority */
    WriteBuffer();
    GetResource(Guard);
    BufferNotEmpty = True;
    ReleaseResource(Guard);
    ChainTask(Read);
}

TASK(Read)
    /* Lowest priority. */
    ReadBuffer();
    GetResource(Guard);
    if( BufferNotEmpty ) {
        ReleaseResource(Guard);
        /* !!! Race condition occurs here !!! */
        ChainTask(Read);
    } else {
        ReleaseResource(Guard);
    }
```



```
    /* !!! Race condition occurs here !!! */  
    TerminateTask();  
  }  
}
```

Code Example 6.6: A System where a Race Condition can Occur

In Code Example 6.6, between the resource being released and the task terminating, Read can be preempted by Write. When task Write chains task Read, the activation will be lost. This is because Read is still running. In other words a task is being activated, but it is not in the suspended state.

To solve this problem, you can allow queued activations of the Read task. This means that you should make the task BCC2. See Section 4.5.2 for more details.

6.10 Summary

- Resources are used to provide mutual exclusion over access to shared data or hardware resources.
- Tasks and ISRs can share any number of resources.
- All `GetResource()` and `ReleaseResource()` calls must be properly nested.
- All resources must be released before the task or ISR terminates.
- The scheduler can be used as a resource, but internal resources should be used in preference, if possible.
- Internal resources provide a cost free mechanism for controlling pre-emption between a group of tasks and ISRs

7 Events

In an AUTOSAR OS system, events are used to send signal information to tasks. This chapter explains what events are, how to configure them and how to use them at runtime.

Events can be used to provide multiple synchronization points for extended tasks. A visualization of synchronization is shown in Figure 7.1.

An extended task can wait on an event, causing the task to move into the waiting state. When an event is set by a task or ISR in the system, the waiting task is transferred into the ready state. When it becomes the highest priority ready task it will be selected to run by RTA-OS3.0.

Events are owned by the extended task with which they are associated. Usually, an extended task will be an infinite loop that contains a series of guarded wait calls for the events it owns. The event mechanism therefore allows you to build event driven state machines using OSEK.

If timing behavior is important in your system, all of your extended tasks (in other words, any task that waits for an event) must be lower priority than the basic tasks.

7.1 Configuring Events

Events are configured using `rtaoscfg`. The maximum number of events that can exist in your application is determined by your target hardware. You should consult the *RTA-OS3.0 Target/Compiler Port Guide* for your port to find out how many events you can have per task.

When an event is declared it must have:

- A name.
Names are used only to indicate the purpose of an event at configuration time.
- At least one task that uses it.
- An event mask.

The event name that is specified in `rtaoscfg` is used as a symbolic name for the event mask at run-time. A mask is an N-bit vector with a single bit set, where N is the maximum number of events on which a task can wait. The set bit identifies a particular event.

The event name is used at run-time as a symbolic name for the mask. The mask is configured by selecting the bit which indicates the event. Figure 7.2

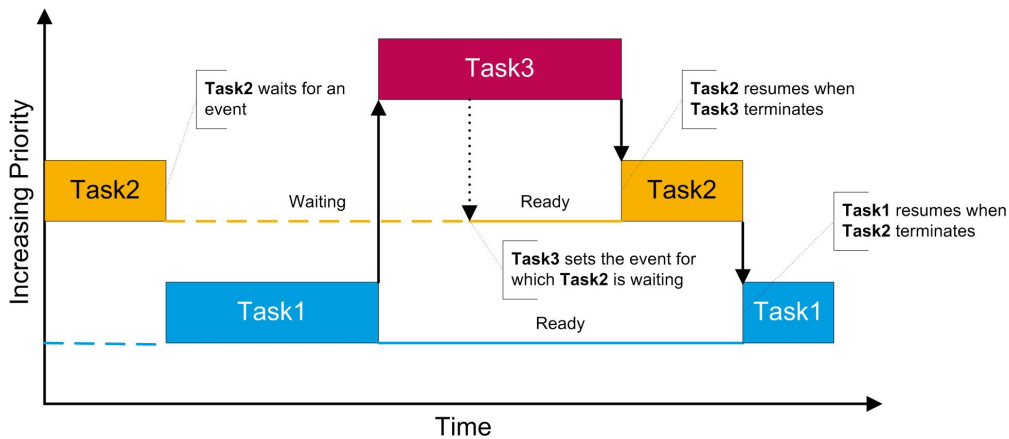


Figure 7.1: Visualizing Synchronization

shows that an event called Event1 has been declared which will be using bit nine in the event mask.

If an event is used by more than one task, each task has its own individual copy. When an event is set, a task must be specified at the same time. So, for example, if you set an event called Event2 for a task called t3, this has no effect on Event2 for the task t4.

7.1.1 Defining Waiting Tasks

Waiting tasks are selected using **rtaoscfg**. If you declare a task that waits on an event, it automatically means that it will be treated as an extended task.

Figure 7.3 shows that an event Event1 has been declared and that the tasks t1 and t2 have been configured to wait on the event.

An extended task that waits on an event will usually be auto-started and the task will never terminate. When the task starts executing, all the events it owns are cleared by RTA-OS3.0.

7.2 Waiting on Events

A task waits for an event using the `WaitEvent(EventMask)` API call. The EventMask must correspond to the one that is declared in **rtaoscfg**.

The `WaitEvent()` takes an event as its sole parameter. When the call executes there are two possibilities:

1. The event has not occurred

In this case the task will enter the waiting state and RTA-OS3.0 will run the highest priority task in the ready state.

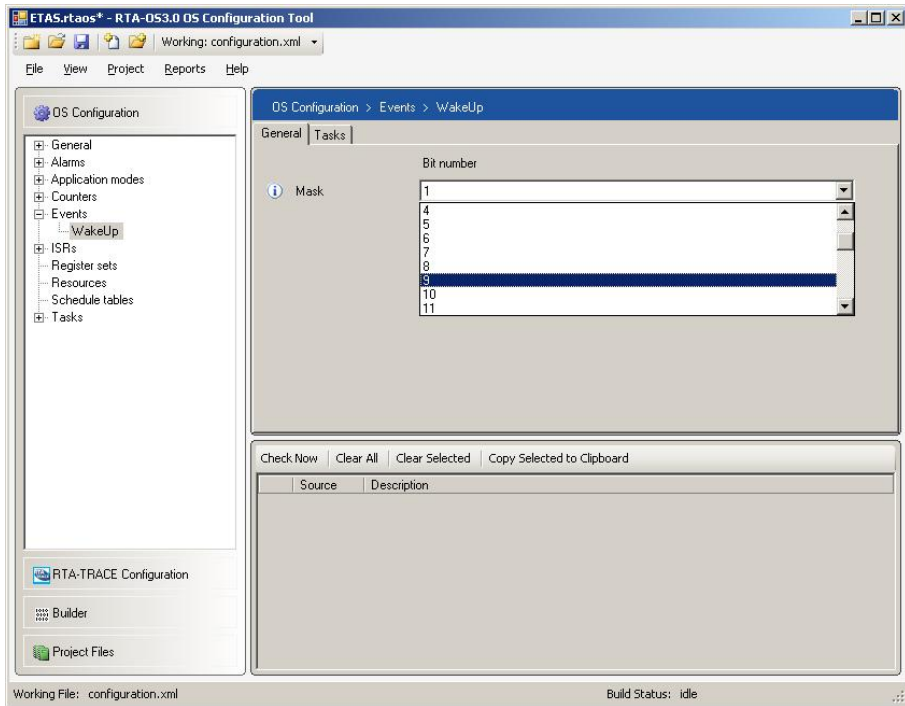


Figure 7.2: Configuring an Event mask in **rtaoscfg**

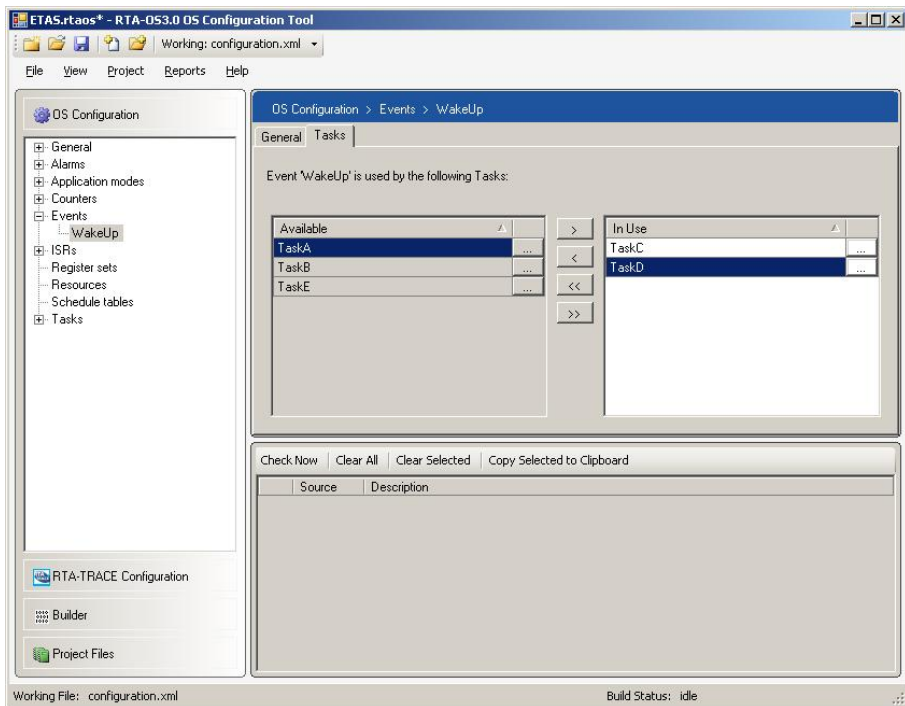


Figure 7.3: Selecting the Task to Wait on an Event

2. The event has occurred

In this case the task remains in the running state and will continue to execute at the statement immediately following the `WaitEvent()` call.

7.2.1 Single Events

To wait on a single event you simply pass in the event mask name to the API call. Code Example 7.1 shows how a task can wait for events.

```
#include <Os.h>
TASK(ExtendedTask) {
    ...
    WaitEvent(Event1); /* Task enters waiting state in API call
        if Event1 has not happened */
    /* When Event1 is set, ExtendedTask resumes here */
    ...
}
```

Code Example 7.1: Waiting on an Event

In AUTOSAR OS it is illegal to set events for a task that is in the suspended state. In practice this means that the structure of a task that waits on events is typically an infinite loop that waits on events as shown in Code Example 7.2.

```
#include <Os.h>
TASK(ExtendedTask){
    /* Entry state */
    while(true){
        WaitEvent(Event1);
        /* State 1 */
        WaitEvent(Event2);
        /* State 2 */
        WaitEvent(Event3);
        /* State 3 */
    }
    /* Task never terminates */
}
```

Code Example 7.2: Simple 3-state State Machine with Events

7.2.2 Multiple Events

Because an AUTOSAR OS event is just a bit mask, you can wait on multiple events at the same time by bit-wise 'OR'ing a set of bit masks.

When your task waits on multiple events it will be resumed when any one of the events upon which you are waiting occurs. When you resume from

waiting on multiple events, then you will need to work out which event (or events) has occurred.

OSEK provides the `GetEvent()` API call so that allows you to get the current set of events that are set for the task.

Code Example 7.3 shows how a task can wait on multiple events simultaneously and then identify which of the events has been set when it resumes.

```
#include <Os.h>
TASK(ExtendedTask){
    EventMaskType WhatHappened;
    while(true){
        WaitEvent(Event1|Event2|Event3);
        GetEvent(Task1, &WhatHappened);
        if( WhatHappened & Event1 ) {
            /* Take action on Event1 */
            ...
        } else if( WhatHappened & Event2 ) {
            /* Take action on Event2 */
            ...
        } else if( WhatHappened & Event3 ) {
            /* Take action on Event3 */
            ...
        }
    }
}
```

Code Example 7.3: Waiting on Multiple Events

7.2.3 Deadlock with Extended Tasks

While AUTOSAR OS provides freedom from deadlock in mutual exclusion over a critical section (see Chapter 6) you are not protected from building systems with events that can deadlock. If you have extended tasks that mutually set and wait on events sets, then it is possible that two (or more) tasks will be waiting on events that are only set by other tasks that are waiting. It is, of course, impossible for basic tasks in the system to deadlock, even if there are deadlocking extended tasks present.

Code Example 7.4 shows two tasks that will deadlock if there no other task set either Ev1 or Ev2.

```
#include <Os.h>
TASK(Task1) {
    while (1) {
        WaitEvent(Ev1);
    }
}
```

```

        /* Never reach here - DEADLOCKED with Task2! */
        SetEvent(Task2,Ev2)
    }
}
TASK(Task2) {
    while (1) {
        WaitEvent(Ev2);
        /* Never reach here - DEADLOCKED with Task1! */
        SetEvent(Task1,Ev1)
    }
}

```

Code Example 7.4: Deadlock with Extended Tasks

OS configuration does not capture which tasks/ISRs set events, only which tasks can wait on events. It is therefore impossible for RTA-OS3.0 to statically determine whether your extended tasks will deadlock or not. However, the following design approaches may help:

- use basic tasks only;
- analyze your code to show that there is no circular waiting of events on the transitive closure of all SetEvent()/WaitEvent() pairs.

7.3 Setting Events

Events are set using the SetEvent() API call.

The SetEvent() call has two parameters, a task and an event mask. For the specified task, the SetEvent() call sets the events that are specified in the event mask. The call does not set the events for any other tasks that share the events.

You can bit-wise 'OR' multiple event masks in a call to SetEvent() to set multiple events for a task at the same time

Events cannot be set for tasks that are in the suspended state. So, before setting the event, you must be sure that the task is not suspended. You can do this using the GetTaskState() API call, but note that there is a potential race-condition when this is called for tasks of higher priority than the caller. The caller may be preempted between the call to the API and the evaluation of the result and the state of the task that was requested may have changed in the intervening time.

An extended task is moved from the waiting state into the ready state when any one of the events that it is waiting on is set.

Code Example 7.5 shows you how a task can set events.

```
#include <Os.h>
TASK(Task1) {
    TaskStateType TaskState;

    /* Set a single event */
    SetEvent(Task2, Event1);

    /* Set multiple events */
    SetEvent(Task3, Event1 | Event2 | Event3);
    ...
    /* Checking for the suspended state */
    GetTaskState(Task2,&TaskState);
    if (TaskState != SUSPENDED) {
        SetEvent(Task2, Event1);
    }
    ...
    TerminateTask();
}
```

Code Example 7.5: Setting Events

A number of tasks can wait on a single event. However, you can see from Code Example 7.5 that there is no broadcast mechanism for events. In other words, you cannot signal the occurrence of an event to all tasks waiting on the event with a single API call.

Events can also be set by alarms and schedule tables.

7.3.1 Setting Events with an Alarm

Alarms can be used to periodically activate extended tasks that don't terminate. Each time the alarm expires, the event is set. The task waiting on the event is then made ready to run.

7.3.2 Setting Events with a Schedule Table Expiry Point

Expiry points on schedule tables can be used to program (a)periodic activations of extended tasks that do not terminate. Each time the expiry point is processed, the event is set. The task waiting on the event is then made ready to run.

7.4 Clearing Events

An event can be set by any task or ISR, but it can only be cleared by the owner of the event.

When a task waits on an event, and the event occurs, then a subsequent call to `WaitEvent()` for the same event will return immediately because the event is still set.

Before waiting for the event occurring again the last event occurrence of the event must be cleared.

Events are cleared using the `ClearEvent(EventMask)` API call. The `EventMask` must correspond to the one that is declared.

Code Example 7.6 shows how a task typically uses `ClearEvent()`.

```
#include <Os.h>
TASK(ExtendedTask){
    EventMaskType WhatHappened;
    ...
    while( WaitEvent(Event1|Event2|Event3)==E_OK ) {
        GetEvent(Task1, & WhatHappened);
        if(WhatHappened & Event1 ) {
            ClearEvent(Event1);
            /* Take action on Event1 */
            ...
        } else if( WhatHappened & (Event2 | Event3 ) {
            ClearEvent(Event2 | Event3);
            /* Take action on Event2 or Event3*/
            ...
        }
    }
}
```

Code Example 7.6: Clearing Events

When a task terminates all the events that it owns are cleared automatically.

7.5 Simulating Extended Tasks with Basic Tasks

Basic tasks can only synchronize at the start or end of task execution.

If other synchronization points are required then the event mechanism provides one way to do this. However, extended tasks typically have greater overheads than basic tasks. On resource-constrained systems, synchronization can be built using basic tasks only.

For example, if a task is built as a state machine (using a C switch statement, for instance) then you can set a state variable, issue a `TerminateTask()` call and wait for re-activation. Code Example 7.7 shows how this can be achieved.

```
#include <Os.h>
```

```

/* Create a "State" variable that remains in scope between task
   activations */
uint8 State;
TASK(Task1) {
    switch (State) {
        case 0:
            /* Synchronization point 0. */
            State = 1;
            break;
        case 1:
            /* Synchronization point 1. */
            State = 2;
            break;
        case 2:
            /* Synchronization point 2. */
            State = 0;
            break;
    }
    TerminateTask();
}

```

Code Example 7.7: Multiple Synchronization Points in a Basic Task

7.6 Summary

- Events are synchronization objects that can be waited on by extended tasks.
- An event can be used by multiple tasks.
- Setting an event is not a broadcast mechanism to signal all tasks that are waiting.
- Tasks, ISRs, alarms and schedule tables can set events.

8 Counters

Counters register how many “things” have happened in the OS in terms of ticks. A tick is an abstract unit. It is up to you to decide what you want a tick to mean and, therefore, what are the “things” the counter is counting.

You might define a tick to be:

- Time, for example a millisecond, microsecond, minute etc and the counter then tells you how much time has elapsed.
- Rotation, for example in degrees or minutes, in which case the counter would tell you by how much something has rotated.
- Button Presses, in which case the counter would tell you how many times the button has been pressed.
- Errors, in which case the counter is counting how often an error has occurred.

An ISR (or sometimes a task) is used to drive a counter. The driver is responsible for making the correct RTA-OS3.0 API call to “tick” the counter or to tell RTA-OS3.0 that the counter has “ticked” to a required value.

8.1 Configuring Counters

Each counter has 4 mandatory attributes:

Name is the name of the counter. RTA-OS3.0 creates a handle for each counter using an identifier of the same name as the counter.

Type defines the counter model. AUTOSAR provides two models

Software counters are those where the count value is maintained internally by the OS. You will need to provide a counter driver that tells the RTA-OS3.0 to increment the counter by one tick. Further details are provided in Section [8.2.1](#).

Hardware counters are those where a peripheral maintains the count value. You will need to provide a counter driver that tells the OS when a requested number of ticks have elapsed. The OS will also require your driver to provide implementations of callback routines that RTA-OS3.0 uses to manage the peripheral at runtime. Further details are provided in Section [8.2.2](#).

A software counter is sufficient when you need a relatively low resolution, for example one millisecond or greater. You should use a hardware counter when you need very high resolution for example in the mi-

crosecond range, or where you need to accurately synchronize scheduling of tasks in RTA-OS3.0 to an external source, for example a TPU or a global (network) time source.

Maximum Value defines the maximum count value for the counter. All counters wrap around to zero on the tick after the maximum allowed value has been reached¹. In many cases, you will simply use a full modulus wrap for the counter, so this will be 65535 ($2^{16} - 1$) for a 16-bit counter and 4294967295 ($2^{32} - 1$) for a 32-bit counter. The maximum counter value for your port can be found in your *RTA-OS3.0 Target/Compiler Port Guide*. This corresponds to the AUTOSAR OS counter attribute MAXALLOWEDVALUE.



For hardware counter you must ensure that MAXALLOWEDVALUE+1 is equal to the modulus of the peripheral.

Minimum Cycle defines the shortest number of ticks allowed when setting a cycle value for an alarm or a schedule table offset. In most cases, you will want this to be 1 tick. However, if you want to build systems where you enforce a minimum separation between alarms on the counter, then you may choose a larger value. This corresponds to the AUTOSAR OS counter attribute MINCYCLE.

Ticks per base is a legacy attribute from AUTOSAR OS that defined the number of underlying counter driver ticks required for each tick on the counter. You can assign any value to this attribute because it is not used by RTA-OS3.0. This corresponds to the AUTOSAR OS attribute TICKSPERBASE.

There is an additional optional attribute:

Seconds Per Tick defines the duration of a tick of the counter in seconds. This should be defined if you want to use the tick/time conversion features provided by AUTOSAR OS. Further details are given in Section 8.5.

Figure 8.1 shows how a counter called MillisecondCounter is declared.

8.2 Counter Drivers

RTA-OS3.0 does not take control of any of your hardware to provide counter drivers. This makes RTA-OS3.0 very easy to integrate with any tick source for example timer ticks, error counts, button presses, TPU peripherals, etc. This means that you need to provide a driver for every counter you declare in RTA-OS3.0 and interface this to the OS.

¹This means that the maximum allowed value is equal to the modulus-1 of the counter.

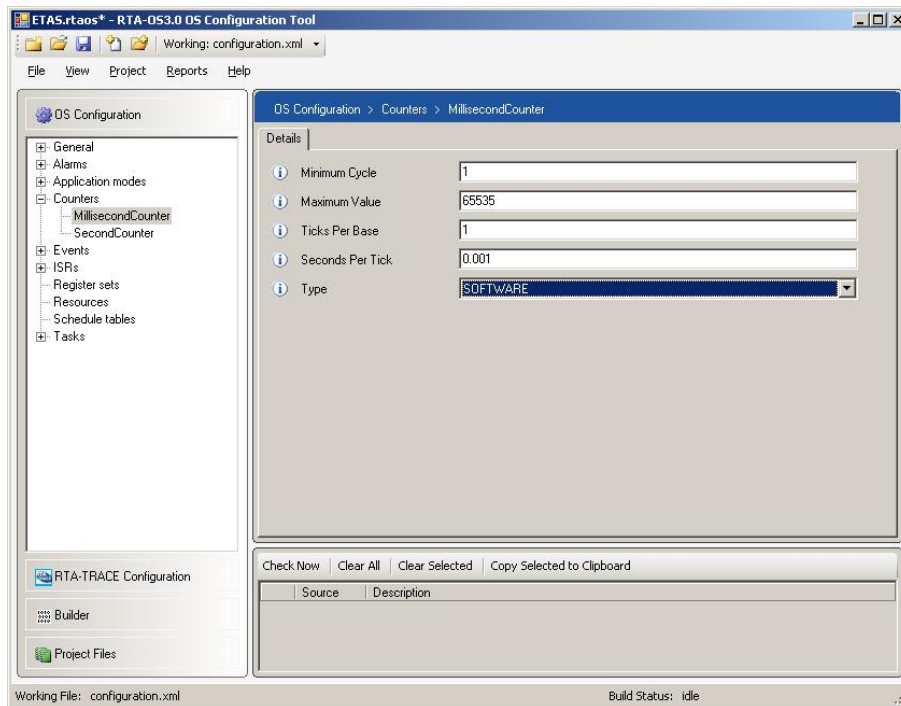


Figure 8.1: Declaring a Counter

The interface between the driver and the counter depends on the counter's type:

Software Counters are incremented by an API call.

Hardware Counters The count value is held in an external hardware peripheral. Your application must provide a more complex driver that tells RTA-OS3.0 when a requested number of ticks have elapsed. RTA-OS3.0 uses special callbacks to set a requested number of ticks, cancel a request, get the current count value and get the status of the counter.

8.2.1 Software Counter Drivers

For each of your software counters, you need to provide the driver that provides the tick. All software counters are initialized to zero by RTA-OS3.0 during `StartOS()` and count upwards.

The software counter driver model is standardized in AUTOSAR OS and is shown in Figure 8.2.

Incrementing Software Counters

You use the API call `IncrementCounter(CounterID)` to increment the counter value held in RTA-OS3.0. The software counter wraps around to zero when

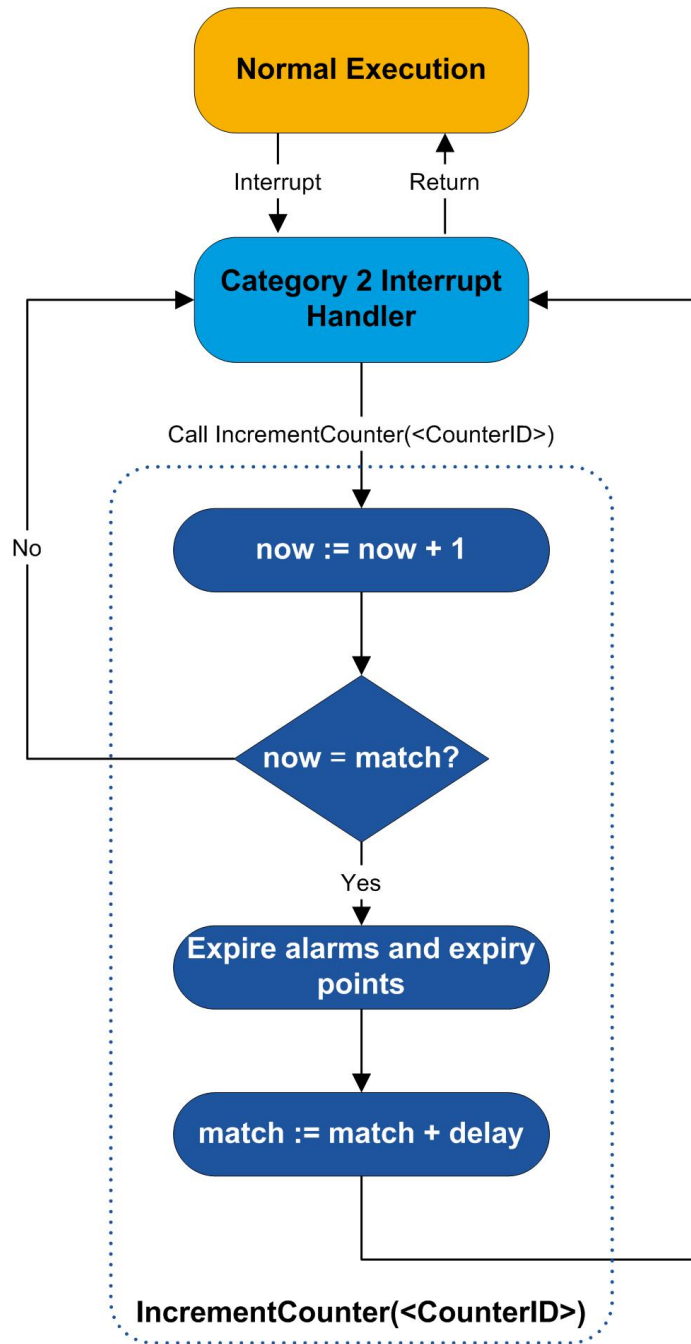


Figure 8.2: Ticked Counter Driver Model

one is added to MAXALLOWEDVALUE.

You can make the call to `IncrementCounter(CounterID)` from most places in your application code. One of the most common uses of a counter is to provide a time-base to RTA-OS3.0 for activating tasks based on alarms (see Chapter 9) or schedule tables (see Chapter 10). In this case, you will need to provide a periodic timer interrupt that calls `IncrementCounter(CounterID)` on each expiry.

Code Example 8.1 shows how a millisecond interrupt would driver a counter called `TimeCounter`.

```
#include <Os.h>
ISR(HandleTimerInterrupt) {
    DismissTimerInterrupt();
    IncrementCounter(TimeCounter);
}
```

Code Example 8.1: Using a periodic interrupt to tick a software counter

Another common use of software counters is as part of a fault-tolerant system where you take some action when an error threshold is exceeded. A software counter can be used to register the number of errors and you can then use an alarm to trigger a recovery action (for example, activate an error recovery task).

Code Example 8.2 shows how a task called `Critical` might log errors on a counter called `ErrorCounter`.

```
#include <Os.h>
TASK(Critical){
    ...
    if (Error) {
        IncrementCounter(ErrorCounter);
    }
    ...
    TerminateTask();
}
```

Code Example 8.2: Using a periodic interrupt too tick a software counter

Static Counter Interface

As the AUTOSAR API call takes the name of a counter as a parameter, this means that RTA-OS3.0 must internally de-reference the parameter before updating the OS data structures. It also means that the compiler needs to push a parameter on the stack on entry.

Typically however, you know at build time which counter you will be ticking from where. You will also probably be driving the counter from an interrupt handler - the last place where you need to waste time unnecessarily.

RTA-OS3.0 recognizes this and generates a dedicated API call called `Os_IncrementCounter_<CounterID>()` for each counter that has been declared in the configuration file (where `CounterID` is the name of the counter).



The API call `Os_IncrementCounter_<CounterID>()` is not necessarily portable to other AUTOSAR OS implementations.

As an example, consider an application containing two counters, one called `TimeCounter` and one called `AngularCounter`. **rtaosgen** will generate the two API calls shown in Code Example 8.3.

```
Os_IncrementCounter_TimeCounter();  
Os_IncrementCounter_AngularCounter();
```

Code Example 8.3: Static Software Counter Interface

The interrupt handlers that you supply to service the timer and angular interrupts must call these API calls.

Code Example 8.4 shows how these interrupt handlers might look.

```
#include <Os.h>  
ISR(HandleTimerInterrupt) {  
    ServiceTimerInterrupt();  
    Os_IncrementCounter_TimeCounter();  
}  
ISR(HandleAngularInterrupt) {  
    ServiceAngularInterrupt();  
    Os_IncrementCounter_AngularCounter();  
}
```

Code Example 8.4: Interrupt Handlers for Code Example 8.3

If you have multiple software counters that you need to tick at the same rate, then you can make multiple `Os_IncrementCounter_<CounterID>()` calls within your handler as shown in Code Example 8.5

```
#include <Os.h>  
ISR(MillisecondInterrupt) {  
    ServiceTimerInterrupt();  
    Os_IncrementCounter_Counter1();  
    Os_IncrementCounter_Counter2();  
    ...  
    Os_IncrementCounter_CounterN();  
}
```


}

Code Example 8.5: Making multiple calls to the static software counter interface



There is an `Os_IncrementCounter_<CounterID>()` API call available for each counter you declare. These ‘static’ API calls are faster and use less RAM than the AUTOSAR `IncrementCounter(<CounterID>)` API call because the calls do not require a parameter and do not need to work out which counter is being ticked. You should decide which version is appropriate for your application and choose accordingly.

8.2.2 Hardware Counter Drivers

For each of your hardware counters, you need to provide the hardware counter driver that calls RTA-OS3.0 and a set of callbacks that are used by RTA-OS3.0. As with software counters, RTA-OS3.0 provides a well-defined interface for connecting the advanced counter driver to the OS.



The AUTOSAR OS standard does not specify a standard API call for dealing with hardware counters. If you are porting your application from another OS to RTA-OS3.0, then you may need to change the hardware counter driver API calls.

For each hardware counter, RTA-OS3.0 knows what the next action driven by the counter is, whether that is to expire an alarm or process an expiry point on a schedule table or both. RTA-OS3.0 also knows how many ticks need to elapse before this happens. This is called the **match** value.

When you use a software counter, *the driver tells RTA-OS3.0* each time a tick has elapsed. RTA-OS3.0 counts ticks internally and, when the match value is reached, the action is taken. RTA-OS then calculates the next match value and the process repeats.

By contrast, when you use an hardware counter, *RTA-OS3.0 tells the driver*, through a callback function, when the next action is needed. Your peripheral counts the requested number of ticks and generates an interrupt when the correct number have elapsed. In the interrupt handler you make the `Os_AdvanceCounter_<CounterID>()` API call to tell RTA-OS3.0 to process the next action due on `CounterID`. RTA-OS3.0 does this and the process repeats.

The driver model is shown in Figure 8.3.

Normally, you will use an interrupt to drive both software and hardware counters. With a software counter, an interrupt occurs for each counter tick, *whether or not there is anything for RTA-OS3.0 to do*. With a hardware counter, an interrupt occurs only when RTA-OS3.0 needs to do something. This means that hardware counters reduce interrupt interference to the ab-

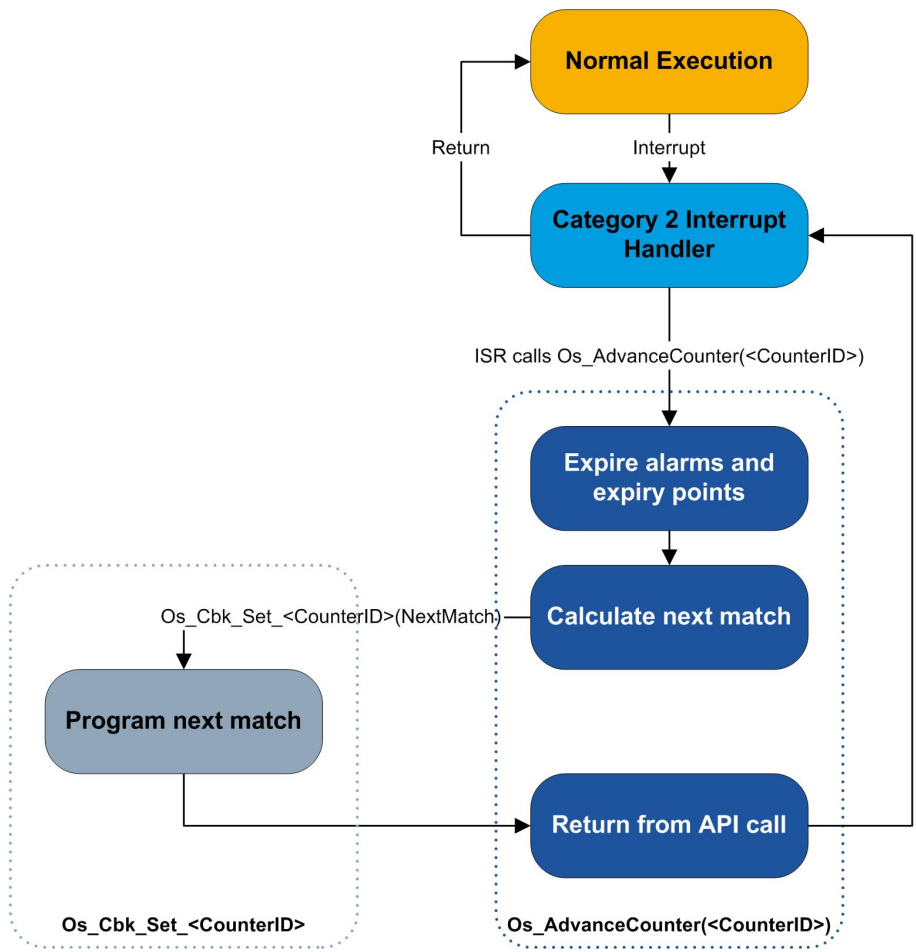


Figure 8.3: Advanced Counter Driver Model

solute minimum required.

Advancing Hardware Counters

You use the API call `Os_AdvanceCounter_<CounterID>()` to tell RTA-OS3.0 that the match value has been reached.



You are responsible for writing the driver that calls `Os_AdvanceCounter_<CounterID>()` and ensuring that the next action is taken at the correct time.

The `Os_AdvanceCounter_<CounterID>()` API call cause the next alarm and/or expiry point to be processed and will set up the next match value by calling a callback you provide or, if there are no actions left to do (i.e. there are no active alarms or schedule tables on the counter), cancel interrupts from the driver. More detailed information about writing hardware counter drivers can be found in Chapter 11.

Callback Functions

For a software counter communication is one way - the driver tells RTA-OS3.0 when a *single* tick has happened. For hardware counters the driver has to tell RTA-OS3.0 when *multiple* ticks have happened. However, RTA-OS3.0 also needs to tell the driver to driver do things. This is done using a set of callback functions that provide an abstraction between RTA-OS3.0 and any type of peripheral you want to use as the driver. The exact functionality of the callbacks depends on the peripheral you are using as your hardware counter driver. Further information on writing callbacks can be found in Chapter 11.

However, by way of a short overview, four callbacks are required:

`Os_Cbk_Set_<CounterID>`

This callback sets up the state for an interrupt to occur when the next action is due. The callback is passed the absolute value of the counter at which an action should take place. For counters, this callback is used in two distinct cases:

1. Starting

Setting the initial interrupt source when a schedule table or an alarm is started on the counter.

2. Resetting

Shortening the time to the next counter expiry.

The second case is needed because you can, for example, make a `SetRelAlarm(WakeUp, 100)` call when the next interrupt is due in more than 100 ticks.

Os_Cbk_State_<CounterID>

This callback returns whether the next action on the counter is pending or not and, if the action is not pending, the number of ticks remaining until the match value is reached.

Os_Cbk_Now_<CounterID>

This callback needs to return the current value of the external counter. This is used for the `GetCounterValue()` API call. See Section 8.4.

Os_Cbk_Cancel_<CounterID>

This callback must clear any pending interrupt for your counter and ensure that the interrupt cannot become pending until a `Os_Cbk_Set_<CounterID>()` call is made. If you do not cancel all the alarms on the counter and/or stop schedule tables driven by the counter, then this call is not needed.

8.3 Accessing Counter Attributes at Runtime

The RTA-OS3.0 API call `GetAlarmBase()` always returns the configured counter values. The structure of `GetAlarmBase()` is shown in Code Example 8.6.

```
AlarmBaseType Info;  
GetAlarmBase(Alarm2, &Info);  
MaxValue = Info.maxallowedvalue;  
BaseTicks = Info.ticksperbase;  
MinCycle = Info.mincycle;
```

Code Example 8.6: Using `GetAlarmBase()` to read static counter attributes

The configured values are can also be accessed as symbolic constants in the form shown below.

- `OSMAXALLOWEDVALUE_<CounterID>`
- `OSTICKSPERBASE_<CounterID>`
- `OSMINCYCLE_<CounterID>`

So Code Example 8.6 above could also have been written as shown in Code Example 8.7:

```
MaxValue = OSMAXALLOWEVALUE_Alarm2;  
BaseTicks = OSTICKSPERBASE_Alarm2;  
MinCycle = OSMINCYCLE_Alarm2;
```

Code Example 8.7: Using macros to read static counter attributes

8.3.1 Special Counter Names

If a counter with the name `SystemCounter` is created, then it is possible in AUTOSAR OS to access the associated counter attributes with a short form of the macros by omitting the trailing `_CounterID`:

```
OSMAXALLOWEDVALUE_SystemCounter → OSMAXALLOWEDVALUE
OSTICKSPERBASE_SystemCounter     → OSTICKSPERBASE
OSMINCYCLE_SystemCounter         → OSMINCYCLE
```

RTA-OS3.0 generates both forms of the macros for `SystemCounter` and you can use either version.

The `SystemCounter` also provides an additional macro to get the duration of a tick of the counter in nanoseconds called `OSTICKDURATION`. This macro requires the counter attribute “Seconds Per Tick” to be configured.

8.4 Reading Counter Values

You may find that your application has the need to be able to read the current value of a counter at runtime. For example, you might want to know how many errors an error counter has logged, how many times a button has been pressed or how much time has elapsed.

The current value of a counter can be read at runtime by calling the `GetCounterValue()` API as show in Code Example 8.8.

```
TickType HowMany;
GetCounterValue(ButtonPresses, &HowMany);
```

Code Example 8.8: Using `GetCounterValue()`

When you use `GetCounterValue()` you should be aware that:

- counters wrap around from `MAXALLOWEDVALUE` to zero, so the calculation needs to compensate for the wrap
- preemption can occur at the point the call returns meaning that when you resume the value of ‘Now’ will be old.
- when using a hardware counter, the counter driver will still be incrementing when the call returns. Even when preemption does not occur, the calculation performed immediately will be based on old data.

If you need to perform a simple calculation to work out how many ticks of the counter have elapsed since a previously read value, then you can avoid this potential race-condition by using the `GetElapsedCounterValue()` API call. The call takes a previously read counter value as input and calculates the

ticks that have elapsed, including compensation for the counter wrapping. The calculation occurs at OS level (i.e. with interrupts disabled) so does not suffer from preemption effects.

Code Example 8.9 shows how you might use this feature to measure the end-to-end (response) time of a task.

```
#include <Os.h>
TickType Start;
ISR(CaptureTrigger){
    /* Dismiss interrupt */
    GetCounterValue(TimeCounter,&Start);
    ...
    ActivateTask(GenerateResponse);
}
TASK(GenerateResponse){
    TickType Finish;
    CalculateValue();
    WriteToDevice();
    GetElapsedCounterValue(TimeCounter,&Start,&Finish);
    ...
    TerminateTask();
}
```

Code Example 8.9: Using GetElapsedCounterValue()

If your counter is counting time ticks (as in Code Example 8.9), then this is referred to in AUTOSAR OS as a “free running timer”. There is nothing special about this type of counter - it is identical to any other type of counter - the only distinction is that the counter is one which is driven by a timer tick source.


The intended use of the free running timer functionality is to measure short, high accuracy, durations at runtime. If you need to do this, then you will probably need to use a hardware counter to get the required counter resolution.

8.5 Tick to Time Conversions

It is common for counters to be used as a time-base reference for the OS. For most of the applications that you write, the relative timing of events will be the real-time values determined by your system requirements. You will most likely think about system configuration in terms of real-time values, nanoseconds, milliseconds etc, rather than in the more abstract notion of ticks.

If a counter configuration parameter ‘Seconds Per Tick’ has been configured, then RTA-OS3.0 generates macros for you to use to convert between ticks

and real-time units.

 *AUTOSAR OS states that tick to time conversion is for hardware counters only. However, the feature is generally useful for both software and hardware counters and the AUTOSAR XML configuration language supports configuration for both types of counter. In RTA-OS3.0 this anomaly is resolved by providing tick to time conversion for both software and hardware counters. However, you should note that the provision of these macros for software counters is not necessarily supported by other AUTOSAR OS implementations.*

The following macros are provided:

- `OS_TICKS2NS_CounterID(ticks)` converts ticks to nanoseconds
- `OS_TICKS2US_CounterID(ticks)` converts ticks to microseconds
- `OS_TICKS2MS_CounterID(ticks)` converts ticks to milliseconds
- `OS_TICKS2SEC_CounterID(ticks)` converts ticks to seconds

The values returned by these macros are of `PhysicalTimeType` rather than `TickTypes` that are used by the API calls that you might use the macros with, so you will need to cast them to an appropriate type.

Code Example 8.10 shows how these macros might be used in your application code to program a timeout using a statically defined “timeout” value.

```
#define TIMEOUT_MS 100 /* Set a timeout to be 100ms */
TickType TimeoutInTicks;
TimeoutInTicks = (TickType)((PhysicalTimeType)TIMEOUT_MS/
    OS_TICKS2MS_TimeCounter(1));
SetRelAlarm(TimeoutAlarm, TimeoutInTicks, 0);
```


Code Example 8.10: Programming an alarm with time rather than ticks (1)

In addition to these macros RTA-OS3.0, generates a macro called `OSTICKDURATION_<CounterID>` that returns the duration of a counter tick in nanoseconds so this makes it extremely useful if you want to program alarms of a fixed time, even if you change the underlying counter tick rate. Code Example 8.11 shows how Code Example 8.10 can be reworked using the `OSTICKDURATION_<CounterID>` macro. This version offers slightly better performance because the duration of a single tick does not need to be calculated at runtime.

```
#define TIMEOUT_NS 100000000 /* Set a timeout to be 100ms */
TickType TimeoutInTicks;
```

```
TimeoutInTicks = (TickType)(TIMEOUT_NS/  
    OSTICKDURATION_TimeCounter);  
SetRelAlarm(TimeoutAlarm, TimeoutInTicks, 0);
```

Code Example 8.11: Programming an alarm with time rather than ticks (2)

 The `OSTICKDURATION_<CounterID>` macros are provided by RTA-OS3.0 and are not part of the AUTOSAR OS standard. Use of the macros is not portable to other implementations.

8.6 Summary

- Counters are used to register a count of some tick source.
- Counters are either software or hardware counters. You need to provide the appropriate driver for the type of the counter you configure.

9 Alarms

It is possible to construct systems that activate tasks at different rates using ISRs. However, for complex systems, this can become inefficient and impractical. Alarms provide a more convenient, and more portable, way of scheduling systems.

The alarm mechanism consists of two parts:

1. A counter.
These were covered in [Chapter 8](#).
2. One or more alarms attached to the counter.

The alarm part specifies an action to perform when a particular counter value is reached. Each counter in your system can have any number of alarms attached.

An alarm is said to have expired when the value of a counter equals the value of an alarm attached to the counter. On expiry, RTA-OS3.0 will perform the action associated with the alarm. The action could be to activate a task, to execute an alarm callback routine, set an event or tick a software counter.

The alarm expiry value can be defined relative to the actual counter value or as an absolute value. If the alarm expiry is defined as relative to the actual counter, it is known as a relative alarm. If it is defined as an absolute value, it is known as an absolute alarm.

Alarms can be configured to expire once. An alarm that expires once is called a single-shot alarm.

An alarm can also be specified to expire on a periodic basis. This type of alarm is called a cyclic alarm. You can find out more about cyclic alarms in [Section 9.2](#).

9.1 Configuring Alarms

There are three parts to alarm configuration:

1. Naming - Each alarm in your system needs to be allocated a unique name. As for other OS objects, this is the name that you will use in your code to refer to the alarm at runtime.
2. Association of a counter - An alarm is statically bound to a counter at configuration time. Any setting of the alarm is done in terms of ticks of the associated counter.

3. Specification of the alarm's action.

Each alarm that you create is associated with up to 4 actions:

1. Activate a task.
2. Raise an event.
3. Execute a callback function.
4. Increment a (software) counter.

If you need to activate multiple tasks, set multiple events, make multiple callbacks or increment multiple counters on expiry, you will need multiple alarms with the same expiry value. (Schedule Tables provide an alternative mechanism that allows you to activate multiple tasks and/or set multiple events simultaneously. You can read about Schedule Tables in Chapter 10).

9.1.1 Activating a Task

The most common action for an alarm is to activate a task. This is the basis for building systems with periodically activated tasks - you create an alarm for each task and then program the alarm to occur at the required period. Figure 9.1 shows how to configure an alarm to activate a task.

In AUTOSAR OS, you may only activate a single task for each alarm. If you need multiple tasks to run when an alarm expires, then you can do this either by creating multiple alarms or by using task activation chains (see Section 4.9.1).

9.1.2 Setting an Event

An alarm can set an event for a specified task. When an event is set with an alarm, it has the same properties as it would if it were set using the `SetEvent()` API call. This means you need to specify both the event and the task for which the event is to be set. Figure 9.2 shows you how to set an event action for an alarm.

9.1.3 Alarm Callbacks

Each alarm can have an associated callback function. The callback is simply a C function that is called when the alarm expires.

Figure 9.3 shows how to configure a callback routine for an alarm.

Each callback routine must be written using the `ALARMCALLBACK()` macro, shown in Code Example 9.1.

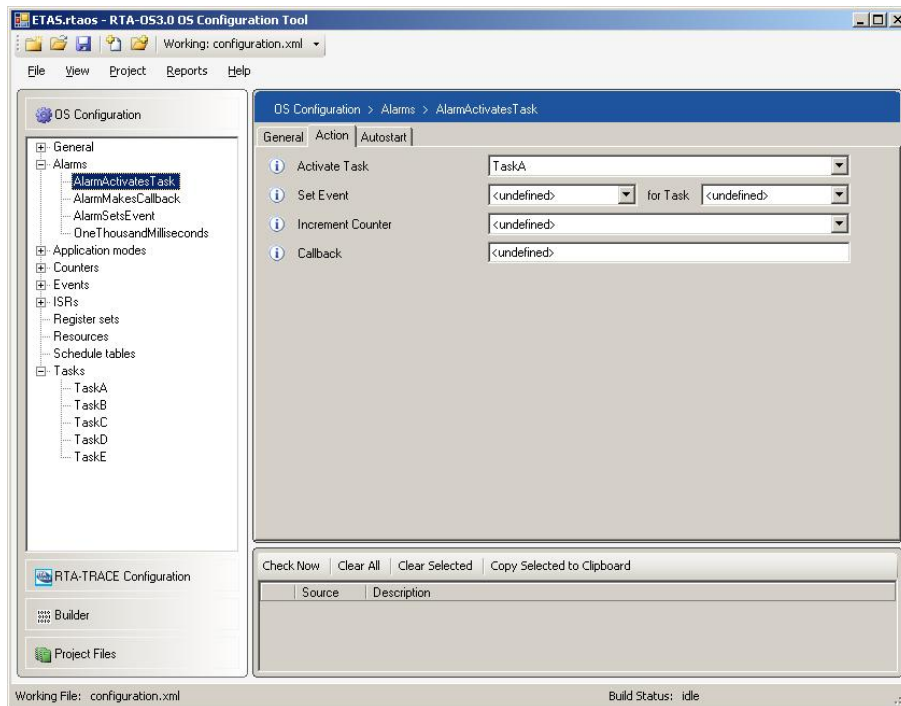


Figure 9.1: Activating a Task with an Alarm

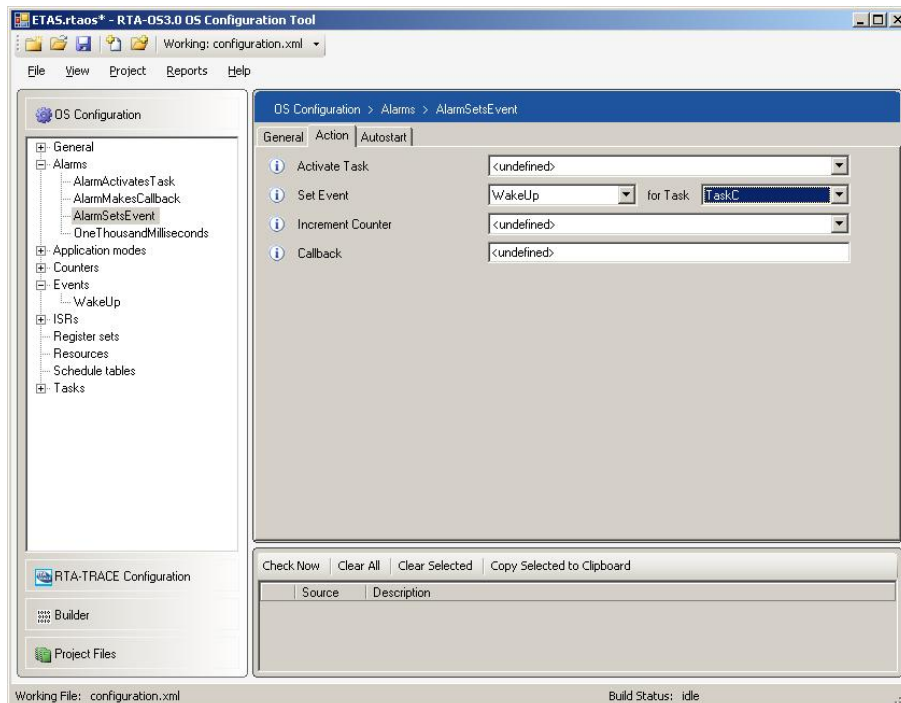


Figure 9.2: Setting an Event for a Task with an Alarm

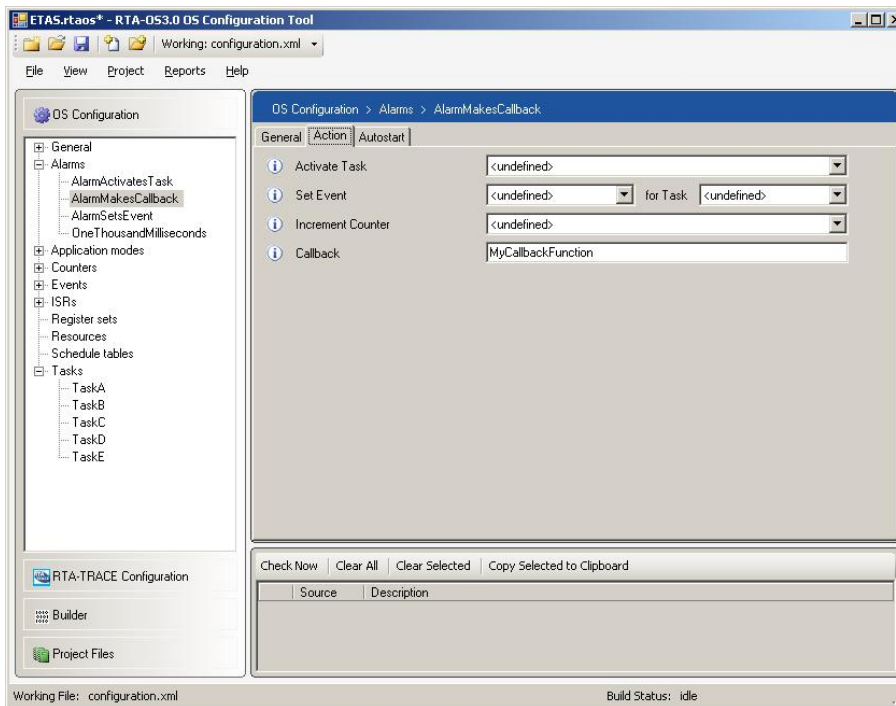


Figure 9.3: Configuring a Callback Routine for an Alarm

```
ALARMCALLBACK(UserProvidedAlarmCallback) {

    /* Callback code. */

}
```

Code Example 9.1: An Alarm Callback



Callback routines run at OS level, which means Category 2 interrupts are disabled. You should therefore aim to keep your callback routines as short as possible to minimize the amount of blocking that your tasks and ISRs suffer at runtime.

The only RTA-OS3.0 API calls that you can make inside the callback are the `SuspendAllInterrupts()` and `ResumeAllInterrupts()` calls.

9.1.4 Incrementing a Counter

Incrementing a software counter from an alarm allows you to cascade multiple counters from a single ISR. A counter ticked from an alarm inherits the period of the alarm. So, if you have an alarm that occurs every 5 milliseconds, you can use the alarm to drive a second ticked counter that ticks every 5 milliseconds. Figure 9.4 shows you how this is configured in RTA-OS3.0.

Code Example 9.2 shows how you would drive `Counter1ms` from an interrupt.

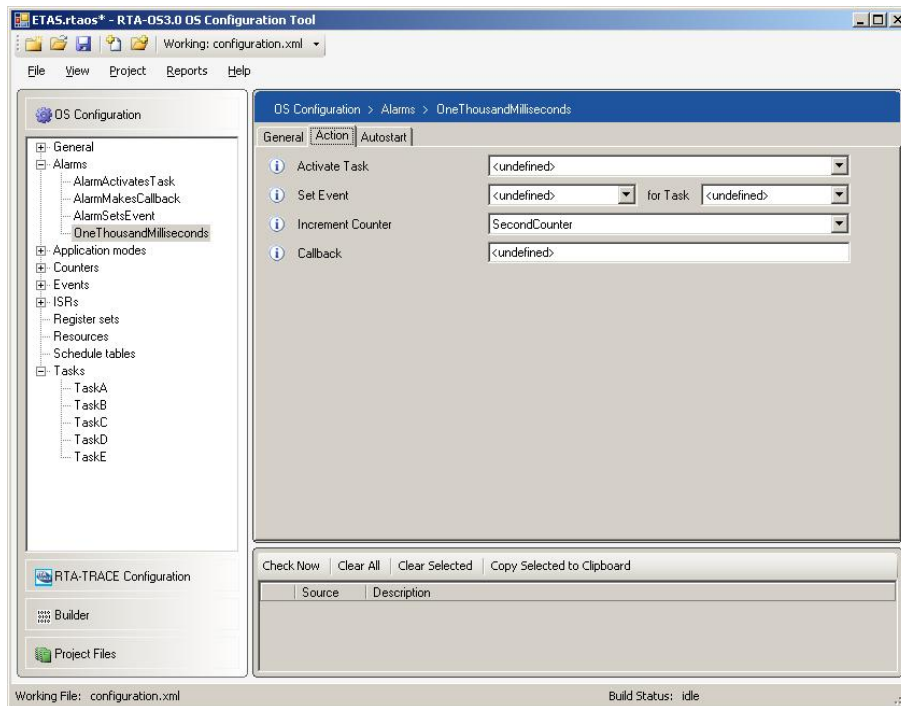


Figure 9.4: Cascading counter increments from an alarm

Every fifth interrupt registered on Counter1ms would cause the alarm to expire and increment the cascaded Counter5s :

```
#include <Os.h>
ISR(MillisecondInterrupt){
    CLEAR_PENDING_INTERRUPT();
    Os_IncrementCounter(Counter1ms);
    /* Every 5th call internally performs Os_IncrementCounter(
        Counter5ms) */
}
```

Code Example 9.2: Cascading Counters

Cascaded counters must have a tick rate that is an integer multiple of the counter driving the alarm. You can configure systems with multiple levels of cascading. However, RTA-OS3.0 will generate an error if you try and configure a system with a cycle in the cascade or you try and increment a hardware counter.



The timing properties of a cascaded counter are defined relative to timing properties of the first counter in the cascade. The earliest counter in the cascade therefore determines the base tick rate from which all other counters are defined. If you change the tick rate of the earliest counter, then the entire timing behavior of the application will be scaled accordingly.

9.2 Setting Alarms

Two API calls are provided for setting alarms:

- `SetAbsAlarm(AlarmID, start, cycle);`
Sets the alarm to expire when the counter value next reaches the value `start`. You should be aware that if the underlying counter already has value `start` when the call is made, then the alarm will not occur until the counter has ‘wrapped around’.
- `SetRelAlarm(AlarmID, increment, cycle);`
Sets the alarm to expire `increment` ticks from the current count value when you make the call. This means that `increment` is a tick offset from the current counter tick value.

In these two API calls, a `cycle` value of zero ticks indicates that the alarm is a single-shot alarm, which means that it will expire only once before being canceled. A `cycle` value greater than zero defines a cyclic alarm. This means that it will continue expiring every `cycle` ticks after the first expiry has occurred. Setting a non-zero `cycle` value gives you an easy way to configure periodic alarms that occur with a periodicity of `cycle` ticks.

Selecting Parameters

If the activated task is BCC1 or ECC1/2 there will be no queued activation. This means that if the `start` or `increment` value is very short, or the `start` value is very close to the current counter value, then this may cause undesired side effects. The alarm will try to activate the task while a previously activated instance is still executing. The activation would be lost and an `E_OS_LIMIT` error would be raised (see Chapter 13 for more information about error codes and how to debug use of RTA-OS3.0 at runtime). You must make sure that enough time is allowed for the task to complete before the next alarm which results in a re-trigger of the task occurs.

9.2.1 Absolute Alarms

Single Shot

An absolute alarm specifies the absolute value of the underlying counter at which the alarm expires. Single shot absolute alarms are useful for monitor-

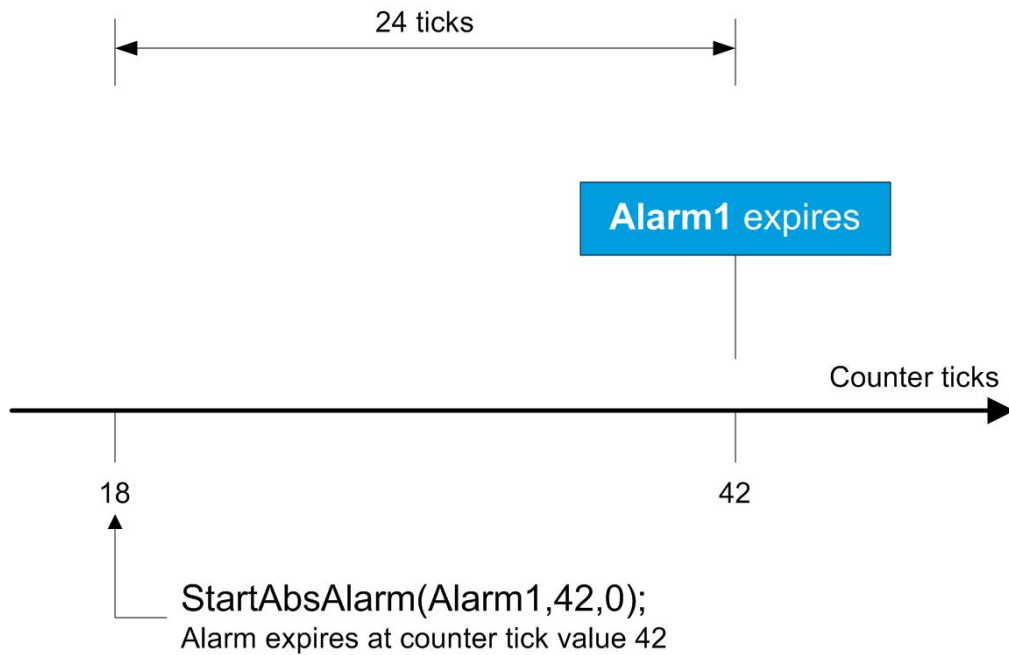


Figure 9.5: Illustration of an Absolute Single Shot Alarm

ing things against a pre-defined threshold value - the alarm can be configured to expire when the threshold is exceeded. You might want to count the number of errors that occur in data samples taken at runtime and then trigger a recovery action when the number of errors reaches a dangerous level. This is shown in Code Example 9.3.

```
/* Expire when counter value reaches 42. */
SetAbsAlarm(DangerLevelReached, 42, 0);
```

Code Example 9.3: Absolute single shot alarm

Code Example 9.3 is illustrated in Figure 9.5.

A single shot alarm is useful when you need to program a timeout that waits for a fixed amount of time and then takes an action if the timeout occurs.

Cyclic

If an absolute alarm specifies a non-zero cycle value then it will first expire at the specified start tick and then every cycle ticks thereafter. This is shown in Code Example 9.4.

```
/* Expire when counter value reaches 10 and then every 20 ticks
   thereafter */
SetAbsAlarm(Alarm1, 10, 20);
```

Code Example 9.4: Absolute cyclic alarm

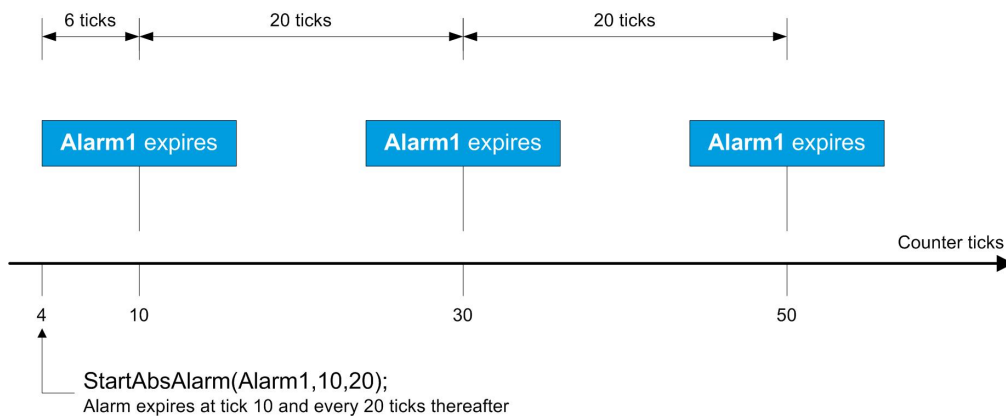


Figure 9.6: Illustration of the Absolute Cyclic Alarm

The behavior of the code example is illustrated in Figure 9.6.

For absolute alarms, an absolute start value of zero ticks is treated in the same way as any other value - it means expire the alarm when the counter reaches the value zero.

For example, if the current counter value was zero then you would not see your alarm expire until the `MAXALLOWEDVALUE+1` number of counter value ticks had happened. On the other hand, if the counter value was already at `MAXALLOWEDVALUE`, then you would see the alarm expire on the next tick of the counter.

Setting Alarms in the past

With an absolute alarm it is possible to set the start time to be a value that is already in the past. This does not mean that the alarm will not happen. Recall that counters wrap around when they reach `MAXALLOWEDVALUE`. So, when you set an alarm in the past you might have to wait up to `MAXALLOWEDVALUE+1` (i.e. the counter modulus) ticks until the alarm occurs.

⚠ *If you set an alarm to start at tick T and the value of the counter is already T then the alarm will not expire immediately. This is because T is already in the past when the alarm is set.*

A common error is to set an absolute alarm to occur at zero when the OS starts and then wonder why it does not occur when expected. This is because zero is already in the past! The effect is shown in Figure 9.7.

Synchronizing Absolute Cyclic Alarms to a Counter Wrap

Setting an alarm to occur periodically at a known synchronization point is extremely important for real-time systems. However, in AUTOSAR OS, it is not possible to set an absolute alarm to occur periodically each time the underlying-

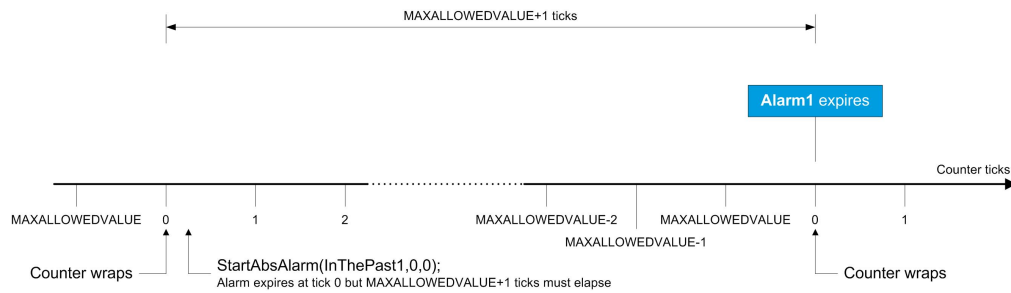


Figure 9.7: Setting an alarm in the past

ing counter wraps around.

For example, assume you have a counter that counts in degrees with a resolution of one degree and you want to activate a task at “top dead center”, i.e. on each revolution of the crankshaft.

For example, assume that the counter has a modulus of 360 ticks. What you need to say is `SetAbsAlarm(Alarm1, 0, 360)`. This is forbidden by the AUTOSAR OS standard because the cycle parameter cannot be greater than `MAXALLOWEDVALUE`, which is always the modulus-1 (in this case 359).

If you need this type of functionality, you must provide code that resets an absolute single-shot alarm each time the alarm expires.

For example, if `Task1` is attached to `Alarm1`, then the body of `Task1` will need to reset the alarm when the task is activated as shown in Code Example 9.5.

```
TASK(Task1) {
    /* Single-shot alarm reset at top dead center = 0 = 360
       degrees. */
    SetAbsAlarm(Alarm1, 0, 0);
    /* User code. */
    TerminateTask();
}
```

Code Example 9.5: Resetting an Alarm when a Task is Activated

9.2.2 Relative Alarms

Single-Shot

A relative alarm specifies the absolute value of the underlying counter at which the alarm expires. Single shot relative alarms are useful when you want to timeout some activity at runtime. For example, you might want to wait for an external event and then activate a task if the event does not occur.

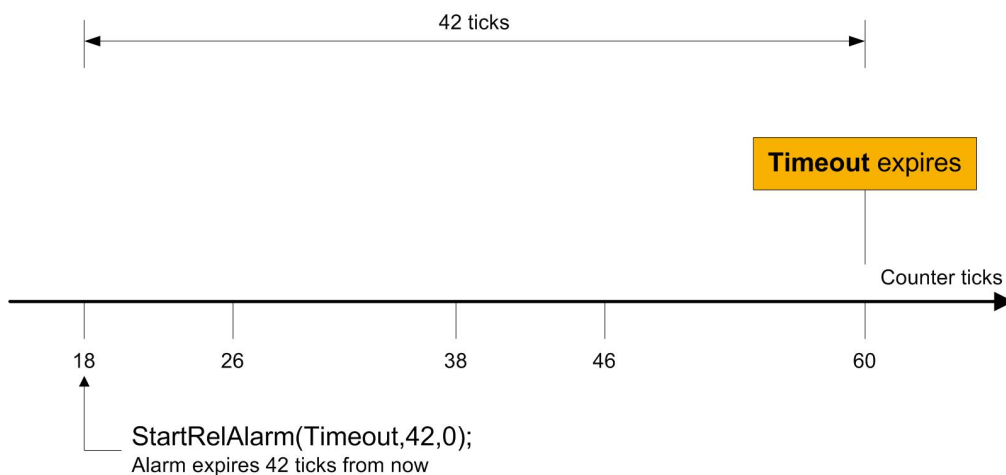


Figure 9.8: Illustration of a Relative Single Shot Alarm

Code Example 9.6 shows how an absolute single shot alarm can be set.

```
/* Timeout 42 ticks from now */
SetRelAlarm(Timeout, 42, 0);
```

Code Example 9.6: Relative single shot alarm

Code Example 9.6 is illustrated in Figure 9.8

A single shot alarm is useful when you need to program a timeout that waits for a fixed amount of time and then takes an action if the timeout occurs.

In AUTOSAR OS, the use of zero for increment in `SetRelAlarm()` is forbidden. If you use zero for increment, then an `E_OS_VALUE` error will be returned.

Cyclic

Code Example 9.7 shows a relative alarm that expires after 10 ticks and then every 20 ticks thereafter.

```
/* Expire after 10 ticks, then every 20 ticks. */
SetRelAlarm(Alarm1, 10, 20);
```

Code Example 9.7: Relative cyclic alarm

In Figure 9.9, you can see how this alarm can be visualized.

9.3 Auto-starting Alarms

It is possible to start alarms by calling `SetRelAlarm()` or `SetAbsAlarm()` in the main program. However, the easiest way to set cyclic alarms is to make them auto-started. Auto-started alarms are started during `StartOS()`.



Figure 9.9: Illustration of a Relative Cyclic Alarm

Auto-started alarms can be set on a per application mode basis so you can choose in which application modes the alarm is auto-started. Each auto-started alarm must also specify whether it is started at an absolute or a relative counter value and the associated increment/start and cycle parameters must be configured.


 *Even though alarms may be started in different application modes it is not possible to assign different auto-start parameters for each mode.*

Figure 9.10 shows how alarms can be set to auto-start from the Startup Modes pane.

RTA-OS3.0 ensures that software counters are initialized to zero during `StartOS()` (hardware counters will be set to the value configured by your own application initialization code). As a result of this, you must take care if you use the a start time of zero ticks for an absolute alarm because the zeroth tick has already happened when the alarm is are started. The alarm will be started but will not expire occur until the associated counter has wrapped around. On a 16-bit counter ticked every millisecond you would need to wait just over 65 seconds for this to happen, and on a 32-bit counter just under 48 days. Specifying that the alarm starts on the first (or later) tick means that the initial expiry will occur on the next tick of the counter.

Auto-started absolute alarms are useful if you require alarms to be synchronized to each other (i.e. the relative expiries between alarms have to occur a pre-configured number of ticks apart).

9.4 Canceling Alarms

You can cancel an alarm using the `CancelAlarm()` API call.

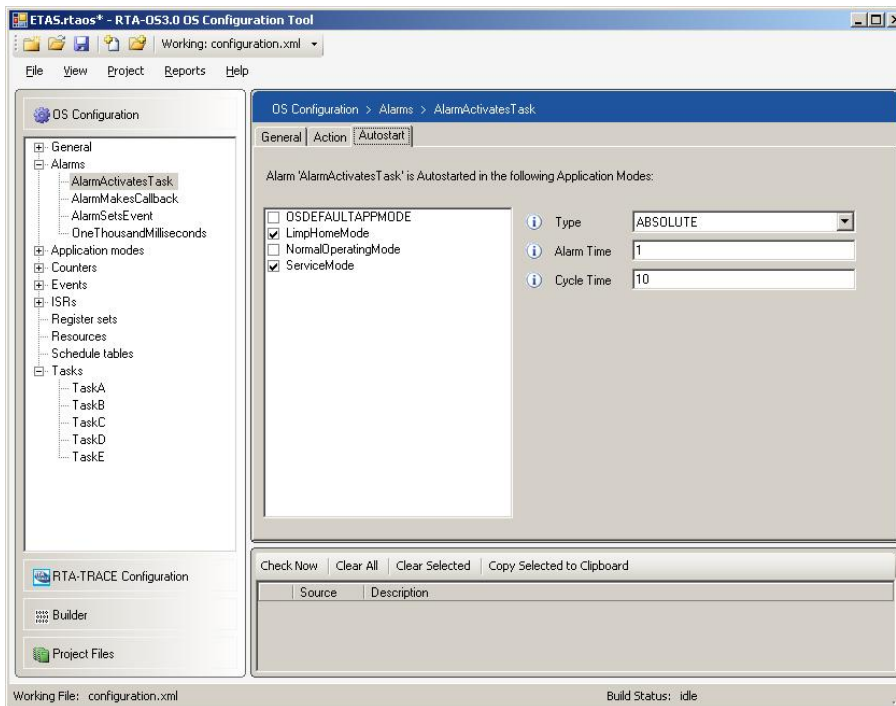


Figure 9.10: Auto-starting Alarms

An alarm may, for example, need to be canceled to stop a particular task being executed. An alarm can be restarted using the `SetAbsAlarm()` or the `SetRelAlarm()` API call.

9.5 Working out when an Alarm will occur

If you need to work out when an alarm will occur, for example, to avoid setting an absolute alarm when the absolute value has already been reached, then you can use the `GetAlarm()` API call.

The call returns the number of ticks remaining before the specified alarm expires. If the alarm is not set, then the API call returns the value `E_OS_NOFUNC` and the number of ticks to expiry is undefined. It is recommended that the return value of the call is checked before using the result. Code Example 9.8 shows the use of the API call.

```
TickType    TimeToExpiry;
TickType    SafetyMargin = 100;
StatusType  IsValid;
IsValid = GetAlarm(Alarm1, &TimeToExpiry);
if (IsValid != E_OS_NOFUNC) {
    if (TimeToExpiry <= SafetyMargin) {
        Log(InsideSafetyMargin);
    }
}
```

```
}
```

Code Example 9.8: Getting the time to expiry

You should exercise caution when making runtime decisions based on the number of ticks returned by the call, especially if the underlying counter has a high resolution. As with reading counter values with `GetCounterValue()`, preemption can occur between getting the value and using it for calculation. This means that you may read a (long) time to expiry but then be preempted to resume shortly before the alarm expires (or even after it has expired).

9.6 Non-cyclic (aperiodic) Alarms

Cyclic alarms are only useful for programming cyclic behavior. In many systems, for example those that need to execute tasks periodically to poll data sources, this is ideal. However, you may need to program systems where the time between successive expiries of an alarm needs to change at runtime. For example, you might be calculating an engine shaft speed and using this to program the duration of spark or injection timing.

Aperiodic behavior with alarms need to be programmed using single-shot alarms that are set to the next expiry value by the activated task.

In Code Example 9.9, a task runs every millisecond and polls a counter that registers degrees of rotation of a crankshaft. The task calculates the position and speed of the crank. The speed is used to determine the duration of the spark timing. The spark is started and an alarm is set to expire after `SparkTiming` ticks.

```
TASK(MillisecondTask) {
    ...
    GetElapsedCounterValue(ShaftEncoder,&Position,&
        DegreesRotation);
    RevsPerMinute = (DegreesRotation/360) * 1000 * 60;
    SparkTiming = Lookup(RevsPerMinute);
    if (Position = 90) {
        StartSpark();
        SetRelAlarm(TimeCounter, SparkTiming, 0); /* Activates
            SparkOff on expiry */ }
    }
    ...
    TerminateTask()
}
TASK(SparkOff){
    StopSpark();
    TerminateTask();
}
```

}

Code Example 9.9: Aperiodic Alarm Example

9.7 Summary

- Alarms are set on an underlying counter.
- You can set multiple alarms on each counter.
- Each alarm specifies an action, either:
 - activation of a task,
 - setting an event,
 - execution of a callback, or
 - ticking a ticked counter.
- Alarms can be set to expire at an absolute or relative (to now) counter value.
- Alarms can be auto-started.

10 Schedule Tables

In Chapter 9 you saw that you can build systems requiring periodic and aperiodic behavior relatively easily. However, one of the limitations of alarms is that you can only perform one action per alarm. If you need to build a system where you have a phased sequence of task activations and guarantee some separation in time (temporal separation) then you need to be quite careful how you start and stop the alarms.

While it is possible to build such a system with alarms, there is nothing, other than code review, that prevents the timing properties of the application being accidentally modified at runtime. Furthermore, you saw that if you wanted to define multiple task activations at a single point in time, you were forced to create multiple alarms when what you really want to do is to activate multiple tasks from a single alarm.

AUTOSAR OS addresses the limitations of alarms by providing an OS object called a schedule table.

A schedule table comprises a set of expiry points that occur on statically configured offsets from a notional zero. The offsets are specified in ticks of a statically bound counter - just like the expiry of alarms. The key difference between schedule tables and alarms is that the expiry points on a schedule table always maintain their relative separation (to each other). The schedule table can be started and stopped as a composite unit and, whenever it is restarted, the expiry points always have the same relative execution behavior.

Schedule tables adopt the following terminology:

Initial Offset is the offset to the first expiry point on the schedule table. It is therefore the smallest offset configured.

Duration is the number of ticks from zero before the schedule table stops.

Final Delay is the difference between the offset to the final expiry point and the duration. It is therefore equal to the value of duration minus the longest offset.

Delay is the number of ticks between adjacent expiry points and is equal to the longer offset minus the shorter offset. If the schedule table repeats, then the delay between the last and the first expiry point is equal to the Final Delay plus the Initial Offset.

An expiry point is similar to an alarm in that it indicates a point in time at which RTA-OS3.0 needs to take some action. The difference between expiry

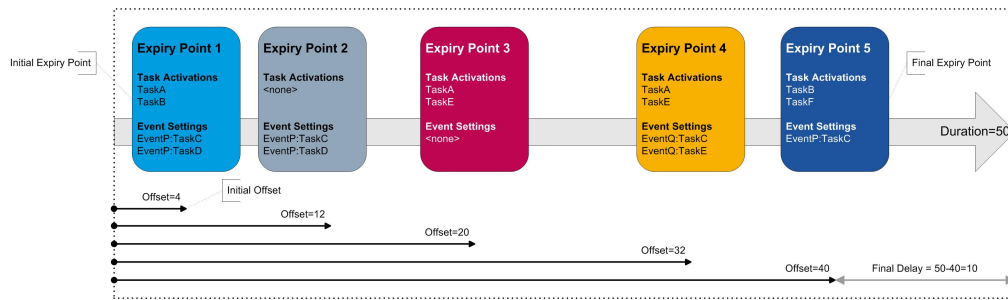


Figure 10.1: Visualizing a Schedule Table

points and alarms lies in what actions can be taken and is shown in the following table.

Action	Alarm	Expiry Point
ActivateTask()	Yes - one task	Yes - multiple tasks
SetEvent()	Yes - one event	Yes - multiple events
Callback	Yes	No
IncrementCounter()	Yes	No

Figure 10.1 shows the anatomy of a schedule tables with 5 expiry points and a duration of 50 counter ticks. When the schedule table was started¹, each expiry point would occur every 50 ticks with offset ticks from the notional zero point.



We use the term notional zero to mean reference starting point on the schedule table from which offsets are measured. It is important to understand that this is nothing to do with values on the underlying counter. When a schedule table is started (see Section 10.2) the notional zero will be mapped onto the appropriate “now” value of the underlying counter.

10.1 Configuring a Schedule Table

Each schedule table is driven by a counter. The counter provides the schedule table with a tick source. You can use the same counter to drive multiple schedule tables. However, at runtime you can only have one schedule table per counter in the running state at any point in time. You can also share a counter between schedule tables and any number of alarms.

Each schedule table must define a duration in ticks of the underlying counter. The duration must be in the range MINCYCLE to MAXALLOWEDVALUE of the counter.

A schedule table is ‘single shot’ so it stops automatically when duration ticks on the counter have elapsed from when the schedule table starts. Single-

¹And assuming it was configured as repeating

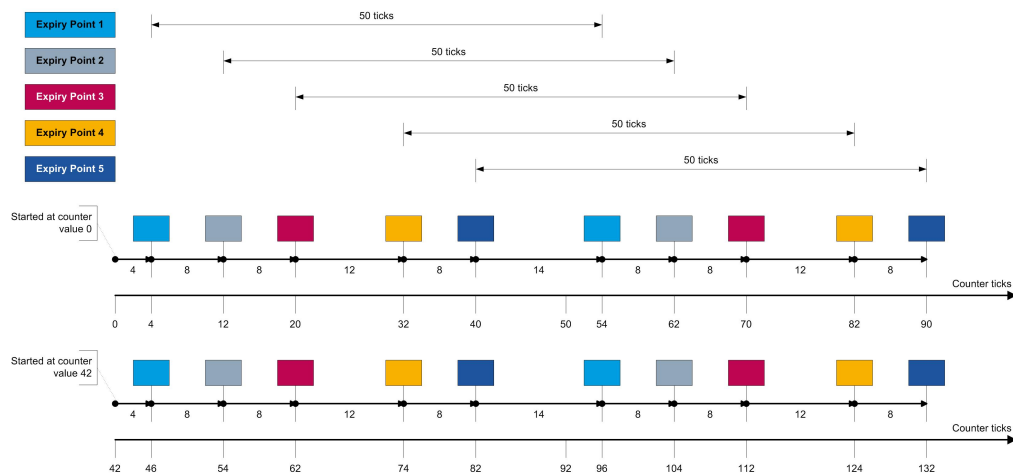


Figure 10.2: Visualizing a Schedule Table

shot schedule tables are useful when you want to start a phased sequence of actions, for example when building closed-loop control systems. However, a schedule table can be configured as repeating so that it repeats on a period of duration ticks. When a schedule table is repeating, every expiry point will occur with a period of duration ticks.

Figure 10.2 shows how the schedule table in Figure 10.1 would run when started on a underlying counter value of 0 ticks and on underlying counter value of 42 ticks.

Figure 10.3 shows the configuration of a schedule table called MasterPlan.

10.1.1 Configuring Expiry Points

Each schedule table contains one or more expiry points. An expiry point marks the place on the table where an action has to take occur. Each expiry point has the following attributes:

- Zero or more tasks to activate
- Zero or more events to set for a specified task
- An offset from the notional zero

It is not possible to have an expiry point with no action - you must activate at least one task or set one event.

The offset sets the number of ticks on the schedule table from a notional zero at which the expiry point needs to be processed. Thus, the offset specifies when expiry point actions happen on the schedule table. An offset on the

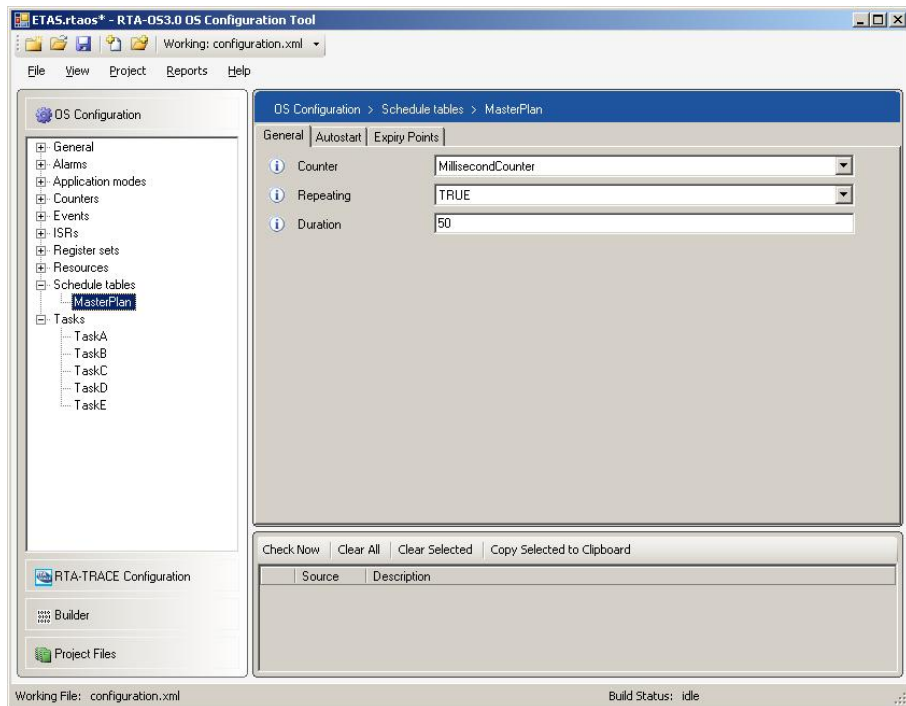


Figure 10.3: Schedule Table Configuration

schedule table can be zero or in the range MINCYCLE to the duration of the schedule table. Similarly, the delays between adjacent expiry points must also lie in this range. Values less than MINCYCLE are not allowed because they cannot be programmed on the counter. (e.g. if the counter has a MINCYCLE of 10 then a value of 5 cannot be set).

Figure 10.4 shows how to specify expiry points.

The upper part of the workspace shows the expiry points and their associated offsets. The lower part of the workspace shows the actions for the selected expiry point and the control for adding and removing expiry points.

10.2 Starting Schedule Tables

10.2.1 Absolute Start

The `StartScheduleTableAbs(ScheduleTableID, Start)` API call is used to start a schedule table at an absolute counter value as shown in Code Example 10.1/

```
/* Start Schedule Table Tbl 6 when the counter reaches tick 6*/
StartScheduleTableAbs(Tbl, 6);
```

Code Example 10.1: Using `StartScheduleTableAbs()`

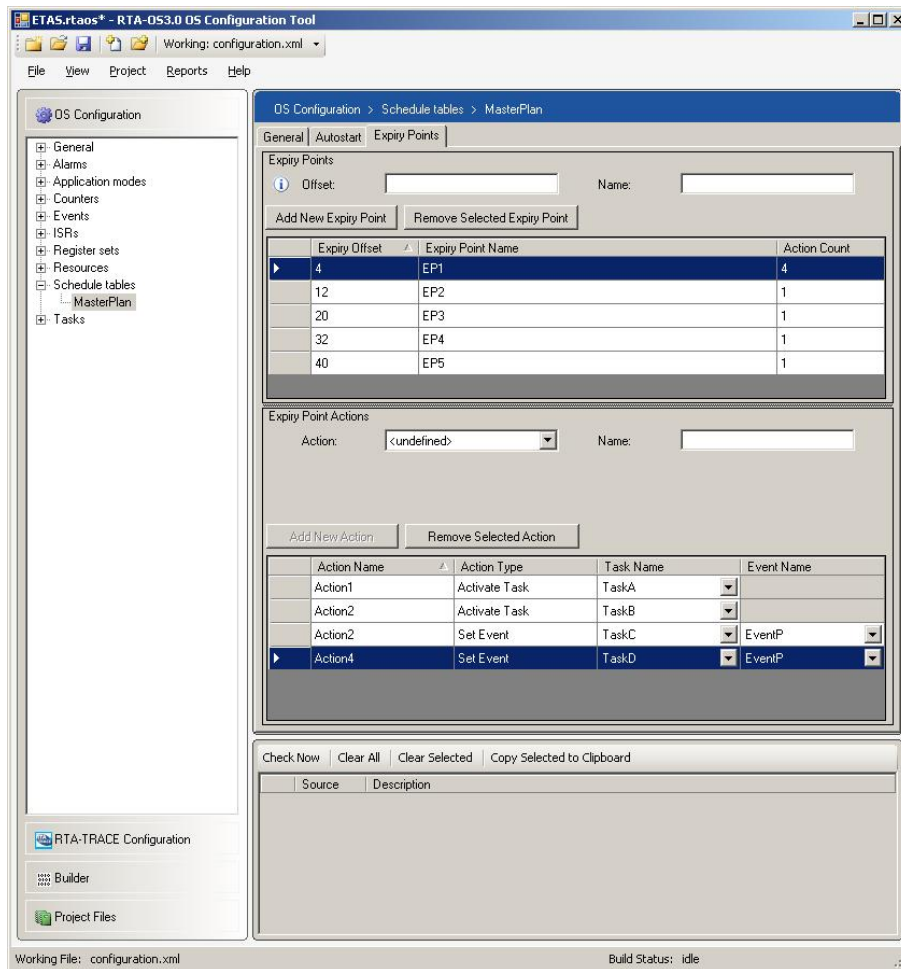


Figure 10.4: Specifying Expiry Points

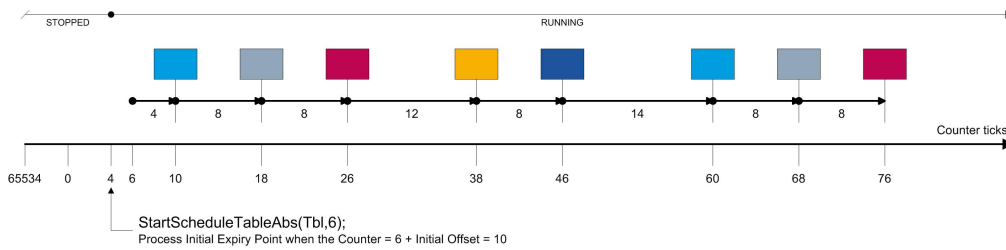


Figure 10.5: Starting a schedule table at an absolute count value

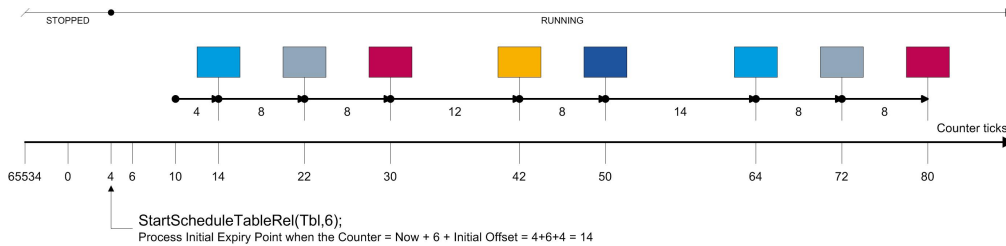


Figure 10.6: Starting a schedule table at a relative count value

The schedule table is in the SCHEDULETABLE_RUNNING state when the call returns. The first expiry point will be processed after Start plus initial offset ticks have elapsed. Figure 10.5 shows the schedule table from Figure 10.1 when started according to Code Example 10.2.

This is extremely useful for building schedule tables that are synchronized to specific values of an external (hardware) counter.

10.2.2 Relative Start

The StartScheduleTableRel(ScheduleTableID, Offset) API call is used to start a schedule table at a relative number of ticks from now.

The offset parameter of the StartScheduleTableRel() call specifies the relative number of ticks from now at which RTA-OS3.0 will process the first expiry point and can be zero. This is the same concept as the increment parameter that you use to set a relative alarm (see Section 9.2.2).

```
/* Start Schedule Table Tbl 6 ticks from now */
StartScheduleTableRel(Tbl, 6);
```

Code Example 10.2: Using StartScheduleTableRel()

The schedule table is in the SCHEDULETABLE_RUNNING state when the call returns. The first expiry point will be processed after Start plus initial offset ticks after the current count value has elapsed. Figure 10.6 shows the schedule table from Figure 10.1 when started according to Code Example 10.2.

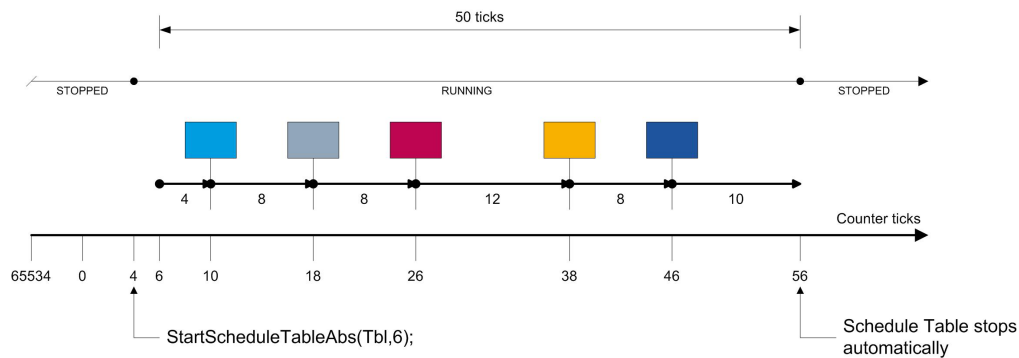



Figure 10.7: Non-repeating schedule tables stop automatically

 You must make sure that the Offset value that is passed to `StartScheduleTableRel()` is sufficiently long, so that it has not already expired before the call returns. This is only an issue for schedule tables driven by a hardware counter. For schedule tables that are driven by a software counter, the counter cannot be incremented while the `StartScheduleTableRel()` is executing because both API calls execute at OS level thus are serialized.

10.3 Stopping Schedule Tables

A schedule table can be stopped at any point by calling the `StopScheduleTable(ScheduleID)`. A call to stop a schedule table stops the processing of any remaining expiry points.

A schedule table that is not configured as repeating will stop automatically final delay ticks after RTA-OS3.0 has processed the final expiry point as shown in Figure 10.7.

A repeating schedule table will run until it is stopped by calling `StopScheduleTable()` or the table is switched by calling `NextScheduleTable()` (see Section 10.4).

You can re-start a schedule table that has been stopped by calling `StartScheduleTable[Abs|Rel]()`. The schedule table does not restart immediately - it will start at the start or offset passed into the API call - and the table will re-commence at its notional zero as shown in Figure 10.8. There is no mechanism for starting a schedule table part-way through.

10.4 Switching Schedule Tables

You can switch from one schedule table to another at runtime using the `NextScheduleTable()` API call. The switch between schedule tables always occurs at the end of the table - i.e. final delay ticks after the final expiry point is processed. Code Example 10.3 shows how the API call is made.

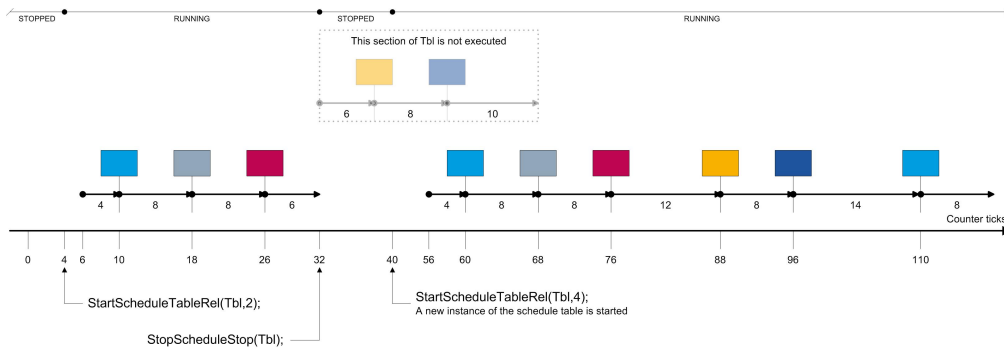


Figure 10.8: Schedule tables always start at their notional zero

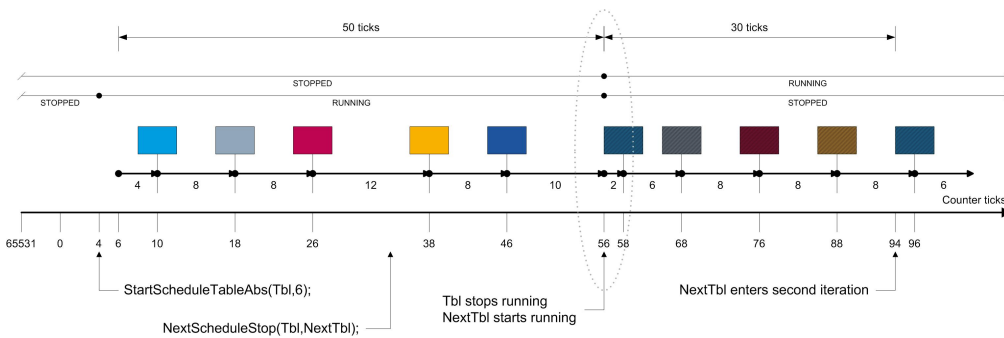


Figure 10.9: Switching a schedule table

```
/* Start NextTbl after Tbl has finished */
NextScheduleTable(Tbl, NextTbl);
```

Code Example 10.3: Switching a schedule table

The only restriction for switching schedule tables is that both the current and the next table must use the same underlying counter. There are no requirements that the schedule tables have the same number of expiry points, the same duration, the same initial offset, etc.

When a call to NextScheduleTable() is made, the delay between the final expiry point on Tbl and the first expiry point on NextTbl is given by:

$$\text{Delay} = \text{Current.FinalDelay} + \text{Next.InitialOffset}$$

Figure 10.9 shows the process of switching from one schedule table (with duration 50 ticks) to another schedule table (with duration 30 ticks).

! *If the Current schedule table has a final delay of zero ticks and the Next schedule table has an initial offset zero, then the delay between expiry points will be zero ticks.*

If you make multiple calls to `NextScheduleTable()` while `Current` is running then the `Next` table that runs will be the one you specified in your most recent call.

10.5 Schedule Table Status

You can query the state of a schedule table using the `GetScheduleTableStatus()` API call. The call returns the status through an out parameter. Code example 10.4 shows how to get the status.

```
ScheduleTableStatusType State;  
GetScheduleTableStatus(Table, &State);
```

Code Example 10.4: Getting the status of a schedule table

The status will be either:

SCHEDULETABLE_STOPPED if the table is not started.

SCHEDULETABLE_RUNNING if the schedule table is started.

SCHEDULETABLE_NEXT if the schedule table has been started by a call to `ScheduleTableNext()` but is not yet running (because another schedule table on the same counter has not yet finished).

10.6 Summary

- Schedule tables provide a way of planning a series of actions statically at configuration time.
- A schedule table is associated with exactly one OSEK counter, may specify a duration, and contains one or more expiry points.
- Expiry points in RTA-OS3.0 are created implicitly by specifying offsets for stimuli implemented on a schedule table.
- You can switch between schedule tables, but only at the notional end of the table.

11 Writing Hardware Counter Drivers

You have seen that RTA-OS3.0 provides a simple, elegant and powerful interface for driving counters. The hardware counter driver mechanism provides great flexibility by placing the software/hardware interaction in the domain of user-supplied code. This allows easy integration of drivers for novel hardware and application requirements, and the ability to “piggyback” driver operation on hardware that is also used for other functions.

As the owner of your hardware you know how you want to use it in your application and therefore you are responsible for providing the hardware counter driver functions.

This chapter offers some guidelines to help you in the construction of hardware counter drivers. Much of this knowledge has been gained while constructing drivers for assorted peripheral timers, but it should be applicable to other peripherals which increment in response to some external event (e.g. interrupts generated by the rotation of a toothed wheel).

The example code is structured for ease of explanation and understanding. Different control structures may result in small improvements in the quality of generated code on some targets (e.g. replacing a **while(1)** loop using **if . break** exits with a **do . while** loop with appropriately modified conditions). If you choose to make this type of optimization, then you should take care to ensure that the required semantics and orderings of operations are maintained (e.g. note that the “&&” logical and operator in C imposes both ordering and lazy evaluation).

11.1 The Hardware Counter Driver Model

The hardware driver concept assumes an underlying free-running peripheral counter. The counter has an initial value established by the user, counts up from zero and wraps back to zero as it reaches its modulus.



These are the assumptions of the model. In later sections of this chapter you will see how to implement this model with hardware which does not require these constraints.

A hardware counter driver uses the `Os_AdvanceCounter_<CounterID>` API call to tell RTA-OS3.0 to expire an alarm and/or the schedule table expiry points associated with a counter as soon as possible after it/they become due and to program the next alarm or expiry point.

In this chapter we use the terms:

now is the counter’s current (continuously increasing) value.

old is the previously programmed compare value.

match is the (absolute value of the) count at which the next alarm or schedule table expiry point is due.

- is a binary subtraction modulo the counter's modulus.

The code examples in this chapter make use of the functions:

```
clear_pending_interrupt()  
set_pending_interrupt()  
enable_interrupt_source()  
disable_interrupt_source()
```

These functions refer to operations performed on the status/control registers of the counter peripheral used to provide the hardware counter functionality. You are responsible for providing these functions (or equivalent code) in your hardware drivers.

11.1.1.1 Interrupt Service Routine

Typically you will call RTA-OS3.0's hardware driver interface from a user-supplied Category 2 ISR..

The ISR is triggered by each **match** and will call `Os_AdvanceCounter_<CounterID>()` to tell RTA-OS3.0 that a **match** has occurred. RTA-OS3.0 will then setup the delay until the next **match**. In general, there are three classes of behavior the ISR. These are described here, along with their implications for system behavior and schedulability analysis, in order that appropriate choices can be made when implementing the ISR component of hardware counter drivers.

Simple handlers can deal with a single **match** value being processed per ISR. This class of handler must complete before the next interrupt becomes due.

Re-triggering handlers can deal with one or more matches becoming due before it completes handling of the interrupt by which it was first triggered. Such a handler processes one **match** per invocation, and exits with the invoking interrupt still pending if another **match** is already due.

Looping handlers can deal with one or more matches becoming due before it completes handling of the interrupt which first triggers it. Such a handler is able to process multiple expiries in turn, and only exits when either no **match** is due or when an interrupt is pending. Any interrupt handler which is capable of looping is a looping handler.

When it can be guaranteed the handler can complete before the next **match** becomes due then a simple handler is the best choice because they typi-

cally have a smaller worst-case execution time than re-triggering or looping handlers. The choice between re-triggering and looping is influenced by the following factors:

1. Some hardware will not support re-triggering behavior, so a looping handler must be used.
2. When the interrupt that invokes the handler is at the same level as another interrupt in the system, and that other interrupt has a higher arbitration precedence (i.e. will be handled first if both are pending) then a re-triggering handler is preferred because it reduces latency for the other interrupt. In practice, this is of particular concern for architectures with a single interrupt priority level.
3. A re-triggering handler typically has smaller execution time than a looping handler when a single **match** is processed. Note that it is not normally relevant that a looping handler may be “more efficient” when several expiries are handled in one invocation. Worst case behavior occurs when each **match** is handled by a separate invocation.

A simple handler is recommended if the handler’s worst case response time (i.e. the time between the interrupt becoming ready and the handler terminating) is known to be smaller than the minimum interval between interrupts. If this cannot be guaranteed, then a re-triggering handler should be used unless the hardware characteristics prohibit it.

11.1.2 Callbacks

Recall from 8.2.2 that four callbacks are also required as part of the hardware counter driver:

1. `Os_Cbk_Now_<CounterID>` which must return the **now** value of the peripheral counter.
2. `Os_Cbk_Cancel_<CounterID>` which clears any pending interrupt for the counter and ensures that the interrupt will not become pending until after a `Os_Cbk_Set_<CounterID>()` call has been made. This behavior is required if any of the following conditions apply to your application:
 - (a) the alarms driven by the counter are never stopped directly by the application calling `CancelAlarm()`;
 - (b) the schedule tables driven by the counter are stopped directly by the application calling `StopScheduleTable()`;
 - (c) you have a schedule table that does not repeat (in this case RTA-OS3.0 may need to cancel the interrupt when the schedule table stops).

If none of these conditions apply, then you can simply provide a 'stub' call to implement `Os_Cbk_Cancel_<CounterID>`.

3. `Os_Cbk_State_<CounterID>` is called by RTA-OS3.0 when `GetAlarm()` or `GetScheduleTableStatus()` is called by your application code and the relevant alarm or schedule table is running. The call returns an `Os_CounterStatusType` which is a C struct of the form:

```
struct {
    TickType    Delay;
    boolean     Pending;
    boolean     Running;
}
```

The `Delay` field, when defined, gives the number of ticks from the previous **match** at which the next **match** is due, i.e. `Delay` is the relative time between matches. The `Pending` field is set to true if the next **match** is already pending. When the `Pending` field is false, then the `Delay` holds the relative number of ticks from **now** that remain until the next **match** becomes due. This behavior is required if the application interrogates the status.

4. `Os_Cbk_Set_<CounterID>` establishes a state in which an interrupt will become due the next time the counter matches the supplied value. The callback is passed the absolute **match** value at which the next **match** is due. The callback is used to start the counter and also to shorten the time to the next **match**. This secondary behavior is needed because you can set alarms (or start schedule tables) that need to begin at an **match** closer to now than the currently programmed **match** value.

All of the hardware driver callbacks run at OS level. This means that they will not be preempted by Category 2 ISRs and do not, therefore, need to be reentrant.

11.2 Using Output Compare Hardware

This section considers the construction of drivers for output compare (sometimes known as compare/match) counter hardware. Such hardware has the property that an interrupt is raised when a counter value (advanced by some outside process such as a clock frequency or events detected by some sensor) matches a compare value set by software. It is assumed that both the counter value and the current compare value can be read by software. In this section, it is assumed that the registers of the counter hardware are mapped to the variables `OUTPUT_COMPARE` and `COUNTER`.

The section outlines appropriate call back functions, followed by several interrupt handlers making different assumptions about required behavior and hardware facilities.

Initially, a counter with the same modulus as TickType is considered. TickType usually has a modulus of 2^{16} on 16-bit targets and 2^{32} on 32-bit targets.

With full modulus arithmetic, the number of ticks in a delay can be determined by subtracting the start value from the end value. When the current counter value (COUNTER) is subtracted from the next compare value (OUTPUT_COMPARE), the result is the number of ticks before the **match** is reached. If this value is read after the next **match** is set, and found to be greater than the currently required delay, then the counter has passed the next **match** and there will be an extra modulus wrap (i.e. TickType ticks) before the compare occurs. This can happen if the delay before the next **match** is very short (for instance, one tick), in which case there is a race condition between the counter passing the intended **match** and the setting of the **match**.

11.2.1 Callbacks

Cancel

The `Os_Cbk_Cancel_<CounterID>()` call must ensure that no further interrupts will be taken. This is a hardware dependent operation that would typically be achieved by disabling interrupt generation by the counter device.

```
FUNC(void, OS_APPL_CODE) Os_Cbk_Cancel_<CounterID>(void){
    clear_pending_periodic();
    disable_interrupt_source();
}
```

Now

The `Os_Cbk_Now_<CounterID>()` call reads the free-running counter to provide the current **now** value.

```
FUNC(TickType, OS_APPL_CODE) Os_Cbk_Now_<CounterID>(void){
    return (TickType)COUNTER;
}
```



Special care may be required when reading the counter on 8-bit devices to ensure that a consistent value is obtained: in some cases, the high and low bytes must be read in a particular order in order to latch then release the counter. Similar considerations may apply when writing compare values.

Set

The `Os_Cbk_Set_<CounterID>()` call causes the interrupt to become pending when the counter value next matches the supplied parameter value. This is achieved by disabling compare matching, clearing any pending interrupt, setting the compare value, and ensuring that the interrupt is enabled. If the hardware does not provide the ability to disable compare matching, this can be simulated by setting the compare value to one less than the current counter value (thus ensuring that a **match** will not occur before the next time that the compare value is set).

Note that it may not be necessary to disable compare matching. If it can be guaranteed that a **match** will not occur between system start up and the **match** at which the hardware counter is started, then disabling compare matching is not necessary. In the example below, this is achieved by setting the compare register to the previous value of the counter, thus ensuring that a “match” interrupt will not be generated until ticks equal to the modulus of the counter have occurred. This will be long enough to perform the rest of the `Os_Cbk_Set_<CounterID>()` function. (Note that this approach can only be used if the compare register is not shared with anything else).

```
FUNC(void, OS_APPL_CODE) Os_Cbk_Set_<CounterID>(TickType Match)
{
    /* prevent match interrupts for modulus ticks*/
    OUTPUT_COMPARE = COUNTER - 1;
    clear_pending_interrupt();
    OUTPUT_COMPARE = Match;
    enable_interrupt_source();
}
```

The code is carefully structured to avoid two potential race conditions that can arise from dismissing the interrupt in a way that can result in unexpected interrupts being generated or expected interrupts being lost. These race conditions are as follows:

1. Pre-existing values of the compare and counter values may lead to an interrupt being raised before the compare register is set, which results in a situation where the interrupt appears to have been caused by the action of `Os_Cbk_Set_<CounterID>()` (rather than previous compare/counter values).
2. Using the `clear_pending_interrupt()` call after the compare register is set avoids the first race condition (without the need to disable the match interrupt), but may result in the situation where a very short delay (for instance, one tick after the value of the counter register when

`Os_Cbk_Set_<CounterID>()` is called) is ignored. In some cases, a full counter wrap will occur before the compare causes an interrupt. Depending on the hardware, this may result in no interrupt occurring (even after a counter wrap).

In all situations, careful consideration should be given to the use of very short delays, as the counter may reach the next **match** even before it has been set, particularly if the execution path between user code which reads the current value of **now**, calculates the next **match** and sets the **match** is long. If this occurs, a full counter wrap will need to occur before the **match** occurs.

In the above example, match interrupts are prevented by means of changing the output compare register. In subsequent examples, the way in which this is achieved is not specified. Rather, it is assumed that a function `disable_compare()` is provided to prevent the hardware from generating match interrupts.



If the counter is used for some other purpose (in addition to its function as the driver for the hardware counter), the `disable_compare()` function must not halt the counter as this will lead to counter drift for other users. The re-enabling of compare matching needs to be done atomically with the assignment of the compare register. If this is not done, another race condition may exist if a short delay is set into the output compare register.

The callback shown above only works for alarms/schedule tables that you do not adjust once they have been started. If you plan to make `Set[Abs|Rel]Alarm()` calls or to `NextScheduleTable()` calls, then you need a different `Os_Cbk_Set_<CounterID>()` callback. The callback needs to be able to reset a currently programmed **match** value for a new **match** that is due to occur means it is fewer ticks from **now** than the old **match** value.

We also assume that delays due to higher priority interrupts are relatively small compared with an entire wrap of the counter modulus.

A naïve implementation would (atomically) reprogram the compare value with **match**. This is wrong because a higher priority interrupt (e.g. Category 1) could delay the write to the hardware register, so that by the time you write **match** to the compare register, **now** is already greater than **match**. This would cause all processing of the whole schedule to cease for 2^{16} (or 2^{32} or modulus) ticks. In fact, it is perfectly possible that, by the time we are ready to write **match** to the compare register, **now** is already greater than both **match** and **old**.

Your implementation of `Os_Cbk_Set_<CounterID>()` must distinguish between the starting case (where interrupts are stopped) and the resetting case

(where the schedule is running and it is being used to shorten the delay to an existing OLD compare value).

In this second case, your implementation of `Os_Cbk_Set_<CounterID>()` must return with the compare register containing the new **match** value and either;

- **now** has not exceeded **match**; or
- the compare interrupt flag is already pending. Note that if the interrupt flag is pending, it does not matter if **match** or even **old** has been passed by **now** as the hardware counter driver code you write that deals with `Os_AdvanceCounter_<CounterID>` will (eventually) catch up to the correct time.

First you must write the new **match** to the compare register:

- If **now** is between **match** and **old**, i.e. $\text{old} - \text{match} > \text{now} - \text{match}$, then **now** has already passed **match**. You must ensure that the interrupt flag is pending before returning.
- If **now** is not between **match** and **old** then either you can return with no flag pending or both **match** and **old** have been passed and you must ensure the pending flag is set before returning. You can test for both values having been passed using $\text{now} - \text{old} < \text{old} - \text{now}$.

```
FUNC(void, OS_APPL_CODE)Os_Cbk_Set_<CounterID>(TickType Match){
    TickType Old = (TickType)COMPARE;
    TickType Now = (TickType)COUNT;
    /* Update COMPARE with new Match */
    COMPARE = Match;
    if ((Old-Match > Now-Match) || (Now-Old < Old-Now)){
        set_pending_interrupt();
    }
}
```

State

The `Os_Cbk_State_<CounterID>()` call is only made when the alarm or schedule table is running and must first check whether the next **match** has already occurred (i.e. the interrupt is pending, this can occur because all of the callbacks are executed at OS level, which will prevent the resulting ISR from preempting the currently executing task). If this is not the case, then the remaining time to the next **match** is also required.

```

FUNC(void, OS_APPL_CODE) Os_Cbk_State_<CounterID>(
    Os_CounterStatusRefType State){
    State.Delay = OUTPUT_COMPARE - COUNTER;
    State.Running = True
    if (interrupt_pending()) {
        State.Pending = True;
    } else {
        State.Pending = False;
    }
}

```



The Delay value is calculated before checking whether the interrupt is pending. This is necessary to avoid a race condition in which the interrupt becomes pending after checking but before calculating Delay, which would result in an invalid value.

11.2.2 Interrupt Handlers

Simple

In the simplest case, it is only necessary to clear the interrupt and make the required `Os_AdvanceCounter()` call. `Os_AdvanceCounter()` calls the callback `Os_Cbk_Set_<CounterID>()` to program the next **match**. This assumes that the latency of the handler to the statement at which it has set the next **match** value - i.e. after the call to `Os_AdvanceCounter()` on the compare value) is known to be less than the shortest time between two matches driven by the counter, so the **match** will be ahead of **now**.

```

#include <Os.h>
ISR(Advanced_Driver)
{
    clear_pending_interrupt();
    Os_AdvanceCounter_<CounterID>();
}

```

It is essential that the **match** is always advanced to be ahead of **now**. If the **match-now** is shorter than the handler response time, then this will not be the case and an additional full wrap of the peripheral counter will be introduced before the next **match** occurs. In order to verify that a simple handler may be used safely, you should use schedulability analysis to verify that the simple handler can complete before its next invocation.

Re-Triggering

When matches may be too close together for the handler to advance the compare value before the next **match** is due, the handler must account for the situation in which the next **match** is already due.

This example considers the use of an output compare timer with hardware interlocking to prevent the accidental clearing of an interrupt which is raised during the clearing sequence. It is assumed that for this type of interlock, clearing the interrupt is achieved by reading the status register, then writing the status register (with a bit pattern that clears the interrupt bit). In this example, the interlock consists of two functions:

1. `prepare_interrupt_clear()`
2. `commit_interrupt_clear()`

While the driver is still running, the **match** is advanced (in the case of a full wrap, advancing by 0 is correct) and the first part of the interrupt clearing sequence is performed (reading the status register). Then a check is made that the new **match** is ahead of **now**. If this check shows that an interrupt will not be raised when the counter advances to the compare value (i.e. the next **match** is not yet due), then the interrupt clearing sequence is completed (by writing to the status register with the flag bit clear). If the check fails (i.e. the new expire is already due) then the interrupt is left pending and the handler will be re-triggered to deal with the **match**.



*The two-stage interrupt clearing sequence is required to avoid a race condition in which the counter reaches the next **match** between being tested and the interrupt being cleared. This would otherwise result in the interrupt for the next **match** being cleared. The required hardware behavior is that if the interrupt is raised again after the first stage of the sequence, then the second stage will not clear the interrupt.*

A similar approach can be taken with devices where the interrupt can be re-asserted by software. In these case, the interrupt can be cleared on entry to the handler, then re-asserted if the next **match** is due. In this case no race condition can occur (assuming there is no problem associated with software asserting an interrupt which the hardware is already asserting).

```
ISR(OutputCompareInterrupt){
    Os_CounterStatusType State;
    TickType remaining_ticks;
    Uint16 clear_tmp;

    Os_AdvanceCounter_<CounterID>();

    Os_Cbk_State_<CounterID>(&State);

    if (State.Running == True) {
        OUTPUT_COMPARE += State.Delay;
        clear_tmp      = prepare_interrupt_clear();
    }
}
```

```

    remaining_ticks = OUTPUT_COMPARE - COUNTER;
    if ((State.Delay == 0)
        || ((remaining_ticks != 0)
            && (remaining_ticks <= State.Delay))) {
        commit_interrupt_clear(clear_tmp);
    }
}
}
}

```



Some output compare hardware requires that the compare register be written to arm each interrupt. In such cases it is necessary to structure the code (as is the case above) so that the compare register is written to its previous value in the case of a Delay value of zero.

Looping

This section considers a generic looping ISR structure TickType modulus counter with programmable output compare.

```

#include <Os.h>
ISR(Advanced_Driver){
    Os_CounterStatusType State;
    TickType remaining_ticks;

    clear_pending_interrupt();

    while(1) {
        Os_AdvanceCounter_<CounterID>();
        Os_Cbk_State<CounterID>(&State)

        if (State.Running == False) {
            /* Exit 1: all alarms/schedule tables stopped */
            return;
        }

        OUTPUT_COMPARE += State.Delay;

        if (State.Delay == 0u) {
            /* Exit 2: full wrap */
            return;
        }

        remaining_ticks = OUTPUT_COMPARE - COUNTER;

        if ((remaining_ticks != 0u) &&

```

```

        (remaining_ticks <= State.Delay)) {
        /* Exit 3: match is in the future */
        return;
    }

    if (interrupt_pending()) {
        /* Exit 4: interrupt pending */
        return;
    }
}
}
}

```

This interrupt handler first dismisses the invoking interrupt, then enters a loop which processes the **match** and checks whether any further matches need to be processed by this invocation. This check has four exit conditions, which must be evaluated in the order shown.

Exit 1 is taken if the counter/schedule has stopped, so no further action is necessary. If the counter has not stopped, then the next **match** is set to the required number of ticks (which will be zero in the case of a full wrap). Checks must then be made to determine whether an interrupt will be raised when the next **match** is due.

Exit 2 is taken if the Delay value indicates that a full wrap of the timer is required before the next **match** is due. Therefore, no change to the **match** value is necessary. A Delay value of 0 ensures that the new **match** is ahead of **now** (and consequently that the interrupt will be asserted when it is reached). Exiting here ensures that the following checks will not immediately identify a **match** between **now** and the **match** when a full wrap has been requested and the counter has not yet moved on¹.

Exit 3 is taken if the current timer value has not yet reached the next **match**. This check is done by determining if the time until the next interrupt (i.e. `OUTPUT_COMPARE - COUNTER`) is less than the Delay until the next **match**. Note that the cast to `TickType` is necessary to ensure that the counter modulo behavior is accounted for. The counter modulus must be the same `TickType` for this to work correctly (Section 11.2.3 explains how to cope with a hardware modulus not equal to `TickType`). If the counter has moved on by less than Delay ticks, then an interrupt will be raised at the correct time and the handler can exit, otherwise, the new **match** may be missed.

¹It is assumed that the interrupt will not be re-asserted while the counter and **match** continue to **match**, only when the **match** first occurs. If this is not the case, it must be ensured that the handler never exits in that state, perhaps by avoiding Delay values of zero.

Exit 4 accounts for a race-condition between setting the next **match** and checking that it is ahead of the counter, since the counter can advance before the Exit 3 check is made. If Exit 3 is not taken, the next **match** is due **now**. If the interrupt is pending, Delay has already been recognized by the hardware, so the handler can exit and be re-invoked by the pending interrupt (it would not be acceptable to exit with an interrupt pending with no **match** due). Note that this construction means that it does not matter whether the interrupt is pending or not when Exit 3 is not taken because the counter has advanced by exactly the Delay value: either the pending interrupt or looping results in the next **match** being processed.

If no exit is taken, then the next **match** is due (or overdue) and another call to `Os_AdvanceCounter()` is made. The next **match** is processed and the exit checking is repeated.

Note that the typical behavior of this handler is expected to be a single `Os_AdvanceCounter()` call, because the next **match** will be in the future (i.e. it behaves just like a simple handler). Consequently, the handler should be as fast as possible for that case (since the worst-case behavior occurs when each **match** is triggered by a separate interrupt).



*It is important that you understand the interrupt behavior of the counter/compare hardware in use. When the **match** is set equal to the counter, there are three possible behaviors: the interrupt becomes pending as the **match** is set, the interrupt becomes pending as the counter moves beyond the **match**, or the counter needs to completely wrap around before the interrupt becomes pending again.*

In the example above, the test for Exit 3 assumes the counter hardware exhibits the first or third behavior. With the second behavior, it is necessary to exit if `remaining_ticks` is zero, as the interrupt will be asserted after the counter and **match** value have been observed as equal.

11.2.3 Handling a Hardware modulus not equal to TickType

The driver outlines presented in Section 11.2 so far have assumed that the counters and compare registers are the same width as `TickType` and arithmetic is unsigned modulo `TickType`. Some hardware may not have this property.

There are two cases:

1. the modulus of the hardware is less than the `TickType`
2. the modulus of the hardware is greater than the `TickType`

Both cases can be handled by changing aspects of the driver. The following sections discuss the changes in more detail.

Modulus less than TickType

In this case, we assume that the counter itself wraps to zero after some value ($m - 1$) (i.e. the counter has modulus m , where m is smaller than `TickType`). This increases the complexity of the drivers, but might be imposed by hardware behavior or may be necessary to support some other system requirement. For example, a timer set up with a modulus of 50000 and tick of 1ms could provide a 50ms interrupt via overflow used to drive a software counter and output compare interrupts used to provide drive a hardware counter.

Such a modulus requires modification to calculations which derive new compare values and which check the relationship between compare and counter values. The following example assumes that `TickType` has modulus 2^{16} .

If m is 2^x . (where $x < 16$), then it is simple to apply explicit modulus adjustments to arithmetic results by ANDing with $2^x - 1$. For 8 bit modulus, this would allow a compare value to be advanced by:

```
new_match = (old_match + Status.Delay) & 0xFF;
```

A similar operation can be applied to the result of calculating the ticks remaining to the **match**.

The calculations become more complex if the modulus value is not a power of two. Possible techniques are presented below.

Calculating of the new compare value must account for four possible results when the new `Delay` value is added to the **old** compare value is calculated using the `TickType` modulus of 2^{16} :

1. The `Delay` is zero. A full modulus wrap leaves the compare value unchanged.
2. The result is greater than the **old** compare value, but less than m . The result is the desired result.
3. The result is greater than m . The result of the addition needs to be wrapped at m . This can be achieved by subtracting m , avoiding the (often costly) modulus operator.
4. The result is less than the **old** compare value. The result of the addition wrapped at 2^{16} , so the result must have $(2^{16} - m)$ added to it to give the result of wrapping at m .

Note that if m is less than or equal to half the arithmetic modulus (i.e. less than or equal to half of 2^{16}), then the fourth case can never occur.

When checking whether the new output compare value has been set ahead of the counter, we consider three circumstances. No subtraction underflows the 2^{16} arithmetic modulus.

1. The Delay is zero, so the next **match** is known to be in the future. The handler is required to complete in less than the counter modulus.
2. The next **match** is greater than or equal to the counter so we can subtract counter from compare to give the interval until the next **match** then check whether this is less than or equal to the required Delay (otherwise, the next **match** has already occurred).
3. The next **match** is less than the counter value. Subtracting the **match** from counter gives the interval that remains when the interval to next **match** is subtracted from the modulus. Thus, we can calculate the interval to next **match** as $m - (\text{COUNTER} - \text{OUTPUT_COMPARE})$ and then check this result against the required Delay.

The same approach can be applied to the calculation of remaining time to **match** in the `Os_Cbk_State_<CounterID>()` call back.

Adding the mechanisms described above to conditions to the `ISR(OutputCompareInterrupt)` driver gives the following:

```
#include <Os.h>
/* The next line should result in a constant being substituted.
   We assume that the expression will be evaluated at compile
   time, avoiding modulus overflow at run time. m is the
   timebase modulus */

#define CMP_ADJUST ((TickType)65536u - m)

ISR(OutputCompare_SmallModulus){
    Os_CounterStatusType State;
    TickType counter_cache, remaining_ticks, new_match;
    clear_pending_interrupt();

    while(1) {

        Os_AdvanceCounter_<CounterID>();
        Os_Cbk_State_<CounterID>(&State);
```

```

if (State.Running == False) {
    /* Exit 1: alarms/schedule tables stopped */
    return;
}

if (State.Delay == 0u) {
    /* OUTPUT_COMPARE = OUTPUT_COMPARE if
     * needed to arm next interrupt */
    /* Exit 2: full wrap */
    return;
}

new_match = OUTPUT_COMPARE + State.Delay;

if (new_match > OUTPUT_COMPARE) {
    if (new_match >= m) {
        new_match -= m;
    }
} else {
    new_match += (CMP_ADJUST);
}

OUTPUT_COMPARE = new_match;
counter_cache = COUNTER;

if (new_match >= counter_cache) {
    remaining_ticks = new_match - counter_cache;
} else {
    remaining_ticks =
        m - (counter_cache - new_match);
}

if ((remaining_ticks != 0u)
    && (remaining_ticks <= State.Delay)) {
    /* Exit 3: match in the future */
    return;
}

if (interrupt_pending()) {
    /* Exit 4: interrupt pending */
    return;
}
}
}

```

Modulus greater than TickType

The alternative case is where a hardware counter has a modulus that exceeds `TickType`. With a little care, such counters can be used to provide the behavior required for a `TickType` with a modulus of 2^{16} . We restrict our consideration to modulus values that are a power of two (e.g. a 32 bit counter). In these cases, the low 16 bits of the counter have the desired behavior, but overflow effects must be taken into account.

When the compare value is advanced in the interrupt handler, overflow from the bottom 16 bits must be propagated through the rest of the compare register. In addition, a Delay of 0 indicates that 2^{16} must be added to the compare value. Since the **match** can never be advanced by more than this, checks for the timer having passed the **match** can be carried out using the low 16 bits of the counter and compare registers.

When the `Os_Cbk_Set_<CounterID>()` call back is used, the **match** must be set so that it matches the counter when the low 16 bits of the counter next have the same value as the parameter passed to `Os_Cbk_Set_<CounterID>()`. This can be achieved as follows (assuming that counter and compare are 32 bit unsigned values):

```
FUNC(void, OS_APPL_CODE) Os_Cbk_Set_<CounterID>(TickType Match)
{
    uint32 to_compare;

    disable_interrupt_source();
    disable_compare();
    clear_pending_interrupt();

    OUTPUT_COMPARE = (COUNTER & 0xFFFF0000ul) | Match;
    to_compare      = OUTPUT_COMPARE - COUNTER;

    if ((to_compare == 0ul) || (to_compare >= 0x10000ul) {
        if(!(interrupt_pending())) {
            OUTPUT_COMPARE += 0x10000ul;
            to_compare      = OUTPUT_COMPARE - COUNTER;
            if ((to_compare == 0ul) || (to_compare >= 0x10000ul)){
                if(!(interrupt_pending())) {
                    OUTPUT_COMPARE += 0x10000ul;
                }
            }
        }
    }
    enable_interrupt_source();
}
```


The operations are carried out with interrupts from the hardware device disabled, in order to make them atomic with respect to the handler. First any pending interrupts are cleared. This must be done after disabling comparison (for instance, setting the **match** to ensure that a pending interrupt can only be due to a **match** with the new **match**). Then, the compare register is set to the counter value with its lower 16 bits replaced by the Match parameter.

If the **match** lies in the future by less than 2^{16} ticks, then it has been set correctly. If there is a pending interrupt then the **match** must have been reached so the interrupt should be handled. Otherwise, the **match** is advanced by 2^{16} . The check must then be repeated to account for a race in which the counter could overtake the next **match** before it has been set. Checking twice is sufficient, assuming that the `Os_Cbk_Set_<CounterID>()` call completes in less than 2^{16} timer ticks.

This code assumes that the interrupt may or may not be pending if the **match** is set equal to the counter. If the interrupt is known to become pending when (or after) the two match, then the check for `to_compare` being zero should be removed.

Note that this function can be much simplified based on knowledge of application behavior. For example, if the counter is zeroed at startup and is started only once less than Match ticks after startup, then it is sufficient to set the compare value to Match.



Modulus 2^{16} behavior is not exhibited by the low 16 bits of a counter which has a modulus that is not a power of two: the last interval before the timer wraps consist of (counter modulus MOD 2^{16}) ticks.

11.3 Free Running Counter and Interval Timer

The counter compare handlers described in Section 11.2 allow the implementation of drift-free hardware counter drivers. However, not all target platforms provide such counter facilities.

Drift can be avoided when using a down counter if a separate free running counter is also available. The free running counter is used to provide a drift-free time reference, and the down counter is set up to interrupt when the next **match** becomes due. Some jitter (delay) may be introduced to individual matches due to delays in setting the interval for the down counter, but these do not accumulate (such jitter can be accounted for in the same way as jitter introduced in the handling of the interrupt). In this section, the down counter is considered to provide registers `COUNTER` and `DOWN_COUNTER` that can be used as variables. As in the previous example, both registers are taken to be `TickType` wide registers, and the values they use are taken to be unsigned `TickType` size integers.

11.3.1 Callbacks

All of the callbacks in this section assumed that the next **match** value is maintained in software and used in calculation of the down count value to the next interrupt. This can be declared as follows:

```
TickType next_match;
```

Cancel

The `Os_Cbk_Cancel_<CounterID>()` callback function only needs to disable the interrupt so the implementation is the same as before.

```
FUNC(void, OS_APPL_CODE) Os_Cbk_Cancel_<CounterID>(void){
    clear_pending_periodic();
    disable_interrupt_source();
}
```

Now

The `Os_Cbk_Now_<CounterID>()` callback function needs to return the value of the free-running counter.

```
FUNC(TickType, OS_APPL_CODE) Os_Cbk_Now_<CounterID>(void){
    return (TickType)COUNTER;
}
```

Set

Things start to change with the `Os_Cbk_Set_<CounterID>()`. The callback needs to set the `DOWN_COUNTER` so that it reaches zero (and interrupts) at a relative number of ticks from **now**. This is done by subtracting the `COUNTER` value from the `Match` value.



This relies on all three counters having the same modulus.

The callback must also log the next **match** value from the absolute `Match` parameter value passed into the call by RTA-OS3.0 (this will be used by the `Os_Cbk_State_<CounterID>()` callback later).

```
FUNC(void, OS_APPL_CODE) Os_Cbk_Set_<CounterID>(TickType Match)
{
    /* Record value at which expire is due */
    next_match = Match;
    disable_compare();
    clear_pending_interrupt();

    /* set up interrupt when counter reaches match value */
}
```

```

    DOWN_COUNTER = next_match - COUNTER;
    enable_interrupt_source();
}

```

State

Note that the `Os_Cbk_State_<CounterID>()` call, below, could return `DOWN_COUNTER` as the `Status.Delay` value. If there is any jitter introduced by setting the down counter, this will reflect in the time at which the next **match** will be signaled, rather than when it is due. However, particularly with a non-TickType modulus where more calculation is avoided, the following may be acceptable.

```

FUNC(void, OS_APPL_CODE) Os_Cbk_State_<CounterID>(
    Os_CounterStatusRefType State){
    State.Delay    = next_match - COUNTER;
    State.Running = True;

    if (interrupt_pending()) {
        State.Pending = True;
    } else
        State.Pending = False;
    }
    return;
}

```

11.3.2 ISR

This demonstrates a looping form of ISR: it loops until no due matches remain, rather than handling one **match** per invocation of the routine, as in a re-triggering form of ISR.

```

#include <Os.h>
ISR(IntervalTimerInterrupt){
    Os_CounterStatusType State;
    TickType              remaining_ticks;

    clear_pending_interrupt();

    while(1) {
        Os_AdvanceCounter_<CounterID>();
        Os_Cbk_State_<CounterID>(&State)

        if (State.Running == True) {
            /* Exit 1: all alarms/schedule tables stopped */
            return;
        }
    }
}

```

```

    }

    next_match += State.Delay;
    /* Subtract adjustment for delay before COUNTER is set */
    remaining_ticks = next_match - COUNTER;

    if (State.Delay == 0u) {
        DOWN_COUNTER = remaining_ticks;
        /* Exit 2: full wrap */
        return;
    }

    if ((remaining_ticks != 0u) &&
        (remaining_ticks <= State.Delay)) {
        DOWN_COUNTER = remaining_ticks;
        /* Exit 3: counter set for next expire */
        return;
    }

    /* assume we only get an interrupt due to setting the
       counter and we only set the counter when we are going to
       exit so no need to test for pending interrupt */
}
}

```

Note that exit 2 assumes that setting the counter to zero will result in an interrupt after one full wrap of ticks.

11.4 Using Match on Zero Down Counters

Some hardware might not provide a free running counter (or you might not want to use this for your hardware driver).

In this case you will have to use just the interval timer. This example assumes a 16-bit decrementing counter that raises an interrupt on reaching 0, and continues to decrement. Because the counter continues to decrement, the start point for the new countdown can be determined by adding the Delay to the counter value (assuming modulo 2^{16} arithmetic). It is desirable to minimize drift during the counter update. Preventing interrupts during the update, and adding an adjustment for the known time taken for update (to both the counter and next_match), may be able to reduce this to one tick per counter adjust (assuming the counter is asynchronous to the update, there will always be some uncertainty). counter_adjust is introduced to allow calculation of a **now** value: subtracting the counter value from next_match gives this. Note that the counter update and counter_adjust update must

be atomic with respect to any call to obtain **now** for this to give the correct result.

When the driver is not running, the down counter is assumed to free-run. From start-up it runs downwards from zero and the value of **now** is (0 - counter). `counter_adjust` is used to hold the actual tick value that a *free running* counter would have reached the next time the `DOWN_COUNTER` has the value 0. this means that `counter_adjust` can be used to synthesize a virtual free-running counter for the purposes of the hardware counter driver.

11.4.1 Callbacks

Cancel

Canceling the driver is achieved as before.

```
FUNC(void, OS_APPL_CODE) Os_Cbk_Cancel_<CounterID>(void){
    clear_pending_periodic();
    disable_interrupt_source();
}
```

Now

The `Os_Cbk_Now_<CounterID>` callback cannot just return the value of the `DOWN_COUNTER` because the counter is not free running or monotonically increasing. Instead, the **now** value is calculated by subtracting the `DOWN_COUNTER` value from the `counter_adjust` to give the virtual free-running value.

```
FUNC(void, OS_APPL_CODE) Os_Cbk_Now_<CounterID>(void){
    return (counter_adjust - DOWN_COUNTER);
    /* counter_adjust is still correct adjustment
       * as counter runs to and through 0 */
}
```

Set

The race conditions discussed in Section 11.2.1 are still present in this model. If the interrupt is dismissed before the `DOWN_COUNTER` is set, there is a risk that an interrupt may occur between dismissing the interrupt and setting the down counter. If the interrupt is set after the down counter is set, a small delay could result in the expected interrupt being discarded. In the absence of specialized hardware protection, this can be avoided by the `disable_compare()` function setting the counter to modulus - 1, then dismissing the interrupt between determining the `AdjustedMatch` value and setting the counter (as shown in the above example).

```
TickType counter_adjust = 0;
```

```

FUNC(void, OS_APPL_CODE) Os_Cbk_Set_<CounterID>(TickType Match)
{
    TickType AdjustedMatch;
    AdjustedMatch = Match - (counter_adjust - DOWN_COUNTER);

    /* dismiss interrupt in a way that avoids race conditions */
    disable_compare();
    clear_pending_interrupt();

    DOWN_COUNTER    = AdjustedMatch;
    counter_adjust += AdjustedMatch;
    enable_interrupt_source();
}

```

State

Os_Cbk_State_<CounterID>() needs to set the Delay and can simply read the value of the DOWN_COUNTER to get this. The rest of the callback is identical to the others you have seen in this chapter.

```

FUNC(void, OS_APPL_CODE) Os_Cbk_State_<CounterID>(
    Os_CounterStatusRefType State){
    State.Delay = DOWN_COUNTER;
    State.Running = True;
    if (interrupt_pending()) {
        State.Pending = True
    } else {
        State.Pending = False;
    }
}

```

11.4.2 Interrupt Handler

The following example shows an appropriate interrupt handler.

```

#include <Os.h>
ISR(MatchOnZeroInterrupt){
    Os_CounterStatusType State;
    TickType             counter_cache;

    clear_pending_interrupt();

    while(1) {
        Os_AdvanceCounter_<CounterID>();
        Os_Cbk_State_<CounterID>(&State);
    }
}

```

```

if (State.Running == True) {
    /* Exit 1: all alarms/schedule tables stopped */
    return;
}

if (State.Delay == 0) {
    /* Exit 2: full wrap */
    return;
}

counter_cache = COUNTER + State.Delay;
COUNTER = counter_cache;
counter_adjust += State.Delay;

if ((counter_cache != 0u) &&
    (counter_cache <= State.Delay)) {
    /* Exit 3: next match not yet been reached */
    return;
}

if (interrupt_pending()) {
    /* Exit 4: interrupt pending */
    return;
}
}
}

```

The condition on Exit 3 assumes that the interrupt becomes pending when (not after!) the counter reaches zero, but may not do so if it is set to zero (if the counter is zero then the **match** is due and will be dealt with either by looping or re-entering via the pending interrupt). The same counter value must be used for both parts of the test otherwise races can occur if the counter changes between the two comparisons (hence the use of `counter_cache`).

If the behavior of the interrupt when the counter is set to zero is known, then the code can be simplified by removing Exit 4 and the associated test (since the interrupt status when `counter_cache` is zero will be known). If setting the counter to zero never causes the interrupt to become pending then that is the only change required. If setting the counter to zero always causes the interrupt to become pending, then Exit 3 should only check for `counter_cache` less than or equal to `Delay`. If the counter is zero, the interrupt will be pending and will cause the next event to be handled.

In the case of a very fast running clock (where the clock speed is greater than or equal to the processor speed), it will be necessary to add a correction to the counter to offset the number of ticks that occur between reading the counter and setting its new value. In any case, a drift of up to one tick cannot be avoided whenever the down counter is set. On a multiple interrupt level platform, it is desirable to disable all interrupts whilst reading/writing COUNTER to avoid the possibility of interruption between these operations, resulting in a large amount of drift.

11.5 Software Counters Driven by an Interval Timer

Using a periodic interval timer (or any per-event interrupt source) with an interrupt on zero can be used to synthesize a free-running counter in software. However, a handler of this form is of limited practical interest because there is one interrupt per tick. This means it is identical to incrementing a software counter. It is recommended that you use the software counter driver model instead.

11.6 Summary

- You need to provide an hardware driver for every hardware counter and advanced schedule.
- The driver interface comprises:
 - A Category 2 interrupt handler that tells RTA-OS3.0 to take action; and
 - Four callback functions used by RTA-OS3.0 to control the counter/schedule.
- If possible, you should use a free running counter with associated compare hardware and a simple interrupt handler.
- More advanced models can be supported though the interface if required.
- It is essential that you understand how your hardware generated the counter tick source and what happens when an interrupt from the device occurs.

12 Startup and Shutdown

Some operating systems that you might have used before will take control of the hardware. RTA-OS3.0, however, is different.

Initially the operating system is not running, so you are free to use the hardware as if no real-time operating system is being used. Until you explicitly start the operating system with an API call, it is not running.

RTA-OS3.0 can be started in different application modes. A mode is a set or subset of the complete application functionality that corresponds with a specific function of the application. You will learn more about application modes in Section [12.2.2](#).

12.1 From System Reset to StartOS()

This section looks at what has to be done between an embedded processor “coming into life” when power is applied and the StartOS() API call being made to start RTA-OS3.0 and your application. The details of what goes on in this period are naturally dependent on the particular embedded processor in use - the underlying principles are however the same. You should read this section in conjunction with the reference manual for your target processor and apply the concepts we describe to your own platform.

12.1.1 Power-on or Reset

When power is applied to an embedded processor, or the processor is reset, the processor does one of two things (depending on the type of processor).

It may start executing code from a fixed location in memory, or it may read an address from a fixed location in memory and then start executing from this address. The fixed location in memory that contains the address of the first instruction to execute is often called the “reset vector” and is sometimes an entry in the interrupt vector table.

In a production environment, the reset vector and/or the first instruction to be executed is usually in non-volatile memory of some variety. In a development environment it is often in RAM to permit easy re-programming of the embedded processor. Some evaluation boards (EVBs) have switches or jumpers that permit the reset vector and/or the first instruction to be in EEPROM or RAM.

Going from power-on or reset to the first instruction being executed is often referred to as “coming out of reset”. After a processor has come out of reset it usually:

- has interrupts disabled,
- is in supervisor mode (if the processor supports it) - i.e. it can execute

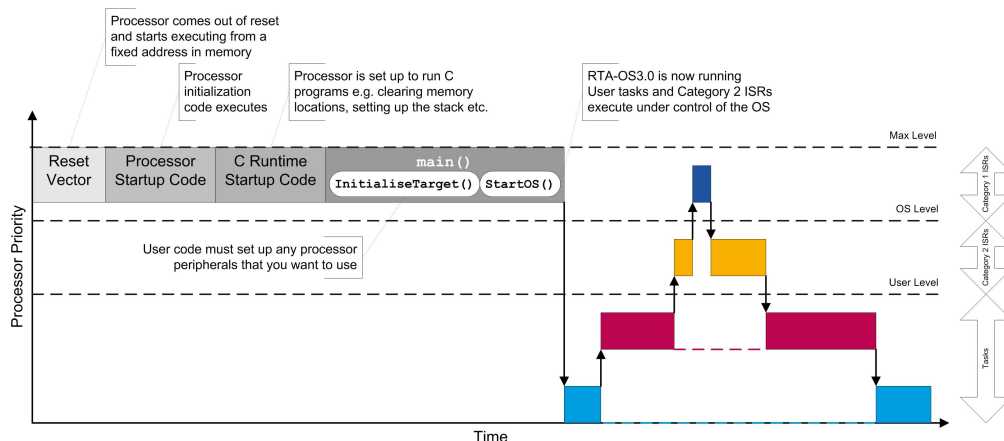


Figure 12.1: System Startup

all instructions and access all addresses without causing an exception and has all forms of memory and I/O protection turned off.

- is in single-chip mode (if the processor supports it) - i.e. the chip is in a “self-contained mode” where external memory is not usable and external buses are disabled.

12.1.2 C Language Start-up Code

It is possible to have any code you would like executed when a processor comes out of reset but it is normal if using a high-level language such as C for this bootstrap code to be supplied with your compiler.

The compiler vendor supplies an object module or library that contains the bootstrap code. The bootstrap code usually does two key things:

1. it carries out basic processor configuration, for example bus configuration, enabling of access to internal RAM, etc.
2. it invokes the C language start-up code. Most of this is concerned with initializing data structures, clearing memory, setting up the stack pointer, etc.

Directives in the object module/library or in the linker configuration file are used to ensure that the bootstrap code (and reset vector value if needed) are placed in the correct location in memory.

The C language start-up code is usually supplied by the compiler vendor in an object module with a name like `crt0` or `startup` and the code can usually be identified in a map file by looking for a symbol with a name something like `__start` or `__main`. The source to this module is usually available to you.

For some target hardware, ETAS supplies a different version of the standard startup code that should be used with RTA-OS3.0 applications. The *RTA-OS3.0 Target/Compiler Port Guide* and the example supplied with RTA-OS3.0 will tell you how to use this.

The start-up code initializes the C language environment. For example, it sets up the stack pointer, the heap used for `malloc()` and it initializes global variables by copying their default values from ROM into RAM. Finally, the start-up code invokes the application start-up code.

12.1.3 Running `main()`

The application start-up code is typically in a function called from `main()`. The application start-up function has two things to do to work with RTA-OS3.0:

1. Initialize the target hardware into a state where RTA-OS3.0 and the application can run
2. Call `StartOS()` to start RTA-OS3.0 running.

For example the application start-up code for an RTA-OS3.0 application may look like:

```
OS_MAIN(){
    InitializeTarget();
    StartOS(OSDEFAULTAPPMODE);
    /* Never reach here */
}
```

The macro `OS_MAIN()` is provided for your convenience by RTA-OS3.0 to mark the main function of your application - you do not have to use this to work with RTA-OS3.0. The macro is used to handle the cases where using **`void main(void)`** is forbidden by the compilers.

The `InitializeTarget()` function in the above example need to be written by you to initialize the target hardware. The remainder of this section describes the types of things that you may have to do to initialize target hardware into a state where your application and RTA-OS3.0 can run. This description is necessarily generic as every embedded processor is slightly different. It is probably wise to read this section in conjunction with the *RTA-OS3.0 Target/Compiler Port Guide* for your processor and the processor's reference guide.

Setting up Memory

In general, memory configuration is carried out by the bootstrap code that is run before the application start-up code is executed. In more complex

embedded processors. However, the memory configuration set-up by the bootstrap code may not be what is required for the application. For example, if the processor has internal RAM and an external memory bus then it is most likely that the bootstrap code will have configured the processor to use the internal RAM. If your application needs to use RAM on the external memory bus, then you will need to configure the processor to use the external RAM. Configuring access to RAM typically involves programming bank select and mask registers - however the details depend on the embedded processor.

Setting up Peripherals

Most embedded applications make use of peripheral devices which may be part of the embedded processor or attached through I/O or memory buses. Examples are CAN controllers, Ethernet controllers and UARTs. It is generally a good idea to set-up peripheral devices before RTA-OS3.0 is started since at this point the application code cannot be preempted and has complete control over interrupts.

Setting up Interrupts

Interrupt sources for Category 1 and 2 interrupts should be configured before `StartOS()` is called. Typically, you should ensure that the IPL is set to OS level and then both configure interrupt sources. You can also enable Category 1 interrupt sources [here](#).



Do not enable Category 2 interrupts before calling `StartOS()` as this can result in a race condition where the interrupt needs to be handled before RTA-OS3.0 has been initialized. You should use the `StartupHook()` to enable Category 2 interrupt sources. This model means that Category 2 interrupts will not be generated until `StartOS()` lowers the IPL just before it enters the idle mechanism.

On some microcontrollers it will be necessary to program priority registers in the hardware that configure interrupt priorities. The values you program must match the priority values that you told RTA-OS3.0 at configuration time, otherwise your application will not work properly. On targets where this is the case, RTA-OS3.0 will usually provide helper code so that you can do this job correctly. You should check the *RTA-OS3.0 Target/Compiler Port Guide* for any special instructions relating to target initialization.

Enabling Interrupts

Category 1 interrupts may also be enabled so that they generate interrupts immediately as the handling of Category 1 interrupts is completely outside the scope of RTA-OS3.0.

Category 2 interrupt sources must not actually generate interrupts until af-

ter `StartOS()` has completed initialization. You must not enable Category 2 interrupt sources before calling `StartOS()`. If you do this, then you can get a race condition where the interrupt occurs before RTA-OS3.0 is correctly initialized.



Enabling Category 2 interrupt sources before `StartOS()` will result in undefined behavior.

RTA-OS3.0 provides a safe way to enable Category 2 interrupt sources using the `StartupHook()` which is described in Section [12.2.1](#).

Setting up Timers

Most embedded applications use hardware timers. Timers are usually configured to “tick” and generate interrupts at a fixed frequency. The ISR associated with the timer interrupts then either activates a task directly or ticks an OSEK counter (i.e. calls `IncrementCounter(CounterID)`).

Setting up a hardware timer depends on the design of the timer but there are two common forms:

1. a count register is set to zero and a match register is set to the maximum value for the count register. The count register is incremented by the processor at a given frequency and, when it reaches the value in the match register, it generates an interrupt and resets the count register to 0.
2. a count register is loaded with the number of ticks to occur before an interrupt should be generated. The processor decrements the count register at a given frequency. When the register reaches zero, an interrupt is generated. Usually the ISR that handles the interrupt is responsible for reloading the count register.

The frequency at which timers must run will depend on your application. It is vital that all counters run at the frequency specified in their definition. If you have told RTA-OS3.0 that a counter driven by a timer has a particular tick rate, i.e. you have specified the “Seconds Per Tick” attribute, then you must make sure that your timer hardware is configured to give a tick at the same rate.

12.2 Starting RTA-OS3.0

Once your hardware is initialized, you can start RTA-OS3.0

RTA-OS3.0 is started only once a `StartOS()` call is made. This call is usually made from the main program. It is up to you to perform any hardware initial-

ization that is necessary for the application. The initial state of RTA-OS3.0 is described in the *RTA-OS3.0 Reference Guide*.

StartOS(AppModeID) takes a single application mode parameter. This parameter is either the default mode OSDEFAULTAPPMODE or another mode that has been configured in [rtaoscfg](#).

Have a look at the example main function in Code Example 12.1, which starts the operating system in the default application mode.

```
#include <Os.h>
OS_MAIN(){
    InitializeTarget();
    StartOS(OSDEFAULTAPPMODE);
    /* Never reach here */
}
```

Code Example 12.1: Example Main Function

The call to StartOS() does not return. Once the RTA-OS3.0 is initialized, all interrupts are enabled and the Os_Cbk_Idle() runs until a higher priority task or ISR occurs.

Most RTA-OS3.0 API calls can be made from the idle mechanism. However, you cannot use any calls that would require the idle mechanism to terminate (for example, it is not possible to call TerminateTask() from the idle mechanism).



You should not make RTA-OS3.0 API calls that manipulate OS objects or enable Category 2 interrupts before calling StartOS().

RTA-OS3.0 can be suspended by disabling all Category 2 interrupts and ensuring that they will not be raised on some future event, such as an output compare match.

RTA-OS3.0 will be suspended when no Category 2 interrupts are raised and the idle mechanism is running. You can resume RTA-OS3.0 by re-enabling Category 2 interrupts and then resume making RTA-OS3.0 calls.

12.2.1 Startup Hook

The Startup Hook is called by RTA-OS3.0 during the StartOS() call after the kernel has been initialized, but before the scheduler is running.

StartOS() raises the interrupt priority level (IPL) to OS level as soon as it is called and lowers it to user level just before it returns. This means that the startup hook runs with Category 2 ISRs masked. That means you can safely enable interrupt generation in StartupHook() knowing that it will not actually



Figure 12.2: Execution of the Startup Hook

result in an interrupt occurring until StartOS() has completed initialization and RTA-OS3.0 is ready to run. At this point StartOS() unmask Category 2 interrupts and the OS is running.

Figure 12.2 shows the execution of the Startup Hook relative to the initialization of RTA-OS3.0.

Code Example 12.2 shows how Startup Hook should appear in your code.

```
FUNC(void, OS_APPL_CODE) StartupHook(void) {
    /* Startup hook code. */
    EnableIOInterrupts();
    EnableTimerInterrupts();
    ...
}
```

Code Example 12.2: Using the Startup Hook

The Startup Hook is often used for the initialization of target hardware (for example the enabling of interrupts sources that have been configured in by the code you executed before the call to StartOS()).

12.2.2 Application Modes

Applications can be started in different modes, which might represents part of the complete functionality. These modes could correspond with specific functions of the application. You could have, for example, an end-of-line programming mode, a transport mode and a normal mode.

You can define as many application modes as you want. Figure 12.3 shows how to declare different application modes in `rtaoscfg`.



You must declare an application mode called OSDEFAULTAPPMODE.

StartOS(MyAppMode) will start RTA-OS3.0 in MyAppMode and you can use the GetApplicationMode() API call to work out which mode you are in. This means that you can write application code that is mode-dependant. Code Example 12.3 shows how a task can be written so that it has different behavior in different modes.

```
TASK(Moded) {
    AppModeType CurrentMode;
```

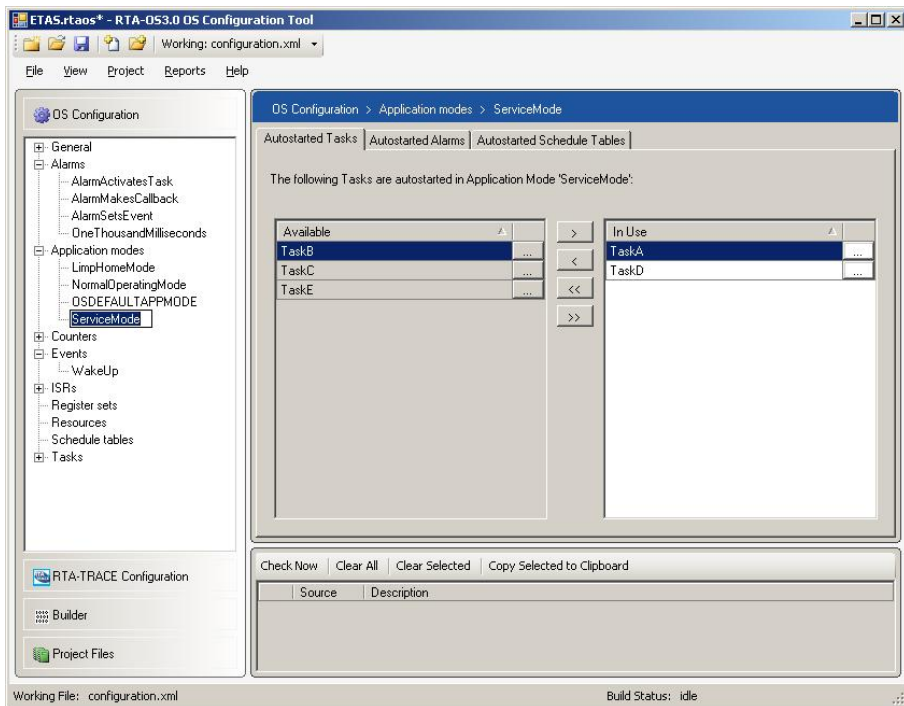


Figure 12.3: Configuring Application Modes

```

GetApplicationMode(&CurrentMode);
switch (CurrentMode) {
    case DiagnosticMode:
        DoExtendedFunctionality();
        break;
    case LimpHome
        DoBasicFunctionality();
        break;
    default:
        DoNormalFunctionality();
        break;
}
...
}

```

Code Example 12.3: Adding moding to a task

Application modes can also be associated with a set of tasks, alarms and schedule tables that are started automatically when the operating system starts. This means you can customize what happens during `StartOS()` for each of your declared modes.

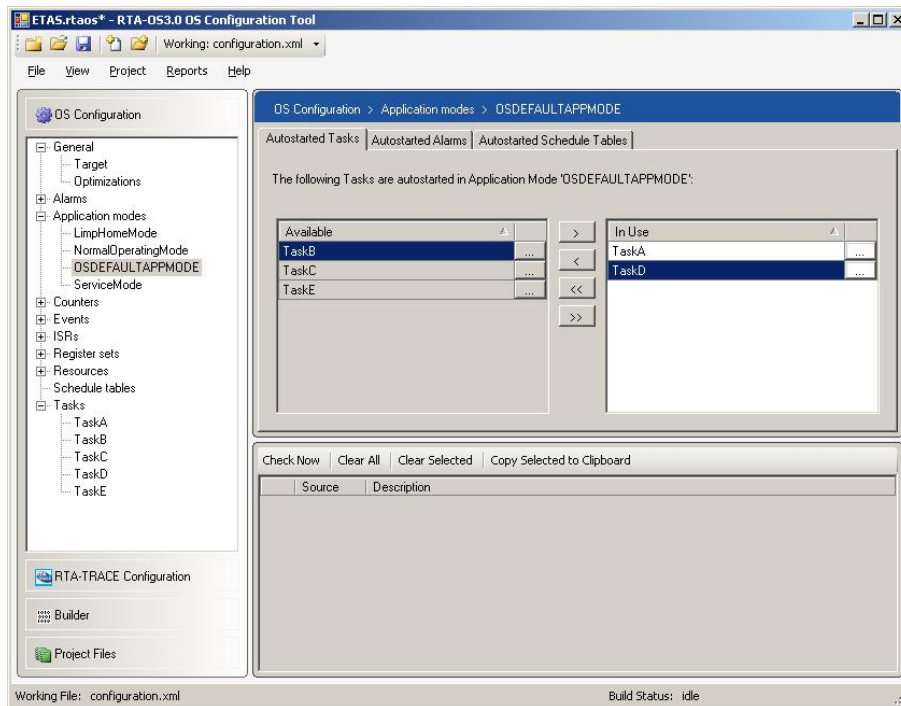


Figure 12.4: Declaring an Auto-started Task

Auto-starting Tasks

Any task can be auto-started in any application mode. When you auto-start a task the OS activates the task during the call to `StartOS()` i.e. an `ActivateTask()` API call is made internally. If you auto-start a basic task then it will have run and terminated before you reach the `Os_Cbk_Idle`. If you auto-start an extended task then it will run and either reach its first `WaitEvent()` API call for an event that has not yet been set or it will have terminated before you reach the `Os_Cbk_Idle`.



You do not need to auto-start tasks that you don't need to run immediately on startup. Tasks that are not auto-started can still be activated and run at a later stage through normal activation operations, expiry of alarms, processing of schedule table expiry points, etc.

Figure 12.4 shows that Task1 has been auto-started in OSDEFAULTAPPMODE and Production application modes.

Auto-starting tasks is typically useful for two cases:

1. Running an initialization task before other tasks in the system start to execute.

If you need to do this, then you must ensure that the auto-started task has a higher priority than any of the tasks that need to run after the

initialization task.

2. Starting extended tasks.

You will recall from Section 7.2 that you cannot set events for extended tasks in the suspended state and that the structure of the task is typically an infinite loop and a series of `WaitEvent()` calls. By auto-starting extended tasks you can avoid any potential errors that may occur by setting events on auto-starting extended tasks.



Auto-started tasks execute in priority order, from the highest to the lowest priority. If a higher priority task sets events for a lower priority task, then the events will be processed by the lower priority task when it executes.

Auto-starting Alarms

Alarms can also be auto-started in any application mode. When `StartOS()` returns, all auto-started alarms will have been set.

Auto-started alarms can be started at either an absolute or relative tick value. This has the same behavior as `SetAbsAlarm()` and `SetRelAlarm()` respectively and configuration uses the same types of parameters. If an alarm is auto-started, then you must specify an alarm time and a cycle time. The same restrictions apply for these parameters as for the `offset`, `start` and `cycle` parameters to the alarm API calls:

	Alarm Time		Cycle Time	
	Min	Max	Min	Max
Relative	1	MAXALLOWEDVALUE	MINCYCLE	MAXALLOWEDVALUE
Absolute	0	MAXALLOWEDVALUE	MINCYCLE	MAXALLOWEDVALUE



If you auto-start an alarm in absolute mode with alarm time zero, then the alarm will not expire until a full modulus wrap of the underlying counter has occurred (i.e. after `MAXALLOWEDVALUE+1` ticks have elapsed) because 0 is already in the past. For example, if you have an alarm on a millisecond counter then it will not occur until 65536ms (65.5 seconds) have elapsed.

Figure 12.5 shows you how an alarm is configured for auto-starting.

Auto-started alarms are useful when you want to start a set of cyclic (periodic) tasks when the OS starts. If you are using alarms to start multiple tasks and you need the tasks to run at specific cyclic rates *relative to each other*, then you must make sure that the alarms are auto-started. This is the only way to guarantee alarm synchronization.

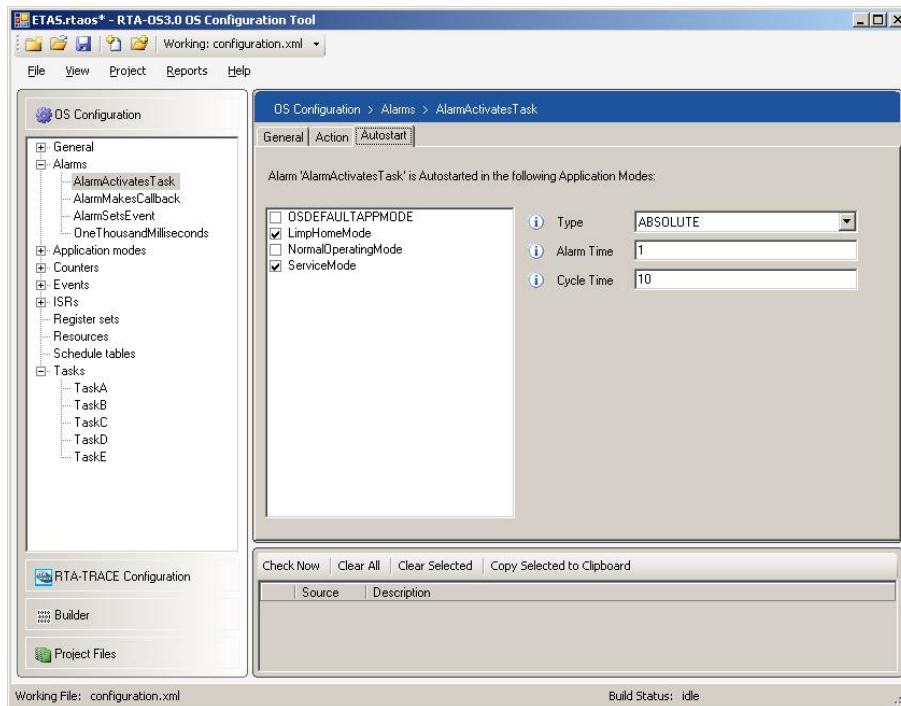


Figure 12.5: Auto-starting an Alarm

Auto-starting Schedule Tables

Schedule tables can be auto-started in any application mode. When `StartOS()` returns, all auto-started schedule tables will be running.

Like alarms, schedule tables can be started at either an absolute or relative tick value. This has the same behavior as `StartScheduleTableAbs()` and `StartScheduleTableRel()` respectively and configuration uses the same types of parameters. If a schedule table is auto-started, then you must specify an absolute start value or a relative offset depending on the mode in which you start the schedule table. The same restrictions apply for these parameters as for the offset, start and cycle parameters to the schedule table start API calls:

	Relative Offset		Absolute Value	
	Min	Max	Min	Max
Relative	1	MAXALLOWEDVALUE	-	-
Absolute	-	-	0	MAX



Schedule tables that are started with an absolute value zero will not expire until a full modulus wrap of the underlying counter has occurred (i.e. after `MAXALLOWEDVALUE+1` ticks have elapsed) because the tick value of zero is already in the past when the schedule table is started.



Figure 12.6: Execution of the Shutdown Hook

12.3 Shutting Down RTA-OS3.0

The operating system can be shutdown at any point by making the `ShutdownOS()` API call. When this happens, RTA-OS3.0 will immediately disable interrupts and then enter an infinite loop. If you have configured the `ShutdownHook()` it is called before the infinite loop is entered.

The `ShutdownHook()` is always passed a parameter that can be used to determine the reason for shutdown and then take any necessary action.

12.3.1 Shutdown Hook

The Shutdown Hook is called during the execution of the `ShutdownOS()` API call. Figure 12.6 shows the execution of the Shutdown Hook with respect to a `ShutdownOS()` API call.

Code Example 12.4 shows how Shutdown Hook might appear in your code.

```
FUNC(void, OS_APPL_CODE) ShutdownHook(StatusType Error) {
    /* Shutdown hook code. */
    switch (Error) {
        case E_OK:
            /* Normal shutdown */
            break;
        default:
            /* Abnormal shutdown */
            LogError();
            break;
    }
    for(;;); /* Wait for reset */
}
```

Code Example 12.4: Using the Shutdown Hook


You should not normally return from the `ShutdownHook()`. If you do then RTA-OS3.0 will disable all interrupts and enter an infinite loop running at OS level.

12.4 Restarting RTA-OS3.0

AUTOSAR OS does not provide any mechanism for restarting the OS at runtime other than through a watchdog reset. This is an unfortunate side-effect of `StartOS()` not returning when a `ShutdownOS()` call is made. This is a significant shortcoming in the AUTOSAR standard because it is an extremely

common requirement to be able to restart the OS in different modes during runtime. For example, an ECU may have a power-saving mode or a “limp-home” mode.

RTA-OS3.0 removes this limitation by providing two API calls that are used in combination to restart the OS.

 *Restarting of the OS is unique to RTA-OS3.0 and is not part of the OSEK or AUTOSAR standards. Use of the features described in this section are therefore not portable to other implementations.*

The API call `Os_SetRestartPoint()` places a marker in your code to which the API call `Os_Restart()` jumps when the call is made. `Os_SetRestartPoint()` cannot be made once `StartOS()` has been called and therefore must occur before the `StartOS()` call for restart to be possible.



It is only possible to restart RTA-OS3.0 once it has been shutdown. You can only call `Os_Restart()` from the `ShutdownHook()`.

Using this feature allows you to jump back to any arbitrary point in your pre-`StartOS()` initialization, so you can place code to initialize other parts of the system outside the OS.

Code Example 12.5 shows how you might use structure of the main program when using `Os_SetRestartPoint()` to place a marker.

```
AppModeType StartupAppMode;
OS_MAIN(){
    InitializeTarget();
    /* Set up normal application mode */
    StartupAppMode = NormalOperation;
    Os_SetRestartPoint(); /* We will return here on restart */
    switch (StartupAppMode) {
        case NormalOperation:
            /* Do mode-specific initialization */
            break;
        case LimpHome:
            /* Do mode-specific initialization */
            break;
        ...
    }
    StartOS(StartupAppMode);
}

FUNC(void, OS_APPL_CODE) ShutdownHook(StatusType Error){
    ...
}
```

```

if (FailureDetected == True) {
    StartupAppMode = LimpHomeMode;
    Os_Restart();
    /* Never reach here */
}
...
}

```

Code Example 12.5: Using `Os_SetRestartPoint()` and `Os_Restart()`

12.5 Summary

- RTA-OS3.0 will not work unless everything is located in the right place in memory.
- The target hardware must be initialized before RTA-OS3.0 can run.
- RTA-OS3.0 does not run until the `StartOS()` call is made.
- RTA-OS3.0 can be stopped at any time using the `ShutdownOS()` call.
- RTA-OS3.0 can be restarted by using the `Os_SetRestartPoint()` call to place a restart marker before `StartOS()`, using the `Os_Restart()` call to jump back to the marker and calling `StartOS()` to restart RTA-OS3.0.
- Tasks, alarms and schedule tables can be auto-started in different application modes.

13 Error Handling

Many of the RTA-OS3.0 API calls return an error code at runtime which tells you whether the OS detected an error during the execution of the API call or not. The set of error codes that are returned depend on two things:

1. the build status of the OS
2. the API call itself

The OS provides two types of build status:

Standard status does a minimum amount of runtime error checking and is intended for production builds of your application (i.e. the build that you will send into series production after you have gained sufficient confidence that your application is free from errors). Four classes of error are detected:

1. E_OK - no error was detected. It is possible that this is because no error checking was done. In this case the call will not have modified the state of the OS (it will have silently failed).
2. E_OS_LIMIT - an internal limit of the OS was reached, for example you tried to activate a task more often than your configuration allows.
3. E_OS_NOFUNC - the call cannot be made
4. E_OS_STATE - the call cannot be made because the object is not in a valid state

Extended status performs the checks as standard build, but adds a significant amount of extended error checking to check for all reasonable violations of OS API usage. There are too many errors to list here, but they fall into 3 classes:

1. E_OK - no error was detected. It is possible that this is because no error checking was done. In this case the call will not have modified the state of the OS (it will have silently failed).
2. E_OS_<standard_code> - an error case defined by the AUTOSAR (or OSEK) OS standard occurred.
3. E_OS_SYS_<vendor_code> - an error case defined by ETAS occurred, in addition to the cases identified by the E_OS_<standard_code> codes occurred.



You are strongly encouraged to use extended status during in the early stages of development so that you can debug any problems arising from incorrect use of the RTA-OS3.0 API. When you are sure that you are using the OS correctly, you can use standard status to check non-functional properties of your application like production memory sizes and performance.

Each API call that returns an error code will return a different set of values depending on what type of errors can occur when the call is made.

Common (extended build) errors are:

Error Code	Meaning
E_OS_ID	You made an API call on the wrong type of object
E_OS_VALUE	A parameter is outside a permitted range
E_OS_CALLEVEL	You made an API call from the wrong place

You can find out which API calls return which error codes, and what each code means for the specific API (and therefore what you might have to do to fix the error) by referring to the *RTA-OS3.0 Reference Guide*.

13.1 Centralized Error Handling - the ErrorHook()

The common way of checking errors from either standard or extended status builds is to use the error hook which provides a “catch all” error handler. If the ErrorHook is enabled, then it is called by RTA-OS3.0 when any API call is about to return an error code that is not E_OK. The error code is passed into the Error Hook routine and you can use it to work out which error has occurred.

Figure 13.1 shows how the Error Hook is enabled.



If you enable the error hook then you must provide an implementation. If you do not provide an implementation then your program will not link correctly.

Depending on the severity of the error, you can decide whether to terminate (by calling ShutdownOS()) or to resume (by handling or logging the error and then returning from ErrorHook()). Code Example 13.1 shows you the usual structure of the Error Hook.

```
FUNC(void, OS_APPL_CODE) ErrorHook(StatusType status) {
    switch (status) {
        case E_OS_ACCESS:
            /* Handle error then return. */
            break;
        case E_OS_LIMIT:
            /* Terminate. */
```

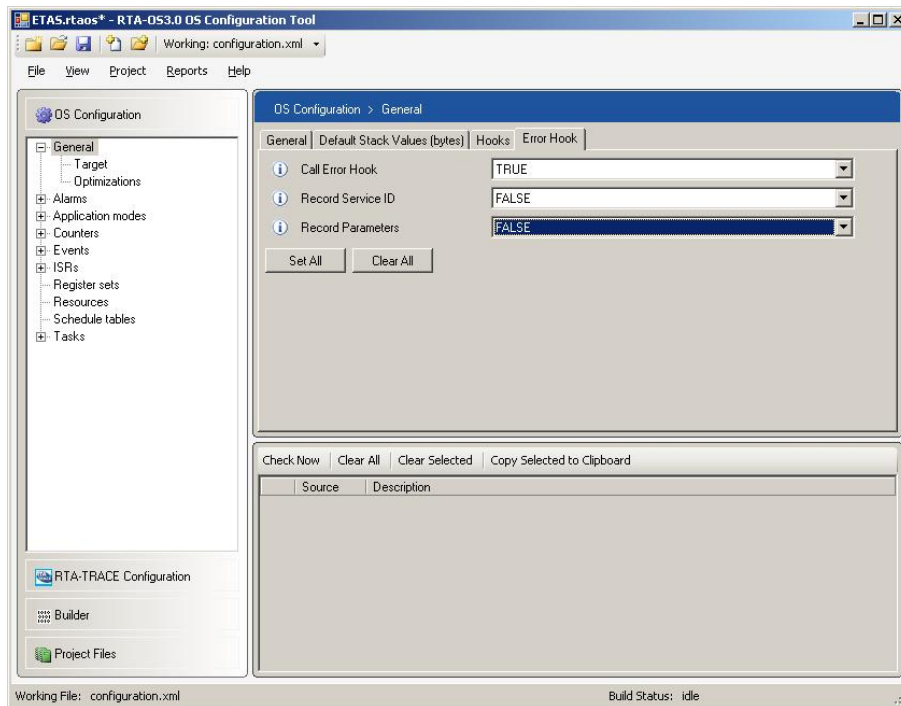



Figure 13.1: Configuring the Error Hook

```

ShutdownOS(status);
default:
break;
}
}

```

Code Example 13.1: Suggested Structure of the Error Hook

As with the other hooks, RTA-OS3.0 defines the macros `OS_ERRORHOOK` when the error hook is configured, allowing you to conditionally compile the hook.

```

#ifdef OS_ERRORHOOK
FUNC(void, OS_APPL_CODE) ErrorHook(StatusType status) {
    ...
}
#endif /*OS_ERRORHOOK*/

```

Code Example 13.2: Conditional compilation of the ErrorHook

The Error Hook is adequate for coarse debugging - it tells you that something has gone wrong. For example, if you get `E_OS_CALLEVEL`, then you know that you have made an API call from the wrong context *somewhere* in your code but you have no indication where it might be. You really need to know more about the error so that you can remove the bug. In this case, you need to

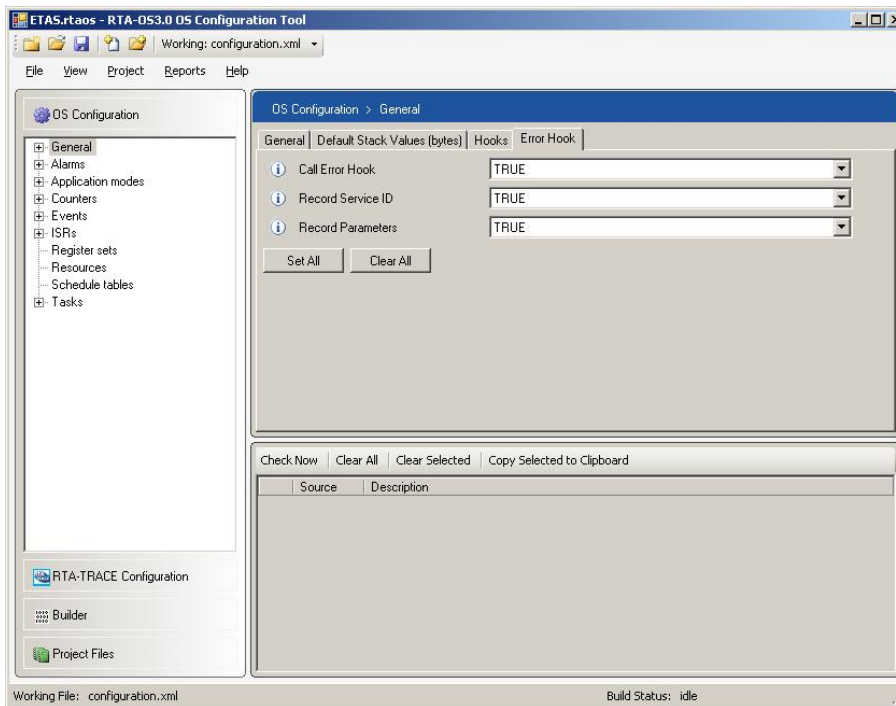


Figure 13.2: Configuring Advanced Error Logging

know which API call resulted in the error being generated. You might find in some cases that knowing which parameters were passed to an API call when it failed helps you to debug a problem. This information is available at run-time by configuring advanced error logging.

13.1.1 Configuring Advanced Error Logging

Three levels of detail are available:

1. Do not record the service details (default)
2. Record the API name only.
3. Record the API name and the associated parameters.

Figure 13.2 shows how the level of detail is defined in `rtaoscfg`.

If you choose not to record the service details, your application does not need to pay the additional overheads associated with collecting this information.

Using Advanced Error Logging

When error logging is enabled, RTA-OS3.0 provides a set of macros for accessing the name and the associated parameters of the API call that caused the error.

You can find out which API call caused the error using the `OSErrorGetServiceId()` macro. This macro returns an `OSServiceIdType` of the form `OSServiceId_<API name>`. If, for instance, an `ActivateTask()` call results in an error, `OSErrorGetServiceId` will return `OSServiceId_ActivateTask`.

The parameters to the API call are available using macros in the form shown in Code Example 13.3. A macro is defined for each parameter of each API call.

`OSError_<API Name>_<API Parameter Name>`

Code Example 13.3: Advanced Error Logging

Using the `ActivateTask()` example again, `OSError_ActivateTask_TaskId` will return the `TaskId` parameter passed to `ActivateTask()`. This additional error logging information can be usefully incorporated into the `ErrorHook()` code. This is shown in Code Example 13.4.

```
FUNC(void, OS_APPL_CODE) ErrorHook(StatusType status) {
    OSServiceIdType callee;
    switch (status) {
        case E_OS_ID:
            /* API call called with invalid handle. */
            callee = OSErrorGetServiceId();
            switch (callee) {
                case OSServiceId_ActivateTask:
                    /* Handle error. */
                    break;
                case OSServiceId_ChainTask:
                    /* Handle error. */
                    break;
                case OSServiceId_SetRelAlarm:
                    /* Handle error. */
                    break;
                default:
                    break;
            }
            break;
        case E_OS_LIMIT:
            /* Terminate. */
            ShutdownOS();
        default:
            break;
    }
}
```

```
}
```

Code Example 13.4: Additional Error Logging Information

The macros for obtaining the API name and the associated parameters should only be used from within the Error Hook. The values they represent do not persist outside the scope of the hook.



When you use extended error logging, the value returned by `OSErrorGetServiceId()` may be misleading. This generally happens when API calls have a side effect. For example if you activate a task from a schedule table expiry point and that task activation results in an error, then `OSErrorGetServiceId()` will return `OSServiceId_ActivateTask` even though the API call that you made was `Os_AdvanceCounter()`.

13.1.2 Working out which Task is Running

When debugging your RTA-OS3.0 applications, you will probably want to know which task or Category 2 ISR is responsible for raising the error. OSEK OS provides the `GetTaskID()` API call to tell you which task is running.

Code Example 13.5 shows you how to do this.

```
TaskType CurrentTaskID;
/* Pass a TaskRefType for the return value of GetTaskID() */
GetTaskID(&CurrentTaskID);
if (CurrentTaskID == Task1) {
    /* Code for task 1 */
} else {
    if (CurrentTaskID == Task2) {
        /* Code for task 2 */
    }
    ...
}
```

Code Example 13.5: Using `GetTaskID()`

13.1.3 Working out which ISR is Running

AUTOSAR OS extends the OSEK scheme to Category 2 ISRs with the `GetISRID()` API call.

Unlike `GetTaskID()`, `GetISRID()` returns the ID of the ISR through the return value of the function rather than as an out parameter to the function call. If you call `GetISRID()` and a task is executing, then the function returns `INVALID_ISR`.

The following code shows how to use `GetISRID()` together with `GetTaskID()`.

```

ISRType CurrentISRID
TaskType CurrentTaskID;
/* Is an ISR running? */
CurrentISRID = GetISRID();
if ( CurrentISRID != INVALID_ISR ) {
    if (CurrentISRID == ISR1) {
        /* Work out which ISR */
    }
} else {
    GetTaskID(&CurrentTaskID);
    if ( CurrentTaskID == Task1 ) {
        /* Work out which task */
    }
}
}

```

13.1.4 Generating a Skeleton ErrorHandler()

Writing error hooks that trap the types of errors that your configure may generate can be time consuming and error-prone. RTA-OS3.0 can help this activity by generating the framework for the ErrorHandler() that includes checking for all types of error, for all API calls.

The framework ErrorHandler() is generated using the following **rtaosgen** command line:

```
C:\>rtaosgen --samples:[ErrorHandler] MyConfig.xml
```

This generates an error hook in Samples\Hooks\ErrorHandler.c that you can use in your application. If the file is already present, then **rtaosgen** will generate a warning. If you want to overwrite an existing file, then you can use:

```
C:\>rtaosgen --samples:[ErrorHandler]overwrite MyConfig.xml
```

13.2 Inline Error Handling

An alternative to the ErrorHandler() is to check the API return codes inline with calling. This means that you can build some degree of run-time fault tolerance into your application.

This may be useful if you want to check for error conditions that can occur in the Standard status (such as ActivateTask() returning E_OS_LIMIT). Code Example 13.6 shows you how this can be done.

```

TASK(FaultTolerant){
    /* Do some work */

```

```

    if (ActivateTask(HelperTask) != E_OK) {
        /* Handle error during task activation. */
    }
    TerminateTask();
}

```


Code Example 13.6: Inline Error Checking

13.3 Conditional Inclusion of Error Checking Code

If you are adding code to check for runtime errors that only occur in extended status, then you do not want to go through your application by hand to remove this code at when you change to standard status.

RTA-OS3.0 provides two macros that allow you conditionally include/exclude code during development:

`OS_STANDARD_STATUS` is defined when standard status is configured
`OS_EXTENDED_STATUS` is defined when extended status is configured

 *The macros `OS_STANDARD_STATUS` and `OS_EXTENDED_STATUS` are provided by RTA-OS3.0 only and are not necessarily portable to other implementations.*

13.4 Summary

- AUTOSAR OS provides facilities for debugging through the Error Hook which provides a mechanism for trapping exceptional conditions at runtime. It can provide a resumption model of exception handling.
- Further information on the source of an error is available through macros accessible in the `ErrorHook()`.

14 Measuring and Monitoring Stack Usage

RTA-OS3.0 provides stack monitoring features that can be used during development to check whether you get any unexpected stack overruns.

When stack monitoring is configured, RTA-OS3.0 also provides features for measuring the stack usage of each task and ISRs at runtime. This can be used to identify which tasks consume what stack space and can help provide information that might be useful for optimizations (for example, identifying which tasks could share an internal resource to reduce the amount of stack required).

You may also want to collect accurate stack usage information for each task so that the stack allocations you specify are not pessimistic - i.e. you don't tell RTA-OS3.0 that tasks use more stack space than is really necessary.

14.1 Stack Monitoring

A common problem when building embedded systems is that of stack overrun, i.e. tasks and or ISRs consuming too much stack space at runtime.

AUTOSAR OS allows you to monitor the stack for overruns. When stack monitoring is enabled, RTA-OS3.0 checks on each context switch whether the stack has exceeded its pre-configured stack allocation value (see Section 4.6.2).



Category 1 ISRs in your system bypass RTA-OS3.0 completely and therefore consume stack without OS knowledge. If your Category 1 ISRs result in stack problems then these will not be detected by RTA-OS3.0.

RTA-OS3.0 calls ShutdownOS(E_OS_STACKFAULT) when a stack fault is identified. This is the behavior required for AUTOSAR but this is not very useful because it does not allow you to try and identify what has failed and by how much the stack has been overrun¹ In RTA-OS3.0 you can you can override this behavior and trap problems with the `Os_Cbk_StackOverrunHook()` instead. Section 14.2 provides more details.

Stack monitoring impacts both the memory footprint and the run-time performance of RTA-OS3.0 and is therefore disabled by default. Stack monitoring is enabled in **General → Stack Monitoring Enabled**. Figure 14.1 shows how to select your chosen option.

When you configure Stack Monitoring you need to define a stack allocation budget for each task and Category 2 ISR. This figure must include the stack required by your application and the stack required for the RTA-OS3.0 context. Section 14.3 explains how to use RTA-OS3.0's stack measurement fea-

¹In extreme cases, it may not be possible for you do anything, but one advantage of the single stack model used by RTA-OS3.0 is that you can add a system-wide stack safety-margin at link time and then use this 'spare' stack space for debugging if a stack fault occurs.

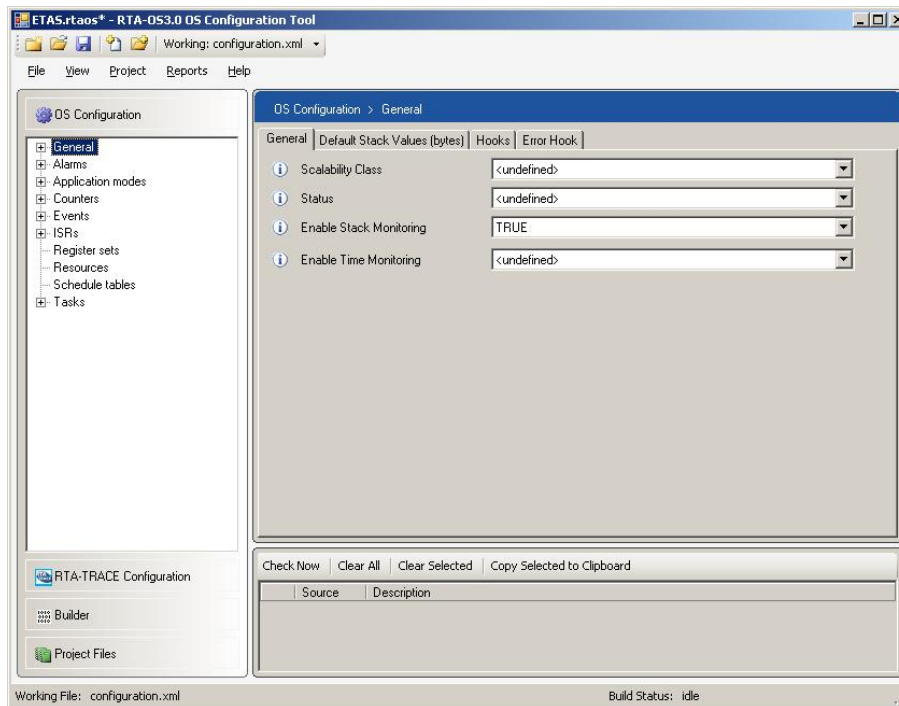


Figure 14.1: Enabling Stack Monitoring

tures to get this data.

RTA-OS3.0 provides 2 ways to define the stack allocation:

1. Task/ISR defaults
2. Per task/ISR configuration

If a per task/ISR value is configured for a task/ISR, then this overrides the default value.

14.1.1 Setting Defaults

Default settings set the stack allocation for all tasks, all Category 2 ISRs and all Category 1 ISRs. You can see how to do this in Figure 14.2. If no other stack allocation is specified elsewhere, then RTA-OS3.0 uses the default value.

14.1.2 Configuring Stack Allocation per Task/ISR

Each task and ISR can specify its own stack allocation as part of the task/ISR configuration. Figure 14.3 shows how this is configured for tasks, ISRs have a similar configuration element. Whenever you specify a stack allocation value for a task/ISR the value configured overrides any default value that you might have set.

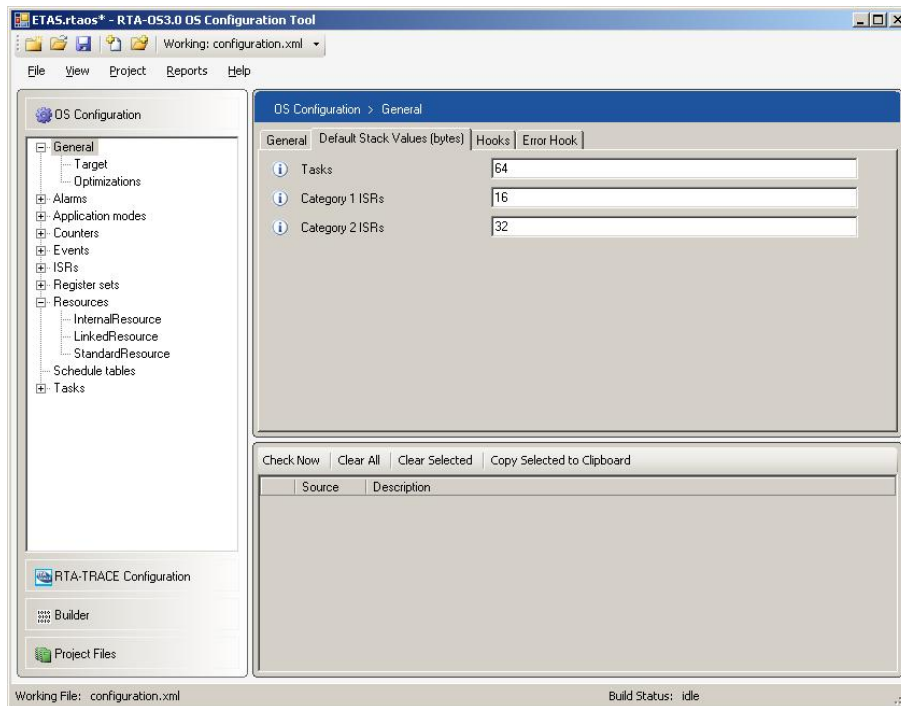


Figure 14.2: Setting default stack allocation

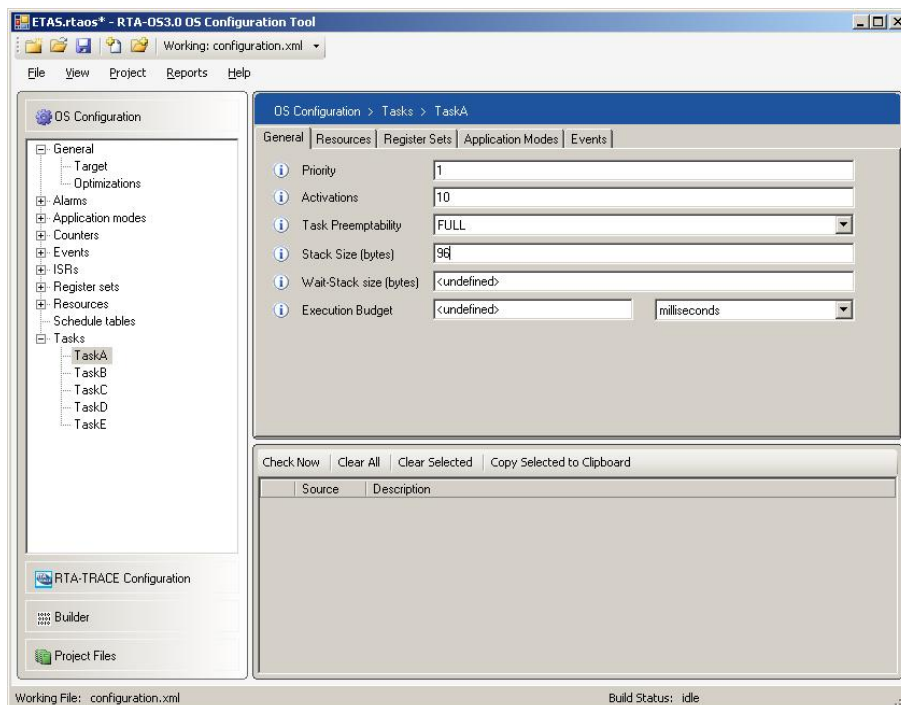


Figure 14.3: Setting Stack Allocation for Tasks

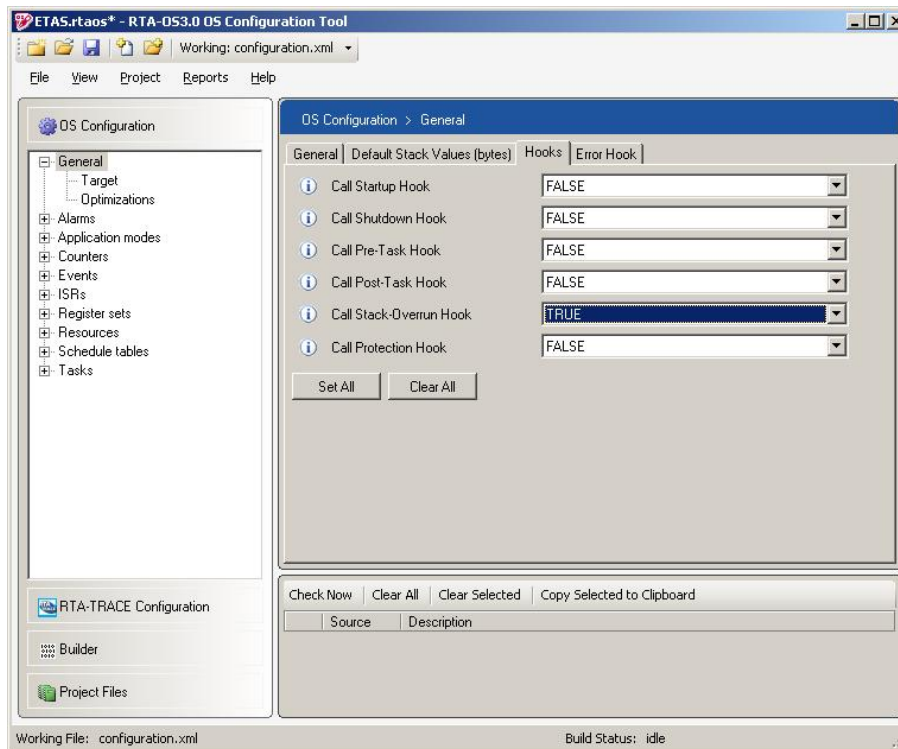


Figure 14.4: Configuring the stack overrun hook

14.2 Using the `Os_Cbk_StackOverrunHook()`

Recall from Section 4.6.5 that RTA-OS3.0 can be configured to call the `Os_Cbk_StackOverrunHook()` when problems with extended task management are detected at runtime. The same hook is used by RTA-OS3.0 for reporting stack overruns detected by stack monitoring.

If you configure RTA-OS3.0 to use the `Os_Cbk_StackOverrunHook()` as shown in Figure 14.4 then RTA-OS3.0 will call the hook when a problem is detected by stack monitoring.

ETAS *Calling `Os_Cbk_StackOverrunHook()` when a problem is detected by stack monitoring is an RTA-OS3.0 extension to AUTOSAR OS and is not portable to other implementations.*

The hook is passed a parameter indicating the number of bytes by which the stack has overrun and a reason for the problem. Stack monitoring adds another reason - `OS_BUDGET` - to those presented in Section 4.6.5. `OS_BUDGET` indicates that a task has exceeded its stack allocation.

`OS_BUDGET` is similar to `OS_ECC_START` - it identifies a situation where the stack has overrun. The difference between the two cases is that `OS_ECC_START` only occurs when an extended task is started (basic tasks that

exceed their configured stack allocation do not result in this error) whereas OS_BUDGET problems are detected on every context switch for every type of task and ISR.

As with the ErrorHook() you can make calls to GetTaskID() and GetISRID() to identify what was executing at the point the problem occurred. Code Example 14.1 shows an example Os_Cbk_StackOverrunHook().

```
FUNC(void, OS_APPL_CODE) Os_Cbk_StackOverrunHook(
    Os_StackSizeType Overrun, Os_StackOverrunType Reason) {
    ISRType CurrentISRID
    TaskType CurrentTaskID;

    /* Work out what has failed */
    CurrentISRID = GetISRID();
    if ( CurrentISRID != INVALID_ISR ) {
        /* An ISR has overrun */
        if (CurrentISRID == ISR1) {
            /* Work out which ISR */
        }
    } else {
        /* It must be a task that has overrun */
        GetTaskID(&CurrentTaskID);
        if ( CurrentTaskID == Task1 ) {
            /* Work out which task */
        }
    }

    /* Work out why */
    switch (Reason) {
        case OS_BUDGET:
            /* Problem: The task/ISR exceeded its stack
             allocation */
            /* Solution: Add Overrun to the stack allocation */
            break;
        case OS_ECC_START:
            /* Problem: Some lower priority task on the stack
             has used too much stack space */
            /* Solution: Enable stack monitoring to find out
             which task */
            break;
        case OS_ECC_WAIT:
            /* Problem: The extended task had consumed too
             much stack space then executing WaitEvent() */
            /* Solution: Add Overrun to the WaitEvent() stack
```

```

        allocation */
    break;
}
}

```

Code Example 14.1: The Stack Overrun Hook



When `Os_Cbk_StackOverrunHook()` is entered this indicates that your system is not behaving as expected. You should not return from the `Os_Cbk_StackOverrunHook()`. Entering the hook usually means that your stack is corrupt. If you do return from the hook then the behavior of your application is undefined.

14.3 Measuring Stack Usage

The figures that you supply for stack monitoring represent the worst-case stack used by each task and should be the sum of the space required by the task. This includes the context for RTA-OS3.0 and the space required for worst-case function call tree made by the task (where worst-case means the tree that results in the most stack space being used by the task).

ETAS *Stack measurement is a feature of RTA-OS3.0 and is not portable to other implementations of the OSEK or AUTOSAR OS standards.*

The `Os_GetStackUsage()` API call is used for stack measurement in RTA-OS3.0. On targets that have a single stack, `Os_GetStackUsage()` returns a scalar value indicating the number of bytes of stack space consumed by the calling Task/Category 2 ISR. If your target has multiple stacks, however, `Os_GetStackUsage()` returns a data structure containing the number of bytes used on each stack. The *RTA-OS3.0 Target/Compiler Port Guide* for your port will tell you how to extract stack space information from this data structure.

RTA-OS3.0 also automatically logs the worst case stack usage seen at runtime for each task and Category 2 ISR. The API calls `Os_GetTaskMaxStackUsage()` and `Os_GetISRMaxStackUsage()` are provided to allow you to find out what has been logged. Logging of worst-case (largest) values is only performed on a context switch or in a call to `Os_GetStackUsage()`.



If your task or Category 2 ISR has not terminated (or entered the waiting state) at least once or called `GetStackUsage()` at least once then RTA-OS3.0 will not yet have logged a value and the `Os_Get[Task|ISR]MaxStackUsage()` calls will return zero.

The values returned are measured from the initial value of the stack pointer at the point RTA-OS3.0 starts the task/ISR. This means that measurements include the stack context required by RTA-OS3.0. However, the stack values returned do not include the stack space required for the calls themselves.

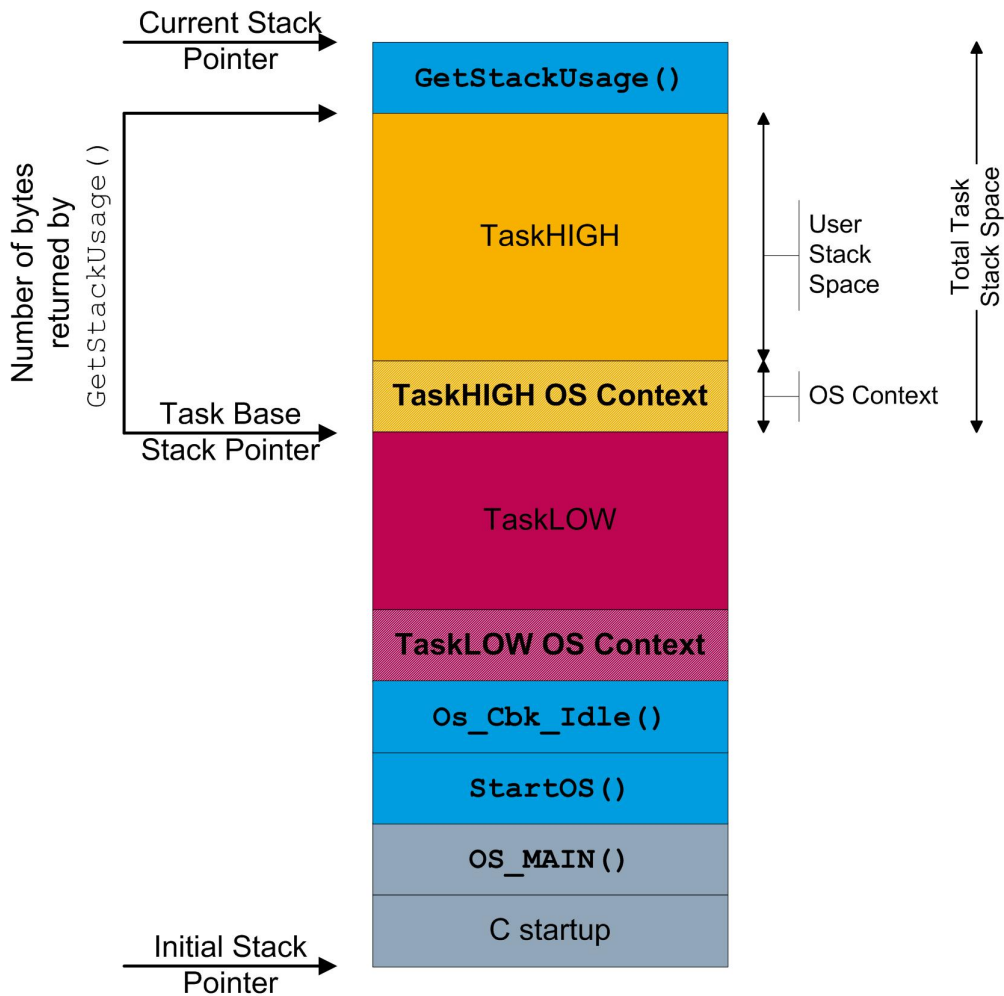


Figure 14.5: Stack Diagram

Figure 14.5 shows the size returned by `Os_GetStackUsage()` when it is called from task `TaskHIGH`.

14.3.1 Marking the Worst Case for Function Calls

To measure the worst-case stack usage for each task or ISR, you need to place a call `Os_GetStackUsage()` call at each leaf of your function call hierarchy.

If you have leaves that are library functions then you will need to make a `Os_GetStackUsage()` call in the parent function and determine the worst-case stack space of the library call. The worst-case stack space requirement for the RTA-OS3.0 API is provided in the *RTA-OS3.0 Target/Compiler Port Guide* for your port. If you make calls to other libraries at the leaves of your call hierarchy, you must contact the vendor to obtain the worst-case stack requirements for the library calls you make.

Code Example 14.2 shows a task that makes a number of function calls. It shows the placement of `Os_GetStackUsage()` calls required to measure stack usage.

```
#include <Os.h>

Os_StackSizeType Measurement1;
Os_StackSizeType Measurement2;
Os_StackSizeType Measurement3;

void Function1(void) {
    ...
    Measurement1 = Os_GetStackUsage();
    ActivateTask(Higher);
    ...
}

void Function2(void) {
    ...
    Function3();
    Measurement2 = Os_GetStackUsage();
    ...
}

void Function3(void) {
    ...
    Measurement3 = Os_GetStackUsage();
    ...
}

TASK(Low) {
    Function1();
    ...
    Function2();
    TerminateTask();
}
```

Code Example 14.2: Measuring Stack Usage

The worst-case stack usage (WCSU) for Code Example 14.2 will be the maximum value of `Measurement1`, `Measurement2` and `Measurement3`. Figure 14.6 show Code Example 14.2 executing. In this case, the WCSU is when task Low calls `Function1()`.

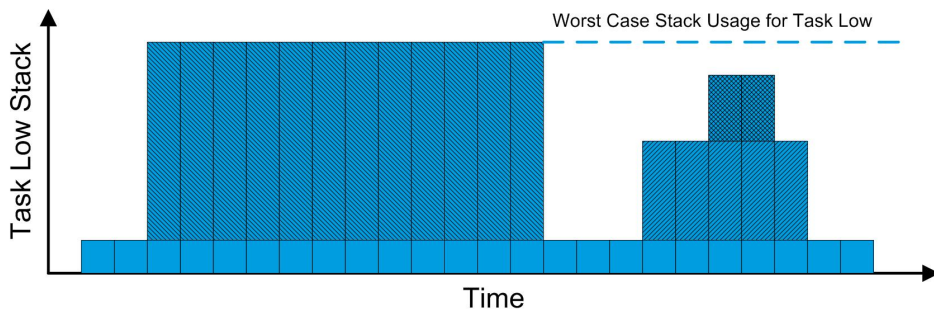
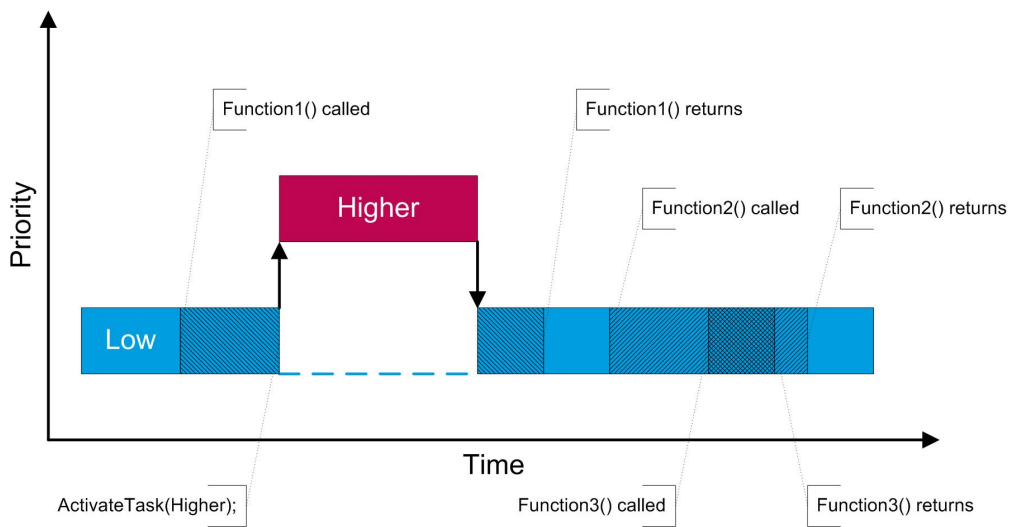


Figure 14.6: Measuring the worst-case stack for the program call tree

14.4 Summary

- RTA-OS3.0 provides in-kernel features that allow you to measure and monitor stack usage at runtime.
- Each task and ISR must specify a stack allocation in bytes for each stack used.
- Arbitrary measurements of the current stack pointer value can be made using the `GetStackOffset()` API call.
- Stack faults can be handled by calling `ShutdownOS()` (as specified by the AUTOSAR OS standard) or can alternatively be re-directed to RTA-OS3.0's `Os_Cbk_StackOverrunHook()` for diagnosis.

15 Measuring and Monitoring Execution Time

ETAS *All timing monitoring and measuring facilities provided by RTA-OS3.0 are not part of the OSEK or AUTOSAR OS standards and are therefore not portable to other implementations.*

RTA-OS3.0 provides facilities for measuring the execution times of user code at the kernel level.

15.1 Enabling Timing Measurement

Time monitoring can be used in both standard and extended builds and is enabled by setting **General → Time Monitoring Enabled** to true. The feature needs access to a free running hardware timer, ideally one that runs at the same speed as your CPU clock because this will allow RTA-OS3.0 to carry out cycle-accurate measurements.

Before you can use time monitoring you need to tell RTA-OS3.0 some details about the timing of the target hardware. There are two values to provide:

1. the instruction cycle rate

This is the rate at which instructions are executed on your target hardware (sometimes called the clock speed).

2. the stopwatch speed

This is the rate at which the stopwatch timer runs. Ideally, this will be the same speed as the instruction cycle rate. However, it might be slower than the CPU instruction rate because your timer module might use some kind of pre-scaler.

RTA-OS3.0 generates a set of macros that encapsulate this information to allow you to scale timing measurements:

Macro	Description
OSCYCLEDURATION	The duration of a CPU instruction in nanoseconds.
OSCYCLESERSECOND	The number of CPU instructions in a second.
OSSWICKDURATION	The duration of a stopwatch tick in nanoseconds.
OSSWICKSERSECOND	The number of stopwatch instructions in a second.

15.1.1 Providing a Stopwatch

The free running timer you provide is called the “stopwatch” and is used by RTA-OS3.0 to measure execution times. RTA-OS3.0 gets access to the stopwatch using a callback function called `Os_Cbk_GetStopwatch()`.

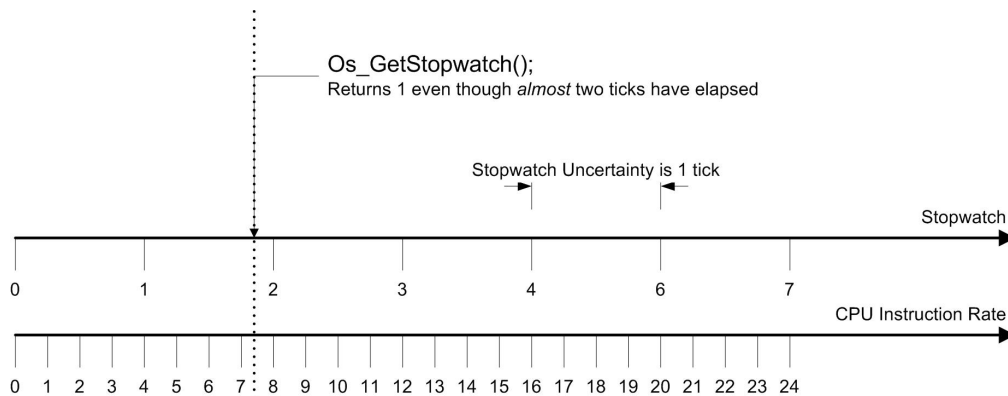


Figure 15.1: Uncertainty in stopwatch measurements



An implementation of `Os_Cbk_GetStopwatch()` must be provided if you are using RTA-OS3.0's time monitoring functionality. Your program will not link correctly if you do not provide this function.

Any code that your application uses to obtain execution times should be conditionally compiled. RTA-OS3.0 provides the macro `OS_TIME_MONITORING`, which allows you to do this. Code Example 15.3 shows an example of conditional compilation when getting the time that a resource is held.

Code Example 15.1 shows a typical example.

```
#ifndef OS_TIME_MONITORING
FUNC(Os_StopwatchTickType, OS_APPL_CODE) Os_Cbk_GetStopwatch(
    void) {
    return (Os_StopwatchTickType)TIMER_CHANNEL_0;
}
#endif /* OS_TIME_MONITORING */
```

Code Example 15.1: Providing a stopwatch

The stopwatch returns ticks and any values reported by RTA-OS3.0 are in terms of ticks on the stopwatch time base. You can use the macros provided by RTA-OS3.0 to convert stopwatch measurements into 'clock time' units like milliseconds, microseconds etc.

Uncertainty in Stopwatch Measurements

If the stopwatch runs slower than the CPU clock, then when RTA-OS3.0 reads the stopwatch, there is a possibility that the time is less than the real amount of time that has elapsed. This occurs because of the difference in resolution of the CPU clock and the stopwatch. Figure 15.1 shows the basic issue - you might read the lower resolution stopwatch just before it will be incremented by the CPU clock.

This difference is called the uncertainty and you will need to compensate for this in any calculations you do that use time measurement.

This does not occur for stopwatches that run at the same rate as the CPU clock because you are already using the maximum possible resolution of time. The stopwatch uncertainty is equal to zero if the instruction cycle rate and the stopwatch speed are equal. In most other cases the uncertainty is one (but see Section 15.1.2).

15.1.2 Scaling the Stopwatch

In most cases your, `Os_Cbk_GetStopwatch()` will return a value read directly from a hardware timer and you will convert timing measurements into 'real' time after measurement.

However, you may prefer to scale the stopwatch directly in the `Os_Cbk_GetStopwatch()` callback so that all times reported by RTA-OS3.0 are already in the units you require. For example, Code Example 15.2 shows how to scale the stopwatch from Code Example 15.1 so that the stopwatch returns a value in nanoseconds.

```
FUNC(Os_StopwatchTickType, OS_APPL_CODE) Os_Cbk_GetStopwatch(  
    void) {  
    return (Os_StopwatchTickType)(TIMER_CHANNEL_0 *  
        OSSWTICKDURATION);  
}
```

Code Example 15.2: Providing a stopwatch

Scaling the stopwatch also has an impact on the stopwatch uncertainty as shown in Figure 15.2.

An appropriate modification to the stopwatch uncertainty calculations you make is to multiply the uncertainty by the scaling factor.

15.2 Automatic Measurement of Task and ISR Execution Times

When your application uses time monitoring, RTA-OS3.0 measures the execution times of each task and Category 2 ISR in your application.

RTA-OS3.0 maintains a log of the longest observed execution time over all executions for each task and Category 2 ISR. The execution time for tasks is measured as follows:

Basic Tasks are measured from their first instruction to the completion of the `TerminateTask()` API call.

Extended Tasks are measured from their first instruction to the first

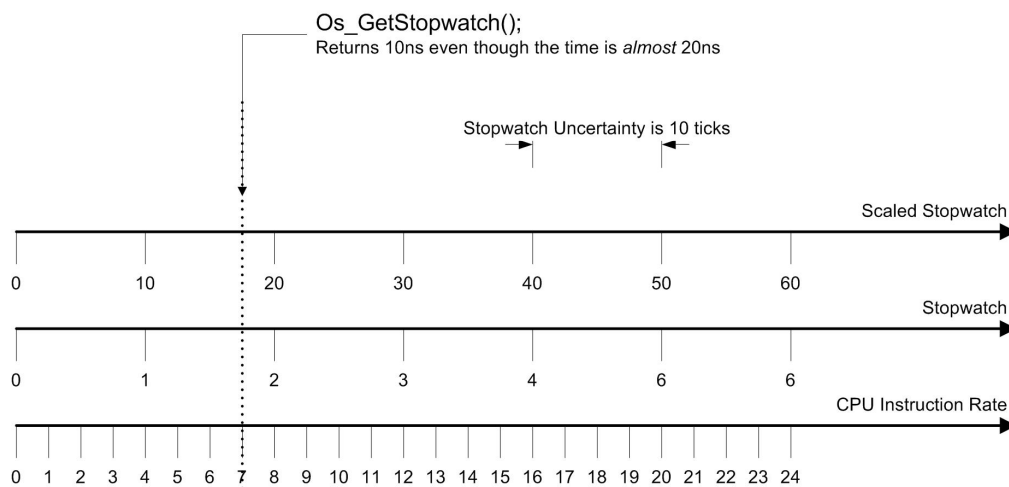


Figure 15.2: Uncertainty in scaled stopwatch measurements

`WaitEvent()`, between adjacent `WaitEvent()` calls and from `WaitEvent()` to the `TerminateTask()` API call.

Pre- and post-task hooks, if configured, are not included in the execution time measurement.

Execution times are measured using the stopwatch provided by `Os_Cbk_GetStopwatch()`. RTA-OS3.0 automatically compensates for preemption during measurement. When a task is preempted then the measurement for the preempted task stops and measurement for the preempting task starts as shown in Figure 15.3.



Measurements are taken on a context switch (or, in the case of extended tasks, the possibility for a context switch). This means that a switch must occur for a time to be recorded. Therefore, a basic task must terminate at least once for a timing measurement to be made and an extended task must either terminate or make a `WaitEvent()` call.

The largest observed execution time for each task/ISR can be read using `Os_GetLargest[Task|ISR]ExecutionTime()` API call. The call returns zero if the task/ISR has not yet completed an execution.

The best place to record task and ISR execution times is in `Os_Cbk_Idle()` since, if the code here executes, you can be guaranteed that there are not tasks or ISRs that are ready to run. Code Example 15.3 shows a typical example.

```
FUNC(boolean, OS_APPL_CODE) Os_Cbk_Idle() {
    #if defined(OS_TIME_MONITORING)
        Os_StopwatchTickType TaskTime;
```

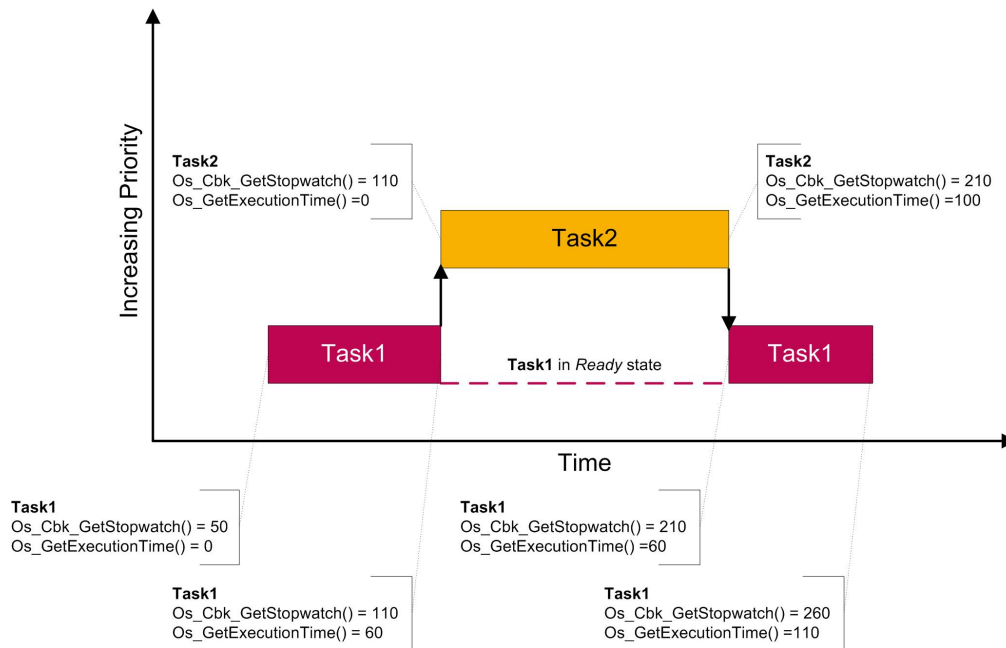


Figure 15.3: Compensating for preemption in timing measurements

```

Os_StopwatchTickType ISRTIME;
GetTaskMaxExecutionTime(MyTask,&TaskTime);
GetISRMaxExecutionTime(MyISR,&ISRTIME);
#endif
return TRUE;
}

```

Code Example 15.3: Reading the longest observed execution times

You can reset a largest time using the `Os_ResetLargest[Task|ISR]ExecutionTime()` API call.

15.3 Manual Time Measurement

RTA-OS3.0's time monitoring provides a API called `Os_GetExecutionTime()` that can be used to get the current stopwatch value. By placing this call before and after any section of code, you can measure the execution time of any fragment of your program. For example:

- you might want to profile the execution of some 3rd party library code
- you may want to debug exactly where time is being consumed by your own applications
- you might want to measure the blocking due to resource locking or the disabling of interrupts

Code Example 15.4 shows how you can measure blocking times. The same principle applies to any code section that you need to measure.

```
TASK(Task1) {
    Os_StopwatchTickType start,finish,correction;
    ...
    #if defined(OS_TIME_MONITORING)
        /* Get time for Os_GetExecutionTime() call itself. */
        start = Os_GetExecutionTime();
        finish = Os_GetExecutionTime();
        correction = finish - start -
            Os_Cbk_GetStopwatchUncertainty();
        /* Measure resource lock time. */
        start = Os_GetExecutionTime();
    #endif
    /* The section of code to measure */
    GetResource(Resource1);
    /* Critical section. */
    ReleaseResource(Resource1);
    #if defined(OS_TIME_MONITORING)
        finish = Os_GetExecutionTime();
        /* Calculate amount of time used. */
        used = finish - start - correction +
            Os_Cbk_GetStopwatchUncertainty();
    #endif
}
```

Code Example 15.4: Measuring Blocking Times

15.4 Imprecise Computation

Because the overheads imposed by time monitoring are small, it can be used for production code. You can exploit this fact to perform imprecise computation.

Imprecise computation is useful in applications that interactively converge on a result. For example, you might use Newton-Raphson to converge on a value.

If a task has not traveled down the worst-case path, then it will not have run in the worst-case execution time. If this is the case, any 'spare' CPU cycles available to the task can be used to refine a result. This technique is illustrated in Code Example 15.5.

```
TASK(NewtonRaphson) {
    TickType Budget = CONFIGURED_EXECUTION_BUDGET;
    TickType LoopTime = TIME_FOR_ONE_ITERATION;
```

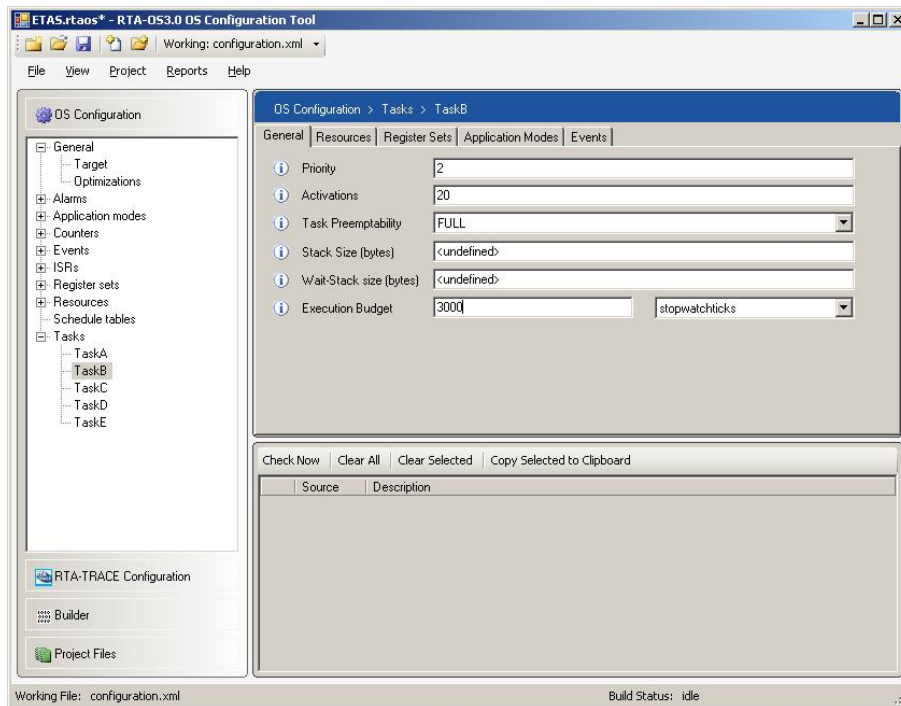


Figure 15.4: Specifying the Execution Time Budgets

```

...
Result = ...;
while ((Budget - Os_GetExecutionTime()) > LoopTime) {
    /* Perform iterative refinement of output. */
    Result = Result - (Function(Result)/Derivative(Function
        , Result));
}
...
}

```

Code Example 15.5: Imprecise Computation

15.5 Monitoring Execution Times against Budgets

Time monitoring also allows you to set budgets for execution times and let RTA-OS3.0 check for violations at runtime. The execution time budgets for each task and Category 2 ISR can be set in your application. These values are optional and do not have to be supplied. Configuration of an execution budget is shown in Figure 15.4.

The type of the budget value can be set as 'clock time' or in terms of stopwatch ticks or CPU cycles. RTA-OS3.0 uses the target timing characteristics to perform any necessary conversions. Figure 15.5 shows how these values

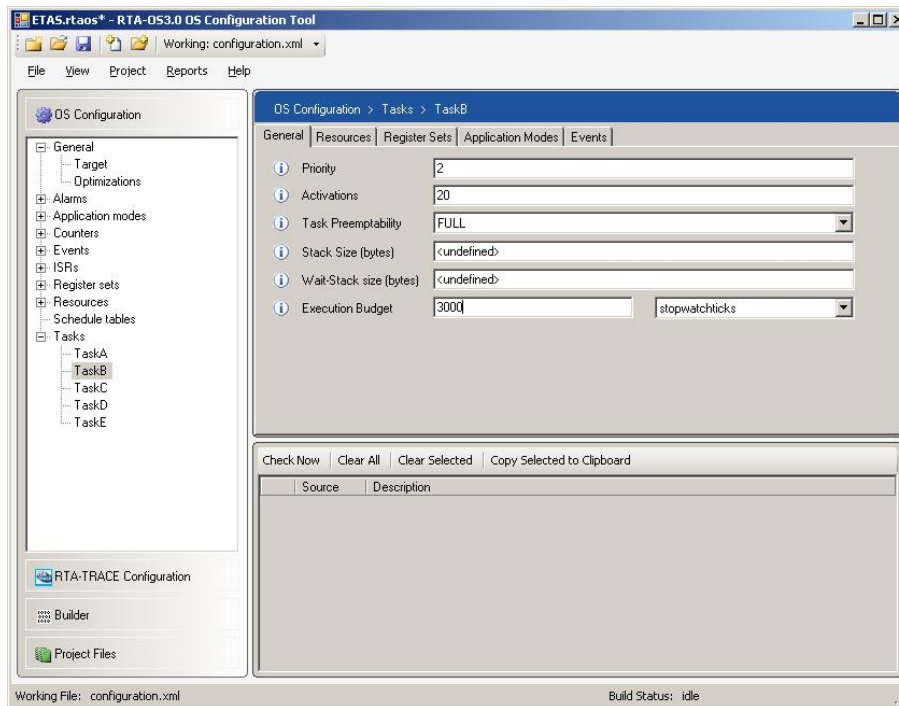



Figure 15.5: Specifying the Instruction Rate and Stopwatch Speed

are set.

When time monitoring is enabled, RTA-OS3.0 will check to see whether tasks or Category 2 ISRs consume more time than is specified in the budget. If the budget is exceeded, then RTA-OS3.0 will call the `Os_Cbk_TimeOverrunHook()` when the task terminates (or, in the case of an extended task, when it calls `WaitEvent()`). This allows you to log the budget overrun. As budgets are checked on a context switch there is the potential for a task or Category 2 ISR to overrun by a large margin before this is actually detected. Figure 15.6 shows what happens when a task overruns.

 *The `Os_Cbk_TimeOverrunHook()` is mandatory if time monitoring is configured in RTA-OS3.0. Your program will not link correctly if you do not provide this function.*

The prototype for `Os_Cbk_TimeOverrunHook()` is shown in Code Example 15.6.

```
#ifndef OS_TIME_MONITORING
FUNC(void, OS_APPL_CODE) Os_Cbk_TimeOverrunHook(void) {
    /* Log budget overruns. */
}
#endif
```

Code Example 15.6: The `Os_Cbk_TimeOverrunHook` Prototype

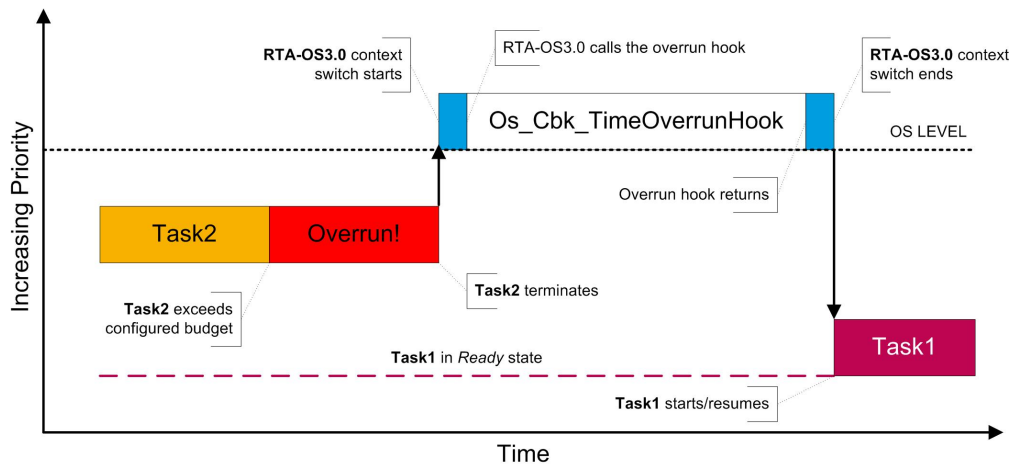


Figure 15.6: Call of the `Os_Cbk_TimeOverrunHook()`

You should be aware that, for extended tasks, the execution time is reset to zero at the start of the task and when resuming from `WaitEvent()`. Normally the budget is used to check the execution time between consecutive `WaitEvent()` calls.

You should also be aware that the execution time is only sampled by RTA-OS3.0 when a task is preempted by another task or ISR or when the task/ISR terminates.



In some unusual circumstances, it is possible for a budget overrun to be missed. This could happen when the interval between preemptions approaches the maximum interval that can be measured by a `Os_StopwatchTickType`. The range of a `Os_StopwatchTickType` is target dependent, but is normally 2^{16} or 2^{32} .

15.6 Summary

- RTA-OS3.0 provide in-kernel features that allow you to measure the execution time of tasks and ISRs at runtime.
- You need to provide access to a free-running timer for RTA-OS3.0 to use as a stopwatch. .
- The worst-case execution time of tasks and ISRs is logged automatically.
- Arbitrary measurements can be made using the `Os_GetExecutionTime()` API.
- If an execution budget is specified for a task or ISR, then RTA-OS3.0 will automatically monitor the task or ISR and generate an error at context switch time if the budget is exceeded.

16 Using an ORTI-Compatible Debugger

ORTI is an acronym that stands for ‘OSEK Run Time Interface’. ORTI was designed to provide a standardized and extensible way for an OSEK operating system to provide internal details of its behavior to a debugger. The design of the ORTI is sufficiently general that it can support operating systems other than OSEK, and in RTA-OS3.0 ORTI support is provided for OSEK OS and AUTOSAR OS features.

ORTI provides a small language that captures two things:

1. how to find objects and variables within the running operating system; and
2. how to interpret or display their values.

This means that ORTI is like a symbol table - telling the debugger which things in memory mean which objects in the OS.

RTA-OS3.0 can generate an ORTI file for your debugger. This means that, during execution of the application, you can observe values of key operating system variables for applications based on RTA-OS3.0.

In this chapter you will learn how to configure the generation of ORTI information for your debugger. A list of compatible ORTI debuggers is provided in the *RTA-OS3.0 Target/Compiler Port Guide* for your port. You will also see the information that RTA-OS3.0 provides about its applications.

For details of how to view ORTI information at runtime you should consult your debugger documentation.

16.1 Development Process

The following steps describe how to use ORTI with your program.

Step 1 Use `rtaoscfg` to enable ORTI debugger support. As ORTI is target-specific, the configuration is done in the “Target Specific” settings. Figure 16.1 shows how this is done.

Step 2 Build the RTA-OS3.0 library. The kernel is instrumented with ORTI support when generated. The ORTI file that you need for your debugger is generated as a file called `<projectname>.orti`.

Step 3 Build the application.

Step 4 Start the debugger, load the application and then load the ORTI file. For details of how to do this, please consult the documentation for your debugger.

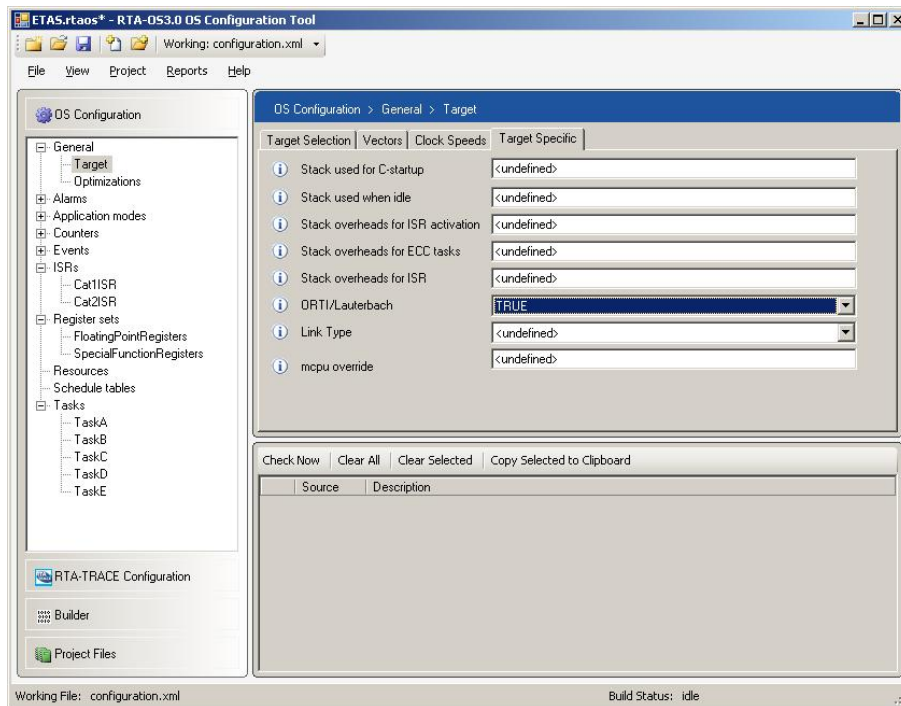


Figure 16.1: Enabling ORTI Support

The debugger will then display the information shown by the ORTI file. The format of this information depends upon the debugger.

16.2 Intrusiveness

ORTI relies upon reading values from the memory of the running application. This means that the presence of ORTI can affect the operation of the application. It is useful to know the extent to which this might happen. ORTI can acquire data via four routes:

1. Constant values within the ORTI file. These are used for quantities that will not change during the execution of an application. These have no impact on the running application.
2. Values generated as part of the normal operation of the application. Data is read from variables that would be present even if ORTI were not. These have no additional impact on the application.
3. Values generated specifically for ORTI support. Such variables constitute a very small extra overhead in the application.
4. Constants generated only for ORTI support. This data amounts to a small overhead in the application. These constants are only generated for debuggers that cannot obtain the information by other means. They

are only present when you specify that you are using a debugger, so you may wish to disable debugger support in your final production release.

16.3 Validity

Many of the values reported by ORTI are simply those contained in the application's memory. Using ORTI to inspect the system before it has been fully initialized will lead to misleading results. RTA-OS3.0 is fully initialized when, as a result of calling `StartOS()`, `Os_Cbk_Idle()` the first task or Category 2 ISR is entered.

Care should be taken where a variable may be cached in a register for a significant portion of its lifetime, especially in the case of register-rich processors. ORTI can only look at the data stored in the variable's memory location. This could be out of date if the register-based copy has been updated recently.

16.4 Interactions

The ORTI output will be correct when the program is stopped at a breakpoint that is:

- In code executed by a task or Category 2 ISR that is outside of any AUTOSAR OS API call.

The ORTI output may be misleading if the application is stopped at a breakpoint that is:

- Within an AUTOSAR OS API call.
- In code executed by a Category 1 interrupt handler.

The output may be misleading because the OSEK data used by ORTI could be in a partially updated state. Normally it is possible to tell if the program is part way through an AUTOSAR OS call by the debugger reporting the name of the function in which the processor stopped.

On a platform with more than two interrupt priority levels, however, a Category 1 interrupt can occur part way through an OSEK call. If the program is stopped at a breakpoint in a Category 1 interrupt handler, it is necessary to use the debugger's stack trace facility to determine the name of the function that was interrupted. The ORTI output can be relied upon, provided that the Category 1 interrupt did not occur within an AUTOSAR OS API call.

16.5 Summary

- RTA-OS3.0 can optionally generate ORTI information for use with a third-party ORTI compatible debugger.

228 Using an ORTI-Compatible Debugger

- ORTI support is port-specific functionality. Additional details on the exact nature of ORTI support for your port can be found in the relevant *RTA-OS3.0 Target/Compiler Port Guide*.

17 RTA-TRACE Integration

RTA-TRACE is a software logic analyzer for embedded systems that provides a set of services to assist in debugging and testing a system including the ability to see exactly what is happening in a system at runtime with a production build of the application software.



RTA-TRACE is a separate product to RTA-OS3.0 and is not supplied with your RTA-OS3.0 installation. For further details about how to obtain RTA-TRACE please contact your local ETAS Sales Office (see Section 18.2).

RTA-TRACE logs trace records in an on-target trace buffer. Each trace record maintains information about what happened, and when, to which object. RTA-TRACE relies on an instrumented OS to gather tracing data. Instrumentation is possible by hand, however, **rtaosgen** can automatically add RTA-TRACE instrumentation to the generated OS kernel. This chapter explains how to use the RTA-TRACE configuration editor provided with the **rtaoscfg** tool to enable this functionality. Section 17.1 describes the basic configuration. RTA-TRACE also provides extensive control on which data is traced and allows you to configure user-defined trace information.

Further details about RTA-TRACE are provided in the RTA-TRACE user documentation. However, you should note the following:

- the information presented in Sections 17.1 and 17.3 augments the information provided in your *RTA-TRACE Configuration Guide* for configuration with RTA-OS3.0's **rtaoscfg** tool.
- RTA-OS3.0 makes some changes to how the RTA-TRACE ECU link works. The information presented in Section 17.4 augments the information provided in your *RTA-TRACE Configuration Guide*.
- for RTA-OS3.0, all RTA-TRACE API calls, callbacks, macros and types and adopt the AUTOSAR naming convention. The changes are as follows:

API Feature	RTA-TRACE	RTA-TRACE with RTA-OS3.0
Call	<name>	Os_<name>
Callback	osTrace<name>	Os_Cbk_Trace<name>
Type	osTrace<name>	Os_Trace<name>
Macro	OSTRACE_ENABLED	OS_TRACE

A complete reference for the modified RTA-TRACE API is provided in the *RTA-OS3.0 Reference Guide*.

17.1 Basic Configuration

The basic configuration parameters RTA-TRACE are shown in Figure 17.1.

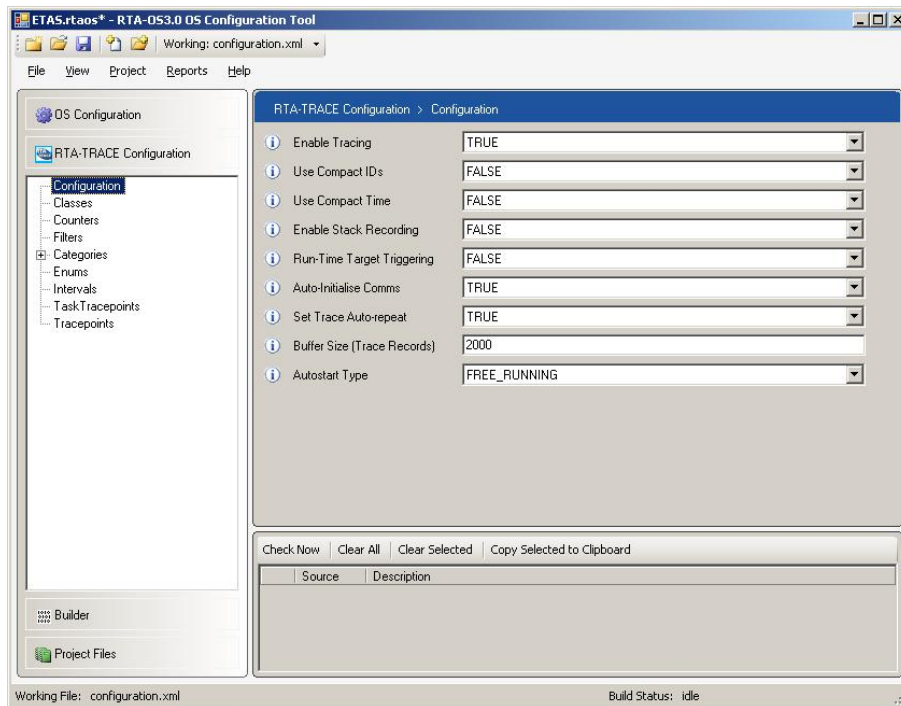


Figure 17.1: Configuring RTA-TRACE

Enable selects whether RTA-TRACE instrumented is added to RTA-OS3.0 or not. If this is not set, then no instrumentation is added to RTA-OS3.0.

Use Compact IDs selects compact trace format which reduces the size of a trace record stored in the trace buffer.

Identifier	Regular IDs	Compact IDs
Task Tracepoint	12-bit (max 4096 IDs)	4-bit (max 16 IDs)
Tracepoint	12-bit (max 4096 IDs)	8-bit (max 256 IDs)
Interval	12-bit (max 4096 IDs)	8-bit (max 256 IDs)
OS Objects	16-bit (max 65536 IDs)	8-bit (max 256 IDs)

For most common applications it is safe to use compact identifiers.

Use Compact Time selects compact (16-bit) or extended (32-bit) time format. This option may not be available for every RTA-OS3.0 port.

Enable Stack Recording selects whether or not to record stack usage or not. When enabled, this logs two trace records for each trace event: one for the event itself and another for the stack size. Enabling this option therefore doubles the amount of trace data that is recorded.

Run-Time Target Triggering selects whether or not runtime target triggering is available.

Auto-Initialize Comms selects whether the RTA-TRACE communications link is initialized by automatically during StartOS(). Setting this configuration item to TRUE means then RTA-OS3.0 will automatically call Os_TraceCommInit() to initialize the communications link. If set too FALSE then you will need to call Os_TraceCommInit() to initialize the RTA-TRACE communication elsewhere in your application. “Auto-Initialize Comms” should be set to FALSE when a debugger link is used to upload trace data from the target to the host PC.

Buffer Size sets the size of the trace buffer reserved on the target for the tracing information. The size is specified in trace records *not* bytes. A trace buffer of 2000 records is recommended as a default setting.

Auto-start Type selects whether tracing is started automatically during StartOS() and which tracing mode is used (Bursting, Free-Running or Triggering). See Section 17.2.

17.2 Controlling RTA-TRACE

RTA-TRACE can be used in three different modes:

Bursting mode handles the buffer as a *linear* buffer and logs trace data until the buffer is full. When the buffer is full tracing stops and the buffer is made available for uploading to the RTA-TRACE host PC. This useful for capturing a ‘one-shot’ log of data. RTA-TRACE is started in this mode using Os_StartBurstingTrace(). Tracing can be automatically re-started after the upload if the call Os_SetTraceRepeat(TRUE) has been made.

Free-running mode handles the buffer as a *circular* buffer and makes data available for uploading to the RTA-TRACE host PC as soon as it has been logged. If data can be uploaded sufficiently often that the buffer is never full, then free-running trace provides a continuous stream of trace data. If the buffer becomes full then tracing is suspended until space becomes available again. RTA-TRACE is started in this mode using Os_StartFreerunningTrace().

Triggering mode handles the buffer as a *circular* buffer and logs continuously. If the buffer overflows then old data is overwritten by new data. Data is not made available for upload until one (or more) user-specified triggers occur. When a trigger occurs the data buffer a user-specified pre-trigger number of trace records is locked and tracing continues until a user-specified post-trigger number of trace records has been logged. When the post-trigger number of records has been logged then tracing stops and the buffer is made available for uploading to the RTA-TRACE host PC. RTA-TRACE is started in this mode using

`Os_StartTriggeringTrace()`. The pre and post-trigger windows are set using `Os_SetTriggerWindow(pre,post)`. Tracing can be automatically re-started after the upload if the call `Os_SetTraceRepeat(TRUE)` has been made.

If you have configured RTA-TRACE to auto-start then RTA-OS3.0 will make the correct RTA-TRACE `Os_Start...()` API automatically during `StartOS()`. If RTA-TRACE is running and you make a `Os_Start...()` then the trace buffer is clear and RTA-TRACE re-starts in the chosen mode.

If you need to stop RTA-TRACE then you need to make the `Os_StopTrace()` API call.

RTA-OS3.0 defines the macro `OS_TRACE` when RTA-TRACE is enabled. You can use this macro to conditionally compile RTA-TRACE code into your application as shown in Code Example 17.1.

```
FUNC(void, OS_APPL_CODE) StartupHook(void)
{
    ...
    #ifdef OS_TRACE
        SetTraceRepeat(TRUE);
        StartBurstingTrace();
    #endif
    ...
}
```

Code Example 17.1: Using the `OS_TRACE` macro

17.2.1 Controlling with Objects are Traced

RTA-TRACE defaults to tracing every type of OS object. Sometimes this might not be appropriate to your application - you may be interested in just a set of tasks or you may need to reduce the amount of data being logged because your data-link has low bandwidth.

RTA-TRACE allows you control over data collection using *classes* and *filters*.

Classes

RTA-TRACE groups trace objects into classes. By default, all classes are traced at runtime. However, to minimize the amount of trace data that is gathered (and therefore minimize the amount of time spent uploading data) you might choose to switch off some classes of tracing.

Each class can be configured as:

Always the class is always traced.

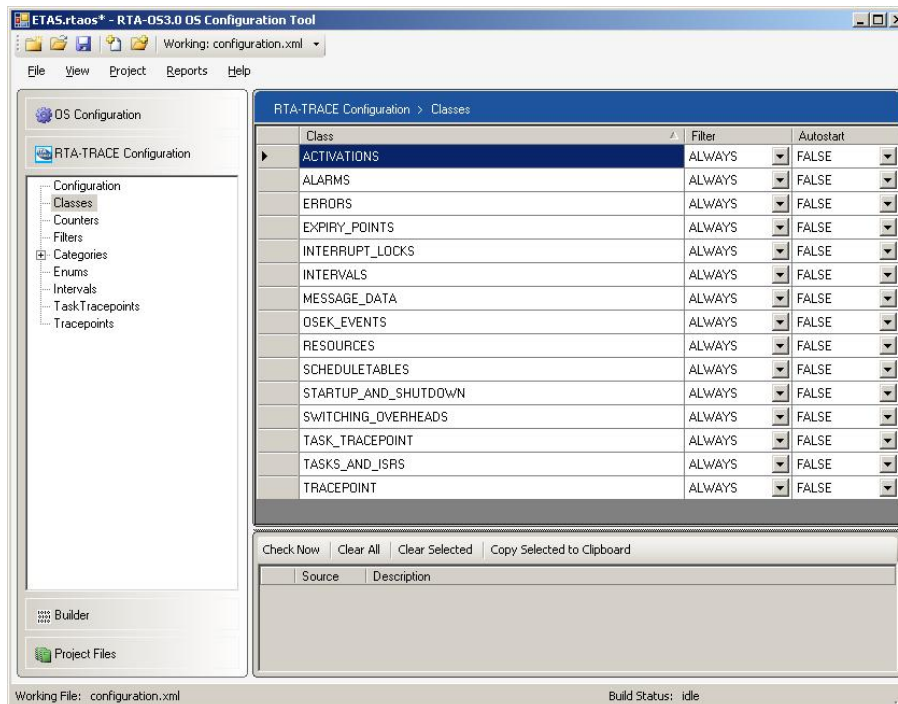


Figure 17.2: Configuring RTA-TRACE classes

Never the class is never traced.

Runtime the tracing of the class can be enabled/disabled at runtime using the API calls `Os_EnableTraceClasses()` and `Os_DisableTraceClasses()`.

Figure 17.2 shows how trace classes can be configured.

Any trace class configured as runtime is disabled when RTA-TRACE starts. However, the runtime classes can be configured to be auto-started when RTA-TRACE starts - TRUE enables runtime tracing of the class and FALSE disables runtime training.

Filters

Filters allow individual tasks and ISRs to be excluded from tracing. As with trace classes, all tasks and ISRs are traced by default, but can be configured as:

Always the task/ISR is always traced.

Never the task/ISR is never traced.

Runtime the tracing of the task/ISR is controlled by the runtime state of the `OS_TRACE_TASKS_AND_ISR_CLASS`.

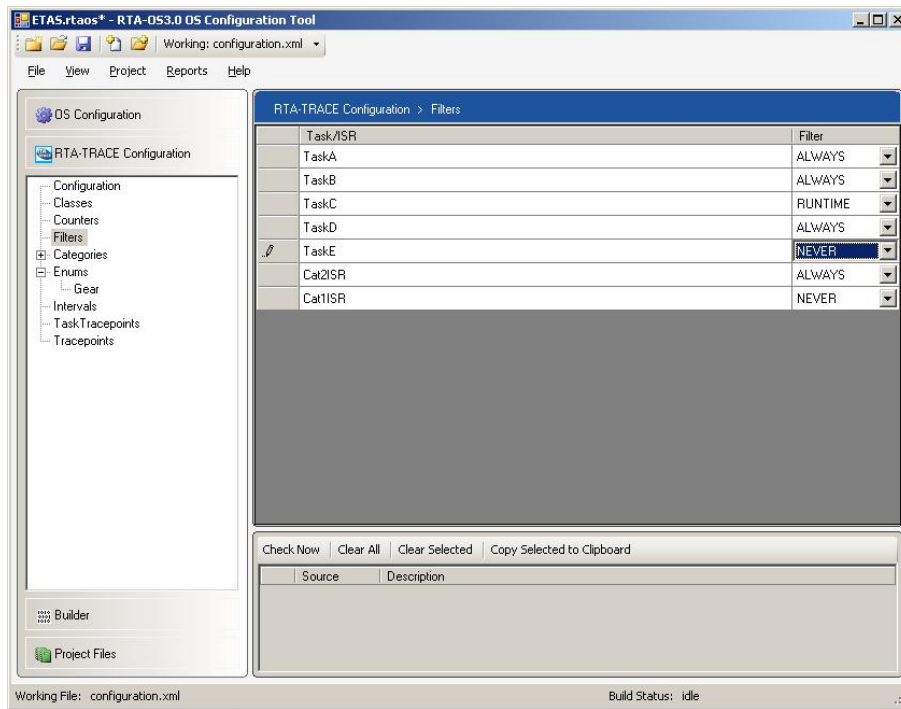


Figure 17.3: Configuring RTA-TRACE filters

Figure 17.3 shows how trace filters can be configured.

The setting of the `OS_TRACE_TASKS_AND_ISR_CLASS` is applied *before* a filter is applied. This means that filter settings for a task/ISR interact with trace classes in the following way:

Class Setting	Filter	Task/ISR Traced?
Never	Never	X
Never	Runtime	X
Never	Always	X
Runtime [Disabled]	Never	X
Runtime [Disabled]	Runtime	X
Runtime [Disabled]	Always	X
Runtime [Enabled]	Never	X
Runtime [Enabled]	Runtime	✓
Runtime [Enabled]	Always	✓
Always	Never	X
Always	Runtime	✓
Always	Always	✓

17.3 User-Defined Trace Objects

RTA-TRACE provides 3 different types of objects that you can configure to help with debugging your application:

Tracepoints are used to log arbitrary data values (for example the value of a variable or content of a data structure) in the trace buffer. Each tracepoint is logged with a timestamp so you can see on the RTA-TRACE visualization what value the data had at what time. A tracepoint can be logged from anywhere in the application.

Task Tracepoints are similar to tracepoints but are displayed on the RTA-TRACE visualization next to the task which logs them

Intervals are used to measure durations of time. An interval has a start and an end marker that can be logged from anywhere in your application. Intervals are particularly useful for measuring end-to-end response times over multiple tasks during program execution.

The following sections describe how to configure these user-defined objects and how to control whether they are logged or not at runtime.

17.3.1 Tracepoints

Each tracepoint requires a unique identifier. This is an integer. The maximum number of tracepoints that can be configured depends on the setting for “Use Compact IDs” (see Section 17.1). If the ID is set to zero, then RTA-OS3.0 automatically allocates a unique ID for the tracepoint.

Each tracepoint can also be associated with a discrete data value or a block of data. RTA-TRACE needs to know how to format the data value supplied and this is configured by specifying a format-string (see Section 17.3.5 for more information about format strings). The format string control how RTA-TRACE will display the data value in the RTA-TRACE GUI. Figure 17.4 shows the configuration of three tracepoints that log data as a signed integer, a hexadecimal value and an unsigned integer respectively.

Any task in the application can log a tracepoint using the following API calls:

- `Os_LogTracepoint()` - log the tracepoint without any associated data
- `Os_LogTracepointValue()` - log the tracepoint with an associated value
- `Os_LogTracepointData()` - log the tracepoint with an associated block of data (specified using a base/bound scheme)

For further details, see the *RTA-OS3.0 Reference Guide*.

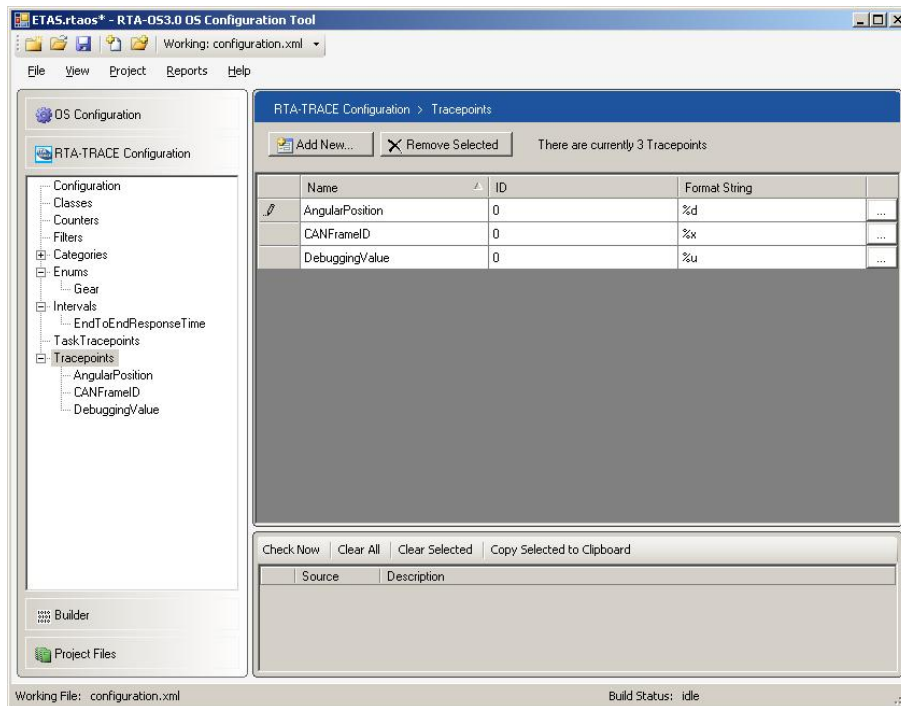


Figure 17.4: Configuring RTA-TRACE tracepoints

17.3.2 Task Tracepoints

Task-tracepoints are configured just like normal tracepoints. See Section 17.3.1 for further details.

Logging a task tracepoint uses a different set of API calls to normal tracepoints:

- `Os_LogTaskTracepoint()` - log the tracepoint against the calling tasks without any associated data
- `Os_LogTaskTracepointValue()` - log the tracepoint against the calling tasks with an associated value
- `Os_LogTaskTracepointData()` - log the tracepoint against the calling tasks with an associated block of data (specified using a base/bound scheme)

For further details, see the *RTA-OS3.0 Reference Guide*.

17.3.3 Intervals

Intervals are used to measure arbitrary times in the application, for example an end-to-end response time. Each interval you want to measure must be

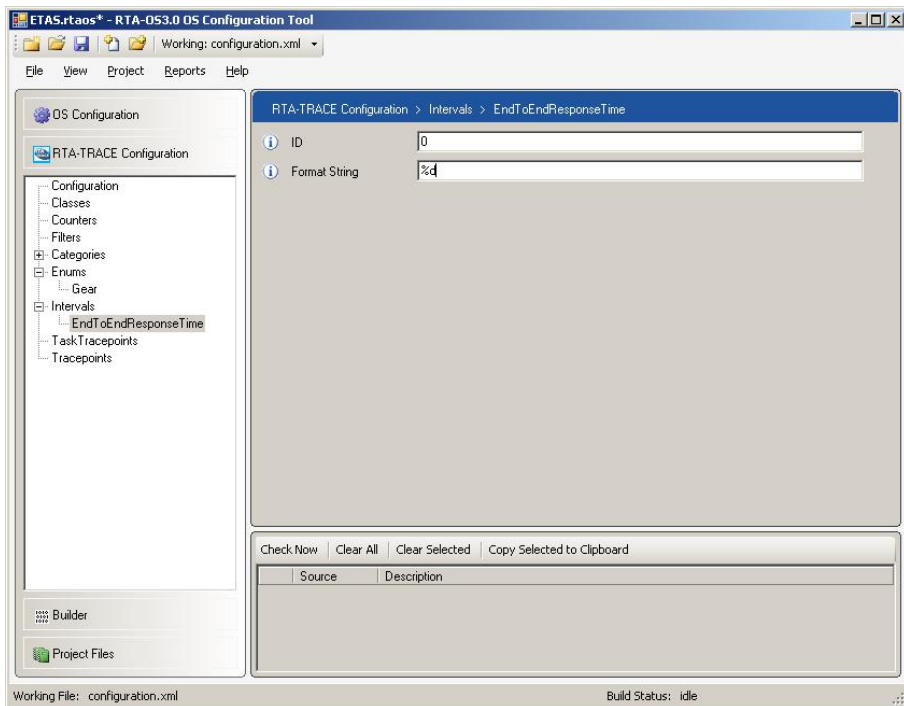


Figure 17.5: Configuring RTA-TRACE intervals

named and allocated an unique identifier. As with tracepoints, an interval identifier is an integer which is specified at configuration time. If a value of zero is configured, then RTA-OS3.0 automatically allocates a unique identifier to each interval.

Figure 17.5 shows how an interval is configured.

Each interval can also be associated with a discrete data value or a block of data. RTA-TRACE needs to know how to format the data value supplied and this is configured by specifying a format-string (see Section 17.3.5 for more information about format strings).

Logging an internal requires you to mark the start and the end of the interval using the following API calls:

- `Os_LogIntervalStart()` - log the start of the interval without any associated data
- `Os_LogIntervalStartValue()` - log the start of the interval with an associated value
- `Os_LogIntervalStartData()` - log the start of the interval with an associated block of data (specified using a base/bound scheme)

- `Os_LogIntervalEnd()` - log the end of the interval without any associated data
- `Os_LogIntervalEndValue()` - log the end of the interval with an associated value
- `Os_LogIntervalEndData()` - log the end of the interval with an associated block of data (specified using a base/bound scheme)

Calls to log with and without data or values can be mixed, as shown in Code Example 17.2.

```
#include <Os.h>
#include "ThirdPartyLibrary.h"
TASK(A) {
    ...
    Os_LogIntervalStart(LibraryCallMeasurement,
        OS_TRACE_CATEGORY_ALWAYS);
    x = CallToLibraryFunction(y,z);
    Os_LogIntervalEndValue(LibraryCallMeasurement,x,
        OS_TRACE_CATEGORY_ALWAYS);
    ...
}
```

Code Example 17.2: Mixing `Os_LogInterval...` calls

For further details, see the *RTA-OS3.0 Reference Guide*.

17.3.4 Controlling which User-Defined Objects are Traced

User-defined objects are logged in the RTA-TRACE trace buffer at runtime. Each API to log a user-specified object takes a parameter defining the trace category for which is logged:

```
Os_Log[[Task]Tracepoint|Interval[Start|End]][Data|Value](...,
    Os_TraceCategoriesType CategoryMask)
```

Trace categories are user-defined names that allow you control whether a user-defined trace object is traced or not at runtime.

Each category has a category bit-mask. The mask is an integer that represents a unique identifier for the category in the trace buffer. The mask can be set to a specific integer value, but it is recommended that you set the mask to zero and let RTA-OS3.0 generate the category mask automatically.



If you choose to set your own mask values then you must ensure that the integer representing the mask is a power of two i.e. 1,2,4,8,16 etc.

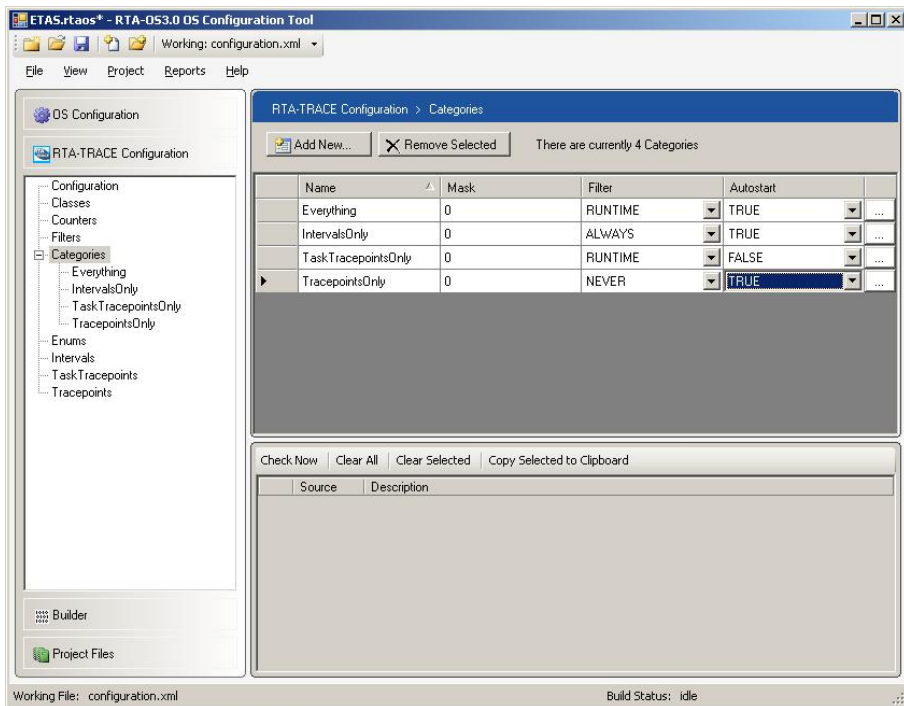


Figure 17.6: Configuring RTA-TRACE categories

As with classes, each trace category can be filtered:

Always the category is always traced.

Never the category is never traced.

Runtime the tracing of the category can be enabled/disabled at runtime.

By default, runtime trace categories are disabled when RTA-TRACE starts. The initial categories configuration allows you to control which of the run-time are enabled when tracing starts.

Figure 17.6 shows how trace categories can be configured.

RTA-TRACE also defines two constant category masks:

1. OS_TRACE_CATEGORY_ALWAYS is always be traced.
2. OS_TRACE_CATEGORY_NEVER is never be traced.

Runtime control for categories is provided though the RTA-TRACE API calls `Os_EnableTraceCategories()` and `Os_DisableTraceCategories()`. Each call takes a category mask (or a bit-wise OR of category

masks) as input. All user tracing can be disabled by calling `Os_DisableTraceCategories(OS_TRACE_CATEGORY_ALWAYS)` and re-enabled by calling `Os_EnableTraceCategories(OS_TRACE_CATEGORY_ALWAYS)`.

17.3.5 Format Strings

Format strings are used to tell RTA-TRACE how to display a user-defined trace item's data. Simple numeric data can be displayed using a single format specifier. More complex data, e.g. a C **struct**, can be displayed by repeatedly moving a cursor around the data block and emitting data according to more complex format specifiers.

If a format string is not supplied, data is displayed in the following manner:

- If the data size is no greater than the size of the target's integer type, data is decoded as if "%d" had been specified.
- Otherwise the data is displayed in a hex dump, e.g.

```
0000 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0010 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
```

The hex dump has a maximum size of 256 bytes.



When format specifiers are given, the target's endian-ness is taken into account. When a hex dump is shown, the target's memory is dumped byte-for-byte. In particular, you may not get the same output from a hex dump as from the %x format specifier.

Rules

Format strings are similar to the first parameter to the C function `printf()`:

- Format strings are surrounded by double-quote (") symbols.
- A format string may contain two types of object: ordinary characters, which are copied to the output stream, and format elements, each of which causes conversion and printing of data supplied with the event.
- A format element comprises a percent sign, zero or more digits and a single non-digit character, with the exception of the %E element.
- The format element is decoded according to the rules in the table below, and the resulting text is added to the output string.
- The special format element %% emits a %.
- In addition to ordinary characters and conversion specifications, certain characters may be emitted by using a 'backslash-escape sequence'. To

emit a double-quote " character, \" is used, and to emit a \ character, \\ is used.

- The optional size parameter to integer format specifiers defines the field's width in bytes. Valid values are 1, 2, 4 or 8.



An important difference from printf() is that the cursor does not automatically move on from the current field when a field is emitted. This is to facilitate multi-format output of a single field.

Format	Element Meaning
%offset@	Moves the cursor offset bytes into the data. This can be used to extract values from multiple fields in a structure.
%[size]d	Interpret the current item as a signed integer. Output the value as signed decimal.
%[size]u	Interpret the current item as an unsigned integer. Output the value as unsigned decimal.
%[size]x	Interpret the current item as unsigned integer. Output the value as unsigned hexadecimal.
%[size]b	Interpret the current item as an unsigned integer. Output the value as unsigned binary.
%enum[:size]E	Interpret the current item as an index into the enumeration class whose ID is enum. Emit the text in that enumeration class that corresponds with the item's value. The enumeration class should be defined using ENUM directives.
%F	Treat the current item as an IEEE 'double'. Output the value as a double, in exponent format if necessary.
%?	Emit in the form of a hex dump.
%%	No conversion is carried out; emit a %.

Enumerations

Sometime you may want RTA-TRACE to display symbolic data for a a given trace value. This is possible in a number of ways with format strings, but one possibility is to use a value to reference an enumeration of symbolic values. Each enumeration you need must be configured before it can be referenced from a format string.

An enumeration is given a name and contains a set of name/value pairs that define the mapping between the value and the associated symbolic name. Figure 17.7 shows how an enumeration of 'Gear' has been configured with a simple mapping between an integer value and the symbolic names of the gears.

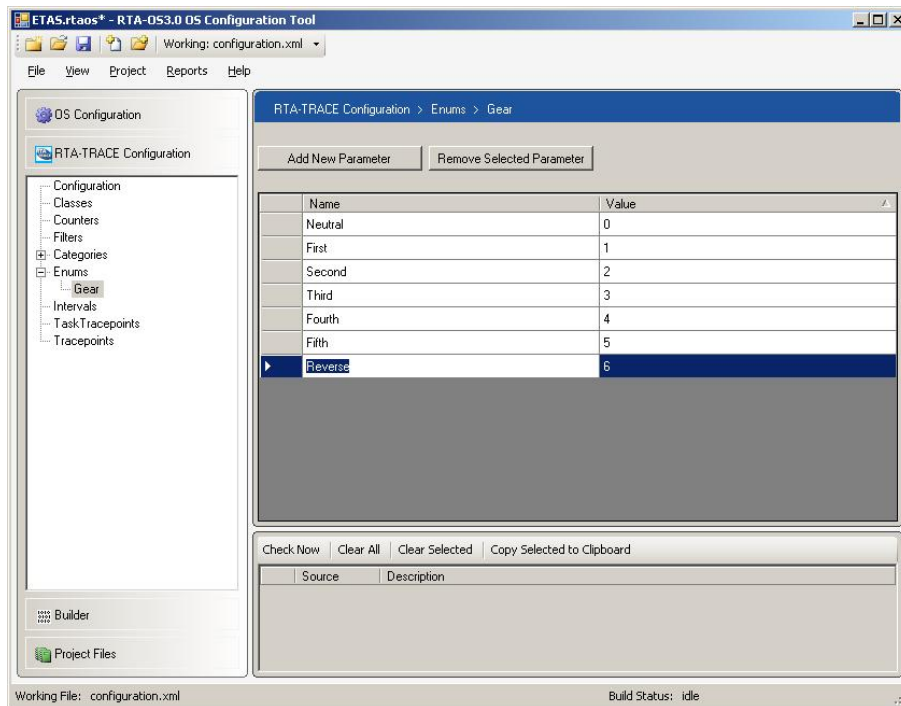


Figure 17.7: Configuring RTA-TRACE enumerations

Example Format Strings

Description	Format	Example	Notes
A native integer displayed in decimal and hexadecimal	"%d 0x%x"	10 0xA	The "0x" is not emitted by the %x format specifier but is specified in literal characters in the string. Absence of size specifier means the target's integer size is assumed. This example is a 16-bit processor.
A single unsigned byte representing a percentage.	"%1u%"	73%	Use of size specifier of 1 byte. Use of %% to emit %.
nt x;int y;structi; on a 32-bit processor.	"(%d,%4@%d)"	(20,-15)	Use of %offset@ to move to byte-offset within the structure.
A value of type enum e_Rainbow, (defined as the colors of the rainbow!)	"%1E"	Yellow	The number 1 refers to the ID of the enum class in the ENUM directives, not to the width of the field.

17.4 ECU Links

RTA-TRACE provides two standard ways to get data from the ECU to the host PC:

- **Debugger Link** - This is a passive data link - it does not require any supporting code in your application. However, you will need to use your debugger¹ to “pull” the contents of the trace buffer from the target to the PC running the RTA-TRACE Server.
- **Serial Link** - This is an active link - you need to provide code in your application to “push” the contents of the trace buffer to from the target to the PC running the RTA-TRACE Server. Both polled and interrupt-driven serial communication is possible.

Other data links may be available - please contact ETAS for details.

The following sections describe how to use the standard data-links in your application.

17.4.1 Debugger Links

The debugger link only transfers data to the RTA-TRACE server once there is a full buffer (or a full trigger window in the case of triggering mode) available for transmission.

When the buffer is full, RTA-TRACE calls the function `Os_TraceBreakLabel()`. You should use your debugger to place a breakpoint on this function so that each time the trace buffer is full, the target is paused and you can then upload the contents of the variable `Os_TraceBuffer[]` to the debugger. Many debuggers can be scripted to perform these steps automatically.

The RTA-TRACE server accepts data in two formats:

1. Lauterbach format
2. CrossView format

These formats are described in the *RTA-TRACE ECU Link Guide*.

Using the debugger link may impact interaction with the target. Each time the trace buffer is full then the target is paused by the debugger and only resumed once the trace buffer has been uploaded as shown in Figure 17.8.

¹A debugger is not supplied with RTA-OS3.0 or RTA-TRACE. A list of compatible debuggers can be found in the *RTA-OS3.0 Target/Compiler Port Guide* for your port.

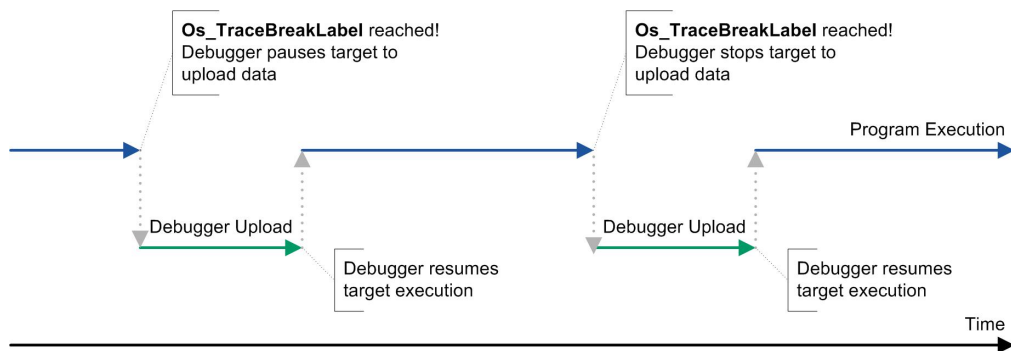


Figure 17.8: Impact of the debugger link with target execution

The debugger link is therefore best used when you need to capture a snapshot trace, such as those obtained using bursting or triggering mode.

17.4.2 Serial Links

The serial link needs your application code to actively transmit data to the RTA-TRACE server over a serial connection. RTA-TRACE manages the emptying of the trace buffer and the state of the serial connection and uses a set of callbacks to control the serial hardware itself.

Initializing the Serial Device

Serial communication is initialized by called `Os_TraceCommInit`. The call must be made before RTA-TRACE is started. If “Auto-Initialize Comms” has been configured then RTA-OS3.0 will automatically call this API during `StartOS()`.

`Os_TraceCommInit` requires you to provide the `Os_Cbk_TraceCommInitTarget` callback that should initialize the serial hardware for your target. The callback should return `OS_TRACE_STATUS_OK` if the initialization was successful and `OS_TRACE_STATUS_COMM_INIT_FAILURE` otherwise. Code Example 17.3 shows how the callback might look in your code.

```
FUNC(Os_TraceStatusType, OS_APPL_CODE) Os_TraceCommInitTarget(
    void)
{
    /* Set baud rate */
    SERIAL_BAUD_REGISTER = 9600; /* baud */

    /* Set transmit enable bit in control register 2 */
    SERIAL_CTRL_REGISTER |= TE_BIT;

    return OS_TRACE_STATUS_OK;
}
```

```
}
```

Code Example 17.3: Initializing the serial hardware

Data Transmission

Data transmission is a two stage process:

1. Check if there is any data to transmit.
2. If data is available then transmit it.

RTA-TRACE will automatically detect when the buffer is full and use this information to make the trace buffer available for transmission. This mechanism is sufficient when using bursting or triggering mode.

If you are free-running mode then this behavior may result in you losing trace records when the buffer becomes full because RTA-TRACE will suspend tracing until the buffer is emptied. However, you can tell RTA-TRACE to check for available data and make it ready for transmission *before* the trace buffer becomes full by calling `Os_CheckTraceOutput()`.

To ensure that the trace buffer is uploaded as quickly as possible you should call `Os_CheckTraceOutput()` as often as you can. A good place to make the call² is in the `Os_Cbk_Idle()` callback as shown in Code Example 17.3 shows how the callback might look in your code.

```
FUNC(boolean, OS_APPL_CODE) Os_Cbk_Idle(void)
{
    #ifdef OS_TRACE
        Os_CheckTraceOutput();
    #endif
    return TRUE;
}
```

Code Example 17.4: Checking if data is available for transmission

`Os_CheckTraceOutput()` has a short execution time so there is no significant overhead on the application if it gets called more frequently than strictly necessary.

When data is available for transmission RTA-TRACE signals this through the callback `Os_Cbk_TraceCommDataReady()`. An implementation of this callback is provided in the RTA-OS3.0 library. However, when using a serial link it is recommended that you provide your own implementation of `Os_Cbk_TraceCommDataReady()` to start the communication process..

²Assuming that there is enough slack time available in your system that the callback runs.

RTA-TRACE provides two mechanisms to transmit data from the target:

1. Asynchronous Dump - transmit the available buffer in single operation.
2. Byte-wise - transmit the available buffer a byte at a time.

Asynchronous Dump

A trace buffer dump is made using the RTA-TRACE API `Os_TraceDumpAsync()`. The call takes a function name as a parameter. The function must be able to transmit a byte of data over the serial line.

You should only call `Os_TraceDumpAsync()` when there is data available which means the call should be made from the `Os_Cbk_TraceCommDataReady()` callback. This means you need to provide an implementation of `Os_Cbk_TraceCommDataReady()` to override the one provided in the RTA-OS3.0 library. Code Example 17.5 shows the implementation of a trace buffer dump.

```
void TransmitByte(uint8 val) {  
    while(!tx_ready) {/* Wait for space in serial device */  
        transmit(val) ;  
    }  
  
FUNC(void, OS_APPL_CODE) Os_Cbk_TraceCommDataReady(void) {  
    Os_TraceDumpAsync(TransmitByte);  
}
```

Code Example 17.5: Implementing a Trace Buffer Dump

Byte-Wise Transmission

Byte-wise transmission feeds one byte of the trace buffer at a time to the serial device. The RTA-TRACE API call `Os_UploadTraceData()` is made to transfer a byte of the trace buffer to the serial device. You need to make the call often enough to ensure that data is transmitted. The call can be made from anywhere in your application code, but you need to be aware that if it is made from a higher priority task then it will affect the responsiveness of lower priority tasks.

If there is data waiting then `Os_UploadTraceData()` attempts to queue the byte for sending as follows:

1. The callback `Os_Cbk_TraceCommTxReady()` is made to check if there is space in the serial device's buffer.

2. If space is available, then the callback `Os_Cbk_TraceCommTxStart()` is made to signal that transmission is about to start. If there is no space then the call returns immediately.
3. The callback `Os_Cbk_TraceCommTxByte()` is made to actually transmit the byte
4. The callback `Os_Cbk_TraceCommTxEnd()` is made to signal that transmission has completed

The RTA-TRACE serial ECU link can operate in either interrupt or polling mode. Interrupt mode prioritizes communication at the expense of the application's timing characteristics. Polling mode prioritizes the application's timing behavior at the possible risk of some loss of trace data. In general it is recommended to use polling mode and if necessary set target-side triggers and filters to generate a smaller volume of data (see Section 17.2.1).

Whichever transmission mode you choose, you always need to provide implementations of the four callback functions:

1. `Os_Cbk_TraceCommTxReady()`
2. `Os_Cbk_TraceCommTxStart()`
3. `Os_Cbk_TraceCommTxByte()`
4. `Os_Cbk_TraceCommTxEnd()`

The following sections explain what these callbacks need to do and how to construct the polled or interrupt mode driver.

Polling Mode

Polled mode requires that you make regular calls to `Os_CheckTraceOutput()` and `Os_UploadTraceData()` to ensure data in the trace buffer is made available to upload and then uploaded before the trace buffer becomes full. Typically, it is sufficient in most system to do this from RTA-OS3.0's idle mechanism, `Os_Cbk_Idle()`, so that when you application has nothing else to do it can be uploading the trace buffer as a 'background' activity.



If you are using RTA-TRACE in free-running mode then you must call `Os_CheckTraceOutput()` regularly. If this is not called in a timely fashion then `Os_UploadTraceData()` will not have any data to transmit. Failing to call `Os_CheckTraceOutput()` regularly will result in the trace buffer becoming full. If this occurs then RTA-TRACE will suspend tracing until the buffer has been emptied or partially emptied and `Os_CheckTraceOutput()` has been called.

You need to provide implementations of the callbacks `Os_Cbk_TraceCommTxReady()` and `Os_Cbk_TraceCommTxByte()`. It is not necessary to do anything for `Os_Cbk_TraceCommTxStart()` and `Os_Cbk_TraceCommTxEnd()` callbacks, but ‘dummy’ implementations need to be provided.

Code Example 17.6 shows a typical polled driver implementation.

```
FUNC(boolean, OS_APPL_CODE) Os_Cbk_Idle(void)
{
    #ifdef OS_TRACE
        Os_CheckTraceOutput();
        Os_UploadTraceData();
    #endif
    return TRUE;
}

FUNC(void, OS_APPL_CODE) Os_Cbk_TraceCommTxStart(void){
    /* Do nothing */
}

FUNC(boolean, OS_APPL_CODE) Os_Cbk_TraceCommTxReady(void){
    return (serial_device_has_space());
}

FUNC(void, OS_APPL_CODE) Os_Cbk_TraceCommTxByte(uint8 byte){
    serial_device_transmit_byte(byte);
}

FUNC(void, OS_APPL_CODE) Os_Cbk_TraceCommTxEnd(void){
    /* Do nothing */
}
```

Code Example 17.6: Polled Transmission

Interrupt Mode

Trace data throughput can be optimized by using the serial module’s ‘Transmit Complete’ interrupt and a user-supplied interrupt handler that calls `Os_UploadTraceData()`. This means that data transmission takes precedence over task execution. Interrupt mode is therefore best suited to bursting and triggered modes where data transmission takes place after trace recording has stopped.



It is not recommended to use interrupt transmission in free-running mode because handling the transmit complete interrupt will affect the timing behavior of the system.

When RTA-TRACE detects that the trace data buffer is ready for transmission in the callback `Os_Cbk_TraceCommDataReady()` is called. You must call `Os_UploadTraceData()` to start the transmission of the trace data.

You will need to configure an RTA-OS3.0 interrupt (either Category 1 or Category 2) using `rtaoscfg` and provide an implementation of the handler.

As with polled mode, implementations of the callbacks `Os_Cbk_TraceCommTxReady()` and `Os_Cbk_TraceCommTxByte()` are required. The functionality of these caThese will be identical to the ones you would write for a polled mode driver.

Interrupt mode uses the callbacks `Os_Cbk_TraceCommTxStart()` and `Os_Cbk_TraceCommTxEnd()` to enable and disable the transmit interrupt.

Code Example 17.7 shows a typical polled driver implementation.

```
ISR(SerialTxInterrupt){
    Os_UploadTraceData();
    dismiss_serial_tx_interrupt();
}

FUNC(void, OS_APPL_CODE) Os_Cbk_TraceCommDataReady(void) {
    Os_UploadTraceData();
}

FUNC(void, OS_APPL_CODE) Os_Cbk_TraceCommTxStart(void){
    enable_serial_tx_interrupt();
}

FUNC(boolean, OS_APPL_CODE) Os_Cbk_TraceCommTxReady(void){
    return (serial_device_has_space());
}

FUNC(void, OS_APPL_CODE) Os_Cbk_TraceCommTxByte(uint8 byte){
    serial_device_transmit_byte(byte);
}

FUNC(void, OS_APPL_CODE) Os_Cbk_TraceCommTxEnd(void){
    disble_serial_tx_interrupt();
}
```

Code Example 17.7: Interrupt Transmission

Mode Summary

The following table gives a summary of what needs to be implemented for polling and interrupt-driven modes of operation.

Callback	Polled-Mode	Interrupt Mode
<code>Os_Cbk_TraceCommTxStart</code>	empty	Enable Tx interrupt
<code>Os_Cbk_TraceCommTxReady</code>	Check for space in serial device	Check for space in serial device
<code>Os_Cbk_TraceCommTxByte</code>	Transmit a byte	Transmit a byte
<code>Os_Cbk_TraceCommTxEnd</code>	empty	Disable Tx interrupt
<code>Os_Cbk_TraceCommDataReady</code>	empty	Call <code>Os_UploadTraceData()</code> ;

17.5 Summary

- RTA-OS3.0 can automatically instrument the kernel library to generate RTA-TRACE profiling information.
- The instrumented kernel logs trace data to an on-target memory buffer.
- The buffer can be emptied by ‘pulling’ the data out using a third-party debugger or by ‘pushing’ the data out over a serial communication link.
- Further information about RTA-TRACE ships with your RTA-TRACE product.

18 **Contacting ETAS**

18.1 **Technical Support**

Technical support is available to all RTA-OS3.0 users with a valid support contract. If you do not have such a contract then please contact ETAS through one of the addresses listed in Section [18.2](#).

The best way to get technical support is by email. Any problems or questions should be sent to: rta.hotline.uk@etas.com

It is helpful if you can provide support with the following information:

- your support contract number.
- your .xml/.rtaos configuration files.
- the error message you received and the file `Diagnostic.dmp` if it was generated.
- the command line that results in an error message.
- the version of the ETAS tools you are using.
- the version of your compiler tool chain you are using.

If you prefer to discuss your problem with the technical support team you can contact them by telephone during normal office hours (0900-1730 GMT/BST). The telephone number for the RTA-OS3.0 support hotline is: +44 (0)1904 562624.

18.2 General Enquiries



Europe

Excluding France, Belgium, Luxembourg, United Kingdom and Scandinavia

ETAS GmbH

Borsigstrasse 14
70469 Stuttgart
Germany

Phone: +49 711 89661-0
Fax: +49 711 89661-300
E-mail: sales.de@etas.com
WWW: www.etas.com

France, Belgium and Luxemburg

ETAS S.A.S.

1, place des États-Unis
SILIC 307
94588 Rungis Cedex
France

Phone: +33 1 56 70 00 50
Fax: +33 1 56 70 00 51
E-mail: sales.fr@etas.com
WWW: www.etas.com

United Kingdom and Scandinavia

ETAS Ltd.

Studio 3, Waterside Court
Third Avenue, Centrum 100
Burton-upon-Trent
Staffordshire DE14 2WQ
United Kingdom

Phone: +44 1283 54 65 12
Fax: +44 1283 54 87 67
E-mail: sales.uk@etas.com
WWW: www.etas.com

USA

ETAS Inc.

3021 Miller Road
Ann Arbor
MI 48103
USA

Phone: +1 888 ETAS INC
Fax: +1 734 997-9449
E-mail: sales.us@etas.com
WWW: www.etas.com

Japan

ETAS K.K.

Queen's Tower C-17F
2-3-5, Minatomirai, Nishi-ku
Yokohama 220-6217
Japan

Phone: +81 45 222-0900
Fax: +81 45 222-0956
E-mail: sales.jp@etas.com
WWW: www.etas.com

Korea

ETAS Korea Co. Ltd.

4F, 705 Bldg. 70-5
Yangjae-dong, Seocho-gu
Seoul 137-889
Korea

Phone: +82 2 5747-016
Fax: +82 2 5747-120
E-mail: sales.kr@etas.com
WWW: www.etas.com

P.R.China

ETAS (Shanghai) Co., Ltd.

2404 Bank of China Tower
200 Yincheng Road Central
Shanghai 200120
P.R. China

Phone: +86 21 5037 2220
Fax: +86 21 5037 2221
E-mail: sales.cn@etas.com
WWW: www.etas.com

India

ETAS Automotive India Pvt. Ltd.

No. 690, Gold Hill Square, 12F
Hosur Road, Bommanahalli
Bangalore, 560 068
India

Phone: +91 80 4191 2585
Fax: +91 80 4191 2586
E-mail: sales.in@etas.com
WWW: www.etas.com

Index

A

- Alarms, [137](#)
 - Absolute, [142](#)
 - Action on expiry, [138](#)
 - Activating Tasks, [138](#)
 - Auto-starting, [146](#)
 - Callbacks, [138](#)
 - Canceling, [147](#)
 - Cyclic, [143](#), [146](#)
 - Incrementing Counters, [140](#)
 - Periodic, *see* Cyclic
 - Relative, [142](#), [145](#)
 - Setting Events, [138](#)
 - Single-shot, [142](#), [145](#)
- Application Modes, [191](#)
- AUTOSAR, [20](#)
 - Include file dependencies, [34](#)
 - Operating System, [20](#)
 - Scalability Class, [20](#)
- AUTOSAR includes
 - Compiler.h, [35](#)
 - Compiler_Cfg.h, [35](#)
 - MemMap.h, [35](#)
 - Platform_Types.h, [34](#)
 - Std_Types.h, [34](#)
- AUTOSAR OS includes
 - Os.h, [39](#)
 - Os_Cfg.h, [39](#)
 - Os_MemMap.h, [39](#)

C

- C Startup Code, [186](#)
- Compilation, [41](#)
- Compiler, [34](#)
- Configuration Files, [29](#)
 - Project Files, [30](#)
 - XML, [29](#)
- Conformance Classes, [51](#)
- Context switch, [68](#)
- Counter
 - Getting the value, [133](#)
- Counter Attributes
 - Accessing at runtime, [132](#)

MAXALLOWEDVALUE, [124](#)

MINCYCLE, [124](#)

TICKSPERBASE, [124](#)

Counter Driver, [160](#)

Counters, [123](#)

Cascading, [140](#)

Free running timers, [134](#)

Hardware, [123](#)

Hardware Driver, [129](#)

Software, [123](#)

Software Driver, [125](#)

Ticks, [123](#)

CPU Clock rate, *see* Instruction Rate

Critical Section, [98](#)

D

- Deadline Monotonic, [45](#)
- Deadlock
 - Freedom from, [100](#)
- Debugging
 - API Usage, [199](#)
 - ORTI, [226](#)
 - RTA-TRACE, [230](#)
 - Stack Monitoring, [207](#)
 - Time Monitoring, [217](#)
- Development process, [24](#)

E

- ECU Link
 - Debugger, [244](#)
 - Serial, [245](#)
 - Asynchronous Dump, [247](#)
 - Byte-wise, [247](#)
 - Driver Callbacks, [248](#)
 - Interrupt Driven, [249](#)
 - Polling, [248](#)
- Error Codes, [199](#)
- Error Handling, [199](#)
- Events, [114](#)
 - Clearing, [120](#)
 - Multiple Waits, [117](#)
 - Setting, [119](#)
 - Waiting On, [115](#)

Extended Status, 199
Extended Tasks
 Risk of deadlock, 118
 Simulation using Basic Tasks, 121

F

Fixed Priority, 45
Free running timer, 134

G

Generated files, 38

H

Hooks
 Error, 200
 PostTask, 76
 PreTask, 76
 Shutdown, 196
 Startup, 190

I

Idle Mechanism, 74
 Limitations, 75
Imprecise Computation, 222
Instruction Cycle Rate, 224
Instruction Rate, 217
Internal Resources, 105
 Shared with interrupts, 105
 Stack Saving with, 109
Interrupts, 40, 83
 Category 1, 84, 90
 Category 2, 84, 91
 Compiler Directives, 90
 Default Interrupt, 94
 Enabling and disabling, 93
 Multi-level, 83
 Nested, 83
 Priority, 84
 Register Sets, 94
 Single-level, 83
ISR, 91

L

Library, 32
 Name of, 39
Linked Resources, 103

M

MISRA, 32
Mutual Exclusion, 98

O

Optimization
 Fast Task Termination, 73
 Omit Schedule () API, 72
 Wait Event Stack, 63
Optimizations
 Stack Reduction, 108
ORTI, 226
OS-level, 87
OSEK, 17
 Operating System, 18

P

Priority Ceiling Protocol, 99
Priority Inversion, 99

R

Rate Monotonic, 45
Register Sets
 Saving in ISRs, 94
 Saving in Tasks, 78
Reports, 32
RES_SCHEDULER, 110
Reset, 185
Resources, 98
 Ceiling Priority, 99
 Internal, 105
 Linked, 103
 Nesting locks, 102
 Race Conditions, 112
 Sharing with Interrupts, 100
Restarting, 196
RTA-TRACE, 230
 Burst Mode, 232
 Categories, 239
 Classes, 233
 Configuration, 230
 ECU Links, 244
 Enumerations, 242
 Filters, 234
 Format Strings, 241

- Free-Running Mode, 232
- Instrumentation, 230
- Intervals, 237
- Task Tracepoints, 237
- Tracepoints, 236
- Triggering Mode, 232
- RTA-TRACE Configuration, 28
- rtaoscfg, 24
 - Builder, 38
- rtaosgen, 37
 - Invoking from rtaoscfg, 28

S

- Sample Code, 39
- Samples, 39
- Schedule Tables, 151
 - Absolute Start, 154
 - Expiry Points, 151, 153
 - Relative Start, 156
 - Switching, 157
- Scheduler, 45
- Scheduling
 - Cooperative, 47, 72
 - Non-Preemptive, 46
 - Preemptive, 45
- Scheduling Policy, 45
- Semaphore, see Resources
- Shutdown, 196
- Shutdown Hook, 196
- Single-Stack, 57
 - Extended Tasks, 58
- Stack, 57
 - Allocation, 61
 - Default Allocation, 63
 - Measurement of, 212
 - Optimization, 63
 - Reducing Size, 109
- Stack Management
 - Overruns, 64
- Stack Resource Protocol, 99
- Standard Resources
 - Stack Saving with, 109
- Standard Status, 199
- Starting RTA-OS3.0, 40
- StartOS, 189
- Startup
 - Activating Tasks, 56
 - Alarms, 146, 194
 - Schedule Tables, 195
 - Tasks, 193
- Startup Hook, 190
- Static Interface
 - Software Counters, 127
- Status
 - Extended, 199
 - Standard, 199
- Stopwatch, 217
 - Uncertainty, 219
- Stopwatch Speed, 224
- SystemCounter, 133
- SystemTimer, see SystemCounter

T

- Tasks, 40, 45
 - Activation, 50
 - Basic, 47
 - Entry Function, 66
 - Extended, 48
 - Maximum supported, 53
 - Optimization, 53, 56, 58
 - Queuing Activations, 51
 - Register Sets, 78
 - Sharing Priorities, 50
 - States, 48
 - Synchronization, 48
 - Termination, 50
- Tick/Time Conversion, 134
- Time Measurement
 - Arbitrary Code, 221
 - ISR, 219
 - Tasks, 219
- Time Monitoring, 217, 223
 - Budgets, 224
 - Resetting Budgets, 225
- Time-base, 134
- Toolchain, 34

U

- User-level, 87

V
Vector Table

Generation, 89
Writing by hand, 89