RTA-OS3.0

VRTA Port Guide

Copyright

Document: 10384(VRTA)-PG-1.0.0

Contents

1 Introduction

RTA-OS3.0 is a small and fast real-time operating system that conforms to both the AUTOSAR OS R3.0 and OSEK/VDX 2.2.3 standards. The operating system is configured and built on a PC for use on a target hardware platform.

This document describes the RTA-OS3.0 VRTA port plug-in that customizes the RTA-OS3.0 development tools for the Microsoft Windows Virtual ECU with the appropriate Windows C compiler. It supplements the more general information you can find in the *RTA-OS3.0 User Guide* and the *RTA-OS3.0 Reference Guide*.

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

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

Chapters 4 to 7 provide reference information including:

- the number of OS objects supported;
- required and recommended toolchain parameters;
- how RTA-OS3.0 interacts with the Virtual ECU, including required register settings, memory models and interrupt handling;
- memory consumption for each OS object;
- memory consumption of each API call;
- execution times for each API call.

For the best experience with RTA-OS3.0 it is essential that you read and understand this document.

1.1 About You

You are a trained embedded systems developer who wants to build real-time applications using a 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.

1.2 Document Conventions

The following conventions are used in this guide:

Choose **File > Open**.	Menu options are printed in **bold, blue** characters.
Click **OK**.	Button labels are printed in **bold** characters
Press <Enter>.	Key commands are enclosed in angle brackets.
The “Open file” dialog box appears	The names of program windows, dialog boxes, fields, etc. are enclosed in double quotes.
`Activate(Task1)`	Program code, header file names, C type names, C functions and RTA-OS3.0. Component API call names all appear in the `courier` typeface.
See Section 1.2.	Hyperlinks through the document are shown in red letters.
ETAS	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.

1.3 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 Installing the RTA-OS3.0 Port Plug-in

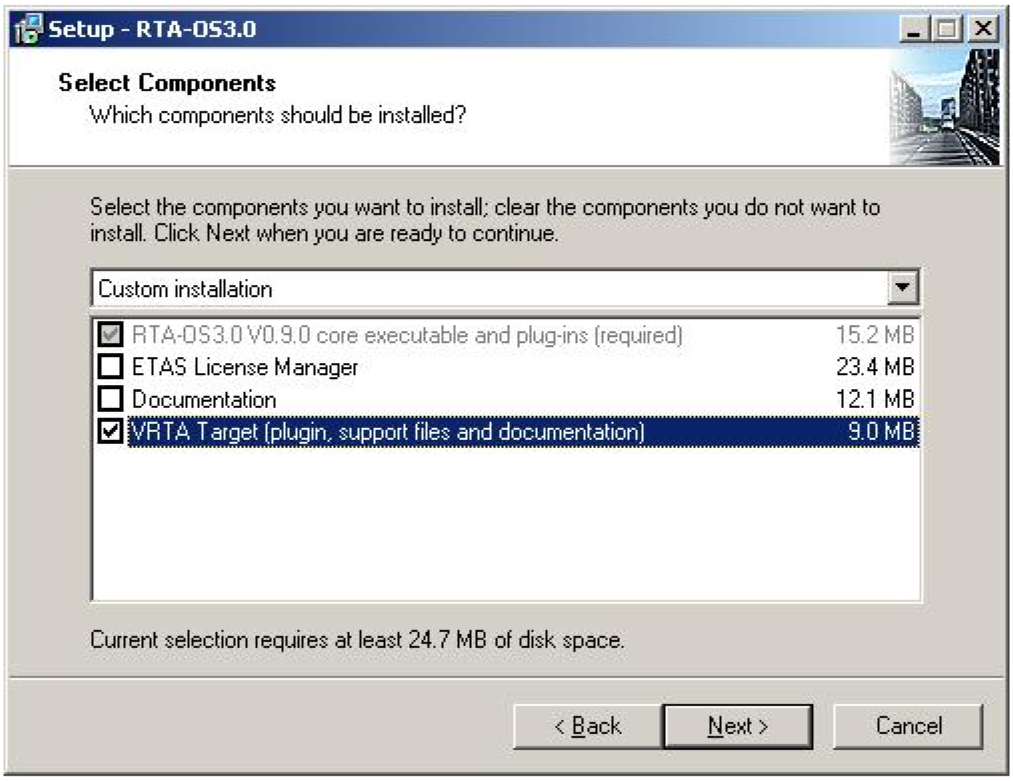
The VRTA port plug-in is installed by default as part of the RTA-OS3.0 tools installation. If you did not to install VRTA when installing the tools, then you will need to do the parts of the tools installation required to install VRTA. Either:

- Double click the executable image; or
- Insert the RTA-OS3.0 CD into your CD-ROM or DVD drive.

  If the installation program does not run automatically then you will need to start the installation manually. Navigate to the root directory of your CD/DVD drive and double click `autostart.exe` or open `Index.htm` in a web-browser of your choice to start the setup.

When asked to “Select Components”, make sure that only the VRTA component is selected as shown in Figure 2.1.

Please refer to the *RTA-OS3.0 Getting Started Guide* for further details regarding the installation of VRTA.


Setup - RTA-OS3.0

Select Components

Which components should be installed?

Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.

Custom installation

✓	RTA-OS3.0 V0.9.0 core executable and plug-ins (required)	15.2 MB
☐	ETAS License Manager	23.4 MB
☐	Documentation	12.1 MB
✓	VRTA Target (plugin, support files and documentation)	9.0 MB

Current selection requires at least 24.7 MB of disk space.

Figure 2.1: Selecting the VRTA installation

3 Verifying your Installation

Now that you have installed both the RTA-OS3.0 tool and an RTA-OS3.0 port plug-in and have obtained and installed a valid license key you can check that things are working.

3.1 Checking the Port

The first thing to check is that the RTA-OS3.0 tools can see the new port. You can do this in two ways:

1. use the **rtaosgen** tool

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

   ```
   rtaosgen
   Version x.y.z.revision, Copyright © ETAS 2008
   Available targets:
    TriCoreHighTec_n.n.n [TC1797]
    VRTA_n.n.n [MinGW,VS2005,VS2008]
   ```

2. use the **rtaoscfg** tool

   The second way to check that the port plug-in can be seen is by starting **rtaoscfg** and selecting **Help → About** drop down menu. This will show information about your complete RTA-OS3.0 configuration.

If the tools can see the port then you can move on to the next stage – checking that you can build an RTA-OS3.0 library and use this in a real program that will run on your target hardware.

3.2 Building the Example Application

Your RTA-OS3.0 port plug-in includes a “HelloWorld” example application that is used to verify that the end-to-end build process is working correctly. This is provided so that you can check that RTA-OS3.0 can build a kernel library with your toolchain and then use the library in a small example application that does just enough to show that the kernel is working correctly.

3.2.1 Preparing the Toolchain

RTA-OS3.0 will need to use your compiler toolchain and expects that all tools are available on the Windows Path.

3.2.2 What does the “HelloWorld” example do?

The example application is a very simple program. It shows preemption between two tasks, `HighPriority` and `LowPriority`. Both tasks run for 2ms.

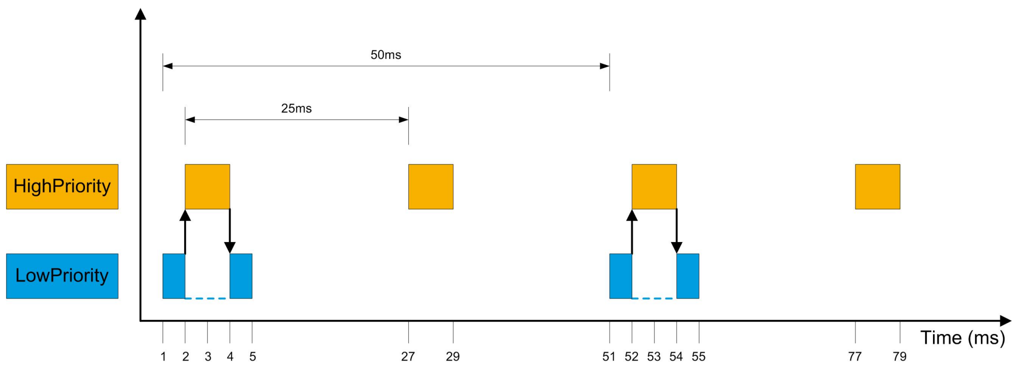


Figure 3.1: Execution of tasks in HelloWorld

Task `HighPriority` is the higher priority task and runs periodically every 50ms. Task `LowPriority` runs periodically every 25ms.

The periodic running of the tasks is achieved using two alarms, `Alarm50` and `Alarm25`, which are attached to a counter called `MillisecondCounter`. The counter is ticked using a 1ms timer interrupt that is handled by the Interrupt Service Routine (ISR) `MillisecondInterruptHandler`. Both alarms are auto-started. `Alarm50` is offset by 1ms relative to `Alarm25`.

When the application runs, Alarm25 will expire after 1ms and activate task `LowPriority`. Task `LowPriority` sets `IO_PIN1` high and runs for 1ms before being preempted by task `HighPriority`.

Task `HighPriority` saves the state of `IO_PIN1` and then sets it low before setting `IO_PIN2` high. It then runs for 2ms before setting `IO_PIN2` low. Finally, it restores the state of `IO_PIN1` and terminates, allowing task `LowPriority` to continue from the point at which it was preempted. Task `LowPriority` then runs for its remaining 1ms.

The pattern of execution is shown in Figure 3.1

3.2.3 Example Files

The “Hello World” example application can be found in the VRTA installation directory:

`<instdir>\Targets\VRTA_1.0.0\Examples\HelloWorld`

Port-Independent Code

The following files are port independent - you should not need to modify these. However, it is useful for you to know what the files contain so you can see how a simple real-time application is constructed.

Filename	Contents
`Task_HighPriority.c`	Source code for the high priority task.
`Task_LowPriority.c`	Source code for the low priority task.
`ISR_Millisecond.c`	Source code for the Category2 interrupt handler.
`Idle.c`	The idle mechanism.
`BusyWait.c`	A busy wait loop. This is used by the `HighPriority` and `LowPriority` tasks so they can execute for the required length of time.
`Main.c`	The main program.

Port-Specific Code

The following files are specific to your target microcontroller and compiler combination:

Filename	Contents
`<target>Def.h`	Register definitions for the target microcontroller.
`Target.c`	Target support code to initialize the hardware and program a 1ms interrupt source.
`Target.h`	Target support macros, including a macro to clear a pending interrupt.

You may also find other files (such as files containing code to handle a reset).

The example application is supplied ready to run on your target hardware. If your target hardware differs from the reference platform, specified in the `README.txt` file, you will need to modify the target-specific aspects defined in `Target.h`, `Target.c` and `<target>Def.h`.

All of the port specific code needs to access the registers of the microcontroller. Sometimes a register definition file is supplied by the compiler vendor to do this and sometimes it isn't.

For commonality across RTA-OS3.0 ports, ETAS supplies a simple register definition file and associated settings to program just the registers needed for the example.

Target.h

The `Target.h` file defines macros for:

- configuring and initializing the I/O port
- setting and reading the state of port pins
- setting a timer for a millisecond interrupt
- dismissing the millisecond interrupt when it occurs

If you want to change the port and/or the pins or change the interrupt source you will need to modify these macros. Details of what configuration changes are required are given in the source file. You may also have to provide additional register definitions through `<target>Def.h` or use a register definition file supplied by your compiler vendor.

Macro	**Description**
`CLEAR_PENDING_INTERRUPT()`	Called from the interrupt handler to clear the pending flag of the millisecond interrupt and to reset an interrupt for another millisecond. On some targets this may be empty as the pending bit gets cleared automatically and/or the millisecond compare is reset automatically.
`SET_PIN_HIGH(pin)`	Sets pin into the high state.
`SET_PIN_LOW(pin)`	Sets the pin into the low state
`GET_PIN_STATE(pin)`	Returns the current state of the pin.
`INIT_PINS()`	Initializes the I/O register used
`IO_PIN1`	The location of pin 1. This is controlled by the `HighPriority` task.
`IO_PIN2`	The location of pin 2. This is controlled by the `LowPriority` task
`IS_PIN_HIGH(state)`	Returns true if state is high and false otherwise.
`TIMER_MILLISECOND`	The number of ticks of the timer required to reach 1ms.

Target.c

The `Target.c` file contains functions to initialize the target hardware and the development environment, as well as stopwatch functions used for timing measurements.

Function	**Description**
`InitEnvironment()`	Called from the main program. Contains code for environment initialization, such as setting the bus clock for the development system.
`InitTarget()`	Called from the main program. Contains code to initialize the timer hardware on the target.
`Os_Cbk_Stopwatch()`	Called to get the current value of a free running counter, called the “Stopwatch” which is used for timing measurement needed to set up the execution times of the `HighPriority` and `LowPriority` tasks.
`Os_Cbk_StartupHook()`	Called from `StartOS()` to enable the periodic interrupt safely. For most targets, the speed of the stopwatch is set to the same frequency as the CPU instruction rate. However, this is not always possible - you should look at the comments in `Target.c` to find out what CPU instruction rate and Stopwatch rate have been used.

There may be additional functions provided.

Build Support

The remaining files are provided to help you build the application or to change how the non-OS parts are configured.

Filename	**Description**
`README.txt`	A description of the low-level details of the application, including the target reference platform, what peripherals are used, where the application is located etc.
`build.bat`	A Windows command shell (DOS shell) script to generate an RTA-OS3.0 kernel library, compile the example and link the example code with the library to produce a downloadable executable image.
`LinkerControl.<ext>`	A control file for the linker where <ext> depends on your linker.

3.2.4 Building “Hello World”

To build the application, open a Windows command prompt and run `build.bat` *Variant*.

Variant is the name of the target variant (i.e. compiler) you want to use to build the example application. Valid options include:

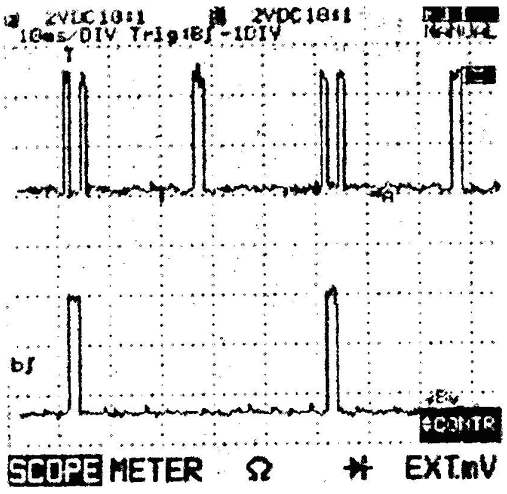


Figure 3.2: Oscilloscope Trace of HelloWorld

- MinGW
- VS2005
- VS2008

3.2.5 Verifying Program Execution

You can monitor task activation by connecting oscilloscope probes to the IO pins defined by `IO_PIN1` and `IO_PIN2`.

Figure 3.2 shows an oscilloscope trace of the state of the IO pins once the program is running. Each vertical grid line represents 10ms of time.

3.2.6 Troubleshooting

If your application doesn't appear to be running, you can use a debugger to verify that the ISR is being called. Place a breakpoint on the first instruction of the `ISR(Os_Entry_MillisecondISR)` and see if your application reaches it.

If the ISR runs this means that the counter is being ticked. You can then set breakpoints on the two tasks (`Os_Entry_HighPriority` and `Os_Entry_LowPriority`) to see whether or not they run. If the tasks run, but you do not see an output, check your hardware initialization.

If your oscilloscope does not show a trace, check the settings for `IO_PIN1` and `IO_PIN2` in `Target.h`. You should also check the `IO_PIN` control macros to ensure that they all reference the same I/O port on your target hardware. If the trace shows different timing behavior, check that your timer hardware is configured correctly and that instruction rate on the target hardware matches that specified in your configuration.

4 Port Characteristics

This chapter tells you about the charcteristics of RTA-OS3.0 for the VRTA port.

4.1 Parameters of Implementation

To be a valid OSEK or AUTOSAR OS, an implementation must support a minimum number of OS objects. The following table specifies the *minimum* numbers of each object required by the standards and the *maximum* number of each object supported by RTA-OS3.0 for the VRTA.

Parameter	Required	RTA-OS3.0
Tasks	16	1024
Tasks not in SUSPENDED state	16	1024
Priorities	16	1024
Tasks per priority	-	1024
Queued activations per priority	-	4294967296
Events per task	8	32
Software Counters	8	4294967296
Hardware Counters	-	4294967296
Alarms	1	4294967296
Standard Resources	8	4294967296
Linked Resources	-	4294967296
Nested calls to `GetResource()`	-	4294967296
Internal Resources	2	no limit
Application Modes	1	4294967296
Schedule Tables	2	4294967296
Expiry Points per Schedule Table	-	4294967296

4.2 Configuration Parameters

The following sections describe the port-specific configuration options for the VRTA port. These settings are accessed on the “Target-Specific” tab that can be found by navigating to the **General → Target** workspace of **rtaoscfg**.

4.2.1 Stack used for C-startup

The amount of stack already in use at the point that StartOS() is called.

4.2.2 Stack used when idle

The amount of stack used in the OS idle state (excluding the C-startup).

4.2.3 Stack overheads for ISR activation

The amount of stack overheads needed to activate a task from within an ISR.

4.2.4 Stack overheads for ECC tasks

The amount of stack overheads needed for an ECC task.

4.2.5 Stack overheads for ISR

The amount of stack overheads needed for a Category 2 ISR.

4.3 Generated Files

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

Filename	Contents
`Os.h`	The main include file for the OS.
`Os_Cfg.h`	Declarations of the objects you have configured. This is included by `Os.h`.
`Os_MemMap.h`	AUTOSAR memory mapping configuration used by RTA-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.

4.4 Type Definitions

4.4.1 Os_StackSizeType

A value representting a size (in bytes) on the stack. As the stack grows in size, these values increase.

Declaration
`unsigned`

4.4.2 Os_StackTraceType

An unsigned type used to represent values on the stack.

Declaration
`unsigned`

4.4.3 Os_StackValueType

A value representing the position of the stack (ESP). As the stack grows in size, these value reduce.

Declaration
`unsigned`

5 Toolchain

This chapter contains important details about RTA-OS3.0 and the appropriate Windows C toolchain. A port of the RTA-OS3.0 is specific to both the target hardware and a specific version of the compiler toolchain. You must make sure that you build your application with the supported toolchain.

In addition to the version of the toolchain, RTA-OS3.0 may use specific tool options (switches). Each tool lists the options that are used by **rtaosgen** to build the kernel. While it is recommended that you use the same options to build application code, there are no restrictions on the use of any tool chain options providing that they do not conflict with options used for RTA-OS3.0.

ETAS has developed and tested RTA-OS3.0 using the tool versions and options indicated in the following sections. Correct operation of RTA-OS3.0 is only covered by the warranty in the terms and conditions of your deployment license agreement when using identical versions and options. If you choose to use a different version of the toolchain or an alternative set of options then it is your responsibility to check that the system works correctly. If you require a statement that RTA-OS3.0 works correctly with your chosen tool version and options then please contact ETAS to discuss validation possibilities.

5.1 Compiler (MinGW)

Name	gcc.exe
Version	Tested on 5.1.3

Options

`-m32` 32 bit i386 code

`-O2` Optimization level

5.2 Librarian (MinGW)

Name	ar.exe
Version	GNU ar 2.11.2

5.3 Linker (MinGW)

Name	g++.exe
Version	Tested on 5.1.3

5.4 Compiler (VS2005)

Name	cl.exe
Version	Tested on Version 14.00.50727.762

Options

`/Z7` Include debug information

`/Od` No optimizations

`/EHsc` Exception handling

`/D_WIN32_WINNT=0x0400` Windows NT upwards

`/D_CRT_SECURE_NO_DEPRECATE` Eliminate deprecation warnings

5.5 Librarian (VS2005)

Name	lib.exe
Version	Tested on Version 8.00.50727.762

5.6 Linker (VS2005)

Name	cl.exe
Version	Tested on Version 8.00.50727.762

Options

`winmm.lib user32.lib ws2_32.lib Advapi32.lib` Libraries

5.7 Compiler (VS2008)

Name	cl.exe
Version	Tested on Version 15.00.30729.01

Options

`/Z7` Include debug information

`/Od` No optimizations

`/EHsc` Exception handling

`/D_WIN32_WINNT=0x0400` Windows NT upwards

`/D_CRT_SECURE_NO_DEPRECATE` Eliminate deprecation warnings

5.8 Librarian (VS2008)

Name	lib.exe
Version	Tested on Version 9.00.30729.01

5.9 Linker (VS2008)

Name	cl.exe
Version	Tested on Version 9.00.30729.01

Options	
`winmm.lib user32.lib ws2_32.lib Advapi32.lib`	Libraries

5.10 Debugger

An application built for the Virtual ECU port is a normal target executable. Any compatible debugger may be used for debugging.

6 Hardware

6.1 Supported Devices

This port of RTA-OS3.0 has been developed to work with the following target:

Name:	Microsoft Windows
Device:	Virtual ECU

The following variants of the Virtual ECU are supported:

- MinGW
- VS2005
- VS2008

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

6.2 Register Usage

6.2.1 Initialization

RTA-OS3.0 requires the following registers to be initialized to the indicated values before `StartOS()` is called.

Register	Setting
none	Initialization is done by the VRTA startup code

6.2.2 Modification

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

Register	Notes
none	Not applicable

6.3 Interrupts

This section explains the implementation of RTA-OS3.0's interrupt model on the Virtual ECU.

6.3.1 Interrupt Priority Levels

Interrupts execute at an interrupt priority level (IPL). RTA-OS3.0 standardizes IPLs across all targets. IPL 0 indicates task level. IPL 1 and higher indicate an interrupt priority. It is important that you don't confuse IPLs with task priorities. An IPL of 1 is higher than the highest task priority used in your application.

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

IPL	ICU	Description
0	0	User (task) level
1-255	1-255	Category 1 and 2 level

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

6.3.2 Allocation of ISRs to Interrupt Vectors

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

Address	Category 1	Category 2
1-32	✓	✓

6.3.3 Vector Table

When “Supress Vector Table Generation” is undefined or is configured as FALSE then **rtaosgen** generates an interrupt vector table for you automatically. You will usually specify that RTA-OS3.0 generates the interrupt vector table automatically.

You are responsible for placing the generated vector table at the correct base address. The following table shows the section (or sections) that need to be located and the associated valid base address:

Section	Valid Addresses
none	Initialization is done by the VRTA startup code.

6.3.4 Writing Category 1 Interrupt Handlers

Category 1 interrupt service routines (ISRs) must correctly handle the interrupt context themselves. RTA-OS3.0 provides the macro CAT1_ISR that expands to the interrupt control directive required by the appropriate Windows C compiler to indicate that a function requires appropriate code to save and restore the interrupt context to be generated.

A Category 1 ISR therefore has the following structure:

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

6.3.5 Writing Category 2 Interrupt Handlers

Category 2 ISRs are provided with a C function context by RTA-OS3.0, since the RTA-OS3.0 kernel handles the interrupt context itself. The handlers are written using the ISR() macro as shown below:

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

You must not insert a return from interrupt instruction in such a function. The return is handled automatically by RTA-OS3.0.

6.3.6 Default Interrupt

The 'default interrupt' is intended to be used to catch all unexpected interrupts. All unused interrupts have their interrupt vectors directed to the named routine that you specify. The routine you provide is not handled by RTA-OS3.0 and must correctly handle the interrupt context itself. The handler must use the CAT1_ISR macro in the same way as a Category 1 ISR (see Section 6.3.4 for further details).

6.4 Memory Model

The following memory models are supported:

Model	Description
32 bit	VRTA is a 32-bit Windows application

6.5 Processor Modes

RTA-OS3.0 can run in the following processor modes:

Mode	Notes
Standard	VRTA runs as a standard Windows executable

6.6 Stack Handling

RTA-OS3.0 uses a single stack for all tasks and ISRs.

No special stack configuration is required for the VRTA port.

7 Performance

This chapter provides detailed information on the functionality, performance and memory demands of the RTA-OS3.0 kernel. RTA-OS3.0 is highly scalable. As a result, different figures will be obtained when your application uses different sets of features. The figures presented in this chapter are representative for the VRTA port based on the following configuration:

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

7.1 Measurement Environment

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

Device	VS2008 on Pentium class PC
CPU Clock Speed	2000.0MHz
Stopwatch Speed	2000.0MHz

7.2 Memory Consumption

7.2.1 OS Object RAM and ROM Usage

Each OS object requires ROM and/or RAM. The following table gives the ROM and/or RAM requirements (in bytes) for each OS object in the RTA-OS3.0 Component. Note that object sizes will vary depending on the project configuration and compiler packing issues.

Object	ROM	RAM
Alarm	2	12
Cat 2 ISR	8	0
Counter	20	4
CounterCallback	4	0
ExpiryPoint	3.5	0
Resource	8	4
ScheduleTable	16	16
Task	16	0

7.2.2 Stack Usage

The amount of stack used by each Task/ISR in RTA-OS3.0 is equal to the stack used in the Task/ISR body plus the context saved by RTA-OS3.0. The size of the run-time context saved by RTA-OS3.0 depends on the Task/ISR type and the exact system configuration. The only reliable way to get the correct value for Task/ISR stack usage is to call the Os_GetStackUsage() API function.

Note that because RTA-OS3.0 uses a single-stack architecture, the run-time contexts of all tasks reside on the same stack and are recovered when the task terminates. As a result, run-time contexts of mutually exclusive tasks (for example, those that share an internal resource) are effectively overlaid. This means that the worst case stack usage of the can be significantly less than the sum of the worst cases of each object on the system. The RTA-OS3.0 tools automatically calculate the total worst case stack usage for you and present this as part of the configuration report.

7.2.3 Library Module Sizes

The RTA-OS3.0 kernel is demand linked. This means that each API call is placed into a separately linkable module. The following table lists the section sizes for each module (in bytes) for RTA-OS3.0 in standard status.

You can ignore modules that start with test_ or target_ - these are an artefact of the generation process.

Library Module	.bss	.data	.rdata	.text
ActivateTask				128
AdvanceCounter				7
CancelAlarm				106
ChainTask				106
ClearEvent				35
DisableAllInterrupts	4			25
DispatchTask				187
EnableAllInterrupts				19

Library Module	.bss	.data	.rdata	.text
GetActiveApplicationMode				10
GetAlarm				190
GetAlarmBase				56
GetCounterValue				47
GetElapsedCounterValue				105
GetEvent				35
GetExecutionTime				45
GetISRID				10
GetIsrMaxExecutionTime				45
GetIsrMaxStackUsage				45
GetResource				79
GetScheduleTableStatus				48
GetStackSize				20
GetStackUsage				45
GetStackValue				25
GetTaskID				18
GetTaskMaxExecutionTime				45
GetTaskMaxStackUsage				45
GetTaskState				61
GetVersionInfo				51
Idle				7
InShutdown				5
IncrementCounter				15
NextScheduleTable				187
Os_Cfg	617		648	314
Os_Cfg_Counters			728	6935
Os_SetJmp				60
Os_Stack		4		91
Os_fsb				4
Os_vectors			388	
Os_wrapper				69
ReleaseResource				72
ResetIsrMaxExecutionTime				35
ResetIsrMaxStackUsage				35
ResetTaskMaxExecutionTime				35
ResetTaskMaxStackUsage				35
RestartOS				
ResumeAllInterrupts				35
ResumeOSInterrupts				35
Schedule				87
SetAbsAlarm				175

Library Module	.bss	.data	.rdata	.text
SetEvent				35
SetRelAlarm				201
ShutdownOS				48
StackOverrunHook				15
StartOS	1			52
StartScheduleTableAbs				199
StartScheduleTableRel				158
StopScheduleTable				96
SuspendAllInterrupts	8			47
SuspendOSInterrupts	8			56
TerminateTask				7
TestErrorHook				13
ValidateCounter				58
ValidateISR				58
ValidateResource				58
ValidateScheduleTable				58
ValidateTask				58
WaitEvent				35
target				293
target_devices	41628	108	233	858
target_get_stopwatch				10
target_test_support		33		154
test_support		18		128
vrtaAppMain				38
vrtaCore	101	25	420	2663
vrtaLoggerDevice		46	259	3387
vrtaSampleDevices		393	1455	7263

7.3 Execution Time

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

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

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

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

- at 150MHz = 50/150 = 0.333μs

While every effort is made to measure execution times using a stopwatch running at the same rate as the CPU clock, this is not always possible on the target hardware. If the stopwatch runs slower than the CPU clock, then when RTA-OS3.0 reads the stopwatch, there is a possibility that the time read is less than actual amount of time that has elapsed due to the difference in resolution between the CPU clock and the stopwatch (the *RTA-OS3.0 User Guide* provides further details on the issue of uncertainty in execution time measurement).

The figures presented in Section 7.3.1 have an uncertainty of 0 CPU cycle(s).

7.3.1 Context Switching Time

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

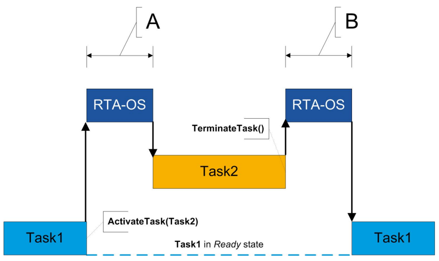
Interrupt latency is the time between an interrupt request being recognized by the target hardware and the execution of the first instruction of the user provided handler function:

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

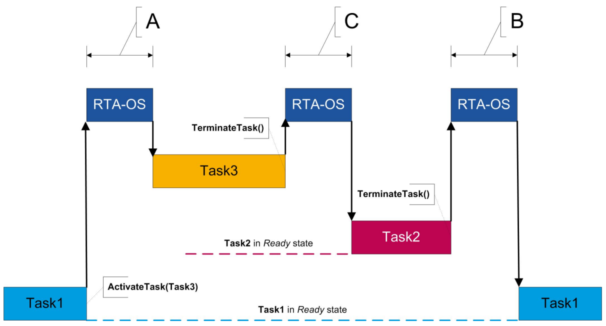
For Category 2 ISRs this is the time required for the hardware to recognize the interrupt plus the time required by RTA-OS3.0 to set-up the context in which the ISR runs.

Figure 7.1 shows the measured context switch times for RTA-OS3.0.

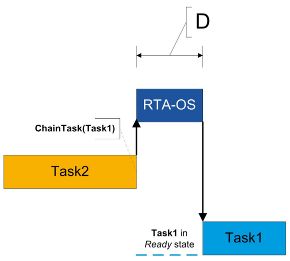
Switch	Key	Execution Time (CPU Cycles)
Task activation	A	- not yet available -
Task termination with resume	B	- not yet available -
Task termination with switch to new task	C	- not yet available -
Chaining a higher priority task	D	- not yet available -
Waiting for an event resulting in transition to the WAITING state	E	- not yet available -
Setting an event results in task switch	F	- not yet available -
Non-preemptive task offers a preemption point (co-operative scheduling)	G	- not yet available -
Releasing a resource results in a task switch	H	- not yet available -
Entering a Category 2 ISR	I	- not yet available -
Exiting a Category 2 ISR and re-suming the interrupted task	J	- not yet available -
Exiting a Category 2 ISR and switching to a new task	K	- not yet available -
Entering a Category 1 ISR	L	- not yet available -



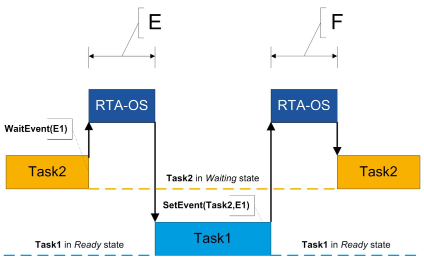
(a) Task activated. Termination resumes preempted task.



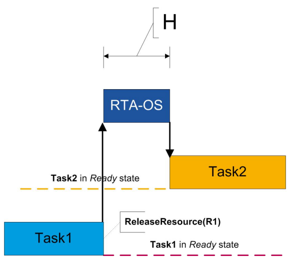
(b) Task activated. Termination switches into new task.



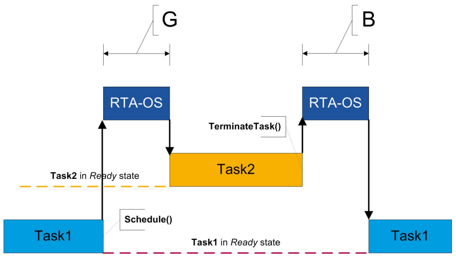
(c) Task chained.



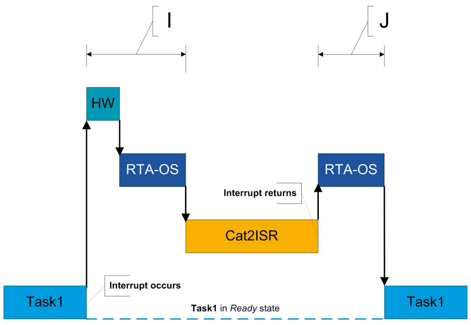
(d) Task waits. Task is resumed when event set.



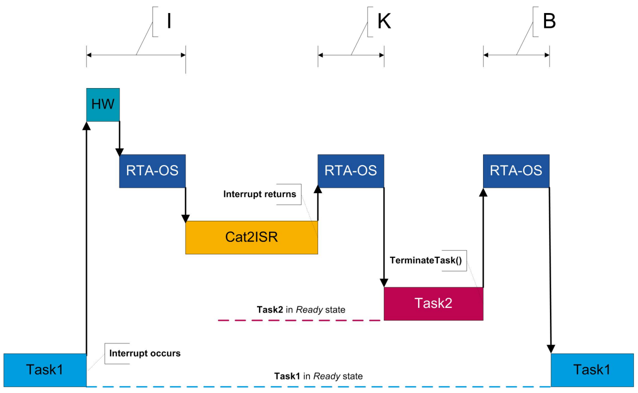
(e) Task switch when resource is released.



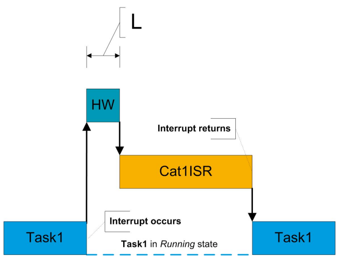
(f) Request for scheduling made by non-preemptive task.



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



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



(i) Category 1 interrupt entry.

Figure 7.1: Context Switching

8 Finding Out More

Additional information about VRTA-specific parts of RTA-OS3.0 can be found in the following manuals:

***RTA-OS3.0 VRTA Release Note*.** This document provides information about the VRTA port plug-in release, including a list of changes from previous releases and a list of known issues.

***RTA-OS3.0 Virtual ECU User Guide*.** This guide explains how to use the Virtual ECU environment included with the VRTA port plug-in.

Information about the port-independent parts of RTA-OS3.0 can be found in the following manuals:

***RTA-OS3.0 Getting Started Guide*.** This document explains how to install RTA-OS3.0 tools and describes the underlying principles of the operating system

***RTA-OS3.0 Reference Guide*.** This guide provides a complete reference to the API, programming conventions and tool operation for RTA-OS3.0.

***RTA-OS3.0 User Guide*.** This guide shows you how to use RTA-OS3.0 to build real-time applications.

9 Contacting ETAS

9.1 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 9.2.

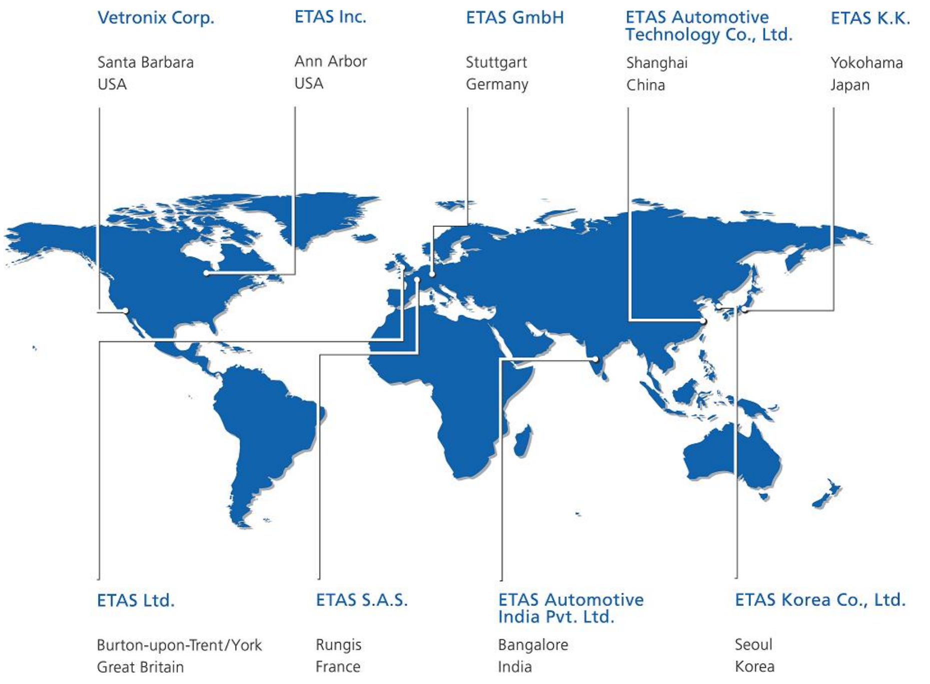
The best way to get technical support is by email. Any problems or questions should be sent to: rta.hotline.uk@etas.com

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.

9.2 General Enquiries



Europe

Excluding France, Belgium, Luxembourg, United Kingdom and Scandinavia

ETAS GmbH		
Borsigstrasse 14	Phone:	+49 711 89661-0
70469 Stuttgart	Fax:	+49 711 89661-300
Germany	E-mail:	sales.de@etas.com
	WWW:	www.etas.com

France, Belgium and Luxemburg

ETAS S.A.S.		
1, place des États-Unis	Phone:	+33 1 56 70 00 50
SILIC 307	Fax:	+33 1 56 70 00 51
94588 Rungis Cedex	E-mail:	sales.fr@etas.com
France	WWW:	www.etas.com

United Kingdom and Scandinavia

ETAS Ltd.		
Studio 3, Waterside Court	Phone:	+44 1283 54 65 12
Third Avenue, Centrum 100	Fax:	+44 1283 54 87 67
Burton-upon-Trent	E-mail:	sales.uk@etas.com
Staffordshire DE14 2WQ	WWW:	www.etas.com
United Kingdom		

USA

ETAS Inc.		
3021 Miller Road	Phone:	+1 888 ETAS INC
Ann Arbor	Fax:	+1 734 997-9449
MI 48103	E-mail:	sales.us@etas.com
USA	WWW:	www.etas.com

Japan

ETAS K.K.		
Queen’s Tower C-17F	Phone:	+81 45 222-0900
2-3-5, Minatomirai, Nishi-ku	Fax:	+81 45 222-0956
Yokohama 220-6217	E-mail:	sales.jp@etas.com
Japan	WWW:	www.etas.com

Korea

ETAS Korea Co. Ltd.		
4F, 705 Bldg. 70-5	Phone:	+82 2 5747-016
Yangjae-dong, Seocho-gu	Fax:	+82 2 5747-120
Seoul 137-889	E-mail:	sales.kr@etas.com
Korea	WWW:	www.etas.com

P.R.China

ETAS (Shanghai) Co., Ltd.		
2404 Bank of China Tower	Phone:	+86 21 5037 2220
200 Yincheng Road Central	Fax:	+86 21 5037 2221
Shanghai 200120	E-mail:	sales.cn@etas.com
P.R. China	WWW:	www.etas.com

India

ETAS Automotive India Pvt. Ltd.		
No. 690, Gold Hill Square, 12F	Phone:	+91 80 4191 2585
Hosur Road, Bommanahalli	Fax:	+91 80 4191 2586
Bangalore, 560 068	E-mail:	sales.in@etas.com
India	WWW:	www.etas.com

Index