RTA-OS3.1

Analysis Visualizer 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 2008-2010 ETAS GmbH, Stuttgart.

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

Document: 10483-AVUG-1.0.0

Contents

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

2 Introduction **11**
- 2.1 Working with the Analysis Visualizer 12

3 Introduction to Schedulability Analysis **14**
- 3.1 What is “real-time” anyway? 14
- 3.2 Testing for Real-Time 14
- 3.3 Analyzing for Real-Time 15
  - 3.3.1 Utilization-Based Schedulability Analysis 15
- 3.4 Deadline Monotonic Analysis 16
  - 3.4.1 Terminology 17
  - 3.4.2 Calculating Response Times 18
  - 3.4.3 Blocking 21
- 3.5 Practical Considerations 22
  - 3.5.1 Operating System Constraints for Analysis . . . 22
  - 3.5.2 Optimism & Pessimism 24
- 3.6 Summary 25

4 Getting Started **26**
- 4.1 Summary 32

5 Basic Modeling **34**
- 5.1 Defining the Kernel 34
- 5.2 Timebases 35
  - 5.2.1 The Stopwatch Timebase 36
  - 5.2.2 Non-Time Timebases 38
  - 5.2.3 Stopwatch Conversions 38
  - 5.2.4 Hints for using timebases 40
- 5.3 Modeling Executable Objects 41
  - 5.3.1 Tasks 43
  - 5.3.2 ISRs 46
  - 5.3.3 The Idle Mechanism 47
- 5.4 Accounting for Blocking 48
  - 5.4.1 Standard Resources 48
  - 5.4.2 Internal Resources 50
  - 5.4.3 Interrupt Masking 50
- 5.5 Modeling Timing Relationships with Transactions 51
  - 5.5.1 How transactions are used 52
  - 5.5.2 What transactions are required? 54
- 5.6 Timelines 55
  - 5.6.1 Arrivalpoint Analysis Clauses 57

5.6.2 Periodic Timelines 57
5.6.3 Single-Shot Timelines 59
5.7 Activators 61
5.7.1 Activator declaration 61
5.8 Timeline Transactions 62
5.9 Bursting Transactions 65
5.9.1 Multiple Arrival Rules 67
5.9.2 Specifying that something happens just once . 68
5.10 Modeling Alarms 69
5.10.1 Reducing Pessimism for a single alarm 70
5.10.2 Reducing pessimism for multiple alarms 72
5.10.3 Alarms that occur once 73
5.11 Modeling Schedule Tables 74
5.11.1 Using Sequential Timelines 74
5.11.2 Using Periodic Timelines 76
5.11.3 Synchronized Schedule Tables 77
5.11.4 Schedule Table Transactions 78
5.12 Auto-started Tasks 79
5.13 The Idle Mechanism 80
5.14 A Note on Deadlines 81
5.15 Summary 81

6 Advanced Modeling 83
6.1 Arbitrary Deadlines and Critical Execution Time 83
6.2 Multiple Profiles 85
6.2.1 Identifying and Referencing Multiple Profiles . . 86
6.2.2 Resource Considerations 87
6.2.3 Restrictions on Profiles and Transactions 88
6.3 Handling Queuing and Buffering 88
6.3.1 Queued Task Activation 89
6.3.2 Buffered Interrupts 92
6.4 Co-operative Tasks 95
6.5 Modeling Extended Tasks 98
6.5.1 Re-factoring the application 98
6.5.2 Partial analysis 99
6.6 Release Delay & Jitter 100
6.7 Response Delay 102
6.8 Accounting for the OS Overheads 103
6.8.1 Interrupt Recognition 103
6.8.2 OS Latencies 103
6.9 Telling the Analysis Visualizer more 105
6.9.1 Analysis Only Transactions 105
6.9.2 Removing interrupt pessimism 106
6.9.3 Non-periodic timebases 108
6.10 Summary 110

7 Performing Analysis **111**
7.1 Schedulability Analysis 111
7.1.1 Running the Analysis 112
7.1.2 Schedulability Analysis Reports 112
7.1.3 Unschedulable Objects 113
7.1.4 Utilization Greater Than 100% 120
7.1.5 Indeterminate Objects 122
7.2 Sensitivity Analysis 125
7.2.1 Performing Sensitivity Analysis 125
7.2.2 Schedulability Analysis Reports 126
7.2.3 Sensitivity of the Idle Mechanism 131
7.3 Priority Optimization 133
7.3.1 Running Priority Optimization 133
7.3.2 Priority Optimization Reports 134
7.3.3 Controlling the Priority Optimization Algorithm 135
7.4 Clock Optimization 136
7.4.1 Running Clock Optimization 137
7.4.2 Clock Optimization Report 137
7.5 Summary 139

8 Tutorials **140**
8.1 Critical execution times and deadlines 140
8.1.1 Example 140
8.1.2 Accounting for Overheads 147
8.2 Execution profiles 147
8.2.1 Example 150
8.3 Shared Resources and Blocking 160
8.3.1 Example 160
8.3.2 Exercise 168
8.4 Periodic timelines and bursting transactions 168
8.4.1 Example 1 170
8.4.2 Example 2 171
8.5 Looping and re-triggering behavior 174
8.5.1 Example 1 175
8.5.2 Example 2 177
8.5.3 Exercise 179
8.6 Allocating priorities 179
8.6.1 Example 180
8.7 Changing processor frequency 187
8.7.1 Example 187

9 Configuration Language Reference 190
9.1 Overview of syntax 190
9.1.1 Notation 190
9.1.2 Declaration rules 191
9.2 Base definitions 192
9.2.1 Time definitions 192
9.2.2 Executable object profiles 193
9.3 Operating system environment definitions 194
9.4 Resource declaration 195
9.5 Timebase declarations 196
9.6 Timebase conversion 198
9.7 Task declarations 199
9.7.1 Task execution profile 200
9.8 Priority constraints 203
9.9 Non-preemption group declarations 204
9.10 Interrupt declarations 205
9.10.1 Interrupt execution profile 206
9.11 Timeline declarations 208
9.11.1 Sequential timelines 208
9.11.2 Periodic timelines 210
9.12 Activator declarations 212
9.13 Transaction declarations 213
9.13.1 Timeline transaction 213
9.13.2 Bursting transaction 213
9.14 Arbitration order 216
9.15 System timing values 217
9.16 Interrupt recognition 218
9.17 Task priority order 219
9.18 Reserved words 219

10 Configuration Language Pre-processor Reference 221
10.1 Preprocessor syntax 221
10.1.1 File inclusion 221
10.1.2 Macro definition 221
10.1.3 Macro undefine 222
10.1.4 Macro expansion 222
10.1.5 Info 222
10.1.6 Warn 222
10.1.7 Error 223
10.1.8 Fatal 223
10.1.9 ifdef 223
10.1.10 ifndef 224
10.1.11 Compatibility with the C preprocessor 224
10.2 Examples of Common Usage 224
10.2.1 Common usage 224
10.2.2 Nesting 228

10.2.3 Precedence order 228
10.2.4 Order of evaluation 228
10.2.5 Macro indirection 229
10.2.6 String concatenation 230
10.2.7 Compatibility with the C preprocessor 231

11 Command Line **233**
11.1 Options 233
11.2 Examples 234

12 Error Codes **236**
12.1 Fatal Messages 237
12.2 Error Messages 249
12.3 Warning Messages 286
12.4 Information Messages 293

13 Finding out more **295**

14 Glossary **296**

15 Contacting ETAS **304**
15.1 Technical Support 304
15.2 General Enquiries 304
15.2.1 ETAS Global Headquarters 304
15.2.2 ETAS Local Sales & Support Offices 304

1 Welcome

Welcome to the Analysis Visualizer!

In this guide you will find out how to build a timing model of an RTA-OS3.x application, perform schedulability analysis on the model using the Analysis Visualizer and interpret the results. Schedulability analysis is a mathematical technique used to prove that an application meets all of its deadlines.

This guide provides a general introduction to the RTA-OS3.x Analysis Visualizer. It describes the basic concepts and functionality of the Analysis Visualizer and gives examples of how to model some fundamental system behavior using the Analysis Visualizer configuration language.

The guide is ordered as follows:

Chapter 2 provides a quick overview of the analysis capabilities of the Analysis Visualizer and how you might use them.

Chapter 3 explains the theory which underlies the Analysis Visualizer. It is useful to understand this in order to understand why the Analysis Visualizer needs to be told different types of information.

Chapter 4 provides a quick tutorial introduction to modeling real-time systems and analyzing them with the Analysis Visualizer.

Chapters 5 and 6 explain how to model RTA-OS3.x-based applications for analysis with the Analysis Visualizer. For basic modeling, it is only required to understand Chapter 5. Chapter 6 can be skipped if necessary.

Chapter 7 explains how to run analyses, interpret the results and fix any problems that may be highlighted.

Chapter 8 provides a set of tutorials.

Chapters 9 to 11 provides a complete technical reference for the Analysis Visualizer, including the configuration language, the configuration language pre-processor, command line options and error codes.

Chapter 13 provides a list of further reading material.

Chapter 14 provides a glossary of terms used in the guide.

1.1 About You

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

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

The intended audience of this manual is designers, developers and engineers who are using RTA-OS3.x to develop real-time systems and wish to demonstrate that their systems are schedulable.

You are assumed to understand the concepts of RTA-OS3.x. In particular, you should have read the *User Guide* and be familiar with concepts it explains before you read this guide.

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 API call names all appear in a `monospaced typeface`.
See Section 1.2.	Hyperlinks through the document are shown in blue letters.

Functionality that is provided in RTA-OS but 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 Introduction

The Analysis Visualizer is a general purpose schedulability analysis tool supplied with RTA-OS3.x. The Analysis Visualizer is suitable for analyzing any system that can be modeled within the constraints imposed by the mathematical theory of schedulability (see Chapter 3).

The Analysis Visualizer is a stand-alone program, called **rtaosanvis**, that takes a model of your OS application as its input and shows the analysis results as its output. The Analysis Visualizer provides the following types of analysis:

Schedulability analysis - Determines whether all tasks and ISRs can meet their deadlines and other constraints in the worst case. If a system cannot meet its deadlines then the analysis will tell you why. The Analysis Visualizer provides extensions to classical schedulability analysis that allow you to determine the maximum buffer sizes required by interrupts. You can use this to guide hardware selection and to determine the maximum activation count for tasks whose activations are queued by the OS.

Sensitivity analysis - Calculates how far a range of timing parameters for each task and ISR can be extended to obtain a system that is just schedulable. Sensitivity analysis can tell you how much longer a task could run for before the system becomes unschedulable, for how long you can hold a resource and how long you can disable interrupts. This can assist you in determining the possibility that the execution time tasks or interrupts can be extended and is an invaluable aid when extending or enhancing a legacy system, without violating its existing performance requirements.

Priority optimization - Allocates task priorities to give the minimum number of preemption levels commensurate with a schedulable system. Research has shown that, even where systems are running at 99% CPU utilization, this technique can be used to modify preemption patterns, which can result in an 8-fold decrease in application stack requirements. This can be used with RTA-OS3.x to gain significant reductions in RAM leading to reduced unit costs.

Clock optimization - Calculates the minimum clock speed for which task priorities can be allocated to give a schedulable system. This can be used to show the slowest speed that the application can run and still meet its deadlines. You can use this functionality to reduce power requirements, to avoid EMC problems or to determine whether cheaper silicon can be used to meet the same performance requirements.

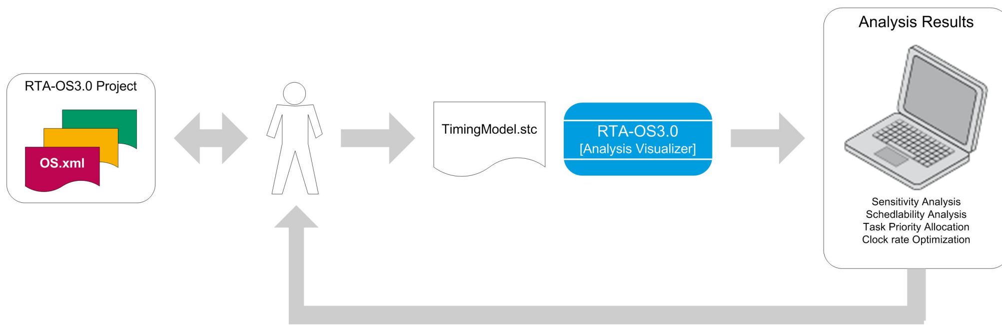
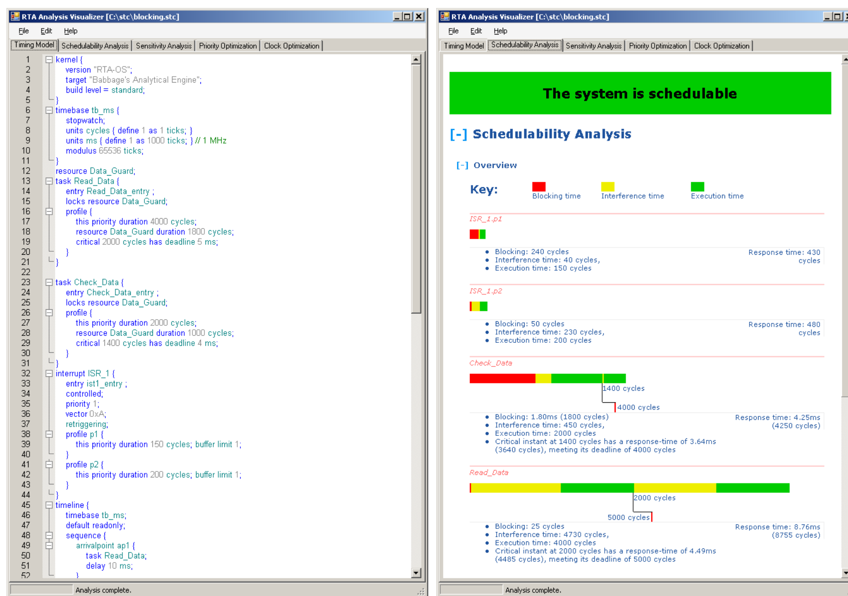


Figure 2.1: User interaction with the Analysis Visualizer



(a) Timing Model (b) Analysis Results

Figure 2.2: Using **rtaosanvis** interactively

2.1 Working with the Analysis Visualizer

The Analysis Visualizer requires a model of the OS configuration to work from. You need to build this yourself. The model is captured using the Analysis Visualizer configuration language, which has a C-like syntax for expressing which OS objects you have and how they interact. The model can be created and edited in the Analysis Visualizer or using a text editor of your choice.

Once you have built your model then it can be analyzed and optimized. You can then use the information you have obtained to improve your RTA-OS3.x configuration. Figure 2.1 shows the basic process.

You can use the Analysis Visualizer in two ways:

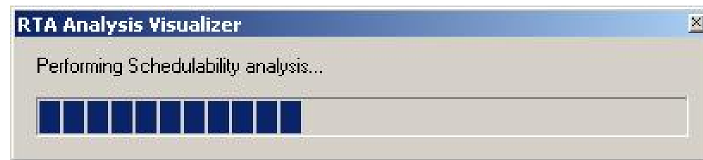


Figure 2.3: Command line progress reporting by **rtaosanvis**

interactive where you use the built-in editor to create your timing model and then the analysis tools to analyze the model. This is shown in Figure 2.2. Results can be exported as HTML files.

command-line where the Analysis Visualizer is given a model and will run the analysis and save the results as a HTML file:

```
C:\working\DevTrunk\Bin>rtaosanvis
    c:\working\UserDocs\STC\examples\blocking.stc
    results.html
```

The Analysis Visualizer will inform you of progress using a progress indicator like that shown in Figure 2.3

In this guide, it is assumed that you are modeling RTA-OS3.x applications and examples focus on how to model an RTA-OS3.x application in the Analysis Visualizer configuration language. However, any OS configuration that can be expressed in the Analysis Visualizer configuration language is amenable to analysis with the Analysis Visualizer.

3 Introduction to Schedulability Analysis

3.1 What is “real-time” anyway?

The meaning of the term ‘real-time’ is frequently misunderstood and often abused. People tend to think that real-time means ‘real-fast’. A true real-time system is one where whatever deadline has been set on an activity taking place, the deadline can always be satisfied. The deadline might be a few nanoseconds or many years, but irrespective of the length of the deadline it is critically important that it can be met, on time, every time. So, in short, real-time is about meeting deadlines.

Deadlines are measured from the point at which work is released into a system. A deadline will be met if the work can be completed *before* the deadline expires. Doing the work itself always consumes some time - this is amount of time is usually called an *execution time*. In the most trivial case, the amount of time required to do work can not be longer than the associated deadline.

If there was only ever one thing to do then things would be easy. Of course, in most systems, there is more that one thing to work on at time, so other jobs may take precedence. This means that when work is released, we may not start it immediately and it may be interrupted while we are doing it. For example, consider a day working in the office when you are interrupted by telephone calls and emails.

It follows that the elapsed time required to *complete* the work may be longer than the time to *do* the work. The elapsed time is known in schedulability analysis terms as the *response time*. If the response time is less than the deadline then the deadline is met.

To be able to show that a deadline is met at runtime we therefore need to work out the longest response time and check that this is less than (or equal to) the required deadline.

3.2 Testing for Real-Time

One approach to solving this problem is to test the system, measuring the response time for each release of work. While this sounds easy, it is extremely difficult in practice. The core problem is that the worst case response time relies on observing the *critical instant*. The critical instant is the point where the worst case release of multiple jobs occurs simultaneously with the worst case time required for each of those jobs to execute. In a real-time system of the type you might build with RTA-OS3.x, this means that the test for response time must ensure that the worst-case phasing of task and interrupt arrivals occur simultaneously with the worst case execution times of each task and interrupt.

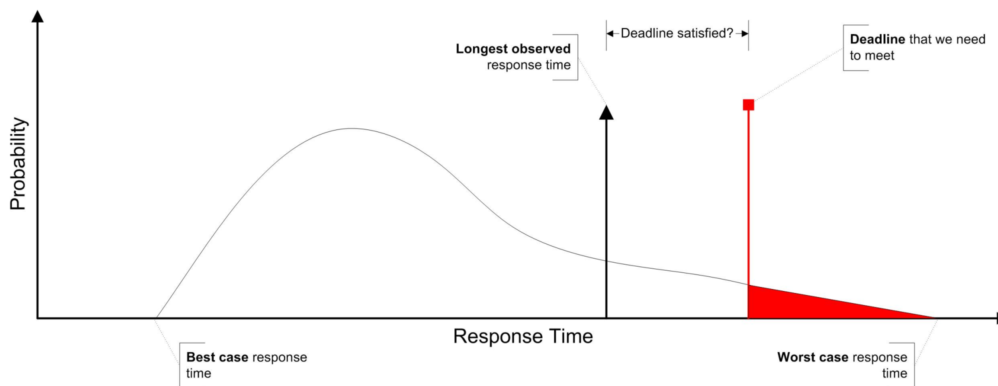


Figure 3.1: Probability density function for response times

Figure 3.1 shows the probability density function of observing the response time of a task/ISR in a preemptive system. The best case time is unlikely to be seen, as too is the worst case. This means that the probability of observing the worst case response time is extremely low. During testing therefore, it is possible to observe a worst case response time that appears to meet a specified deadline but is actually shorter than the real worst case.

This means that there is a significant risk that a system which has been tested for real-time correctness will fail to meet deadlines. This is hardly surprising because testing for real-time relies upon the extremely improbable observation of the worst-case response time.

What is required is an alternative approach that guarantees that the worst-case response time we obtain, and check against the deadline, really is the worst-case. That approach is to determine the response time analytically.

3.3 Analyzing for Real-Time

Schedulability analysis uses mathematics to calculate worst-case response times using knowledge about how long work takes to execute and how often work is released. Recall that with testing the problem is ensuring that the critical instant occurs. Schedulability analysis avoids this problem by calculating the *critical instant* (or likely candidates for the critical instant) and then calculating the response time for every object.

3.3.1 Utilization-Based Schedulability Analysis

An elementary form of schedulability analysis is to check *utilization*, which is the amount of load placed on the CPU.

The utilization of task i in a system can be calculated by dividing its execution time (C_i) by its arrival rate (T_i) and expressing the result as a percentage. By

summing the utilization of all tasks and ISRs in a system the overall utilization can be calculated.

If the CPU is more than 100% loaded then it is obvious that some tasks are going to miss deadlines - there is simply not enough CPU time to do all the work required. However, in 1973, Liu and Leyland proved that a system which satisfied the constraints of Rate Monotonic Analysis (RMA) can meet its deadlines if CPU utilization is less than 69.3%.

The constraints of RMA are:

- No shared resources
- Deadlines are exactly equal to periods
- Unique, static priorities
- Preemptive scheduling
- Tasks with shorter periods/deadlines are given higher priorities
- OS context switching times are zero

The insight provided by Liu and Leyland was that for this type of system, the critical instant occurs when all tasks are released a some notional time zero.

In practice, utilization-based analysis is quite restrictive. The lack of shared resources means that it would not be possible to use this model for practical AUTOSAR OS R3.x systems. While it would be nice is an OS could run in zero time, in practice this is not possible, so there must be some way to account for OS overheads in the schedulability test,

RMA is also unnecessarily pessimistic - there are systems which are schedulable and have significantly higher than 69.3% utilization. If we used RMA as the only guide to schedulability then we would reject systems that are schedulable.

3.4 Deadline Monotonic Analysis

An alternative test for schedulability is Deadline Monotonic Analysis (DMA) which makes the following assumptions about the properties of a system:

Fixed number of tasks. There is a bounded, fixed, number of tasks (in this context, a task is work that needs to be done by the system, in AUTOSAR OS R3.x terms this can be a task or an ISR).

Unique priorities. Tasks have fixed, unique priorities.

No self suspension. Tasks do not suspend themselves during execution.

Independence. Tasks are independent of each other apart from their use of shared protected data. Shared data is protected by a ceiling protocol.

Preemptive Scheduling. Tasks are scheduled preemptively.

DMA is more flexible in the types of systems it can analyze than RMA. In DMA, periods need not be static and deadlines can be shorter than periods (in fact, extensions to basic DMA allow for arbitrary deadlines, including those that are longer than periods). DMA does not require rate-monotonic priority assignment so task priority is not tied to task period as with RMA[1].

In essence, DMA is based on knowing:

1. how often objects are released into the system (e.g. how often do interrupts occur, how often are tasks activated).
2. for how long an object executes.

The analysis approach is a form of divide and conquer - it divides the problem of determining the response time into two simpler problems of identifying arrival patterns and execution times and divides the problem of system schedulability into multiple problems of executable object schedulability.

DMA uses the data supplied to calculate the worst case response time for each executable object. It is then trivial to compare the calculated response time against the required deadline to determine whether or not it is schedulable. If all executable objects in a system are schedulable, then the system as a whole is schedulable.

DMA is the core technology upon which the Analysis Visualizer is based. In this section we look at how basic DMA works because it will help you to understand why you are required to provide the information requested by the Analysis Visualizer and understand how it is used.

3.4.1 Terminology

Figure 3.2 is an annotated time diagram defining the notation used in the analysis.

The term T_i is the *period* of a task i. The period is the minimum time between a task being made ready to run and being made ready again. This model

[1] A Deadline Monotonic *Priority Assignment* would assign higher priorities to tasks with shorter deadlines, however, there is no restriction on priority assignment as far as DMA itself is concerned.

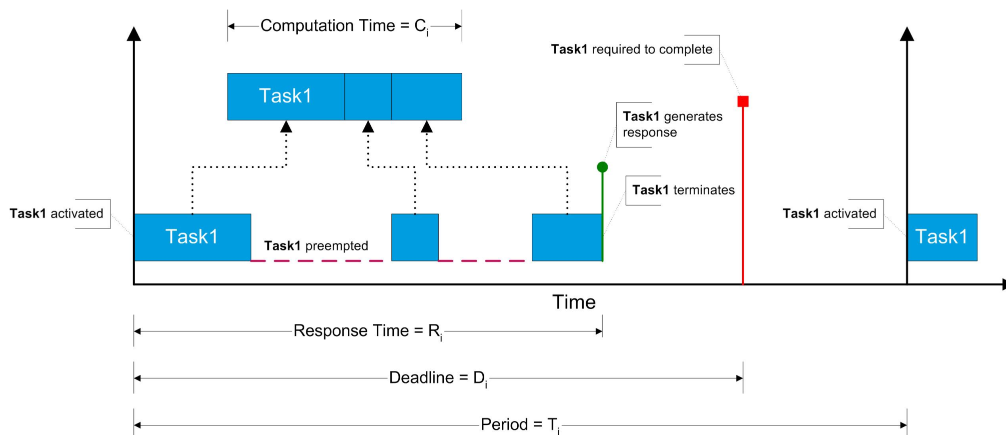


Figure 3.2: DMA Notation

supports both sporadic and strictly periodic tasks. Each time the task i is made ready it may execute for up to C_i processor time. This is known as the worst-case execution time of task i. Note that this time does not include the time for which other tasks and interrupt handlers use the processor: it is the processor time required only by task i. The worst-case response time of a task i is measured from the time the task is made ready to the time the task completes its worst-case execution time C_i. This worst-case response time is denoted R_i. The deadline of a task i is denoted D_i and a task will always meet its deadline if $R_i \leq D_i$.

3.4.2 Calculating Response Times

The basic idea of DMA is to find an equation that will calculate R_i. This time is made up of two times: the time a task takes to execute its own code, and the time it takes for higher priority tasks to execute and finish with the processor. The following equation indicates this:

$$R_i = C_i + I_i \tag{3.1}$$

The term I_i is the preemption time from higher priority tasks and interrupt handlers and is called the *interference*. The problem now focuses on how to find the interference time.

Figure 3.3 shows a task i being preempted by a higher priority task k. Task k initially preempts the task then preempts again when it is made ready for a second time. By the time task k comes back for a third time the lower priority task has already finished, and so this time there is no interference. The number of times that a given task k can preempt a task i while task i is ready is given by :

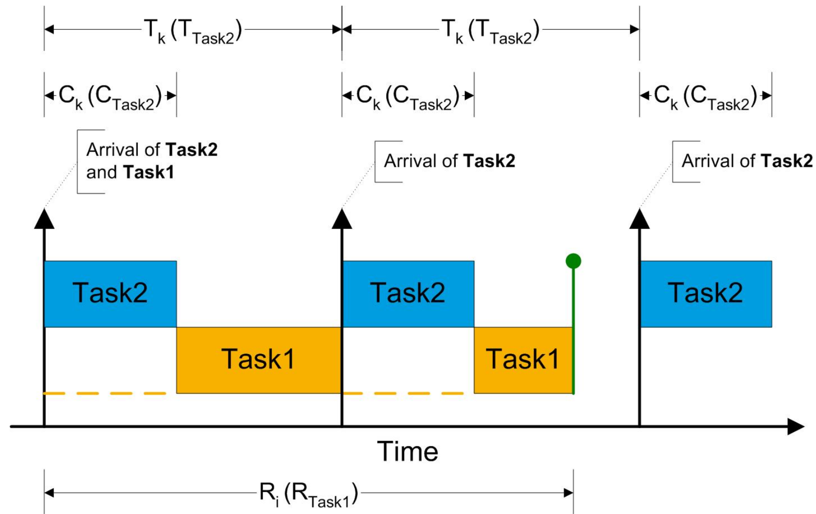


Figure 3.3: Impact of preemption on a lower priority task

$$\left\lceil \frac{R_i}{T_k} \right\rceil \tag{3.2}$$

The total time taken by a higher priority task k when it preempts and executes is simply the number of times it preempts multiplied by its execution time:

$$\left\lceil \frac{R_i}{T_k} \right\rceil C_k \tag{3.3}$$

The symbol $\lceil\rceil$ is the ceiling function that performs a "round up" function. For example $\lceil 1.2 \rceil = 2$, $\lceil 2 \rceil = 2$ and $\lceil 2.1 \rceil = 3$.

To calculate the total interference from *all* higher priority tasks we simply calculate the interference for each higher priority task and then add them together. Assuming that hp_i is the set containing all higher priority tasks (and ISRs) in the system, the calculation can be expressed by the following equation:

$$I_i = \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k \tag{3.4}$$

This can be substituted into Equation 3.1 to give the basic DMA response time equation:

$$R_i = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k \tag{3.5}$$

Task/ISR	Priority	T	C	D
i1	Highest	10ms	0.50ms	3ms
t1		3ms	0.50ms	3ms
t2		6ms	0.75ms	6ms
t3		14ms	1.25ms	14ms
t4	Lowest	14ms	5.00ms	14ms

Table 3.1: ECU Node Specification

You will notice that the term R_i now appears on both sides of the equation - it would appear that we can only find the worst-case response time if we know the worst-case response time! However, we know that R_i will never be less than C_i so this can be solved by forming a recurrence relation:

$$R_i^0 = C_i R_i^{n+1} = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i^n}{T_k} \right\rceil C_k \tag{3.6}$$

This forms the basic DMA equation that can be used to calculate the response time for each task (and ISR) in a system. The recurrence relation will converge on a value for R_i if a value exists. This means that the calculation can halt if the calculated response time is the same as the previously calculated response time, i.e. when:

$$R_i^n = R_i^{n-1} \tag{3.7}$$

Furthermore, because the equation does not oscillate around a solution, as soon as the response times begin to diverge they will continue to diverge. The observation of divergence can therefore also act as a termination condition:

$$R_i^n - R_i^{n-1} > R_i^{n-1} - R_i^{n-2} \tag{3.8}$$

It is trivial to checking schedulability by checking that the calculated response times are less than the associated deadlines:

$$\forall i \in Tasks \bullet R_i \leq D_i \tag{3.9}$$

Example

Assume we have a system with the following of tasks and ISRs:

For brevity, we will just calculate the response time for task t4 and check whether it meets its deadline. We start with an initial R estimate of 5ms, i.e.

Step	R_n	I					R_{n+1}
		i1	t1	t2	t3	Total	
1	5	0.5	(2 x 0.5)	0.75	1.25	3.5	8.50
2	8.5	0.5	(3 x 0.5)	(2 x 0.75)	1.25	4.75	9.75
3	9.75	0.5	(4 x 0.5)	(2 x 0.75)	1.25	5.25	10.25
4	10.25	(2 x 0.5)	(4 x 0.5)	(2 x 0.75)	1.25	5.75	10.75
5	10.75	(2 x 0.5)	(4 x 0.5)	(2 x 0.75)	1.25	5.75	10.75

Table 3.2: Worst-Case Response Time for Task t4

the computation time of task t4. Table 3.2 shows the calculation of worst-case response time for Task t4. The calculation stops at Step 5 because it converges on R_i, i.e. $R_i^5 = R_i^4$.

So the worst-case response time of task t4 is 10.75ms. This is less than the deadline of 14ms and proves that task t4 will always meet its deadline in all situations.

3.4.3 Blocking

So far we have assumed that a higher priority task is not delayed by a lower priority one. This means that no lower priority task is allowed to disable interrupts or even share a resource with a higher priority task. Clearly this assumption is unrealistic and we need to do something about it. The way we deal with this is to allow for *blocking time*, denoted B_i, and equal to the time for which the execution of lower priority tasks can delay a given task i.

What we need to do is be able to work out this blocking time. A desirable property to simplify the calculation of blocking time is that a task is blocked at most once. The worst-case blocking can then be calculated by determining the longest blocking time imposed by any lower priority task. The Priority Ceiling Protocol for resource locking achieves exactly this and allows the calculation of blocking time. Priority Ceiling Protocol introduces the notion of *ceilings*. Each resource has a ceiling priority: this is the priority of the highest priority task that can lock the resource. A task can hold several resources at once, but only if they are locked in a nested pattern for example:

```
GetResource(R1);
  GetResource(R2);
  ReleaseResource(R2);
ReleaseResource(R1);
```

We can more formally define the blocking time for a given task i by defining the following terms:

lp_i	the set of tasks with priorities lower than task i
$locks_{k,i}$	the set of resources locked by a task k where the resources have a ceiling priority equal to or higher than the priority of task i.
$t_{k,s}$	the time for which a task k holds a resource s.

The blocking time B_i is defined as follows:

$$B_i = max_{\substack{\forall k \in lp_i \\ \forall s \in locks_{k,i}}} (t_{k,s})$$

Accounting for blocking time in the calculation of a task's response time is easy - because a task is blocked at most once then we just need to add B_i to the equation for the worst-case response time:

$$R_i^{n+1} = B_i + C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i^n}{T_k} \right\rceil C_k$$

3.5 Practical Considerations

For simple system you can do DMA analysis by hand, but as systems become more complex you need a dedicated tool like the Analysis Visualizer. The Analysis Visualizer calculates the worst-case response times for each task and ISR in your application based on the values you provide for C_i and then checks that all the responses meet their deadlines.

3.5.1 Operating System Constraints for Analysis

The Analysis Visualizer analyses a model of a system. It is therefore necessary that the runtime behavior of an OS-based system does not break the constraints of the model - i.e. the OS configuration must be amenable to DMA. This does not mean that *every* system that can be built with the OS *must* be analyzable. The OS may provide features that are not analyzable, but this is not a problem providing those features are not used in a configuration[2]. This is illustrated in Figure 3.4

The constraints of the DMA model of analysis interact with RTA-OS3.x configurations in the following ways:

Fixed number of tasks. DMA requires a static system with known set of tasks and interrupts. This does not impact AUTOSAR OS R3.x as it is a statically configured OS.

[2] Compare this to a MISRA compliance checker. Any valid C program can be compiled by a C compiler, but only the subset of programs that satisfy the MISRA guidelines can be checked successfully.

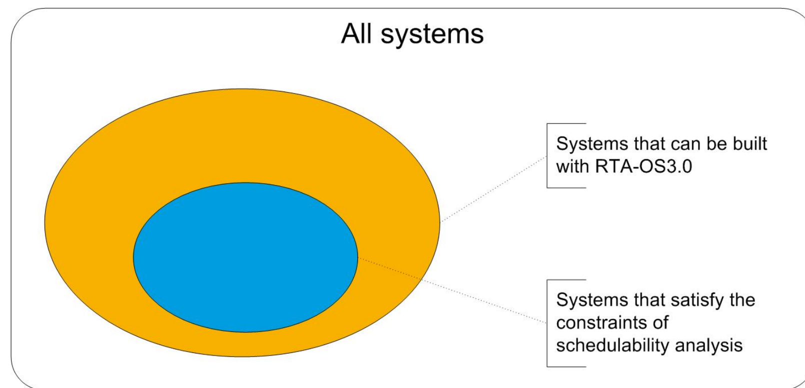


Figure 3.4: Analyzable systems as a subset of buildable systems

Unique priorities If two (or more) objects are released simultaneously then it must be possible to determine the order of execution. This means:

- Tasks must not share priorities. Tasks are allowed to share priority levels, which is what happens when they use AUTOSAR OS R3.x internal resources.
- ISRs should not share priorities where possible.
- When the hardware forces ISR to share priorities, then the arbitration order (i.e. the order in which they are processed by the hardware) needs to be modeled for the Analysis Visualizer.

No self-suspension. DMA does not permit objects to block themselves. This has two impacts for AUTOSAR OS R3.x:

- Extended tasks cannot be analyzed. Ideally, you should not use extended tasks in your system. If you do have extended tasks then they cannot, in general, be analyzed for schedulability. This also means that any tasks of lower priority than the highest priority extended task cannot be analyzed for schedulability (as the interference contributed by the extended tasks cannot be calculated). The Analysis Visualizer does allow you to analyze tasks of higher priority than the highest priority extended tasks subject to accounting for the blocking factors they introduce on higher priority objects. See Section 6.5 for further details.
- The `Schedule()` API call cannot be used to force rescheduling to take place.

Priority Ceiling Protocol. Resources and interrupt locks must be managed according to the priority ceiling protocol. This guarantees that tasks are blocked at most once during their execution and that blocking only occurs at the start of execution. This does not impact AUTOSAR OS R3.x.

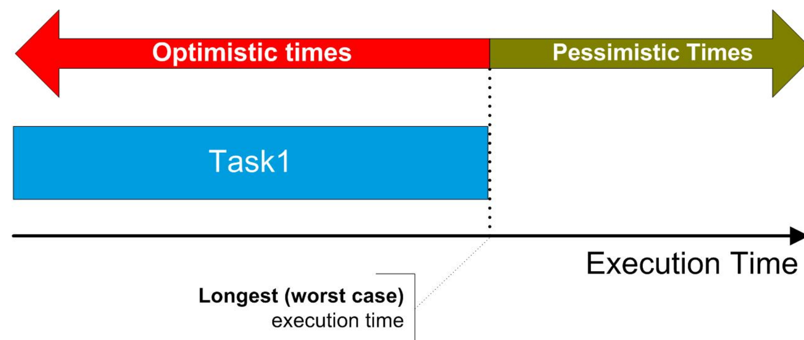


Figure 3.5: Pessimism and optimism

No upwards activation. A task must only activate tasks of lower priority.

3.5.2 Optimism & Pessimism

When you provide data to model the system for analysis, it is important to provide data that is as accurate as possible. Inaccuracy has two effects:

Optimism If you tell the Analysis Visualizer that execution times are shorter that they really are or that arrivals are closer together than they really are then the analysis will be *optimistic*. This means that, even though the Analysis Visualizer will tell you that the system is schedulable then if will fail to meet deadlines at runtime[3]. *You should avoid optimistic figures at all costs if you want to trust the results of the analysis.*

Pessimism If you provide executions times that are longer than normal or arrival rates are more frequent than they really are then he analysis will be pessimistic. This is must less critical - the analysis will report systems as unschedulable when, in fact, they are - so the results will always be safe.

Figure 3.5 shows the relationship between pessimism and optimism for execution times and arrival rates.

If you cannot guarantee that your data is accurate, then supply data that is pessimistic. You can make your data pessimistic by supplying execution times that are longer than the actual execution times, for example. You could also declare delays between releases as shorter than the actual delays.

[3]This is a case of “garbage in, garbage out”.

3.6 Summary

- Real-time is about meeting deadlines.
- The real-time characteristics of an application are difficult to test.
- Real-time behavior can be analyzed off-line using schedulability analysis.
- Analysis requires knowledge about execution times and arrival rates to calculate the worst-case response time for each task and interrupt in a system. If all tasks and ISRs are schedulable then the system is schedulable.
- AUTOSAR OS R3.x-based applications can be designed to fit within the theoretical limits for analyzable systems.
- Data provided for analysis should be as accurate as possible.

4 Getting Started

Before we look in more detail at various areas of functionality we start with a simple example to provide a quick overview of how the Analysis Visualizer operates. You should the Analysis Visualizer and type in any configuration you are shown. Let's consider a preemptive system containing three tasks with following properties:

Task	Period [ms]	Computation time [ms]	Deadline [ms]
Task_3	20	5	20
Task_2	40	10	40
Task_1	80	40	80

The task's period specifies the minimum time between consecutive executions. Computation time is the worst-case execution time from the start of the first instruction in the entry-function to the end of the 'return' instruction. The deadline is the maximum allowable time for the task to complete execution and equals the task's period, i.e. tasks must complete before their next invocation is due. Ignoring any existing overheads, such as task-switching time or task entry-latencies, you can see that this trivial system is just schedulable, with Task_1 finishing execution just in time for its next release. Figure 4.1 shows how the system is scheduled.

All tasks are released at time 0ms. Task_3 has the highest priority, Task_1 the lowest. Therefore Task_3 runs to completion before Task_2 starts (at time 5ms) and Task_2 runs to completion before Task_1 starts (at time 15ms), giving the tasks response times of 5ms and 15ms respectively. Task_1 cannot complete before Task_3 and Task_2 start again and gets preempted by Task_3 at time 20ms, 40ms and 60ms. It also has to wait for Task_2 to complete at time 45ms. However, there is still enough execution time left for Task_1 to complete at time 80ms, the time when all three tasks are released again. The response time of Task_1 is therefore 80ms. This means all three tasks meet their deadlines and the system is schedulable.

One interesting aspect of this system, however, is that it fails the basic schedulability test by Liu and Leyland (1973) because the combined utilization of the system is 100%. This test considers the utilization of sets of tasks where priorities are assigned according to the tasks periods (i.e. shortest period task is assigned the highest priority, etc).

In order to represent this example system in the Analysis Visualizer configuration language you will need to declare the three tasks, a timeline and one transaction for this particular timeline. The timeline defines the sequence in which your tasks will execute and the transaction follows that timeline.

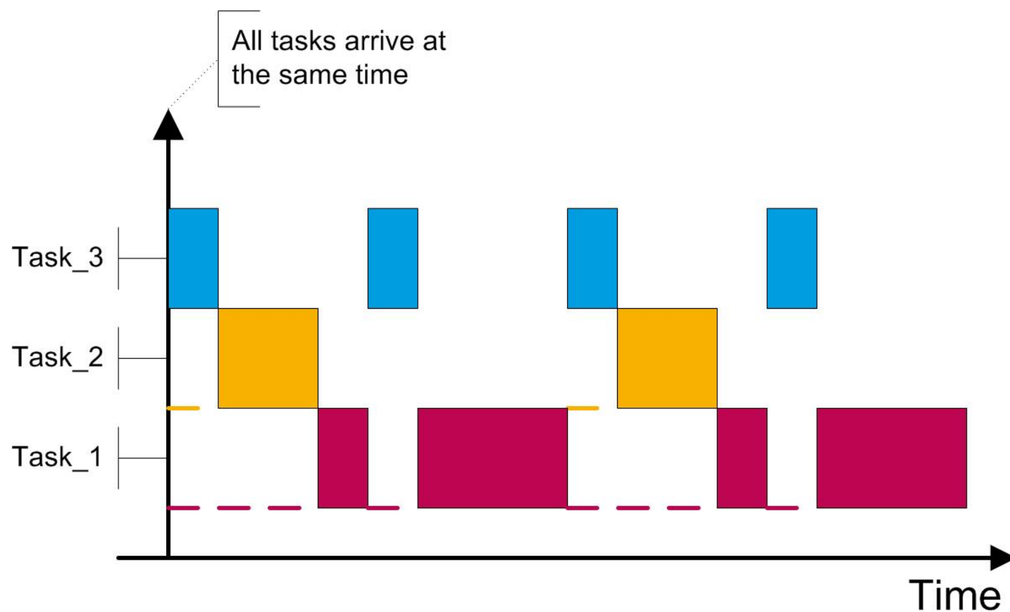


Figure 4.1: Task execution with all tasks arriving at time zero

You start your configuration by specifying the system's environment. The `kernel` clause is usually the first clause in a configuration file and informs the Analysis Visualizer of the RTA-OS3.x version used, the target platform and the build level required. The version and target parameters must be provided and you are free to choose any value you like. Valid values for build level are `standard`, `timing` and `extended` but these do not impact the analysis performed. You configuration should start with the following `kernel` declaration:

```
kernel {
    version "RTA-OS";
    target "Babbage's Analytical Engine";
    build level = standard;
}
```

Every configuration file must have at least the stopwatch timebase declared. As you only need one timebase for this example, declare a millisecond-timebase (we assume the processor clock runs at 100kHz and the stopwatch clock runs at the same rate) directly after the kernel clause:

```
timebase tb_ms {
    stopwatch;
    units cycles { define 1 as 1 ticks; }
    units sec    { define 1 as 100000 ticks; }
    units ms     { define 1000 as 1 sec; }
    modulus 65536 ticks;
}
```

For each task, the task declaration only needs to contain the entry name of the corresponding function and an execution profile which specifies the computation time of the task:

```
task Task_1 {
    entry task1_entry ;
    profile { this priority duration 4000 cycles; }
}

task Task_2 {
    entry task2_entry ;
    profile { this priority duration 1000 cycles; }
}

task Task_3 {
    entry task3_entry ;
    profile { this priority duration 500 cycles; }
}
```

The Analysis Visualizer uses timelines to specify when things happen. The following configuration defines a sequential timeline which describes the timing of the tasks:

```
timeline {
    timebase tb_ms;
    default readonly;

    sequence {
        arrivalpoint ap1 {
            analysis {
                task Task_1;
                task Task_2;
                task Task_3;
                delay 20 ms;
            }
        }
        arrivalpoint {
            analysis {
                task Task_3;
                delay 20 ms;
            }
        }
        arrivalpoint {
            analysis {
                task Task_2;
                task Task_3;
                delay 20 ms;
            }
        }
        arrivalpoint {
```

```
                analysis {
                    task Task_3;
                    delay 20 ms;
                    next ap1;
                }
            }
        }
    }
```

You can see that Task_3 runs every 20ms, Task_2 every 40ms and Task_1 every 80ms.

Finally, you need to tell the Analysis Visualizer that a timeline should be considered for analysis. The Analysis Visualizer uses the notion of a `transaction` to capture this. You will learn more about transactions later, but for now you just need to declare a simple transaction which follows the arrivalpoints in your timeline.

```
transaction t1 {
    start ap1;
}
```

The starting point ap1 is the first arrivalpoint in the timeline defined above, and the transaction will follow the timeline beginning at ap1.

Next, you will need to add a system timings and an interrupt recognition clause. The system timings and interrupt recognition values are target dependent and may differ significantly between different platforms. This tutorial will not use realistic values but only describe the impact certain values can have. In the current example you can ignore any overheads and therefore set every value to zero.

```
system timings { 0; 0; 0; 0; 0; 0; 0; 0; }
interrupt recognition 0 cycles;
```

Finally, you have to specify a task priority order for your system. The Analysis Visualizer requires each task to be allocated a unique priority, so you either add a task priority order clause to your configuration file or select automatic priority allocation (option -p) when running the Analysis Visualizer. For this example you already know that tasks are assigned priorities in deadline-monotonic order, i.e. the task with shortest deadline is assigned the highest priority, etc. Therefore, you can declare the following task priority order:

```
task priority order {
    task Task_3;
    task Task_2;
    task Task_1;
}
```

The structure of your configuration file should now look as follows:

1. Kernel clause
2. One timebase
3. Three tasks (Task_1, Task_2, Task_3)
4. One timeline
5. One transaction (t1)
6. System timings
7. Interrupt recognition
8. Task priority order

After you have saved your configuration file as `getting-started.stc` it is time to check whether the system is schedulable. Select the "Schedulability Analysis" tab in the Analysis Visualizer.

Provided you typed in everything correctly, the Analysis Visualizer should produce output like that shown in Figure 4.2.

All timing values are given in stopwatch cycles and milliseconds. By default 'cycles' is the smallest unit of a stopwatch timebase and, if not defined differently, 1 cycle equals 1 tick. As we expected, all three tasks are schedulable with Task_1 just finishing before its next invocation. The largest blocking time is zero for all three tasks, which is not very surprising, as we ignored all overheads and didn't declare any resource locking.

To get a better understanding on how sensitive the schedule of your system actually is, you should now change one of the delay values in the timeline from 20ms to 19ms and rerun the analysis. This time the Analysis Visualizer reports that the system is not schedulable as shown in Figure 4.3.

The Analysis Visualizer always performs a quick utilization test first to check whether or not utilization is greater than 100%. If it is, then the system cannot be schedulable, so the analysis will fails and the Analysis Visualizer finishes execution. This result, of course, is not surprising, as the original system was already stretched to its limit with a combined utilization of 100%. Therefore, reducing the period to 79ms was not feasible. Now, let's assume the period of 79ms had to stay this way and you would have to reduce your task execution times in order to obtain a schedulable system. The Analysis Visualizer provides special sensitivity analysis for your system which will tell you exactly what changes are needed to make your system schedulable. Select the "Sensitivity Analysis" tab to run sensitivity analysis.

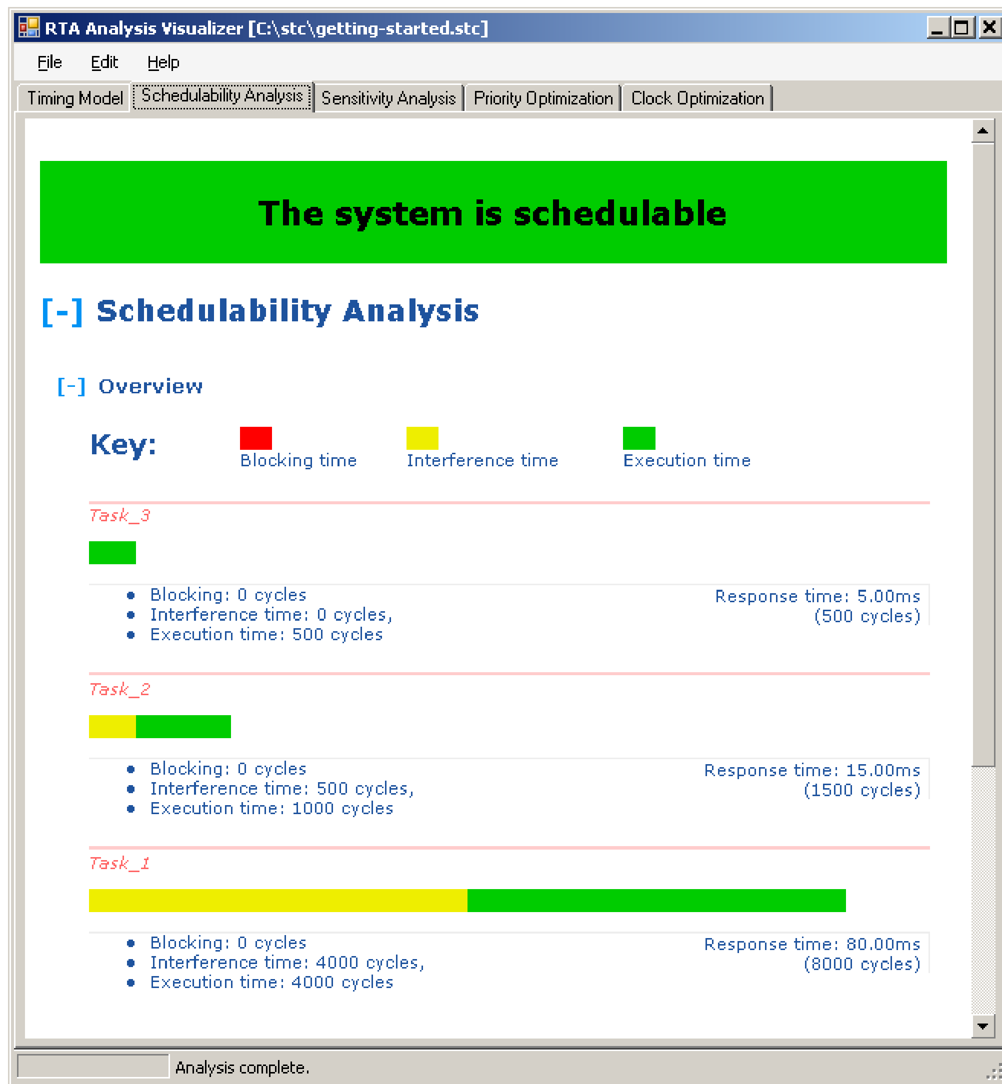


Figure 4.2: Successful schedulability analysis

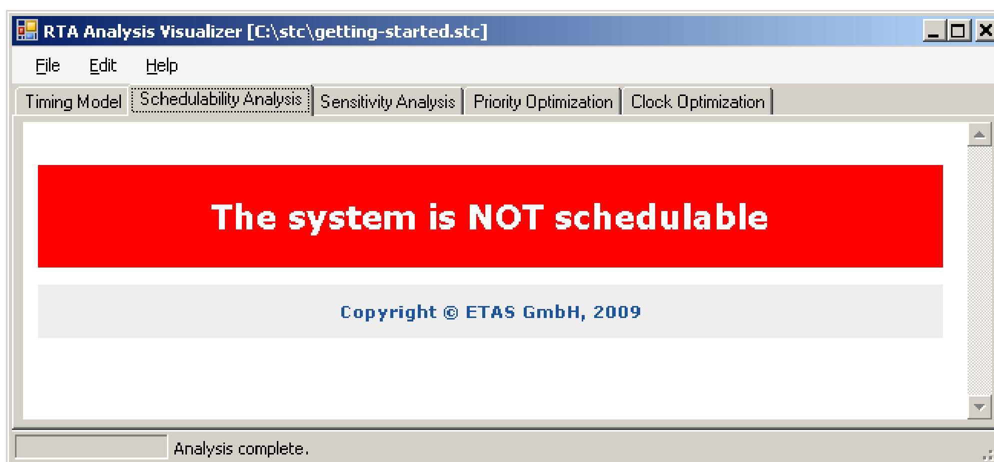


Figure 4.3: Unsuccessful schedulability analysis

The Analysis Visualizer will output the following result shown in Figure 4.4.

For every task the Analysis Visualizer calculates the maximum execution time that, if applied, would allow your system to be schedulable. Alternatively, you can change the processor clock speed and sometimes this might be the only option left to make a system schedulable. Note that if you decide to change task execution times you only need to change one task.

You should now make the suggested changes to the execution time of one of the tasks in your configuration file, then rerun the analysis and check that your system is schedulable.

4.1 Summary

This chapter has shown how a simple, but complete, system can be modeled for analyzed by the Analysis Visualizer. The following chapters explain more about modeling applications. Additional tutorial examples can be found in Chapter 8.

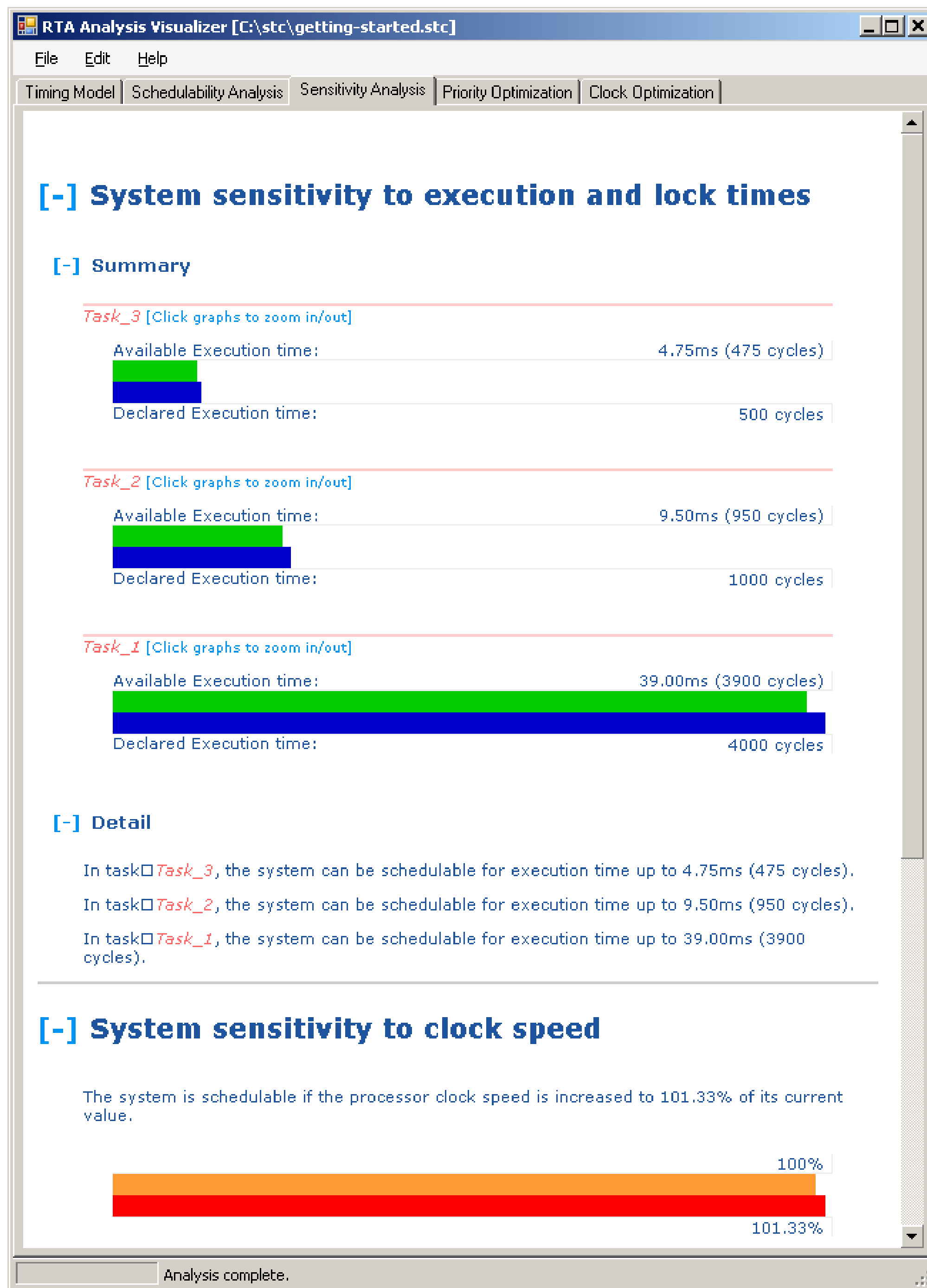


Figure 4.4: Sensitivity analysis

5 Basic Modeling

In this chapter you will learn how to model basic, non-queued, tasks and Category 1 and 2 ISRs. The models you will create will be sufficient for many practical applications. The Analysis Visualizer also provides many advanced features that relax the constraints of classical DMA analysis. Once you have mastered the basic modeling concepts presented in this chapter you should consult Chapter 6 for how to model more sophisticated systems.

To analyze a real-time system you must provide:

- A description of the software architecture in terms of the tasks, interrupts, resources, counters, alarms and schedule tables.
- The execution times for each task and ISR.
- A timing model that defines the timing relationship between executable objects. This defines the periods and deadlines for your application. These are the *release* of time into the system. A task is released when it enters the ready state. An ISR is released when the CPU has recognized the interrupt that causes it
- Target specific timing information (optional).

5.1 Defining the Kernel

The Analysis Visualizer configuration language requires that you provide a `kernel` definition clause. This is not used for analysis.

The kernel clause is the first clause in a configuration file. The version and target parameters must be provided and you are free to choose any value you like.

Valid values for build level are `standard`, `timing` and `extended` but these do not impact the analysis performed. The following example illustrates the syntax :

```
kernel {
    version "RTA-OS";
    target "Babbage’s Analytical Engine";
    build level = standard;
}
```

5.2 Timebases

A timebase defines the units, granularity and range of a counter. A timebase may correspond to a regular time reference that marks the passing of time (for example, a millisecond timer), or to some other kind of application-specific counter (for example a count of the number of times a tooth on a toothed wheel passes in front of a sensor). A counter increment is referred to as a tick.

The value of the timebase ranges from zero up to (but not including) the timebase modulus. The timebase value is always stored as a 32-bit unsigned integer and so the largest timebase modulus is 4294967295 ($2^{32} - 1$).

A timebase is declared in the configuration file as follows:

```
timebase cpu_clock {
    modulus 65536 ticks;
}
```

The above declaration defines a timebase called tb1 that can count 65536 ticks (i.e. it counts ticks from 0 to 65535 inclusive). This is the simplest way of declaring a timebase.

The default unit for a timebase is a tick and a tick can be anything that you want it to be. However, tick values can be assigned symbolic names using a units declaration in the timebase declaration.

```
timebase cpu_clock {
    units s { // A second has 8 million ticks on an 8MHz clock
        define 1 as 8000000 ticks;
    }
    units ms { // There are 1000 milliseconds per second
        define 1000 as 1s;
    }
    units us { // There are 1000 microseconds per millisecond
        define 1000 as 1ms;
    }
    units ns { // There are 1000 nanoseconds per microsecond
        define 1000 as 1us;
    }
    modulus 65536 ticks;
}
```

The above declaration defines a timebase that counts from 0 to 65535. It also defines some units (nanoseconds, `microseconds`, and `seconds`), and maps the units to the timebase by defining one second to be 8,000,000 ticks.

If the conversion between units results in a large rounding error for a constant then the Analysis Visualizer will generate a warning of lost accuracy.

In this example the timebase might correspond to a hardware counter counting from 0 to 65535, driven from the processor clock with a frequency of 8MHz. If the crystal frequency or counter prescale is changed then you only need modify the definition of s.

Your hardware/software implementation of a counter must tick at the rate specified in the configuration file. Otherwise, your definition of units, constants and delays used in timelines (see sub-section 'arrivalpoints and timelines') will be incorrect.

A unit name applies only to the timebase in which it is declared. There is no relationship between the same unit name in different timebases. If you declare the same name in multiple timebases and intend these to represent the same times then you will need to define conversions. The Analysis Visualizer allows this using the "stopwatch" timebase, but it your responsibility to ensure that all stopwatch conversions are correct. For hints about how to do this, see Section 5.2.4.

5.2.1 The Stopwatch Timebase

When the Analysis Visualizer performs its analysis it represents time internally as 'cycles'. Cycles are defined by a a special timebase called the stopwatch timebase. As there can be multiple timebases in the system, the Analysis Visualizer must be told which is the stopwatch timebase. This is done by using the keyword `stopwatch` in the timebase declaration.

Exactly one timebase must be marked a stopwatch timebase in the configuration file.

By default, the Analysis Visualizer assumes that one tick of the stopwatch timebase represents one cycle. The rate for cycles should be as fast as possible, ideally the same as the CPU clock or the fastest time that can be measured in the CPU. It is therefore recommended that the stopwatch timebase is used to model the CPU clock. The following declaration defines a 20MHz CPU clock rate:

```
timebase cpu_clock {
    stopwatch;
    units s {
        define 1 as 20000000 ticks; // 1 tick = 1 cycle
    }
    units ms {
        define 1000 as 1s;
    }
    units us {
        define 1000 as 1ms;
    }
    modulus 65536 ticks;
```

```
}
```

Scaling the Stopwatch

The largest interval of time that can be analyzed by the Analysis Visualizer is $2^{32} - 1$ ticks. The CPU clock frequency will therefore limit the maximum delay or to busy period that can be analyzed. The following table shows how increasing CPU clock rate reduces the maximum analyzable intervals:

CPU Clock Rate (MHz)	Longest Time (secs)
1	4294
16	298
80	53
150	28

However, it may be the case that the intervals are too short for your model. To overcome this, the Analysis Visualizer allows the unit `cycles` to be defined so that the stopwatch timebase case be scaled. Cycles are not allowed to be greater than ticks, e.g. the declaration `units cycles define 1 as 2 ticks` will result in an error message from the Analysis Visualizer.

In the example below, the CPU clock is 20MHz as before, buy cycles have been scaled for analysis so there are 20 cycles in a stopwatch tick.

```
timebase cpu_clock_scaled {
    stopwatch;
    units cycles {
        define 20 as 1 tick;
    }
    units s {
        define 1 as 20000000 cycles; // Note! expressed in cycles
    }
    units ms {
        define 1000 as 1s;
    }
    units us {
        define 1000 as 1ms;
    }
    modulus 65536 ticks;
}
```

Scaling the stopwatch timebase results in pessimism in the analysis because execution times are rounded up by up to scaling factor cycles, and inter-arrival times are rounded down by up to scaling factor cycles.

5.2.2 Non-Time Timebases

The use of timebases for defining time units is merely one way of using timebases. There is no special treatment of time as a unit, and no units are predefined: you can name units as you wish. So you could define a timebase to measure angular rotation rather than time.

```
timebase spindle {
    units deg {
        define 360 as 1128 ticks;
    }
    units rad {
        define 6.283185 as 360 deg; // 2pi radians = 360 degrees
    }
    units mrad {
        define 1000 as 1 rad;
    }
    modulus 360 deg;
}
```

The above example defines a timebase representing a counter than counts from 0 to 1127. The counter could be connected to a sensor that senses a toothed wheel on a motor spindle and is reset to zero when a particular tooth passes the sensor. In the example the sensor generates 1128 pulses for each full rotation of the spindle. The unit deg is defined to represent degrees, with 360 degrees for a full rotation of the spindle. The unit mrad is a milliradian, and one rad is one thousand milliradians.

5.2.3 Stopwatch Conversions

If the application uses any timebases in addition to the stopwatch timebase, it is necessary to supply a conversion factor between these and the stopwatch timebase.

Typically other timebases will be declared if the application uses timelines driven by activators that are triggered by counters running at a different frequency to the stopwatch timebase. In common usage, you may have many timebases that define 'wall clock' time units like seconds, milliseconds, for example:

```
timebase cpu_clock {
    stopwatch;
    units s {
        define 1 as 20000000 ticks; // 20MHz
    }
    units ms {
        define 1000 as 1 s;
    }
    modulus 65536 ticks;
```

```
}
timebase timer1 {
    units ms {
        define 1 as 1 ticks;
    }
    units s {
        define 1 as 1000 ms;
    }
    modulus 65536 ticks;
}
timebase timer2 {
    units s {
        define 1 as 1 ticks;
    }
    modulus 65536 ticks;
}
```

In this case, the stopwatch defines the unit `s` as 20000000 ticks, i.e. the processor is clocked at 20MHz. However, timebases `timer1` and `timer2` also define a units called `s`, but as 1 tick and 1000 ticks respectively.

Recall from Section 5.2 that unit names apply only to the timebase in which they are declared. Assuming that you want the units to mean the same thing everywhere in the model a stopwatch conversion is used to ensure that these units will be converted to cycles correctly:

```
stopwatch conversion {
    on timer1 1s is 1s;
    on timer2 1s is 1s;
}
```

There is a trade-off when using this form of stopwatch conversion: this form of the conversion tells the user nothing about the relative rate of ticks on `slow_tick` to the stopwatch timebase. However if the frequency of either timebase could change, a modification needs to be made in only one place in the configuration file, rather than in the timebase declaration and at the stopwatch conversion.

Where a timebase is associated with non-periodic events, the stopwatch conversion needs to specify the highest frequency that events can arise. For example in the case of a slotted disk (where light passing through a slot falls on a light sensor and causes an interrupt), with 8 equally spaced slots and a maximum rotational speed of 6000rpm, the following timebase and timebase conversion would be used:

```
timebase tb_disk {
    units revolution {
        define 1 as 8 ticks;
```

```
        }
        units mins {
            define 1 as 6000 revolution;
        } // at worst
        units s {
            define 60 as 1 mins;
        }
    }
    stopwatch conversion {
        on tb_disk 6000 revolutions is at worst 1 minute;
    }
```

The optional `at worst` keyword has been used here to show that the conversion represents a worst-case conversion (the disk may often rotate slower than 6000 RPM).

Where the conversion from a timebase to the stopwatch tick results in a non-integral value, rounding is used to ensure that worst case values are used. Execution times, blocking times, jitter and release delay all rounded up. Delays between arrivalpoints and deadlines are both rounded down. Thus 3 ticks on the `slow_tick` timebase, defined above, are equivalent to 23437.5 ($3 \times 8000000 / 1024$) ticks of the stopwatch timebase. If used as an execution time, this is treated as 23438 stopwatch ticks, but if used as a deadline, it is treated as 23437 ticks.

5.2.4 Hints for using timebases

The following hints will help you to make best use of timebases:

- Execution times should be declared in terms of processor cycles. If the processor speed is changed then the 'wall clock time' the code takes to execute will shorten or lengthen accordingly.
- Use timebases to capture units of time that you use with various timing values accurately reflect the meaning of those values. For example, if a delay is measured in seconds to conform to a specification, then you should have a timebase which defines a seconds unit on the timebase with which it is to be implemented.
- Define a timebase for each OS counter you have in your system. This means that the relationship between a tick of the counter and the underlying time is captured by the timebase declaration.

Following these hints will help you to build a timing model that is robust to changes in your configuration file.

To simplify the setting up of timebases, the following example gives a template that uses the Analysis Visualizer macro preprocessor to ensure that 'wall clock' timing units on multiple timebases are correctly mapped to CPU cycles for analysis.

```
// CPU (System) clock frequency = 10MHz
(define FREQUENCY 10000000)

// Prescale for peripheral timer
(define CLOCK_PRESCALE 32)

// Define any units of wall clock time from seconds
(define DERIVE_UNITS_FROM_SECONDS
    units ms {
        define 1000 as 1 s;
    }
    units us {
        define 1000 as 1 ms;
    }
    units ns {
        define 1000 as 1 us;
    }
)

timebase cpu_clock {
    stopwatch;
    units s {
        define 1 as (FREQUENCY) cycles;
    }
    (DERIVE_UNITS_FROM_SECONDS)
    modulus 65536 ticks;
}

timebase peripheral_timer {
    units s {
        define (CLOCK_PRESCALE) as (FREQUENCY) ticks;
    }
    (DERIVE_UNITS_FROM_SECONDS)
    modulus 65536 ticks;
}

// Unify seconds
stopwatch conversion {
    on peripheral_timer 1 s is 1 s;
}
```

5.3 Modeling Executable Objects

In order to perform schedulability analysis of a system, the runtime behavior of each of the executable objects within the system must be modeled. An

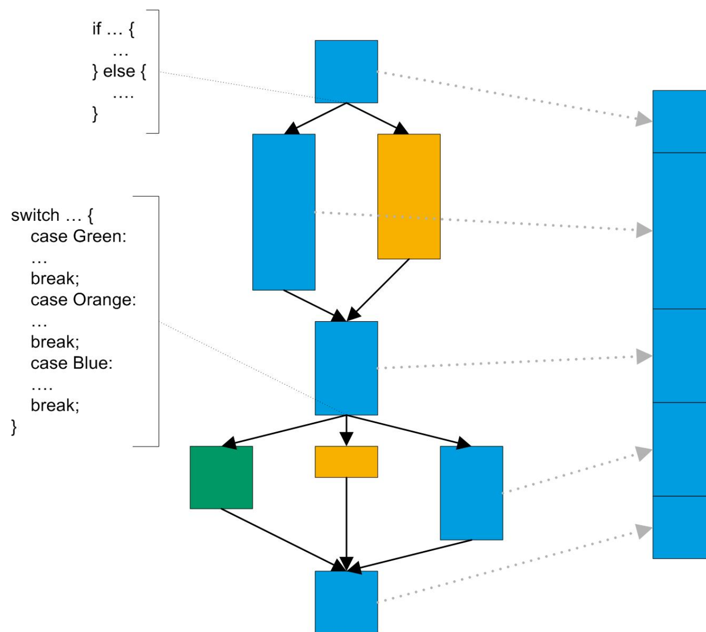


Figure 5.1: The longest path gives the worst case execution time

executable object is a task or an ISR. The execution characteristics of tasks and ISRs are declared in execution *profiles*. At least one execution profile is required for each task and ISR. You can also use multiple profiles to model situations where you know that different invocations of a task/ISR have different execution time. Multiple profiles are considered in Section 6.2.

The execution profile declares the worst-case execution time of the corresponding task and ISR. Worst-case execution times are usually determined by the amount of code executed, so they are measured in processor cycles. This means that if you change the CPU clock rate, the execution time for your tasks and ISRs will scale automatically[1].

The worst-case execution time for tasks and ISRs is measured from the start of the first machine code instruction of the task entry function, through the longest path in time and then to the end of the 'return' instruction as shown in Figure 5.1. It excludes the effects of preemption or interrupts. You are responsible for ensuring that the effects of any cache or instruction pipelines are at their most pessimistic.

There are many techniques available for obtaining a worst-case execution time. Techniques fall into two classes:

[1]Tasks that perform imprecise computation are an exception to this rule. This type of task executes until it observes a value in a particular time. You should, therefore, express execution time using 'real-world' time units.

Static techniques analyze the object code and calculate the worst-case execution times based on instruction set timing and/or models of the processor. Examples of commercial tools are AbsInt's aiT[2].

Dynamic techniques measure execution times on the target and therefore rely on the worst case execution path in the program being executed. There are many measurements approaches, including two provided by RTA-OS3.x:

- time monitoring in the RTA-OS3.x kernel which automatically logs the longest observed execution time. This provides a single worst-case figure.
- RTA-TRACE integration which calculates worst-case execution times offline using logged OS trace data. Using RTA-TRACE can provide additional insight into how different releases of tasks and ISRs execute and this additional information can be used to build advanced timing models. See Section 6.2 for further details.

Third-party tools include Rapita System's Rapitime[3].

The following sections describe how to model the execution time for tasks and ISRs.

5.3.1 Tasks

If `t1_entry` is found to have a worst-case execution time of 600 processor cycles, it will have the following task declaration:

```
task t1 {
    entry t1_entry;
    profile {
        this priority duration 600 cycles;
    }
}
```

By default, an execution time within a profile is specified in terms of the stopwatch timebase (see Section 5.2.1). Other timebases can be used instead if they are explicitly referenced as in the following example.

```
task t1 {
    entry t1_entry;
    profile {
        this priority duration on low_resolution_timebase 150
            ticks;
    }
}
```

[2] www.absint.com
[3] www.rapitasystems.com

Activating Tasks

If a task activates another task, either through the `ActivateTask()` or `ChainTask()` calls then this needs to be captured in the model by specifying an `activates task` clause:

```
task t1 {
    entry t1_entry;
    activates task t2; // By ActivateTask(t2);
    activates task t3; // By ChainTask(t3);
    profile {
        this priority duration 600 cycles;
    }
}
```

Activations must be downwards, i.e. you can only model a system where a task activates only lower priority tasks. The Analysis Visualizer will check for incorrect upwards activations in your model and generate an error if any are found.

Task activations must be recorded even if they are conditional because this represents the worst case.

Task Priorities

Unlike AUTOSAR OS R3.x, where the priority of a task is a number associated with the task itself, an Analysis Visualizer model does not need to care about what priority value is associated with a task, but only with the relative ordering of task priorities.

Task priorities are specified in the `task priority order` clause in the system tail of the configuration file. This is required if more than one task is declared.

```
task priority order {
    task TaskA; // Highest
    task TaskB;
    task TaskC; // Lowest
}
```

The above example specifies the priority order for a system of three tasks: TaskA has the highest priority, TaskB the second highest and TaskC the lowest priority. All tasks declared in the configuration file must appear in the `task priority order` clause. The Analysis Visualizer will generate an error if this is not the case.

The `task priority order` is a total ordering over the tasks in the system. This means that it is not possible to share task priorities (as required by the constraints of DMA) for system that you want to analyze, even though you can build them with AUTOSAR OS R3.x. It is, however, possible to share priority levels by declaring non-preemption groups as described in Sections 5.3.1 and 5.4.2.

Non-Preemptive Tasks

The Analysis Visualizer does not allow you to say that a task is non-preemptive. Instead, a general purpose mechanism called a non-preemption group that allows you to group a set of tasks together that need to run non-preemptively is used[4].

Tasks in the non-preemption group run at the priority level defined by the highest priority task in the same group.(i.e. they run at the ceiling priority of the non-preemption group). To correctly model the notion of a non-preemptive task it therefore follows that one of the tasks in the group must be the highest priority task in the system *even if the highest priority task itself is preemptive*.

Consider a system with the following tasks:

Task	Priority	Non-Preemptive
t5	5	✗
t4	4	✓
t3	3	✗
t2	2	✓
t1	1	✗

The non-preemptive tasks would be modeled using the following non-preemption group:

```
nonpreemption group {
    task t5; // Included as it is highest priority
    task t4;
    task t2;
}
```

Analysis assumes that the tasks in the non-preemption group run at the ceiling priority for their entire duration. This means that the tasks must not use the `Schedule()` call during execution. More information about modeling so-called "co-operative " tasks can be found in Section 6.4.

[4]The same construct will be used in Section 5.4.2 to model tasks sharing an internal resource.

5.3.2 ISRs

The Analysis Visualizer defines two types of interrupts:

controlled interrupts are managed by the OS. In AUTOSAR OS R3.x this maps directly on to Category 2 ISRs.

uncontrolled interrupts are not managed by the OS. In AUTOSAR OS R3.x this maps directly on to Category 1 ISRs.

Each interrupt must define a priority. Priority 1 defines the lowest interrupt priority. The highest interrupt priority will depend on your target microcontroller. The priority clause defines the interrupt priority level (IPL) for the interrupt.

The Analysis Visualizer configuration language requires an interrupt vector to be specified to be syntactically correct. It is not required that this is a valid vector address for your target or that the vector is correct for the interrupt. The vector is not used for analysis.

As with tasks, interrupts also need to specify an execution time in a `profile` clause.

The following is a declaration of a controlled interrupt called `int1`:

```
interrupt int1 {
    entry i1_handler;
    controlled;
    priority 4;
    vector 0x1000;
    profile {
        this priority duration 100 cycles;
    }
}
```

All uncontrolled interrupts must have a priority greater than or equal to the highest controlled interrupt priority. This is the same model as RTA-OS3.x applies for Category 1 and Category 2 ISRs.

Arbitration Ordering

IPLs can be shared between interrupts if this is supported by your target microcontroller. However, when ISRs share an interrupt priority level, you will have to specify an interrupt arbitration order. The arbitration order is the sequence in which interrupts of the same priority are serviced if several are pending at the same time. You can usually find this information in the data

book for your target microcontroller. The arbitration ordering allows the Analysis Visualizer to determine interrupt blocking correctly for the specified interrupts.

```
arbitration order {
    interrupt priority 2 {
        interrupt TimerChannel;
        interrupt IOport;
    }
}
```

The above order specifies that `TimerChannel` is serviced first if both the `TimerChannel` and `IOport` interrupts are pending simultaneously.

You should ensure that the arbitration order that you specify matches the arbitration order of the interrupts with your application. For many processors, the interrupt arbitration order is fixed and can be found in the processor reference manual, for others it can be defined by the application programmer. In both cases, care must be taken to ensure that the information given in the configuration file correctly describes the run-time behavior of interrupts.

5.3.3 The Idle Mechanism

If the idle mechanism makes any API calls it can introduce blocking. This must be considered in the analysis.

A profile for the idle mechanism is specified in the same way as for a task, in fact in the Analysis Visualizer it is called the "idle task". If the idle task has no deadlines to meet, however, the exact value of the execution time specified is irrelevant.

The use of the idle task in this way is shown below:

```
idle task {
    profile {
        this priority duration 1 cycle;
    }
}
```

If the idle task performs resource locking, setting the interrupt priority level or has any deadlines, an idle task profile must be provided that includes these. This profile has the same structure and possible contents as task and ISR profiles.

5.4 Accounting for Blocking

The modeling we have considered so far has been enough to specify execution times of executable objects. In this section we look at how to model blocking.

Tasks and ISRs that get resources or mask interrupts block the execution of higher priority tasks and ISRs. Let's look at an example of a system that contains two tasks. The tasks are called Task1 and Task2 and they share a resource. Task2 has a higher priority than Task1.

If Task2 becomes ready when Task1 owns the resource, it is blocked until Task1 releases the resource.

To determine whether your application is schedulable, the Analysis Visualizer must know for how long resources are held and for how long interrupts are disabled. The Analysis Visualizer does not need to know *when* the resource is locked relative to the start of the task or ISR.

5.4.1 Standard Resources

Every task and ISR that locks a resource needs to declare this in the model as follows:

```
task t1 {
    entry t1_entry;
    locks resource r1;
    locks resource r2;
    profile {
        this priority duration 600 cycles;
    }
}
```

If you do not specify resource and interrupt locking times, then the Analysis Visualizer assumes that the resource is held or that the interrupt is disabled for the entire execution time of the task or ISR. In most cases this will be pessimistic as the resource will only be locked for a part of the execution time.

You can reduce the pessimism in the analysis by adding the time for which a resource is locked to the task's profile. The analysis uses this information to determine how blocking affects the schedulability of the system. The Analysis Visualizer does not need to know the time at which the resource is locked relative to the start of the locking task - the analysis will always assume that this happens at the time that gives the worst case response time.

As an example, consider the following task:

```
TASK(t1){
    ...
    GetResource(r1);
    /* 25 cycles */
    ReleaseResource(r1);
    ...
    GetResource(r2);
    /* 100 cycles */
    ReleaseResource(r2);
    ...
    GetResource(r1);
    /* 75 cycles */
    ReleaseResource(r1);
    ...
    TerminateTask();
}
```

The following profile shows the execution time of the associated task in terms of CPU cycles in addition to showing the durations for which the task locks the specified resources.

```
task t1 {
    entry t1_entry;
    locks resource r1;
    locks resource r2;
    profile {
        this priority duration 600 cycles;
        resource r1   duration  75 cycles;
        resource r2   duration 100 cycles;
    }
}
```

When a resource is locked multiple times then only the longest time for which the resource is continuously locked is required. However, it is recommended that each separate locking duration is specified in order to improve the clarity and maintenance of the configuration file (see for example, the two locking durations of resource r1, above).

```
task t1 {
    entry t1_entry;
    locks resource r1;
    locks resource r2;
    profile {
        this priority duration 600 cycles;
        resource r1   duration  25 cycles; // Optional
        resource r2   duration 100 cycles;
        resource r1   duration  75 cycles;
    }
}
```

You do not need to distinguish whether or not resource requests are nested. the Analysis Visualizer takes account of this automatically during analysis. Even when resources are nested then you must provide the worst case locking time for entire duration the resource is held - irrespective of whether any other resources are locked inside.

5.4.2 Internal Resources

indexBlocking!Internal Resources

Internal resources are modeled using non-preemption groups that you will recall from Section 5.3.1.

A non-preemption group should be defined for each internal resource and the tasks that lock the resource should be listed as the members of the group.

```
nonpreemption group {
    task TaskA;
    task TaskC;
    task TaskH
}
```

A task can belong to multiple non-preemption groups (i.e. it can share multiple internal resources). The Analysis Visualizer will automatically calculate the correct ceiling priorities during analysis.

Note that it is not necessary to specify a `duration` when modeling an internal resource in this way - the Analysis Visualizer already knows how much blocking time will be introduced by looking at the worst case execution times specified for each task.

RTA-OS3.x provides an extension to AUTOSAR OS R3.x that allows internal resources to be shared between tasks and ISRs. However, it is not possible to model this in the Analysis Visualizer.

5.4.3 Interrupt Masking

indexBlocking!Interrupt Masking

All tasks and interrupts can raise the interrupt priority level to prevent tasks and interrupts at the same or lower IPL from executing. In the case of tasks and controlled interrupts, this is done by means of API calls. For uncontrolled interrupts, the IPL is modified by more direct means, such as changing the processor status word.

Like resource locks, there may be several interrupt priority sub-clauses within a single profile. An example of a profile that declares interrupt locks is shown below:

```
profile {
    this priority duration 120000 cycles;
    interrupt priority OS level duration 24000 cycles;
    interrupt priority 42 duration 8000 cycles;
}
```

Each clause specifies the duration at which the task (or ISR) spends at the indicated IPL. In AUTOSAR OS R3.x applications, the use of the `SuspendOSInterrupts()` API call locks to "OS level". The Analysis Visualizer allows you to use the shorthand `OS level` to define this priority level and automatically calculates the value of `OS level` from the interrupt priorities you specify.

To model disabling all interrupts using `SuspendAllInterrupts()` and `DisableAllInterrupts()` you need to specify that you lock to the priority of the highest priority interrupt in your system. In the example above this is assumed to be priority level 42. You will need to identify this priority level yourself and specify the time spent at this level for each task and ISR that executes `SuspendAllInterrupts()` and/or `DisableAllInterrupts()`.

To avoid the need to modify every executable object each time you add a new higher priority interrupt, the use of the Analysis Visualizer macro preprocessor is recommended. A macro called `MAXIMUM` can be defined and used as follows:

```
(define MAXIMUM 42)
...
profile {
    this priority duration 120000 cycles;
    interrupt priority OS level duration 24000 cycles;
    interrupt priority MAXIMUM duration 8000 cycles;
}
```

When a new interrupt is added that is higher priority than the current highest priority interrupt, only the definition of `MAXIMUM` needs to be changed.

5.5 Modeling Timing Relationships with Transactions

Profiles are used to specify the detailed timing behavior of individual executable objects. In order to analyze a system containing these objects, it is necessary to describe timing relationships between execution profiles. For this purpose, the Analysis Visualizer uses a *transaction* to model timing relationships. Any number of transactions can be created and the Analysis Visualizer assumes that all transactions are asynchronous.

Each transaction is used to link the following information:

- The delays between consecutive releases of tasks/ISRs into the system.
- The profiles of tasks and interrupts that are released.
- Delays (if any) between the arrival of the triggering event and the release of tasks and interrupts .

There are two categories of transaction:

timeline transactions allow you to specify a notional timeline of events and associated delays. A timeline transaction is typically used to model the activation of tasks via an alarm or from a schedule table expiry point, this is described in Section 5.8.

bursting transactions provide a shorthand for expressing sporadic events. You typically use an bursting transaction to model the case where an ISR activates a task but you can specify arbitrarily complex bursting clauses. Bursting transactions are also used to model situations where something happens once.

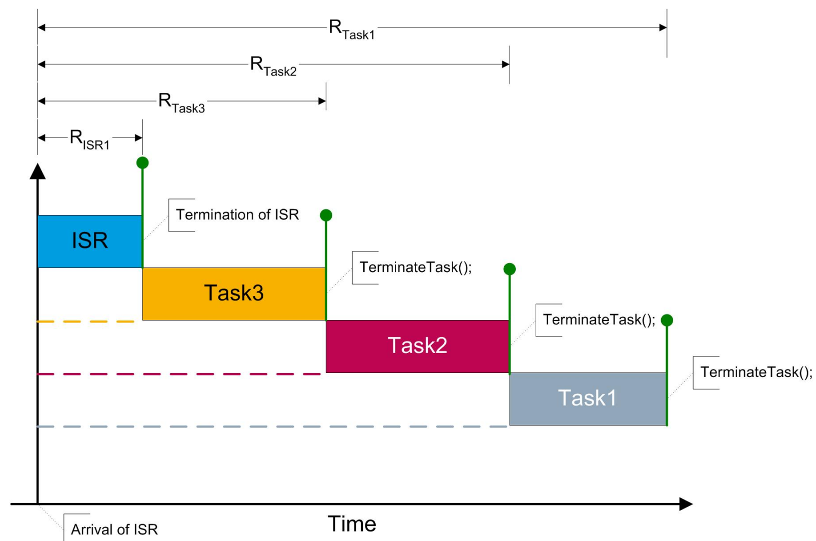
Each profile can be associated with exactly one transaction. In the case of a timeline transaction, the profile can appear multiple times in the same transaction. If you need the same task or ISR to appear in multiple transactions then you will need to create multiple profiles. Section 6.2 describes this advanced form of modeling in more detail.

Note that, so far as the analysis is concerned, there is no difference between an arrival directly causing a set of executable objects to be released and a sequence of executable objects activating other executable objects when upwards activation is forbidden. This is the response time is always measured from the initial trigger, so the observed behavior of the above two situations is identical. This is illustrated in Figure 5.2.

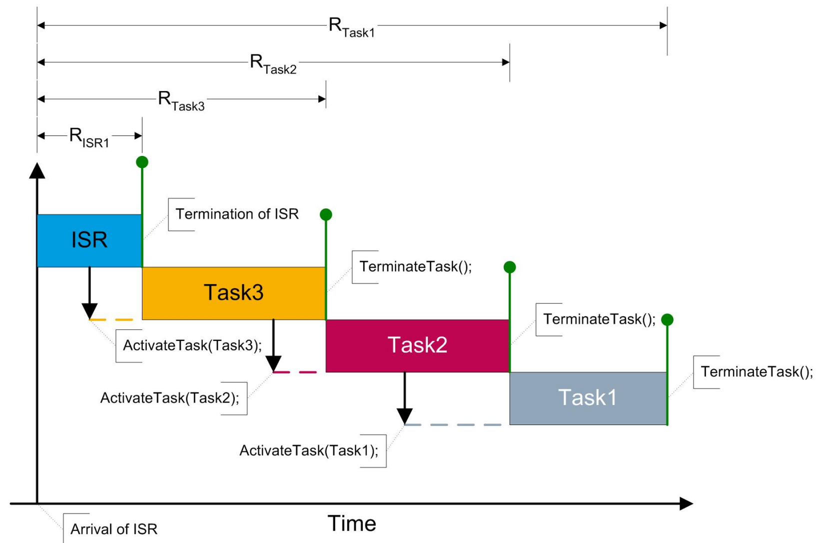
In the second case, even though Task3 activates Task2 and Task2 activates Task1, the effect is as if all three tasks had been released at the same time. This means you do not need to specify *when* a ISR activates a task (or when a task activates another task) - it is sufficient for analysis simply to know that it does.

5.5.1 How transactions are used

DMA analysis is based upon the notion of a *critical instant* - that point in time that represents the worst-case set of arrivals of tasks and interrupts. If the system is schedulable at a critical instant, then if is schedulable for other (non-critical) alignments. Classic DMA has a critical instant that corresponds to every higher priority object being released at the same time as the object



(a) Objects released simultaneously



(b) Objects released by downwards activation

Figure 5.2: Simultaneous versus activation chain release of executable objects

of interest, i.e. there is some notional time zero at which the task under analysis is released together with every higher priority task or ISR. This is why the DMA equation from Chapter 3.4 calculated the interference from all higher priority objects.

However, this model is pessimistic if it is known that some higher priority objects are never released at the same time. It may be the case that the execution of a higher priority task is *offset* from notional time zero in such a way that it does not contribute to the interference suffered by the task under analysis.

Each transaction defined specifies the *offsets* between releases of executable objects. When the Analysis Visualizer runs it calculates the critical instant (or critical instants) for each task by checking every possible alignment of all transactions and then performing the analysis. This means that Analysis Visualizer only includes interference from higher priority tasks that can be released while the task under analysis is in the ready or running state, allowing the analysis to be less pessimistic (i.e. more accurate) than it would be if classical DMA was performed.

5.5.2 What transactions are required?

In typical usage, a transaction will be required for each of the following:

- Each Category 1 interrupt.
- Each Category 2 interrupt that activates tasks directly.
- Each task that is activated by an alarm driven by an RTA-OS3.x software counter.
- The set of alarms driven by an RTA-OS3.x hardware counter.
- Each AUTOSAR OS R3.x Schedule Table.

As the transaction is used as the basis of analysis, any executable objects that are not referenced by a transaction will not be included in the analysis.

Omitting executable objects from the analysis may result in incorrect results. This applies irrespective of whether the omitted objects have deadlines or other timing behavior that needs to be guaranteed: the mere presence of the objects in the system can cause other objects to become less schedulable. This occurs because any low priority tasks in the system lock the interrupt priority level to OS level for short amounts of time, and thus introduce a certain amount of blocking. High priority tasks (that is tasks with a higher priority than any tasks that are analyzed) or interrupts will increase the response time of analyzed executable objects. If these effects are not accounted for in the analysis, the Analysis Visualizer may report that the system is schedulable when it is not.

Before looking at transactions in more detail, it is necessary to understand how to model timelines. The following sections explain how to model timelines and activators before looking at how timeline transactions and bursting transactions are build in Sections 5.8 and 5.9 respectively.

5.6 Timelines

A timeline is used to define when tasks (and optionally ISRs) are released into the system. Each timeline contains a list of arrivalpoint declarations that specify what happens at that point and how long elapses until the next arrivalpoint. The timeline is associated with a timebase so that you can specify arrivalpoint delays in meaningful units.

```
timeline {
    timebase tb_ms;
    sequence {
        arrivalpoint somename ... // Specify which tasks are
            released
        arrivalpoint ...
        arrivalpoint ...
        next ...          // (Optionally) repeat from a named point
    }
}
```

The timeline can also specify a next clause after the sequence of arrivalpoints which specifies a named arrivalpoint at which to repeat processing.

Each arrivalpoint has five properties:

1. An (optional) identifier which is required if you need to refer to the arrivalpoint. Typically only the first arrivalpoint needs to be named as this will be the arrivalpoint from which the timeline is started. Other arrivalpoints need only be named if they need to be referenced by `next` clause. In the above example, the second and third arrivalpoints are not named.

2. A set of task profiles, which specify tasks to be executed.

3. A delay (called ‘delay’) before the next arrivalpoint occurs.

A delay of zero means “the full modulus value of the associated timebase”, not “zero delay”.

4. A (optional) next clause which specifies which arrivalpoint will be processed next. If the next clause is omitted then the next arrivalpoint in the timeline’s configuration is processed.

5. An (optional) analysis clause that is used to provide additional information to the Analysis Visualizer for accurate schedulability analysis.

Delays are expressed in units of the timeline’s timebase. When the delay is converted into ticks of the timebase, it must not exceed the modulus of the timebase. If you want a delay that exceeds the timebase modulus, then you can add arrivalpoints with no tasks to span the large delay required.

The following example shows a timeline with two arrivalpoints that specifies that task t1 and task t2 should be activated together, then there should be a delay of 20 ms, then task t1 should be activated followed by another delay of 20 ms before the repeating:

```
timeline {
    timebase tb_ms;
    sequence {
        arrivalpoint start {
            task t1;
            task t2;
            delay 20 ms;
        }
        arrivalpoint {
            task t1;
            delay 20 ms;
        }
        next start; // Repeat from start
    }
}
```

The next clause does not have to go back to the start of the timeline. This is useful if you want an initialization part where tasks are activated in a special pattern, then the rest of the timeline is the looping part to repeat the activation of tasks. In the configuration file you can only put next after the last arrivalpoint declaration in a timeline or inside an arrivalpoint itself.

A timeline with a loop is a good way to handle periodic tasks. You can construct a repeating timeline where all the periodic tasks appear in the right

places to give the periodic behavior required. The timeline will need to be long enough so that each task appears at least once, and this usually means that the duration of the timeline is as long as the least-common multiple (LCM) of the periods of all the tasks.

5.6.1 Arrivalpoint Analysis Clauses

The Analysis Visualizer uses the information supplied by the arrivalpoint for analysis. When an optional analysis clause is given, the delay and next analysis attributes override the application attributes as far as timing analysis is concerned. When profiles are given in the application attributes and in the analysis-only attributes, all profiles from both sources are considered to be released. For example consider the following arrivalpoint:

```
arrivalpoint {
    task t1;
    delay 20 ms;
    next X;
    analysis {
        task t2;
        delay 10 ms; // Delay will override 20 ms
        next Y;      // Next overrides X with Y
    }
}
```

When this arrivalpoint occurs, the Analysis Visualizer would consider that the arrivalpoint releases both t1 and t2 and, instead of processing arrivalpoint X 20 ms later, arrivalpoint Y would be processed 10 ms later.

5.6.2 Periodic Timelines

It can be quite awkward to have to work out the timeline to activate tasks periodically when there are more than a small number of tasks. For example, assume a timeline is required to model a system where task t1 will be activated every 5ms, task t2 will be activated every 10ms, and task t3 will be activated every 20ms. Task t3 also has an offset of 5ms relative to the start of the timeline. The resulting timeline would be:

```
timeline {
    timebase tb3;
    sequence {
        arrivalpoint start {
            task t1;
            task t2;
            delay 5.0ms;
        }
        arrivalpoint {
            task t1;
            task t3;
```

```
            delay 5.0ms;
        }
        arrivalpoint {
            task t1;
            task t2;
            delay 5.0ms;
        }
        arrivalpoint {
            task t1;
            delay 5.0ms;
        }
        next start;
    }
}
```

As this is a common type of model, the Analysis Visualizer configuration syntax has a shorthand form called a *periodic timeline declaration*. This lets you define the period and offset of each task. Offsets must be less than the period of the task, and at least one task in the timeline must have an offset of zero.

```
timeline {
    timebase millisecond;
    periodic start {
        task t1 every 5ms  offset 0ms;
        task t2 every 10ms offset 0ms;
        task t3 every 20ms offset 5ms;
    }
}
```

This periodic timeline has identical behavior to the sequential timeline, but is must easier to write. When the Analysis Visualizer analyses the system, it will automatically expand the periodic timeline into the sequential timeline, creating arrivalpoints as required.

When the sequential timeline is generated, it will have a number of arrivalpoints equal to the Least Common Multiple (LCM) of the periods in the periodic timelines. You should check that the length of each periodic timeline is reasonable given the task periods, sometimes small rounding errors in the unit conversions can result in a set of periods that have a large LCM, even though the specified values are harmonic. The Analysis Visualizer provides the `-t` option that can be used to check this.

Multiple Offsets

You can give a task more than one offset (as long as all the offsets for the task are less than the period). This makes a timeline where the task is activated several times for each period. You might find this useful when you need the

task to make an output for the first time it runs then a short time later the task to run again and check the output.

```
timeline {
    timebase tb3;
    default readonly;
    periodic start4 {
    task t2 every 100ms offset 0 ms offset 5.5 ms;
    }
}
```

In the above timeline, task t2 will be activated at 0, 5.5ms, 100ms, 105.5ms, 200ms, 205.5ms, and so on (measured relative to the start of the timeline).

5.6.3 Single-Shot Timelines

If the final arrivalpoint does not specify a next clause then the timeline becomes single-shot timeline. This means that once the sequence of arrivalpoints has occurred the timeline is assumed to be stopped. A single-shot timeline is quite useful when you want to cause a sequence of phased activations in response to a sporadic event.

The following declaration specifies a one-shot timeline:

```
timeline {
    timebase tb3;
    sequence {
        arrivalpoint first {
            task t1;
            task t2;
            delay 10.0ms;
        }
        arrivalpoint second {
            task t1;
            task t2;
            delay 5.0ms;
        }
        arrivalpoint third {
            task t1;
            delay 5.0ms; // Don’t care, there is no next
        }
    }
}
```

Repeating a Single-Shot Timeline

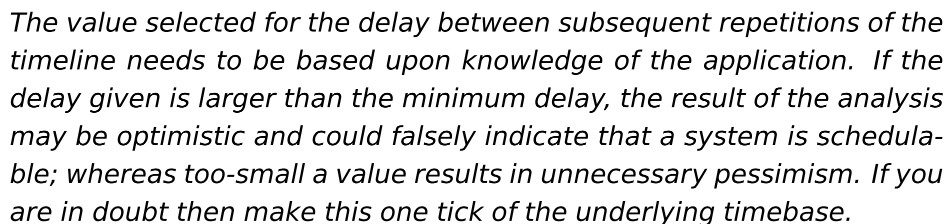
If a transaction is declared which references a single-shot timeline, the analysis will consider this to mean that the timeline only occurs once.

However, a single-shot timeline may be processed repeatedly by the system. In this case, you need to indicate that the timeline is single shot, but that it repeats for analysis purposes. To achieve this, the final arrivalpoint must be annotated with an analysis. For example the final arrivalpoint in a single-shot timeline could be set to the following:

```
timeline {
    ...
    arrivalpoint first {
        ...
    }
    ...
    arrivalpoint third {
        task t1;
        delay 5.0 ms; // Don't care - last point
        analysis {
            next ap_first; // Timeline can be repeated
        }
    }
}
```

Unless the enclosing timeline has a next clause, the delay clause of the final arrivalpoint of a one-shot timeline has no meaning for the application. However, when the analysis clause of this arrivalpoint specifies a "next" arrivalpoint, the analysis will use this value to give the time between repetitions of the timeline. It is good practice to also specify this delay in the analysis clause. This clearly allows a reader of the configuration file to see that the delay has been provided for analysis purposes. Thus the above example might be modified as follows:

```
arrivalpoint third {
    task t1;
    delay 5.0 ms; // this value is ignored for application
       purposes as this arrivalpoint is the last in a one shot
       timeline
    analysis {
        delay 5.0 ms; // in the analysis, this overrides the
           other delay value
        next first;
    }
}
```



The value selected for the delay between subsequent repetitions of the timeline needs to be based upon knowledge of the application. If the delay given is larger than the minimum delay, the result of the analysis may be optimistic and could falsely indicate that a system is schedulable; whereas too-small a value results in unnecessary pessimism. If you are in doubt then make this one tick of the underlying timebase.

5.7 Activators

An activator is used to model the processing of the arrivalpoints in a timeline, activating tasks and waiting for the necessary delays (represented in ticks). An activator provides the encapsulation of the AUTOSAR OS R3.x counter model.

Each activator contains a logical counter that counts ticks and is associated with a timebase, which describes how the counter ticks (the range, granularity, units, etc). The activator uses the counter to determine when the required number of ticks has occurred so that it can process the next arrivalpoint.

An activator can only process timelines that are associated with the same timebase as the activator. This is to make sure that the ticks for arrivalpoint delays are in the same units as the ticks counted by the activator's counter.

While the Analysis Visualizer allows several activators to be associated with the same timeline at the same time, this represents a model which is not permitted in AUTOSAR OS R3.x. In your models, you should make that any timelines you declare are associated with exactly one activator.

There are two types of activator:

coarse activators capture the ticking of counter and model the case of a software counter in AUTOSAR OS R3.x.

fine activators assume that counting is performed by external hardware, typically a timer driver, and the activator processes each arrivalpoint only when a requested number of ticks have occurred. This models the case of a hardware counter in AUTOSAR OS R3.x.

5.7.1 Activator declaration

Assume the following timebase declaration:

```
timebase low_resolution {
    units s {
        define 1 as 1000 ticks; // frequency = 1 kHz
    }
    units ms {
        define 1000 as 1 s;
    }
    modulus 65536 ticks;
}
```

You can declare a coarse activator called `software_counter` associated with timebase `tb4` like this:

```
activator software_counter {
    timebase low_resolution;
    coarse;
}
```

A fine activator is declared in much the same way as a coarse activator but you must specify placeholders for the driver callbacks. Typically you will use this when you have a high resolution timebase:

```
activator hardware_counter {
    timebase high_resolution;
    fine;
    driver callbacks {
        now    Os_Cbk_Now_hardware_counter ;
        cancel Os_Cbk_Cancel_hardware_counter ;
        state  Os_Cbk_State_hardware_counter  ;
        set    Os_Cbk_Set_hardware_counter ;
    }
}
```

The driver callbacks are not used by the analysis and the names given are not important. They are required only to make the configuration file syntactically correct.

5.8 Timeline Transactions

A timeline transaction will directly reference the first arrivalpoint in a timeline. This arrivalpoint and each subsequent one in the timeline represents the arrival of some (usually external) event. In the case of timeline transactions that are driven by activators, the Analysis Visualizer needs to know how to account for the occurrence of interrupt and the associated processing of the arrivalpoint.

Consider the following timeline:

```
timeline {
    timebase tb_millisecond;
    default readonly;
    sequence {
        arrivalpoint ap_first {
            task t3;
            delay 5ms;
        }
        arrivalpoint ap_second{
            task t2;
            delay 8ms;
        }
        arrivalpoint ap_third {
            task t1;
```

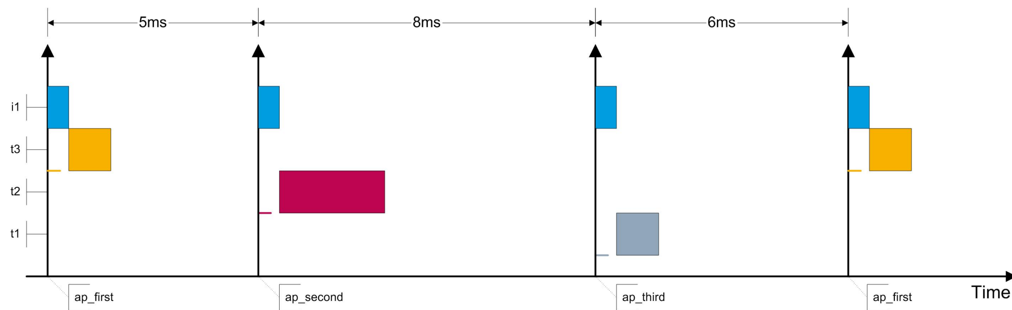


Figure 5.3: Automatic inclusion of a fine activator profile in a timeline transaction

```
                delay 6ms;
            }
            next ap_first;
        }
    }
```

A transaction can be based upon this timeline by specifying the start arrivalpoint (`ap_start`) and by giving the executable object that drives the activator along the timeline as follows:

```
transaction timeline_transaction {
    start ap_start;
    activator act1 driven by i1;
}
```

If the activator that drives this timeline (`act1`) is a fine activator, the executable object referenced by the profile `i1` will run whenever an arrivalpoint on the timeline becomes due. Therefore, to model this correctly, the Analysis Visualizer automatically includes the invocation of `i1` in every arrivalpoint in the timeline. Figure 5.3 shows the inclusion of `i1` and the resulting execution at each arrivalpoint.

This is equivalent to defining the following analysis-only timeline and associated transaction:

```
timeline {
    timebase tb_millisecond;
    default readonly;
    sequence {
        arrivalpoint ap_first {
            analysis {
                interrupt i1;
                task t3;
                delay 5ms;
            }
```

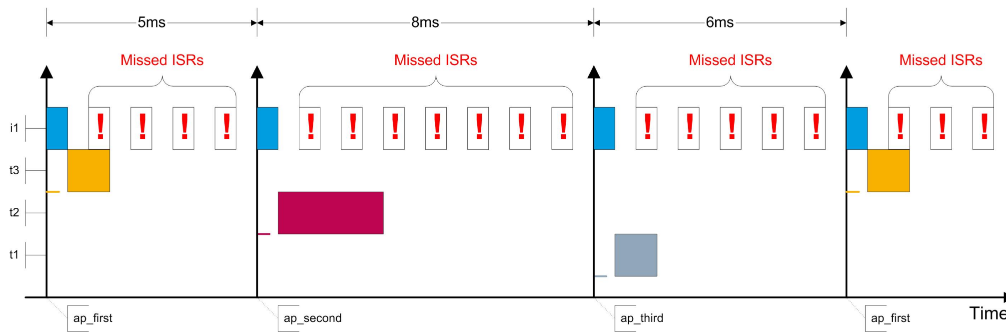


Figure 5.4: Missed profile releases with automatic inclusion of a coarse activator profile in a timeline transaction

```
        }
        arrivalpoint ap_second {
            analysis {
                interrupt i1;
                task t2;
                delay 8ms;
        }
        }
        arrivalpoint ap_third {
            analysis {
                interrupt i1;
                task t1;
                delay 6ms;
                next ap_first;
            }
        }
    }
}
```

If `act1` is a coarse activator, then the profile for `i1` which is responsible for 'ticking' the software counter modeled by the activator is *not* automatically included in the timeline. This is because one executable object may 'tick' more than one activator, and a timeline will typically require more than one tick per activation.

For example, consider assume that `i1` is driven by a 1 millisecond clock interrupt. In this case, including the ISR profile in the timeline will result in four releases of the ISR profile being missed between `ap_first` and `ap_second` and then seven releases being missed between `ap_second` and `ap_third`. The impact of this is shown in Figure 5.4 where the shaded arrivals of the coarse activator are the ones that would be missed.

Instead, when a coarse activator is used to drive a timeline, it must be modeled as a separate transaction. The following example shows how this is modeled as a timeline transaction:

```
timeline {
    timebase tb_millisecond;
    readonly;
    sequence {
        arrivalpoint ap_int {
            analysis {
                interrupt i1;
                delay 1 ms;
                next ap_int;
            }
    }
}
transaction tr_coarse_activator {
    start ap_int;
}
```

The Analysis Visualizer will align the transactions for analysis and then determine schedulability when arrivals align as shown in 5.5

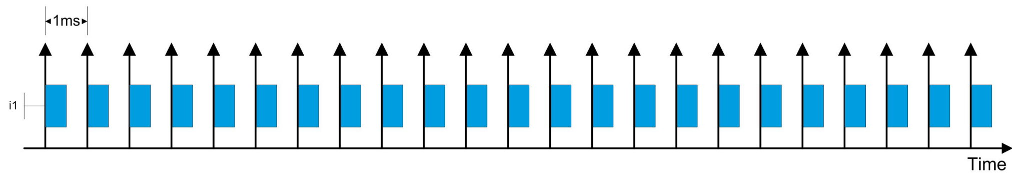
Note that the use of the coarse activator results in additional interference in the execution of the tasks and therefore increases task response time.

5.9 Bursting Transactions

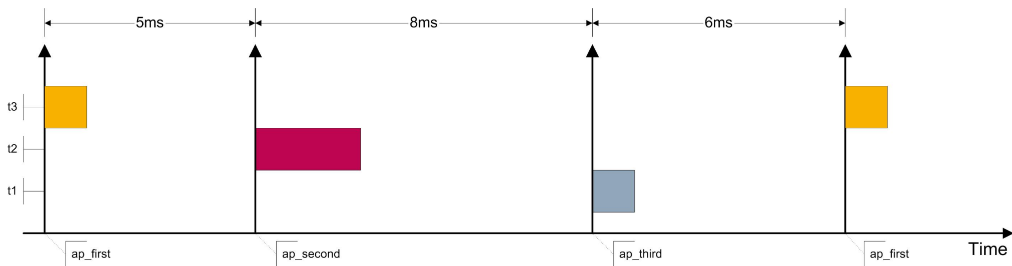
Bursting transactions are a useful means for describing any event that is initiated by an interrupt. Either sporadic or periodic interrupts can be modeled with bursting transactions. They allow the user to express a set of rules describing the arrival pattern of the arrivals.

```
transaction tr2 {
    bursting {
        1 times in 100ms;
    }
    interrupt ist1;
    task task1;
}
```

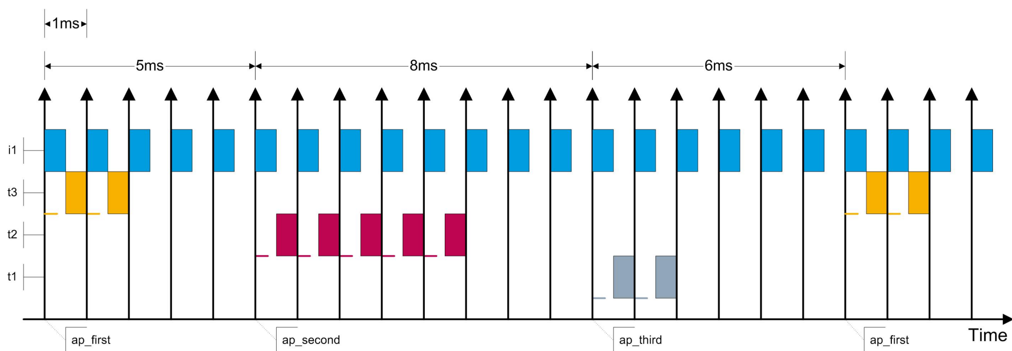
In this case, a single arrival rule is used. This results in a transaction with periodic arrival behavior. Bursting transactions can be used as an alternative way to model timer interrupts, without the need to explicitly provide arrivalpoints. In this case, this transaction has identical behavior to the simple timeline transactions shown in Section 6.9.1



(a) Transaction tr_coarse_activator



(b) Transaction timeline_transaction



(c) Execution of the aligned transactions

Figure 5.5: Alignment of coarse activator transactions for analysis

5.9.1 Multiple Arrival Rules

A more complex example of a bursting transaction uses multiple arrival rules:

```
transaction tr3 {
    bursting {
        1 times in  1 ms; // Rule 1
        2 times in  5 ms; // Rule 2
        3 times in 20 ms; // Rule 3
    }
    interrupt i1;
    task t2;
    task t3;
}
```

In this example, the transaction describes the timing behavior of an interrupt and two tasks. The bursting clause of the transaction specifies the rules governing the frequency of arrivals. In this case, the release will occur:

Rule 1 No more than once in any one millisecond;

Rule 2 No more than twice in any five milliseconds;

Rule 3 No more than three times in any twenty milliseconds.

These rules combine to form a worst-case arrival pattern as follows:

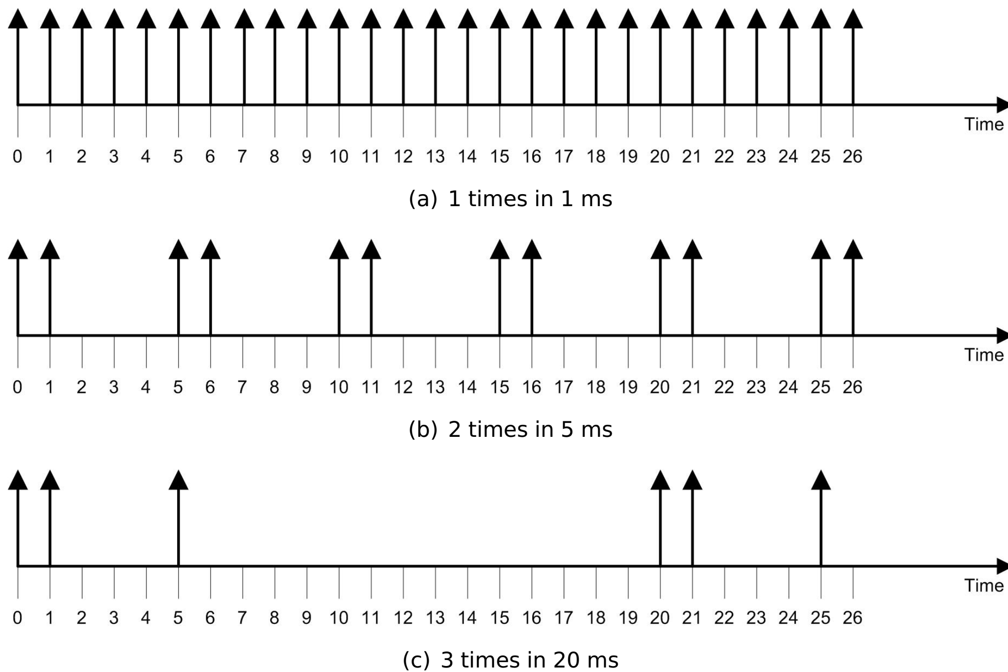
- 0ms, 1ms, 2ms, 3ms. . . (Rule 1 allows the minimum inter-arrival time of 1ms).
- 0ms, 1ms, 5ms, 6ms, 10ms, 11ms. . . (Rule 2 prevents more than 2 arrivals in a period of 5ms, so bursts of 2 are separated by 5ms periods).
- 0ms, 1ms, 5ms, 20ms, 21ms, 25ms. . . (Rule 3 prevents more than 3 arrivals within a 20 ms interval).

Figure 5.6 shows the impact of each rule on the arrivals.

When more than one arrival rule is given, another rule covers the values that are allowed. If values are arranged in increasing order, each successive pair of values (arrivals, interval) must be greater than the previous pair. The rate of arrivals (that is, arrivals/interval) must strictly decrease.

Following on from the previous example you can see that:

- 1 time <2 times <3 times



(a) 1 times in 1 ms

(b) 2 times in 5 ms

(c) 3 times in 20 ms

Figure 5.6: Impact of multiple bursting clauses on arrivals

- 1ms <5ms <20ms
- 1/ms (1 time in 1ms) >0.4/ms (2 times in 5ms) >0.15/ms (3 times in 20ms)

Generally, pessimism in analysis will become lower the more bursting clauses that are given. However, if the bursting interval is greater than the longest busy period for the system, the arrival rule doesn't give you any benefit. So, in this example, if you know that the system will never run for longer than 20ms before the idle task runs, then Rule 3 will not improve the accuracy of the analysis.

5.9.2 Specifying that something happens just once

A bursting clause of '1 times in forever' means that the arrival of the event can only occur once during the operating cycle of the system. This could be used to represent the triggering of a one-off safety device, such as an airbag in a vehicle.

```
transaction tr0 {
    bursting {
        1 times in forever;
    }
    interrupt air_bag;
}
```

When you have a bursting clause of ‘1 times in forever’ the Analysis Visualizer will automatically search for the worst case scenario for the event to occur and then check whether the system is schedulable if it occurs at that time.

5.10 Modeling Alarms

From the preceding discussion about timelines and transactions it should be clear that there are a number of ways available to model AUTOSAR OS R3.x alarms.

The choice of how to model the alarms depends on two factors:

1. what type of counter, software or hardware, is used to drive the alarms.
2. whether the alarms need to be modified after they are initially started.

Alarms represent independent, asynchronous, releases of tasks into the system. This means that they must be modeled as separate transactions. Even when alarms are driven by the same counter, there is no guarantee that they are synchronized with respect to other alarms on the same counter in the general case. This occurs because the alarms times and periods can be modified at runtime with the `SetAbsAlarm()` and `SetRelAlarm()` API calls. Modeling alarms as independent transactions therefore ensures that worst-case timing behavior of the alarms is modeled.

It is only possible to model task activation from an alarm. It is not possible to model event setting or alarm callbacks.

To model alarms, the following transactions need to be defined:

- a bursting transaction for the counter tick
- a bursting transaction for each alarm attached to the counter

All the transactions should use the same timebase.

It does not matter whether alarms are driven by a hardware or a software counter. In the case of a software counter driven alarm, there will always be a tick of the counter. In the case of a hardware counter, because it is possible to set and reset the alarms independently, in the worst case arrival for the counter interrupt will be an expiry of the hardware counter for every tick. It is only possible to reliably account for reduced interrupt interference if the characteristics of the alarms driven by a hardware counter are fixed. See Section 5.10.2 for further details.

For example, assume a system with an interrupt which ticks a software counter every 1ms. There are two alarms in the system. The first alarm

is single-shot and programmed to expire at a range of values between 10 and 200ms. Each expiry results in a new value being programmed. The second alarm is auto-started and runs at a period of 50ms. It is never modified during runtime.

The following declarations model this system:

```
transaction timer_interrupt {
    bursting {
        1 times in 1 ms;
    }
    interrupt timer;
}
transaction alarm1 {
    bursting {
        1 times in 10 ms;
    }
    task t1;
}
transaction alarm2 {
    bursting {
        1 times in 50 ms;
    }
    task t2;
}
```

When the analysis is run, it will check the alignments of the three transactions to identify the worst case phasing of releases (though it should be obvious that this will be every 50ms when the interrupt and both alarms can occur simultaneously).

Note that there is no difference between the models for the single-shot alarm which is programmed multiple times and the alarm with is programmed to run periodically. For the purposes of analysis these difference runtime programming models have the same net effect in the time domain - they release execution time into the system. The Analysis Visualizer does not need to know how the time is released - simply the durations between time being released.

5.10.1 Reducing Pessimism for a single alarm

In the previous example `alarm1` was modeled using the shortest time for the bursting interval. This is because this represents the worst case between two successive expiries of the alarm. If there is additional information available about how `alarm1` is set and reset at runtime then it would be possible to create a more accurate model. For example, assume that it can be identified that `alarm1` is set in the following sequence:

10 ms, 50 ms, 160ms, 30ms, 200 ms, 10 ms, 50 ms, 160ms, 30 ms, 200 ms, 10 ms, . . .

In this case, the arrivals of the alarm can be more accurately modeled as a timeline transaction:

```
timeline alarm1_sequence {
    timebase tb_millisecond;
    sequence {
        arrivalpoint first_expiry {
            analysis {
                task t1;
                delay 10 ms;
            }
        }
        arrivalpoint {
            analysis {
                task t1;
                delay 50 ms;
            }
        }
        arrivalpoint {
            analysis {
                task t1;
                delay 160 ms;
            }
        }
        arrivalpoint {
            analysis {
                task t1;
                delay 30 ms;
            }
        }
        arrivalpoint {
            analysis {
                task t1;
                delay 200 ms;
            }
        }
        next first_expiry;
    }
}

transaction alarm1 {
    start first_expiry;
}
```

5.10.2 Reducing pessimism for multiple alarms

A common use of alarms an AUTOSAR OS R3.x system is to define their periods at configuration time, autostart the alarms at runtime and not make any runtime modifications to their periods or expiry times. However, if the alarms are auto-started and they are not modified at run-time, this model is unnecessarily pessimistic because it fails to account for alarms that run at an offset to each other.

For example, if two alarms run every 10 ms and activate tasks that run for 2 and 3 ms respectively then the analysis must assume in the worst case that the alarms occur simultaneously, resulting in the lower priority task suffering 2 ms interference from the higher priority task and respond in 5 ms in the worst case. However, if it is known that the second alarm is offset from the first alarm by 5ms then the two alarms will not occur simultaneously and, therefore, the lower priority task suffers no interference and responds in 3ms in the worst case.

If it can be guaranteed that two (or more) alarms on the same counter are auto-started and are not used in `SetAbsAlarm()` and `SetRelAlarm()` at run-time, then this pessimism in the analysis can be removed by modeling the alarms as a timeline transaction as follows:

```
task t1 {
    entry t1_entry;
    profile {
        this priority duration 2ms;
    }
}
task t2 {
    entry t2_entry;
    profile {
        this priority duration 3ms;
    }
}
timeline {
    timebase millisecond;
    periodic start {
        task t1 every 10ms offset 0ms;
        task t2 every 10ms offset 5ms;
    }
}

transaction alarm1 {
    start first_expiry;
}
```

If alarms all the alarms associated with a specific hardware counter are modeled on the same timeline transaction, then pessimism can be reduced further by modeling the hardware counter as a fine activator used by the transaction:

```
activator hardware_counter {
    timebase high_resolution;
    fine;
    driver callbacks {
        now    Os_Cbk_Now_hardware_counter ;
        cancel Os_Cbk_Cancel_hardware_counter ;
        state  Os_Cbk_State_hardware_counter  ;
        set    Os_Cbk_Set_hardware_counter ;
    }
}

transaction alarm1 {
    start first_expiry;
    activator hardware_counter driven by timer_interrupt;
}
```

This will mean that the analysis only accounts for the time released by the hardware counter interrupt when there is an alarm to expire. If the hardware counter was running at 1ms resolution then, instead of the analysis assuming that an interrupt occurs every 1ms, it would now from the transaction that an interrupt will only occur every 5ms, resulting in 80% less interrupt interference from the hardware counter driver.

You are restricted to one fine activator driver per transaction so you cannot use this model if there are alarms driven by the same hardware counter that are not included in the transaction, or there are schedule tables that are driven by the same hardware counter.

5.10.3 Alarms that occur once

Alarms that occur at most once can be modeled trivially as bursting transactions:

```
transaction runs_once {
    bursting {
        1 times in forever;
     }
    task emergency_task;
}
```

This model is only appropriate if the alarm occurs once only. This is not the same as a single-shot alarm (i.e. one with a period of zero) that can be set multiple times during system execution which should be modeled in the way described in Section 5.10.

5.11 Modeling Schedule Tables

Schedule tables can be modeled trivially as sequential timelines. You need to declare an arrivalpoint for each expiry point on the schedule table.

The Analysis Visualizer only allows you to model task releases from a schedule table. It is not possible to model event setting. See Section 6.5 for further details.

The schedule table model in AUTOSAR OS R3.x is very similar to the sequential timeline model in the Analysis Visualizer so each schedule table expiry point can be mapped onto an arrivalpoint. However, there are differences in how delays between releases occur:

Schedule Table	Timeline
Expiry points do not have to occur at the notional zero of the table. The first expiry occurs at a (potentially non-zero) number of ticks from the start of the table.	There must be an arrivalpoint at zero offset from the start of the timeline.
Expiry points occur at an absolute offset number of ticks measured from the start of the table	Arrivalpoints occur at a relative delay from the previous expiry point.
A single-shot schedule table ends when a final delay has expired. The final delay is equal to the duration of the table minus the largest offset.	The timeline ends when the final arrivalpoint is processed. Any final delay is ignored as there are no releases after the final arrivalpoint.

The following sections explain how to map the schedule table model onto the timeline model.

5.11.1 Using Sequential Timelines

Any schedule table can be modeled by a sequential timeline.

Each expiry point on a schedule table has a unique offset. This property ensures that expires are totally ordered. Arrivalpoints in the schedule table model must follow the same ordering. Each arrivalpoint must include the same set of tasks released as found on the expiry point.

The delay to the next arrivalpoint should be calculated by subtracting the offset for the current expiry point from the offset of the next expiry point. If the arrivalpoint models the final expiry point then the offset of the next expiry point needs to be replaced by the duration of the schedule table.

For example, consider a repeating schedule table with a duration of 25ms and the following expiry points:

Expiry Point	Offset	Activated Tasks
ep1	0ms	t1, t3
ep2	3ms	t2
ep3	7ms	t1, t2
ep4	19ms	t2, t3

This could be modeled with the following periodic timeline:

```
timeline schedule_table_duration_25ms {
    timebase tb_millisecond;
    sequence {
        arrivalpoint ep1 {
            task t1;
            task t3;
            delay 3ms; // 3ms - 0ms
        }
        arrivalpoint ep2 {
            task t2;
            delay 4ms; // 7ms - 3ms
        }
        arrivalpoint ep3 {
            task t1;
            task t2;
            delay 12ms; // 19ms - 7ms
        }
        arrivalpoint ep4 {
            task t2;
            task t3;
            delay 6ms; // 25ms (duration) - 19ms
        }
        next ep1;
}
```

Note that because the schedule table is defined as 'repeating' then a `next` clause is needed to tell the Analysis Visualizer that the timeline loops back to the first arrivalpoint.

If the schedule table is modified then make sure that the timeline is updated to match the new configuration. In particular, note that changing an offset for an expiry point will mean that the delay for the associated arrivalpoint and every later arrival point will need to be updated.

Non-zero initial delays

If the initial delay is non-zero then an arrivalpoint is required to capture the start of the schedule table. The following example assumes an initial delay of 10 ms:

```
timeline non_zero_initial_offset {
    timebase tb_millisecond;
```

```
        sequence {
            arrivalpoint fake_arrival_at_zero {
                // Do nothing
                delay 10 ms; // Schedule table Initial delay
            }
            ...
    }
```

This declaration ensures that the Analysis Visualizer correctly accounts for the work done by the schedule table driver to start the schedule table, even though no profiles are released.

Single-Shot Schedule Tables

If the schedule table is single-shot then an arrival point will need to be created to model the end of the schedule table and there must be no next clause specified:

```
    timeline single_shot_schedule_table {
        timebase tb_millisecond;
        sequence {
            ...
            arrivalpoint fake_table_end {
                // Do nothing
                delay 1 tick; // Ignored, there is no next.
            }
        }
    }
```

This fake arrivalpoint captures the work that needs to be done by the OS to change the schedule table start from RUNNING to STOPPED as this does not automatically happen when the last sequence of task releases are made from the final expiry point.

Recall however from Section 5.6.3 that the Analysis Visualizer assumes that a single-shot timeline runs once in the entire run-time of the system. If the single-shot schedule table can be started multiple times then the final arrivalpoint will need to model this as suggested in Section 5.6.3.

5.11.2 Using Periodic Timelines

If the schedule table is repeating then it can be modeled as a periodic timeline. This removes the need to convert between offsets on the schedule table and delays on the sequential timeline as offsets can often be used directly.

Rather than map each schedule table expiry point onto an arrivalpoint, each task reference in any expiry points is modeled with a periodic timeline entry.

The period specified for the task should equal the duration of the schedule table.

The offsets used for each periodic timeline entry are equal to the specified expiry point offsets minus the initial offset (i.e. the smallest expiry point offset in the schedule table). If the initial offset is zero, then this means that expiry point offset figures can be copied directly from the OS configuration.

For example, consider a schedule table with a duration of 50ms and the following expiry points:

Expiry Point Offset	Activated Tasks
0ms	t1, t3
3ms	t2
7ms	t1, t2
19ms	t2, t3
21ms	t1
35ms	t3
40ms	t1, t2, t3
47ms	t1

This could be modeled with the following periodic timeline:

```
timeline {
    timebase millisecond;
    periodic start {
        task t1 every 50ms offset 0ms offset 7ms  offset 21ms
            offset 40ms offset 47ms;
        task t2 every 50ms offset 3ms offset 7ms  offset 19ms
            offset 40ms;
        task t3 every 50ms offset 0ms offset 19ms offset 35ms
            offset 40ms;
    }
}
```

5.11.3 Synchronized Schedule Tables

Synchronized schedule tables allow the delays between adjacent expiry points to be modified at runtime within pre-defined bounds) to re-synchronize the schedule table with an auxiliary (so called “global”) time reference.

The Analysis Visualizer does not need to know about any delays becoming longer as this cannot lengthen the busy period of a task or ISR. However, delays which can be shortened need to be modeled.

One approach is simply to use the shortest delays in the timeline model. The shortened delays are defined by Next.Offset-Next.MaxRetard-This.Offset.

A more elegant way to model this is to use the analysis clause of the arrivalpoint to capture the worst case behavior when the full synchronization adjustment is made as follows:

```
arrivalpoint adjustable {
    task t1;
    delay 20 ms; // Next.Offset-This.Offset
    analysis {
        delay 11ms; //  Next.Offset-Next.MaxRetard-This.Offset
    }
}
```

The analysis delay overrides the arrivalpoint delay whenever it is present.

5.11.4 Schedule Table Transactions

A schedule table transaction needs to reference the first arrivalpoint on the schedule table. If the schedule table is driven by a software counter then the counter should be modeled as a coarse activator and referenced in the transaction:

```
activator software_counter {
    timebase millisecond;
    coarse;
}
transaction schedule_table {
    start schedule_table_first_arrival;
}
```

If the schedule table is driven by a software counter, then the counter will need to be modeled as a bursting transaction at the counter tick rate as it was modeled for alarms:

```
transaction software_counter {
    bursting {
        1 times in 1 ms;
    }
    interrupt tick_software_counter;
}
```

Similarly, if the schedule table is driven by a hardware counter then a fine activator should be created and referenced from the transaction:

```
activator hardware_counter {
    timebase millisecond;
    fine;
    driver callbacks {
        now    Os_Cbk_Now_hardware_counter ;
        cancel Os_Cbk_Cancel_hardware_counter ;
```

```
        state  Os_Cbk_State_hardware_counter  ;
        set    Os_Cbk_Set_hardware_counter ;
    }
}
transaction schedule_table {
    start schedule_table_first_arrival;
    activator hardware_counter driven by timer_capcom_interrupt;
}
```

You are restricted to one fine activator driver per transaction so you cannot share a hardware counter between multiple schedule tables or schedule tables and alarms.

5.12 Auto-started Tasks

Where the system includes auto-started tasks, it is possible to use a simple form of timeline transaction to represent them. For example:

```
timeline {
    timebase tb_sw;
    default readonly;
    sequence {
        arrivalpoint ap_autostart {
            analysis {
                task t1;
                task t2;
                delay 0 ticks; // delay is mandatory here
            }
        }
    }
}
transaction tr_autoactivate {
    start ap_autostart;
}
```

The arrivalpoint `ap_autostart` lists the profiles of the tasks that are auto-started. Note that the timeline starting with `ap_autostart` contains no next arrivalpoint. This means that the transaction will consist of only one invocation of `ap_autostart`. This example is used to represent the situation where t1 and t2 are auto-started.

However, if your system contains other timelines, it is possible to include the auto-started task as the first arrivalpoint of the timeline. This results in a less pessimistic analysis, and is therefore preferred. For example, if the application includes a periodic timeline starting with `ap_periodic_thereafter`, the auto-started tasks could be included as follows:

```
timeline {
    timebase tb_sw;
```

```
        default readonly;
        sequence {
            arrivalpoint ap_autostart {
                analysis {
                    task t1;
                    task t2;
                    delay 2 ms;
                }
            }
            arrivalpoint ap_periodic_thereafter {
                task t3;
                delay 1 ms;
            }
            next ap_periodic_thereafter;
        }
    }
```

This timeline is used by a transaction that starts with `ap_autostart`. The analysis delay in `ap_autostart` represents the delay between starting the OS and the point at which the activator processes the first arrivalpoint, i.e. the point at which an auto-started alarm or schedule table will begin to run.

If your system contains multiple timelines, each auto-started task must be included at the start of the timeline that later activates that task. If no timeline activates the task, it can be included at the start of any of the timelines.

5.13 The Idle Mechanism

If the idle task has been modeled (see Section 5.3.3) this will be because it needs to be included in the analysis. Therefore, it must be included in a transaction.

The simplest way to include the idle task is in a bursting transaction:

```
    transaction tr_idle {
        bursting {
            1 times in forever;
        }
       os_idle_task;
    }
```

Alternatively, the idle task can be modeled as an auto-started task for the purposes of analysis. Thus, it can be included as the initial arrivalpoint of a timeline intended for other purposes as an analysis-only arrivalpoint:

```
    timeline {
        timebase tb_sw;
        default readonly;
        sequence {
```

```
        arrivalpoint ap_idle {
            analysis {
                task os_idle_task;
                delay 10ms;
            }
        }
        arrivalpoint tp_periodic {
            task t5;
            delay 5ms;
        }
        next tp_periodic;
        }
    }
    transaction tr_idle {
        start ap_idle;
    }
```

5.14 A Note on Deadlines

Simple executable objects have an implicit deadline: they must complete before they are next released. In AUTOSAR OS R3.x terms this means that you are limited to modeling basic tasks.

The Analysis Visualizer automatically calculates the deadlines for your system using this assumption. For example, a task in a bursty transaction that executes every 20ms must complete before 20ms of time has elapsed. This is referred to as an implicit deadline.

The Analysis Visualizer provides support for specifying explicit deadlines too, both those that are shorter then the inter-arrival time (the task activated every 20ms might have to generate its response no later then 10ms after arrival) as well as deadlines that are longer than the inter-arrival time (which happens when task activations or ISR arrivals are queued by either the OS itself or by peripheral hardware).

5.15 Summary

- Timebases provide a way to capture time (and other types of event steams) upon which all other analysis objects rely.
- It is good practice to specify execution times in terms of cycles of the CPU clock rate (modeled as an appropriate timebase) and to specify arrival times in terms of physical units like milliseconds, degrees etc. This will ensure that if the CPU clock rate is modified then execution times scale correctly but the system's timing requirements remain constant.

- The execution time characteristics of an object - its execution time and the time for which it holds shared resources and/or disabled interrupts - are captured in a profile.
- Each executable object to be analyzed needs to be declared and used in a transaction. Objects not involved in a transaction will be excluded from analysis.
- A model can include multiple transactions.
- Timeline transactions use arrivalpoints to specify when tasks (and possible ISRs) are released. A shorthand form called a periodic timeline is provided for strictly periodic tasks. Timeline transactions should be used to model schedule tables and can be used to model synchronized alarms.
- Bursting transactions specify which tasks and ISRs are released at the specified burst rate. Bursting transactions are used to model interrupts (for example counter tick sources) as well as direct activations of tasks by ISRs. Bursting transactions should be used to model alarms that are not synchronized.

6 Advanced Modeling

In Chapter 5 you saw how to model useful and practical RTA-OS3.x systems for analysis with the Analysis Visualizer. If you are happy with your models then you should skip ahead to Chapter 7 to see how to analyze your models.

This chapter shows you how to build more complicated models, for example those that use queued task activations and/or buffered interrupts. The chapter also shows how to model the situations where a deadline is shorter than the inter-arrival time (period) for a task/ISR and how to provide more accurate (i.e. less pessimistic) models of execution times.

6.1 Arbitrary Deadlines and Critical Execution Time

The most common form of timing constraint that needs to be guaranteed by schedulability analysis is the deadline . From the occurrence of a particular external event, the time taken for the software to respond must be less than some specified deadline. Within the configuration file, the external event is represented as an arrivalpoint. The time at which the system responds to that event is defined by the profile of the executable object that makes the response.

In Chapter 5, no deadlines were specified. The Analysis Visualizer knows that all computation must occur before a task or ISR is next released and can therefore determine an implicit deadline for a executable object automatically from the timing model.

However, it may be the case that execution must complete before the implicit deadline is reached. For example a task may be released every 10ms but must complete within 2ms. Examples of very short deadlines often occur for releases that might only happen once during the lifetime of a system but which must be completed in a very short time, for example when deploying an airbag.

It may also be the case that the work done by the task or ISR to generate the response completes before then end of execution, for example in the following code the task generates the response and then performs some additional computation before terminating:

```
TASK(A){
    /* Do work */
    GenerateResponse();
    LogDiagnosticInformation();
    TidyUp();
    TerminateTask();
}
```

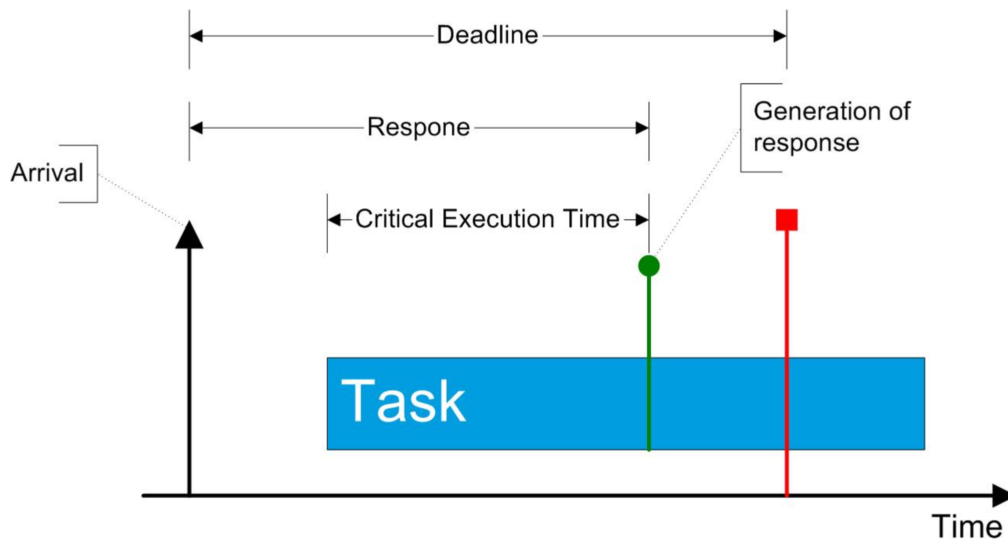


Figure 6.1: Critical execution times and deadlines

Here there is a `critical execution time`, from the task entry function up to and including the call to `GenerateResponse();`, that must complete before the deadline.

The Analysis Visualizer model can capture explicit deadlines and critical execution times for each profile. In order to declare an explicit deadline for a profile, it is necessary to specify a critical execution time and an associated deadline. Figure 6.1 shows these values related to the execution of a task.

The arrival represents the occurrence of the external event that eventually results in the execution of the task. There may be some delay between the arrival event and the start of task execution, for example, a higher priority task or ISR might already be executing.

During the execution of the task, the response is made at the critical point. The critical execution time is the worst-case execution time measured from the start of the task's entry function until the point at which the critical computation is completed.

The deadline gives the longest permitted time between the arrival point and the point by which the critical computation must be completed, and is normally derived from the system requirements.

When defining the value for a deadline, it must be noted that the measurement is from the arrival of the triggering event until the time at which the software executes the instruction representing the critical point. The execution of the critical point will normally result in some response being made to the outside world. The deadline specified should take account of any hardware delays between executing the critical code and the real-world response.

The example below shows a profile that specifies a critical execution time and a deadline:

```
profile {
    this priority duration 1500 cycles;
    critical 500 cycles has deadline 6 ms;
}
```

The critical execution time must be no greater than the execution time for the profile. Note that it can be equal to the execution time for the profile if you want to make an implicit deadline into an explicit one.

There is no constraint on the deadline it can be shorter or longer than the minimum inter-arrival time.

It is not necessary for every profile within a system to define critical points: often ISRs and intermediate tasks have no deadlines directly associated with their activities.

6.2 Multiple Profiles

In the systems we have modeled so far, each task and ISR has been given an execution profile that models its worst case behavior. However, tasks or ISRs can occur in several different execution contexts and execute for different times. If we always use the worst case then this means that the analysis will be pessimistic.

However, the Analysis Visualizer allows additional information to be provided in the model to help reduce pessimism by allowing multiple profiles to be specified for each executable object. Each profile can contains the same type of configuration found in the single profile case.

Multiple profiles are useful when tasks or ISRs have different execution times when called from different contexts, for example:

- An ISR that services several different sources of interrupt and its execution behavior is different for each source.

- A task that implements round-robin scheduling of its activities. For example, the first time it is activated, it performs A; the second time it performs B and so on.
- A task or ISR has different behavior depending on the current application mode.

For example, assuming the following task implementation:

```
TASK(task1){
    if (Condition) {
        /* Short computation. */
    } else {
        /* Long computation. */
    }
    TerminateTask();
}
```

The two paths through the task could be modeled as separate profiles as follows:

```
task task1 {
    entry task1_entry;
    profile short {
        this priority duration 20 cycles;
    }
    profile long {
        this priority duration 1000000 cycles;
    }
}
```

If you declare a profile but do not use it in a transaction then the Analysis Visualizer will assume that the profile does not run and will not include it for analysis.

6.2.1 Identifying and Referencing Multiple Profiles

When an executable object has a single profile then the profile can be referenced using the name of its parent executable object. However, when there are multiple profiles it becomes necessary to reference specific profiles so the Analysis Visualizer can be told *which* profile is released at which time.

Each profile can be allocated a unique identifier by defining a profile name given after the profile keyword. The profile can then be referenced using `objectname.profilename`. For example, assuming a model where `task1` is activated every 10ms but toggles between the short and long paths on each invocation, the two profiles for `task1` would be referenced as follows:

```
timeline {
    timebase tb_short_ticks;
    default readonly;
    sequence {
        arrivalpoint tp_start {
            task task1.short; // Short path
            delay 10ms;
        }
        arrivalpoint {
            task task1.long;  // Long path
            delay 10ms;
        }
        next tp_start;
    }
}
```

6.2.2 Resource Considerations

When constructing multiple profiles for tasks or ISRs that can get resources or disable interrupts, you must consider whether or not each profile gets each specific resource.

In the following example, `Task1` gets resource `Resource1` in one profile and disables OS interrupts in another profile.

```
TASK(Task1) {
    if (Condition) {
        ...
        GetResource(Resource1);
        ...
        ReleaseResource(Resource1);
        ...
    } else {
        ...
        DisableOSInterrupts();
        ...
        EnableOSInterrupts();
        ...
    }
    TerminateTask();
}
```

You only need to specify execution times for the profiles where a resource is used or where an interrupt is disabled. You can enter zero execution times for the profiles that do not lock the resource or disable the interrupt. If any information is missing you may receive inaccurate results from the analysis of your application.

```
task Task1 {
```

```
        entry Task1_entry;
        resource Resource1;
        profile p1 {
            this priority duration 250 cycles;
            resource r2   duration  75 cycles;
        }
        profile p2 {
            this priority duration 100 cycles;
            interrupt priority OS level duration 30 cycles;
        }
    }
```

6.2.3 Restrictions on Profiles and Transactions

A single profile can be associated with:

- exactly one bursting transaction; or
- exactly one timeline transaction (the profile can appear at most once per arrivalpoint).

When an executable object declares multiple profiles then each declared profile must either:

- be associated with a different transaction (for example an ISR with multiple profiles could have each profile in a different bursting transaction or associate each with a different activator); or
- appear on the same timeline transaction (each profile can appear at most once per arrivalpoint and they will need to be buffered (Section 6.3.1).

6.3 Handling Queuing and Buffering

All the timing models in Chapter 5 assumed that deadlines were *equal* to the inter-arrival time - i.e. that executable objects must complete before their next release. The Analysis Visualizer calls this *simple* behavior. Section 6.1 explained how to relax this constraint by showing how arbitrary deadlines could be modeled.

When a deadline is *longer* than the minimum inter-arrival time it follows that one (or more) invocations can be pending at any moment in time. Obviously, if all arrivals are periodic and the deadlines are longer than the period then the system will never be schedulable. However, sometimes the tasks and ISRs may arrive faster than they can be processed for a window of time. You will see this if, for instance, you have to deal with the arrival of bursting messages over a network or if you have a system that queues task activations. In

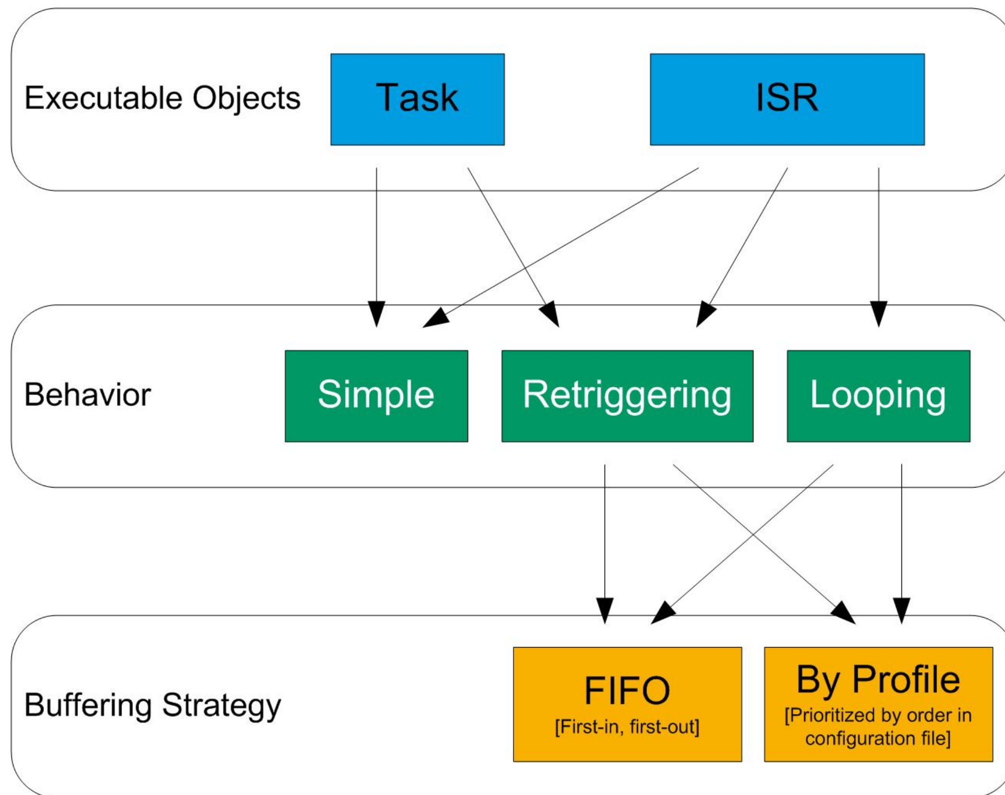


Figure 6.2: Permitted combinations of behavior and buffering strategy

these cases the deadline is longer than the period because the next instance of the task or ISR can arrive before the previous instance was completed.

In additional to *simple* behavior, the Analysis Visualizer can model two additional types of behavior that allow you analyze these types of system called *re-triggering* and *looping* behavior. Re-triggering can be used by tasks and ISRs. Looping can be used only by ISRs. Figure 6.2 shows the permitted combinations of behavior and buffering strategy.

The following sections explain how to uses this feature of the Analysis Visualizer to model task queuing and interrupt buffering.

6.3.1 Queued Task Activation

The Analysis Visualizer allows you to indicate that a task that can be queued by specifying that it is *re-triggering*. Re-triggering tells the Analysis Visualizer that the task will be immediately ready to run again when it terminates.

If you want to model systems that contain queued (BCC2) tasks or if you have a task that activates itself (using `ChainTask()` then you will need to use re-triggering behavior to do this.

As an example, consider the case where a communications controller triggers an interrupt `ist_data` to indicate that data has arrived. ISR `ist_data` transfers the incoming data to a buffer, then activates a task `t_data` to process the data. It is known that `ist_data` can always complete before the next interrupt arrives (and therefore can be implemented as a simple ISR).

```
interrupt ist_data {
    entry ist_data_entry;
    controlled;
    priority 1;
    vector 0xFF;
    resource BufferLock;
    profile {
        this priority duration 43000 cycles;
        resource BufferLock duration 4000 cycles;
    }
}
```

However, `ist_data` activates a task called `t_data` which is relatively complex and cannot always be expected to complete before the buffer is next updated. Thus `t_data` can be declared as a re-triggering task:

Example task declaration:

```
task t_data {
    entry t_data_entry;
    retriggering fifo buffer limit 5;
    resource BufferLock;
    profile {
        this priority duration 350000 cycles;
        resource BufferLock duration 4000 cycles;
    }
}
```

The FIFO buffer limit of 5 is based upon the maximum number of messages that can be contained within the buffer between `isr_data` and `t_data`. This limit specified should match the maximum number of activations that you specify for your queued task activations in the AUTOSAR OS R3.x configuration.

When analysis is performed, the Analysis Visualizer will report if the buffer ever overflows (i.e. whether the maximum activation count for the task is ever exceeded).

The Analysis Visualizer can actually calculate the maximum buffer limit automatically. To use this feature you specify that a buffer limit of `unlimited` and the Analysis Visualizer will report the maximum buffer size required. The previous example could have simply used:

```
task t_data {
    entry t_data_entry;
    retriggering fifo unlimited; // The STC will work out the
        buffer size required
    profile {
        this priority duration 350000 cycles;
        resource BufferLock duration 4000 cycles;
    }
}
```

Multiple Profiles

When a task contains multiple profiles it is possible to tell the Analysis Visualizer how the execution of the queued task activations proceeds. There are two options:

FIFO where profiles are assumed to be released in first-come, first-served order.

By profile where each profile can be assigned its own buffer limit and profiles within the task are assumed to execute in priority order.

Section 6.3.1 has shown how to specify FIFO buffer limits already. To buffer by profile, the task must specify if it is re-triggering and then specify a buffer limit for each profile:

```
task t1 {
    entry t1_entry;
    retriggering; // Activations can be queued
    profile p1 {
        this priority duration 12000 cycles;
        buffer limit 2; // Allow 2 retriggers for p1
    }
    profile p2 {
        this priority duration 500 cycles;
        buffer limit 7; // Allow 7 retriggers for p2
    }
}
```

Retriggers are processed in priority order of the task profiles where the priority of the profile is defined by its occurrence in the Analysis Visualizer configuration file. In the example above, profile `t1.p1` has higher priority than `t1.p2`.

6.3.2 Buffered Interrupts

Interrupts can also be buffered, but this is done by the interrupt hardware rather than by the OS. CAN controllers, for example, often provide some hardware buffering for messages arriving over the network.

There are two ways that ISRs can deal with buffered interrupts:

Looping. The outermost level of the ISR consists of a loop that checks whether unprocessed interrupts remain and, if so, repeats the processing.

Re-triggering. The final instruction(s) of the ISR checks whether unprocessed events remain and, if so, causes the interrupt to trigger the handler again.

The interrupt mechanism on your target platform affects the way that re-triggering is achieved. Usually you must reassert the interrupt.

Normally re-triggering behavior is recommended. There are three factors that will influence your choice of behavior:

1. some hardware will not support re-triggering behavior for interrupts. If this happens, a looping ISR must be used.
2. a re-triggering handler may be better if the interrupt that invokes the handler is at the same level as another interrupt in the system and if that other interrupt has a higher arbitration precedence. Higher arbitration precedence means that it will be handled first if both are pending. This may reduce the amount of blocking suffered by the other interrupt, which is important if your target only supports a single interrupt level.
3. a re-triggering ISR could have a smaller execution time than a looping executable object when a single interrupt is processed. It doesn't matter that a looping handler may be 'more efficient' when several events are handled in one invocation, because the analysis must assume worst-case behavior. This is where interrupts occur in a pattern that results in each one being handled by a separate invocation of the ISR.

If you want the Analysis Visualizer to take account of buffered behavior when analysis is performed, you must specify:

- That buffering is used.

- Whether the buffer is processed by re-triggering the ISR or looping within it.
- The size of the buffer.

Re-triggering

If an ISR is re-triggering then this needs to be indicated in the configuration file:

```
interrupt buffered_isr {
    entry buffered_isr;
    controlled;
    priority 1;
    vector 0xFF;
    retriggering fifo limit 2; // Allow 2 retriggers
    profile {
        this priority duration 200 cycles;
    }
}
```

Looping

Looping is configured in a similar way:

```
interrupt buffered_isr {
    entry buffered_isr;
    controlled;
    priority 1;
    vector 0xFF;
    looping fifo limit 2; // Allow 2 loops
    profile {
        this priority duration 200 cycles;
    }
}
```

Note that the `limit` for the `looping` clause provides an upper bound on how may times the ISR will loop.

Multiple Profiles

Consider an ISR which handles three interrupt sources detected by functions `Source1()`, `Source2()` and `Source3()`.

```
ISR(LoopingHandler) {
  do {
    if (Source1()) {
      /* Handle Source1. */
      /* Clear interrupt source1 */
    } else if (Source2()) {
```

```
        /* Handle Source2. */
        /* Clear interrupt source2 */
      } else if (Source3())    {
        /* Handle Source3. */
        /* Clear interrupt source3 */
      }
    } while (interrupt_pending());
  }
```

Three separate execution profiles are defined for the ISR. They can be characterized by the results of the tests:

- Source1() returns true. The profile for this situation will include the worst-case execution time of the successful check of Source1 handler code.
- Source1() returns FALSE and Source2() returns TRUE. The execution time for this profile will include the worst-case execution time required for the unsuccessful check of Source1, the successful check of Source2 and the Source2 handler code.
- Source1() and Source2() return FALSE and Source3() returns TRUE.

Each of these profiles represents a complete path through the ISR (from the first instruction until the end of the final return instruction) making only one iteration of the do-while loop. Note that no profile exists for the case where all checks fail. This is because there is no way that the interrupt could be entered (or the loop repeated) without one of the above conditions being true.

Representing each of the three profiles within the interrupt declaration is achieved as follows:

```
interrupt LoopingHandler {
    entry ist_handler;
    priority 1;
    looping; // No fifo buffer limit here
    profile pr_source1 {
        this priority duration 10000 cycles;
        buffer limit 2;
    }
    profile pr_source2 {
        this priority duration 15000 cycles;
        buffer limit 3;
    }
    profile pr_source3 {
        this priority duration 35000 cycles;
        buffer limit 1;
    }
}
```

The ISR is the example is defined as `looping`, but no `fifo buffer limit` sub-clause is given. Without this, the order in which the profiles are given represents a priority order. If during analysis three separate transactions release the different profiles simultaneously, profile `LoopingHandler.pr_source1` is assumed to execute first and profile `LoopingHandler.pr_source3` last. A buffer limit is provided for each profile.

If the ISR had been declared as follows:

```
ISR(RetriggeringHandler) {
    if (Source1()) {
      /* Handle Source1. */
      /* Clear interrupt source1 */
    } else if (Source2()) {
      /* Handle Source2. */
      /* Clear interrupt source2 */
    } else if (Source3())    {
      /* Handle Source3. */
      /* Clear interrupt source3 */
    }
    ReassertInterrupt();
}
```

then it would be modeled as:

```
interrupt RetriggeringHandler {
    entry ist_handler;
    priority 1;
    retriggering; // Changed to re-triggering - still no fifo
        buffer limit here
    profile pr_source1 {
        this priority duration 10000 cycles;
        buffer limit 2;
    }
    profile pr_source2 {
        this priority duration 15000 cycles;
        buffer limit 3;
    }
    profile pr_source3 {
        this priority duration 35000 cycles;
        buffer limit 1;
    }
}
```

6.4 Co-operative Tasks

A set of tasks that are co-operative run non-preemptively with respect to other tasks in the set and preemptively with respect to tasks are not in the

set. Within the set of co-operative tasks, each task can tell the OS it can be preempted.

In AUTOSAR OS R3.x co-operative scheduling is achieved by sharing an internal resource between the tasks that are required to run co-operatively and then inserting `Schedule()` API calls in the task bodies to tell the OS that preemption is possible. For example:

```
TASK(Cooperative){
    ...
    Schedule();/* Allow preemption */
    ...
    Schedule();/* Allow preemption */
    ...
    Schedule();/* Allow preemption */
    ...
    TerminateTask();
}
```

The execution time for a co-operative task is calculated in the same way as any other type of task. However, it is necessary to model the blocking time that the co-operative task introduces for co-operative tasks with higher priority.

Each co-operative task is dispatched according to its assigned (or base) priority, but runs at the priority of the highest priority task in the set of co-operative tasks. At each call to `Schedule()`, the task's priority is momentarily lowered to its base priority and the OS checks if a context switch is needed. Conceptually, this is *almost* the same as a task that locks and unlocks a standard resource multiple times during execution as follows:

```
TASK(Cooperative){
    GetResource(Shared);
    ...
    ReleaseResource(Shared);
    /* Allow preemption */
    GetResource(Shared);
    ...
    ReleaseResource(Shared);
    /* Allow preemption */
    GetResource(Shared);
    ...
    ReleaseResource(Shared);
    /* Allow preemption */
    GetResource(Shared);
    ...
    ReleaseResource(Shared);
    TerminateTask();
}
```

Recall from Section 3.4.3 that a higher priority task is blocked at most once during execution and the maximum time for which it is blocked is equal to the maximum time that a lower priority task holds the resource. This means that the blocking time introduced by a co-operative task is the longest of the following durations:

- The task being entered and the first `Schedule()` call.
- Between each `Schedule()` call
- Between the final `Schedule()` call and the termination of the task.

This means that a co-operative task can be modeled as a normal task that locks a standard resource which is used by all co-operative tasks. The resource need not appear in your actual OS configuration file, but it needs to be created in the Analysis Visualizer model. As described in Section 5.4.1, it is good practice to include all lock times in the configuration rather than simply the longest.

For example, a co-operative task with the following structure:

```
TASK(Cooperative){
    /* A */
    Schedule();/* Allow preemption */
    /* B */
    Schedule();/* Allow preemption */
    /* C */
    TerminateTask();
}
```

would be modeled for Analysis Visualizer analysis as:

```
resource Coop;

task Cooperative {
    priority 2;
    locks Coop;
    profile {
        this priority duration 20000 cycles;
        resource Coop duration  5000 cycles; // A
        resource Coop duration  9000 cycles; // B
        resource Coop duration  6000 cycles; // C
    }
}
```

However, this isn't quite accurate because the Analysis Visualizer would allow the task to be preempted between completing `/*C */` and executing

`terminateTask()`[1]. This cannot happen in the co-operative model because the task terminates while holding the resource. The Analysis Visualizer allows a resource lock to be marked as "locked when the task terminates" to model this situation by using the `at exit` syntax:

```
resource Coop;

task Cooperative {
    priority 2;
    locks Coop;
    profile {
        this priority duration        20000 cycles;
        resource Coop duration         5000 cycles; // A
        resource Coop duration         9000 cycles; // B
        resource Coop duration at exit 6000 cycles; // C
    }
}
```

6.5 Modeling Extended Tasks

In general, it is not theoretically possible to determine schedulability for extended tasks because they break the constraint of DMA analysis that forbids tasks from self-suspending during execution. The problem posed by self-suspension is that the worst case response time for a self-suspending task occurs when it remains suspended for the longest time. Unfortunately, this also represents the best-case scenario for all lower priority tasks in the system because this minimizes the interference they suffer due to the self-suspending (and higher priority) task. Conversely, the worst-case interference suffered by the lower priority tasks occurs when the self-suspending task is suspended for the shortest time (typically this would be zero cycles, corresponding to the case that the event upon which the task waits already being set at the point the wait occurs). This represents the best case response-time for the self-suspending task.

In order to determine system schedulability it is therefore required that the self-suspending task suspends for both the shortest and longest times simultaneously. This is obviously impossible.

6.5.1 Re-factoring the application

A solution to this problem is to not allow deadlines to span the time when a task self-suspends. In this case, the system reverts to one where tasks do not self-suspend between being released and generating a response and analysis can proceed as normal. The *User Guide* provides more details on

[1] This is why it was described as *almost* the same as a task that locks and unlocks a standard resource multiple times.

how a system with extended tasks can be re-factored into one comprising basic tasks only. Once in basic tasks only format, it can be modeled as usual.

6.5.2 Partial analysis

If re-factoring is not possible then it is possible to perform a partial analysis of the system. As the Analysis Visualizer cannot determine interference from higher priority extended tasks, it follows that a partial model will only be able to analyze the set of basic tasks that are of higher priority than all extended tasks.

There is are no special changes required to the basic tasks in this set in order to construct this model. However, it is necessary to account for the blocking that the lower priority tasks that are not part of the model introduce on tasks that are part of the model.

The simplest way to do this is by including the required blocking times in the idle task and using the idle task in a bursting transaction that runs once in forever. Even if a lower priority task locks a resource multiple times during multiple invocations, it is sufficient to assume that it is locked once because a higher priority object will be delayed at most once during execution. A `1 times in forever` clause is therefore sufficient to model the blocking from the lower priority task. The Analysis Visualizer will automatically identify the worst place where the blocking could occur when it checks for the alignments of the transactions.

Consider the following system where tasks and ISRs are presented in decreasing priority order:

Task/ISR	Resources			**Task Type**
	r1	**r2**	**r3**	
ISR1	1ms			N/A
Task1	2ms	1ms		Basic
Task2	2ms			Basic
Task3	3ms		5ms	Extended
Task4		2ms		Basic
Task5	4ms		10ms	Extended

The highest priority extended task is Task3. Any lower priority tasks than `task3` cannot be analyzed. However, since Task3, Task4 and Task5 all share resources with the tasks and ISRs that are amenable to analysis, the blocking that they introduce needs to be modeled in the idle task.

```
idle task {
    profile {
        this priority duration 1 cycle;
        resource r1 duration 1ms; // Task3
```

```
            resource r2 duration 2ms; // Task4
            resource r1 duration 4ms; // Task5
        }
    }

    transaction tr_idle {
        bursting {
            1 times in forever;
        }
       os_idle_task;
    }
```

Note that it is not necessary to specify that resource r3 is locked because it is not shared with any tasks in the model and therefore cannot introduce blocking on those tasks when locked by tasks excluded from the model.

6.6 Release Delay & Jitter

In many situations, there is a lag between the arrival of some event and the notional release of the profiles associated with that event. There are many possible causes for this delay, such as slow hardware performing the detection (for example, some A/D converters), or because the release cannot occur until after the event has completed. This delay can be represented by two parameters: release delay and jitter. The release delay represents the shortest time between the event occurring and the notional release of profiles. The jitter represents the difference between the shortest and longest delay time. Thus the sum of release delay and jitter gives the longest possible time between the event occurring and the notional release of profiles.

You can see an illustration of this in Figure 6.3.

The Analysis Visualizer allows you to model the release delay and input jitter for each transaction. The Analysis Visualizer will assume that release delay and jitter are zero if no times are defined. The following example specifies a release delay of 170ns and jitter of 50ns.

```
    transaction delayed {
        release delay 170ns;
        jitter 50ns;
        start some_timeline;
    }
```

Each time a profile in the timeline that includes arrivalpoint `some_timeline` is released, the impact of the release delay and jitter will be taken into account.

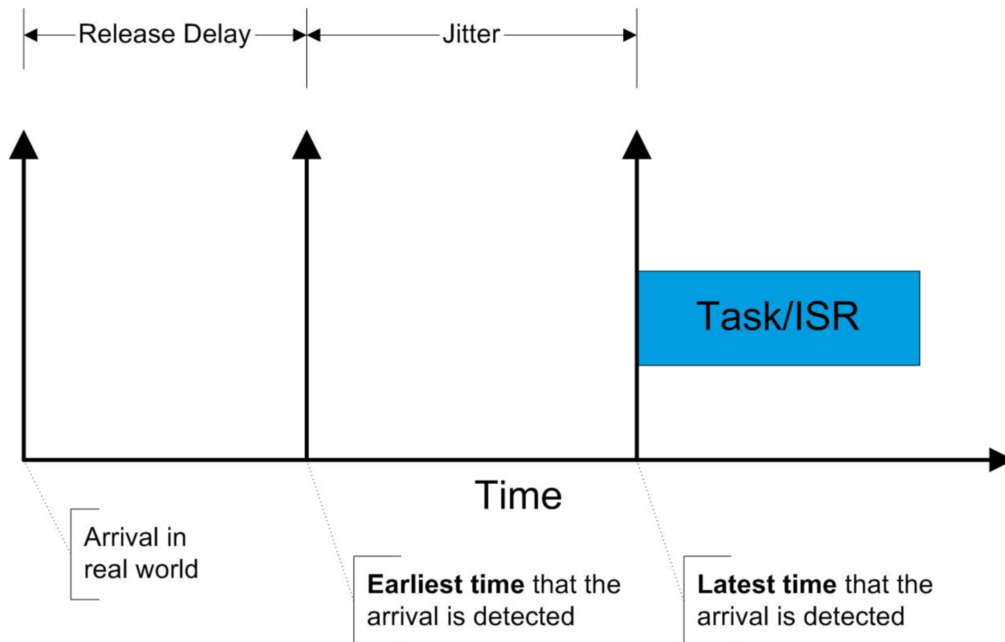


Figure 6.3: Release delay and jitter

If you use a coarse activator then you will have a separate transaction is required for the processed timeline and the activator driver (typically an ISR). Both transactions should normally be declared with the same release delay and jitter.

Bursting transactions can also supply values for the release delay and the jitter.

The following example shows a transaction representing the reception of messages across a network. Messages arrive in bursts of up to five messages every 20ms, with a separation of 1ms between each message. Each message has a minimum transmission time of 300μs and a maximum transmission time of 350μs (which gives a release delay of 300μs and a jitter of 50μs). Upon receiving a message, the ISR `msg_rx` is started, which in turn activates `process_message`.

```
transaction tr_b_delayed {
    release delay 300us;
    jitter 50us;
    bursting {
        1 times in 1ms;
        5 times in 20ms;
    }
    interrupt msg_rx;
    task process_message;
}
```

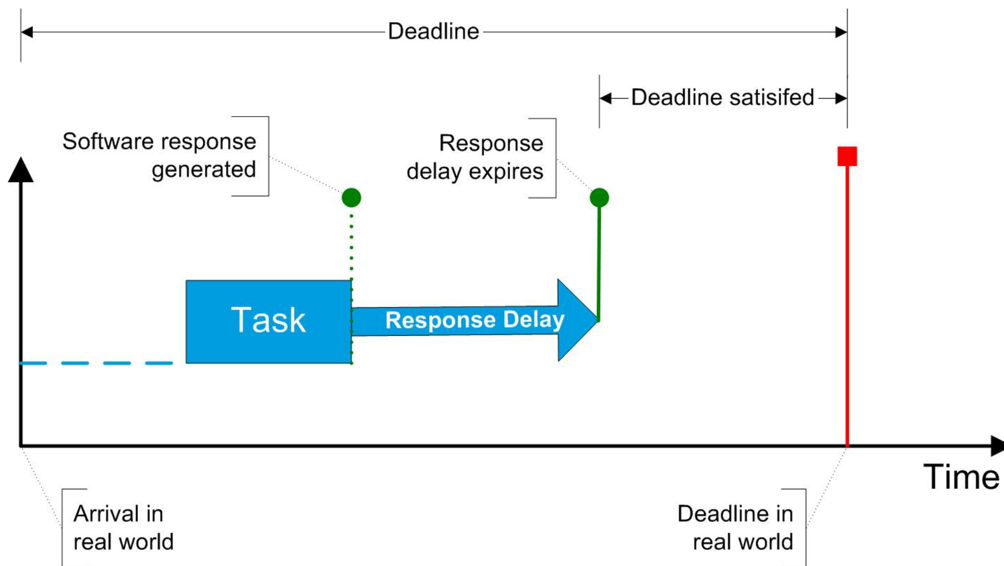


Figure 6.4: Response delay

6.7 Response Delay

Similarly to release delay, there is often a lag between the generation of a response and it actually occurring in the real world. This is called the response delay and it is typically due the impact of inertia of physical devices.

The impact of release delay is trivially a duration which is added to each response time.

Figure 6.4 shows the impact of the release delay on a task's response time.

The response delay is specified by adding an optional `max_response` declaration to a `critical...has deadline` clause as follows:

```
profile {
    this priority duration 25000 cycles;
    critical 10000 cycles has deadline 5 ms max_response 1ms;
}
```

This declaration tells the Analysis Visualizer that a response delay of 1ms should be added to the calculated response time before the deadline is checked.

6.8 Accounting for the OS Overheads

6.8.1 Interrupt Recognition

The interrupt recognition represents the maximum time during which an interrupt will not be recognized by your target hardware. This is a single value and is entered in terms of CPU cycles.

Interrupt recognition time is usually at least equal to the execution time of the longest instruction (unless lengthy instructions can be interrupted part way through). Have a look at the *Target/Compiler Port Guide* and the manufacturers' data book for your target to find out this information.

Interrupt recognition time is specified as follows:

```
interrupt recognition 6 cycles;
```

Interrupt recognition time is treated as blocking time by the analysis. This means that, for the entire duration of the interrupt recognition time, the processor will be executing instructions of a (soon to be interrupted) task, as if no interrupt had occurred. You must make sure that you do not classify interrupt handling overhead as interrupt recognition time.

6.8.2 OS Latencies

In order to provide accurate schedulability analysis, the Analysis Visualizer must be told about how to account for operating system overheads. System timing data describes how many processor cycles key operating system operations consume.

System timing information is specific to a particular hardware configuration. If you change your hardware or locate the application in a different memory area (by moving from on-chip to off-chip ROM, for instance) the system timings will need to be measured again. The values may also differ if you change the characteristics of an application by, for example, adding an extended task or an alarm.

There are 8 core system timings:

task entry latency is the worst-case time taken to context switch into a task. This is the maximum excess time taken to get to the first instruction of a new task, as opposed to simply returning from an OS call without switching.

task switch overhead is the worst-case overhead due to the OS performing a task switch. This is the same as the difference in time between a task activating a higher priority task using an OS call versus making a

non-switching OS call and simply calling the higher priority task's entry function

controlled interrupt entry latency is the worst-case time from the end of the instruction prior to the interrupt being serviced, to the start of the first instruction of the ISR's entry function

controlled interrupt overhead is the worst-case time taken to service a controlled interrupt, excluding the time taken in the handler. It is measured from the end of the instruction prior to the interrupt being serviced to the start of the next instruction in the interrupted code. Assumes no task switch or interrupt in returning from the interrupt.

uncontrolled interrupt entry latency applies only where the OS introduces extra code overhead that executes before the user-supplied uncontrolled interrupt handler. It is the worst-case time from the end of the instruction prior to the interrupt being serviced, up to the start of the first instruction of handler. In most cases, this is zero.

uncontrolled interrupt overhead applies only where the OS introduces extra code overhead on uncontrolled interrupts. It is measured from the end of the instruction prior to the interrupt being serviced to the start of the next instruction in the interrupted code. In most cases, this is zero.

minimum OS level blocking is the worst-case time for which any OS API executes at OS level or above. This is used as the minimum OS level blocking for all tasks and ISRs, apart from the idle task. It will be approximately equal to the longest running API

uncontrolled interrupt blocking is the worst-case time for which any OS call or interrupt wrapper prevents uncontrolled interrupts above OS level from being recognized.

The following example shows how system timings are configured:

```
system timings {
    96;  /* task entry latency */
    274; /* task switch overhead */
    69;  /* controlled interrupt entry latency */
    136; /* controlled interrupt overhead */
    0;   /* uncontrolled interrupt entry latency */
    0;   /* uncontrolled interrupt overhead */
    387; /* minimum OS level blocking */
    0;   /* uncontrolled interrupt blocking */
}
```

System timing values are required for accurate analysis. If you cannot generate these values you will need to supply a set of plausible system timings. You could do this, for example, to scope the timing behavior of a proposed system early in the development lifecycle. If you do not provide any set values, the Analysis Visualizer will assume that they are zero (i.e. that the OS executes in zero time).

6.9 Telling the Analysis Visualizer more

6.9.1 Analysis Only Transactions

In the transactions considered so far, whether bursting of timeline, there is usually an interrupt that occurs which either directly releases tasks or which causes an arrivalpoint on the timeline to be processed. The Analysis Visualizer will typically assuming that the worst case arrival of these transactions occurs when every interrupt becomes pending at the same time.

However, you may know from the implementation of system that some combinations are not possible, for example if one interrupt handler enables a secondary interrupt (say to signify a timeout).

Analysis only transactions can be created that capture this additional information and allow the Analysis Visualizer make the analysis less pessimistic.

For example, consider the timing behavior of an interrupt handler that executes periodically every 10ms and activates a task directly. This could be trivially modeled using a bursting transaction:

```
timeline {
    timebase tb_ms;
    bursting {
        1 times in 10 ms;
    }
    interrupt isr1;
    task task1;
}
```

However, assume that the interrupt also enables a secondary interrupt to occur 7ms later. As the interrupt always occurs as a result of the primary interrupt expiring, it is pessimistic to model this as an interrupt that occurs every 7ms. Instead, this sequencing of interrupts can be modeled using an analysis only transaction:

```
timeline {
    timebase tb_ms;
    sequence {
        arrivalpoint ap {
            analysis {
                interrupt isr1;
```

```
                task task1;
                delay 7 ms;
            }
        }
        arrivalpoint {
            analysis {
                interrupt isr2;
                delay 3 ms;  // isr1 occurs with 10 ms period
                next ap;
            }
        }
    }
}
```

This can then be included in a transaction as follows:

```
transaction tr_simple {
    start ap;
}
```

The start clause indicates that the first arrivalpoint in the transaction is ap.

6.9.2 Removing interrupt pessimism

It is common to use one ISR to tick two different software counters:

```
uint8 count;
ISR(Millisecond){
    count++;
    Tick(MillisecondCounter);
    if (count%1000) {
        Tick(SecondCounter);
    }
}
```

In this case, the `MillisecondCounter` is ticked every time the interrupt occurs and the `SecondCounter` every 1000 occurrences of the interrupt. The worst case execution time for the ISR is when the condition is true, but this only occurs every 1000 interrupts. Assume that the ISR is modeled simply as:

```
interrupt Millisecond {
    ...
    profile {
        this priority duration 150 cycles;
    }
}

transaction {
    timebase tb_ms;
    bursting {
```

```
            1 times in 1ms;
        }
        interrupt Millisecond;
    }
```

This model introduces pessimism in the analysis because the longest path is assumed to occur 999 times more frequently then it really does. This pessimism can be reduced by modeling the interrupt with multiple profiles and then specifying the arrivals with a sequential timeline:

```
interrupt Millisecond {
    ...
    profile normal_case {
        this priority duration 80 cycles;
    }
    profile long_case {
        this priority duration 150 cycles;
    }
}

transaction {
    timebase tb_ms;
    sequence {
        arrivalpoint ap1 {
            analysis {
                interrupt Millisecond.normal_case;
                delay 1ms;
            }
        }
        arrivalpoint ap2 {
            analysis {
                interrupt Millisecond.normal_case;
                delay 1ms;
            }
        }
        ... // 996 additional arrivals
        arrivalpoint ap999 {
            analysis {
                interrupt Millisecond.normal_case;
                delay 1ms;
            }
        }
        arrivalpoint ap1000 {
            analysis {
                interrupt Millisecond.long_case;
                delay 1ms;
                next ap1;
            }
        }
    }
```

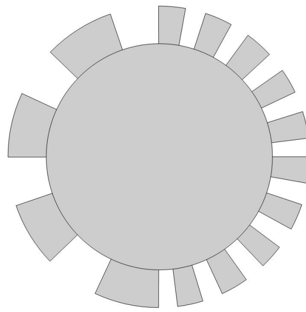


Figure 6.5: Non-uniform timing wheel

```
}
```

6.9.3 Non-periodic timebases

In some applications, where timebases are used to represent the passage of events rather than time, a coarse activator may be triggered at varying intervals. In such a case, a more accurate representation can be created by including the ticks of the timebase as arrivalpoints on a bursting timeline.

For example, consider a toothed wheel (where interrupt `i_wheel` is triggered as each tooth passes a sensor). One half of the wheel has 10 equally spaced teeth and the other half of the wheel has 4 equally spaced teeth, as shown in the Figure 6.5.

A timebase, `tb_wheel`, is defined to indicate the passage of the teeth past the sensor. At the maximum speed, one rotation takes 100ms (this gives a separation of 5ms between the closely spaced teeth and 12.5ms between wider spaced teeth). The arrival of interrupt events can be modeled using a bursting transaction:

```
transaction tr_wheel {
    bursting {
         1 times in 5ms;
        10 times in 50ms;
        11 times in 62.5ms;
        12 times in 75ms;
        13 times in 87.5ms;
        14 times in 100ms;
    }
    interrupt ist_tooth_sensor;
}
```

However, the timebase conversion factor can use only one value, namely the shortest time, as follows:

```
stopwatch conversion {
    on tb_wheel 1 ticks is at worst 5ms;
}
```

If a timeline is defined where a task (t1) is released on every second occurrence of this interrupt, the Analysis Visualizer will treat this as a delay of 10ms between each release of t1 (irrespective of the arrival behavior of the interrupt driving the timeline). This introduced pessimism can be reduced by accurately modeling the delay between activations of t1. This accurate model can be produced by specifying the timeline corresponding to the bursting pattern of the interrupt and associating activations of t1 with this as follows:

```
timeline {
    timebase tb_wheel;
    default readonly;
    sequence {
        // min delay between interrupts is time for smallest
            teeth to pass sensor
        arrivalpoint tp_start { task t1; delay 2 ticks; analysis
            {interrupt i_wheel; delay 5 ms;}}
        arrivalpoint { analysis { task t1; interrupt i_wheel;
            delay 5 ms; } }
        arrivalpoint { analysis { interrupt i_wheel; delay 5 ms;
            } }
        arrivalpoint { analysis { task t1; interrupt i_wheel;
            delay 5 ms; } }
        arrivalpoint { analysis { interrupt i_wheel; delay 5 ms;
            } }
        arrivalpoint { analysis { task t1; interrupt i_wheel;
            delay 5 ms; } }
        arrivalpoint { analysis { interrupt i_wheel; delay 5 ms;
            } }
        arrivalpoint { analysis { task t1; interrupt i_wheel;
            delay 5 ms; } }
        arrivalpoint { analysis { interrupt i_wheel; delay 5 ms;
            } }
        // min delay is time for largest teeth to pass sensor
        arrivalpoint { analysis { task t1; interrupt i_wheel;
            delay on stopwatch 12.5ms; } }
        arrivalpoint { analysis { interrupt i_wheel; delay on
            stopwatch 12.5 ms; } }
        arrivalpoint { analysis { task t1; interrupt i_wheel;
            delay on stopwatch 12.5ms; } }
        arrivalpoint { analysis { interrupt i_wheel; delay on
            stopwatch 12.5 ms; } }
        next tp_start;
    }
}
```

Note that this level of analysis may be unnecessary: if the Analysis Visualizer indicates that the system is schedulable without this expanded timeline, the above expansion can be omitted.

6.10 Summary

- Timing models can specify arbitrary deadlines, provided they are associated with a specification of the critical execution time that must elapse before the response associated with the deadline is satisfied.
- Executable objects can specify multiple profiles if required. Each profile must be uniquely named.
- Tasks and ISRs can include modeling for buffered invocations. This allows queued task activations to be handled and elaborate interrupt handlers (e.g. those handle more than one interrupt source or those that buffer interrupts) to be modeled. Buffer depth can be specified in the configuration or can be calculated automatically by the Analysis Visualizer.
- Cooperative task execution can be modeled by synthesizing the behavior using a standard resource lock.
- Extended tasks cannot be modeled. If a system includes extended tasks, then it is only possible to determine the schedulability of the tasks and ISRs that are higher priority than the highest priority extended task.
- Delays between the arrival of an (external) event and the release of one (or more) profiles can be modeled by configuring release delay and jitter for a transaction. Similarly, delays between the internal response and the external response of a system can be modeled by specifying a response delay.
- OS overheads and target hardware interrupt recognition time can be accounted for in the analysis if true cycle accuracy is required.

7 Performing Analysis

The Analysis Visualizer provides the following types of analysis:

- Schedulability analysis;
- Sensitivity analysis;
- Task Priority Allocation;
- Clock Optimization.

Schedulability and sensitivity analysis are used to tell you about the memory usage and timing behavior of your application. Task priority allocation and clock rate optimization suggest ways that your application can be optimized for either space or time.

7.1 Schedulability Analysis

Schedulability analysis is the most straightforward of the analysis options offered by the Analysis Visualizer. Schedulability analysis operates on a configuration file supplied by the user. The supplied configuration file must describe an analyzable system (that is, it should be constructed using the guidelines in Chapters 5 and 6). For any analyzable system, schedulability analysis reports that a system is either schedulable or not schedulable. If a system is schedulable, this means that all executable objects in the system will always meet all of their deadlines, and will never fail to recognize arrival points. If the system is reported to be not schedulable, this is because either: the system will not meet its deadlines (it is unschedulable), or because it is not possible to determine whether or not the system is schedulable (schedulability is indeterminate or unknown).

Schedulability analysis is used to work out whether each response can be generated before its deadline. The deadline can be either explicitly declared in the profile or implicitly derived from the timing model.

When the schedulability analysis is performed, there are several possible outcomes:

- The system is schedulable.
- The schedulability of tasks or interrupts is indeterminate.
- The system contains unschedulable tasks or interrupts.
- The system exceeds 100% utilization.

Because some assumptions made in modeling the system can be pessimistic, a system that is described as being unschedulable or of indeterminate schedulability may actually be schedulable if the analysis is repeated using

a less pessimistic representation. Note that provided no optimistic assumptions have been made, the Analysis Visualizer will never report a system as being schedulable when it actually is not.

7.1.1 Running the Analysis

Schedulability analysis is performed by selecting the "Schedulability Analysis" tab in the Analysis Visualizer.

Alternatively, it can be run from the command line using:

```
C:\>rtaosanvis --analysis:Schedulability Model.stc Results.html
```

7.1.2 Schedulability Analysis Reports

When the analysis finished, the Analysis Visualizer generates a schedulability analysis report. Each bar in the graphical analysis report shows the response time for the execution profile. Each bar has up to 5 sections:

- Delay/Jitter. This is the maximum amount of between the arrival of the triggering event and the release of the associated task or ISR profile.
- Blocking time. This is the amount of time that the execution profile is prevented from executing by a lower priority profile that holds a shared resource or has disabled interrupts.
- Interference. This is the amount of time that the execution profile is prevented from running by higher priority tasks or ISRs. This is the total amount of time that the profile is preempted during execution.
- Execution time. This is the worst-case execution time that you specified for the execution profile.
- Response delay. This is the time from the response being generated by the software to it being observable in the external environment. This is usually only specified when the response drives some external hardware.

If you have specified explicit deadlines for responses, these are represented on the bar as small tags with the deadline specified. Implicit deadlines are not shown.

In addition to this information, the textual output will tell you about queued activation counts, buffered interrupts and the parts of the system that contribute to the blocking time. For any analyzable system, schedulability analysis reports that a system is either schedulable or not schedulable. If a system is schedulable, this means that all tasks or ISRs in the system will always meet all of their deadlines.

7.1.3 Unschedulable Objects

Individual executable objects within the system may be unschedulable. There are three main reasons for this:

- The simple executable object cannot complete before it is next released.
- The executable object cannot reach its critical point before the deadline.
- The looping/re-triggering executable object has exceeded its buffer limit.

Task or ISR Cannot Complete Before Next Release

Assume a system with the following characteristics:

- Task_1 that is released twice every 3ms (with offsets of 0 and 1ms) and takes 900μs to execute.
- ISR_1 which occurs every 100μs and its handler takes 25μs to execute. The interrupt drives a software counter (modeled as a coarse activator) which drives the scheduling of Task1.

When this system is analyzed, the Analysis Visualizer will produce the result shown in Figure 7.2.

This output shows that neither Task_1 nor ISR_1 is subject to blocking, and that although Task_1 cannot be scheduled, ISR_1 is schedulable. The response time given for ISR_1 represents the longest elapsed time from the interrupt occurring until ISR_1 completes, which is the same as its execution time in this instance.

In this case, the total interference on Task_1 in a 1ms period amounts to 250μs (i.e. the interrupt ISR_1 occurring 10 times in 1ms and taking 25μs execution time), which means that Task_1 does not complete before its next activation. This situation is illustrated in the Figure 7.2. At the completion of a 1ms period, Task_1 has only executed for a total of 750μs.

There are a number of approaches you can use to make this type of system schedulable:

- Reduce the execution time of the task or any other higher priority tasks or interrupts. By reducing execution times you can reduce the amount of interference suffered by lower priority tasks and interrupts.

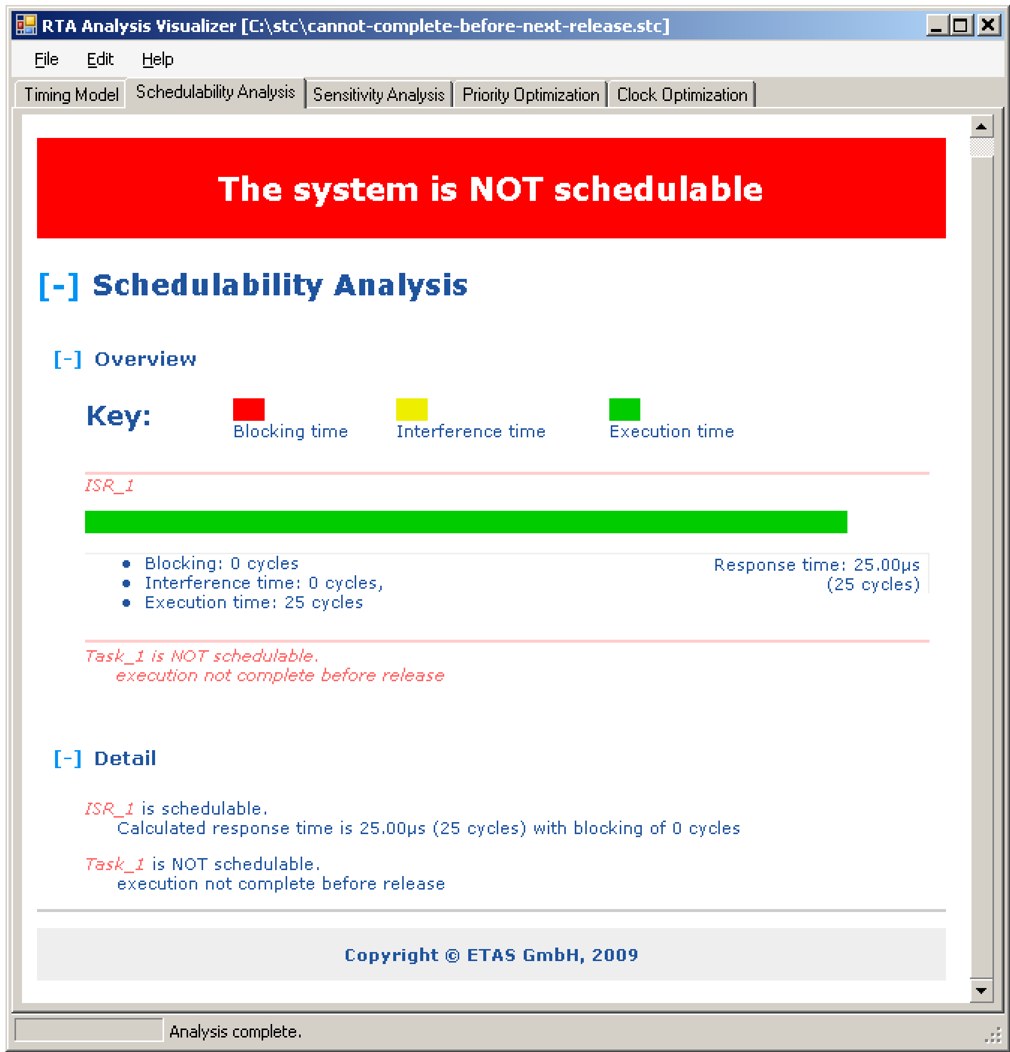


Figure 7.1: Analysis shows task cannot complete before next release

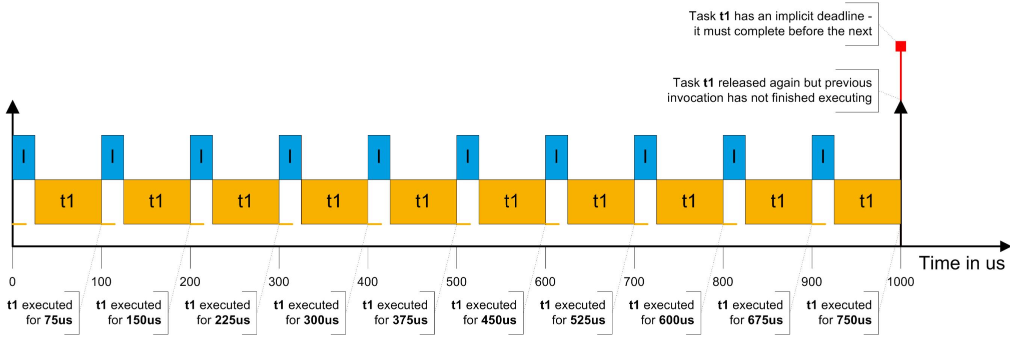


Figure 7.2: Task cannot complete before next release

- If the task or any higher priority tasks or interrupts are periodic, their periods can be increased. If the task being adjusted has multiple offsets, these can be altered.
- Introduce queued activation for tasks or buffer interrupts to ensure that activations made while the tasks or ISRs are executing are not lost.
- Other tasks within the system may be making a specific task unschedulable. You could use best task priorities analysis (which you'll find out more about in Section 7.3) to see if a different priority ordering will make the system schedulable.
- If the unschedulable task or ISR shares a resource with lower priority tasks or ISRs then you could try reducing the amount of time for which the resource is held by these tasks and ISRs. This reduces blocking times and may make the task schedulable.

These measures can also be used where systems are found not to be schedulable for other reasons.

A Task or ISR Cannot Meet its Deadline

If an executable object cannot meet a deadline, this also results in an unschedulable system. There are two different ways in which this is detected:

1. A specific profile is found to be not schedulable because the deadline has been exceeded.

   In this case, other profiles of the same executable object might also be found to be schedulable (or unschedulable). This case applies to all types of executable objects except those that have looping or re-triggering behavior and FIFO execution order.

2. An executable object has been found to be not schedulable because the deadline of one of its profiles has been exceeded. No profiles of this object will be found to be schedulable.

   This case applies only to profiles belonging to looping or re-triggering executable objects with a FIFO execution order.

Consider the following re-triggering interrupt:

```
interrupt isr1 {
    entry isr1_entry;
    uncontrolled;
    priority 1;
    vector 0xA;
    re-triggering fifo buffer limit 4;
```

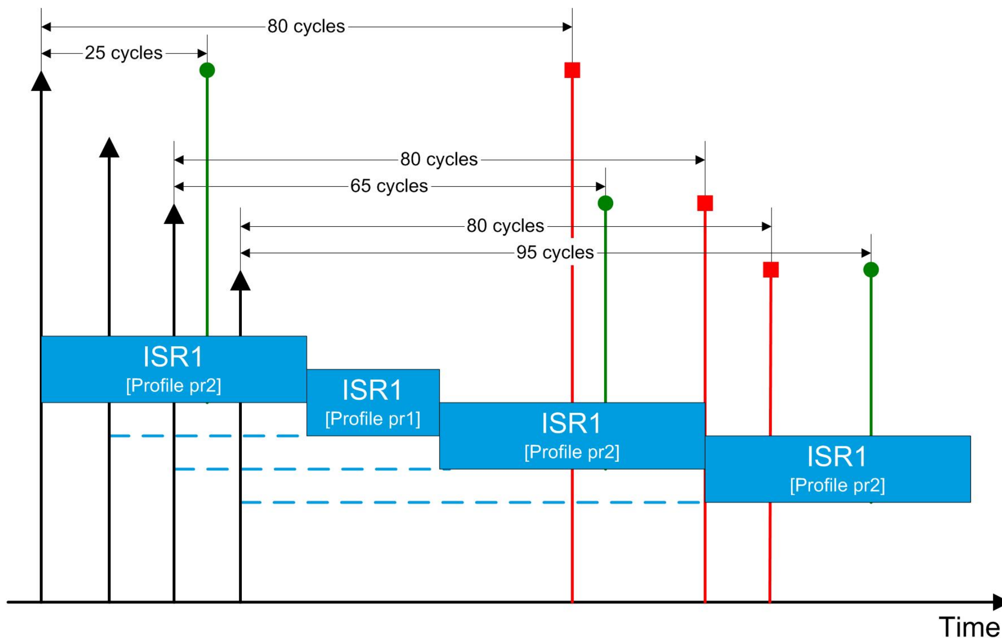


Figure 7.3: Buffered release cannot meet deadline

```
    profile pr1 {
        this priority duration 20 cycles;
    }
    profile pr2 {
        this priority duration 40 cycles;
        critical 25 cycles has deadline 80 cycles;
    }
}
```

If a transaction is declared that can lead to the pattern of interrupts shown in Figure 7.3 then profile pr2 will cause isr1 to be unschedulable.

The first release is handled immediately and starts isr1.pr2. The releases of the interrupt are FIFO buffered until the first instance of isr1.pr2 is complete. The first instance of isr1.pr2 reaches its critical point 25 cycles after the arrival of the interrupt. The second release of isr1.pr2 takes 65 cycles to reach its critical point, just meeting its deadline. The third release of isr.pr2 instance takes 95 cycles and is therefore not schedulable. The Analysis Visualizer will produce the result shown in Figure 7.4.

When a task or ISR is not schedulable because its deadline cannot be met, you can try to:

- Increase the deadline.

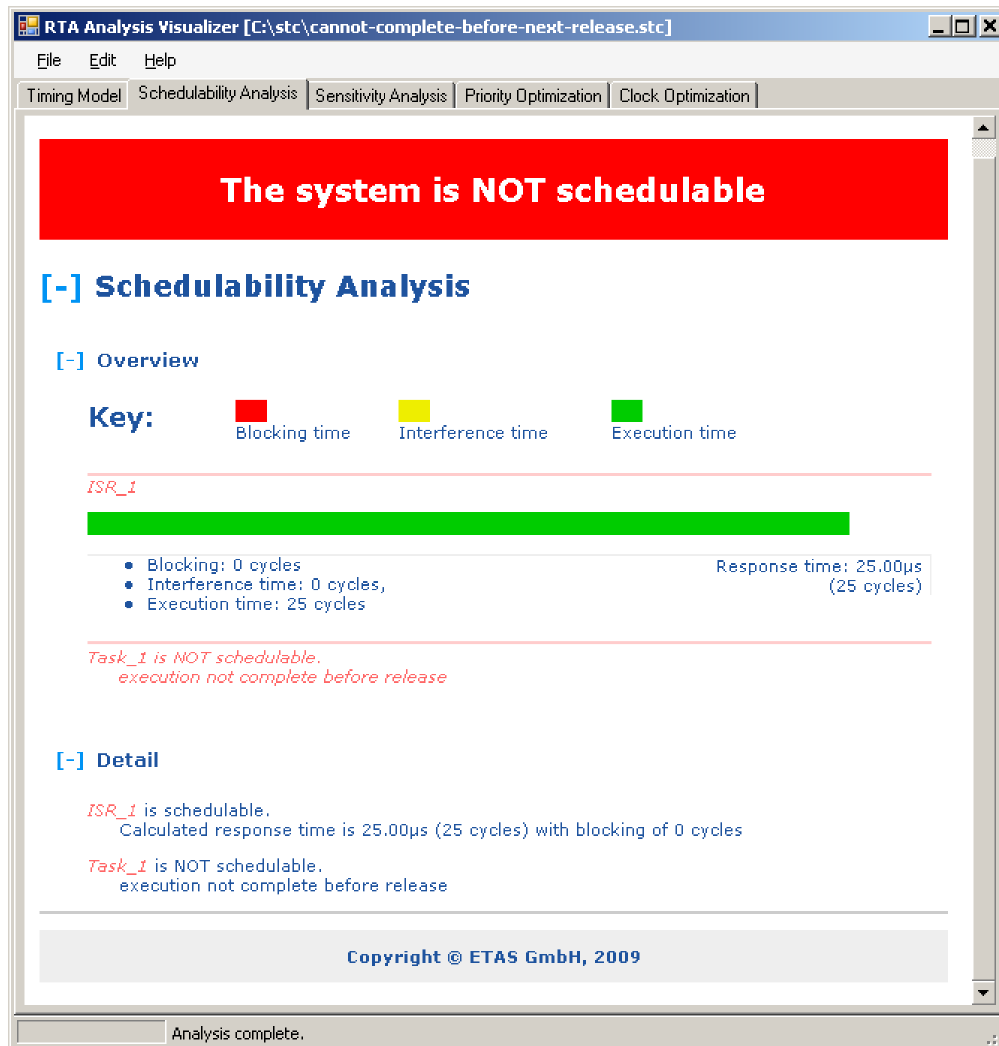


Figure 7.4: Analysis shows task cannot meet deadline

- Move the response generation code earlier in the program. This shortens the amount of time that the task or ISR must execute to generate the response.
- Use the suggestions for unschedulable systems that are mentioned above.

Buffer Limits Exceeded

Sometimes a system will not be schedulable because the buffering for queued task activations is not long enough to hold the maximum number of activations that can occur while the task is running. Similarly, for interrupts that are buffered, the number of interrupts that need to be buffered may exceed the buffer size.

Consider the following configuration:

```
task Task_1 {
    entry task1_entry ;
    re-triggering fifo buffer limit 4;
    profile {
        this priority duration 140 us;
    }
}

interrupt timer_tick {
    entry timer_tick_entry;
    controlled;
    priority 1;
    vector 0xA;
    profile {
        this priority duration 25 us;
    }
}

transaction t1 {
    bursting {
        1 times in 50 us;
        3 times in 170 us;
        5 times in 830 us;
    }
    interrupt timer_tick;
    task Task_1;
}
```

Figure 7.5 shows the results from the Analysis Visualizer for buffered task which takes 140μs to execute and is activated from an ISR with the following bursting rate:

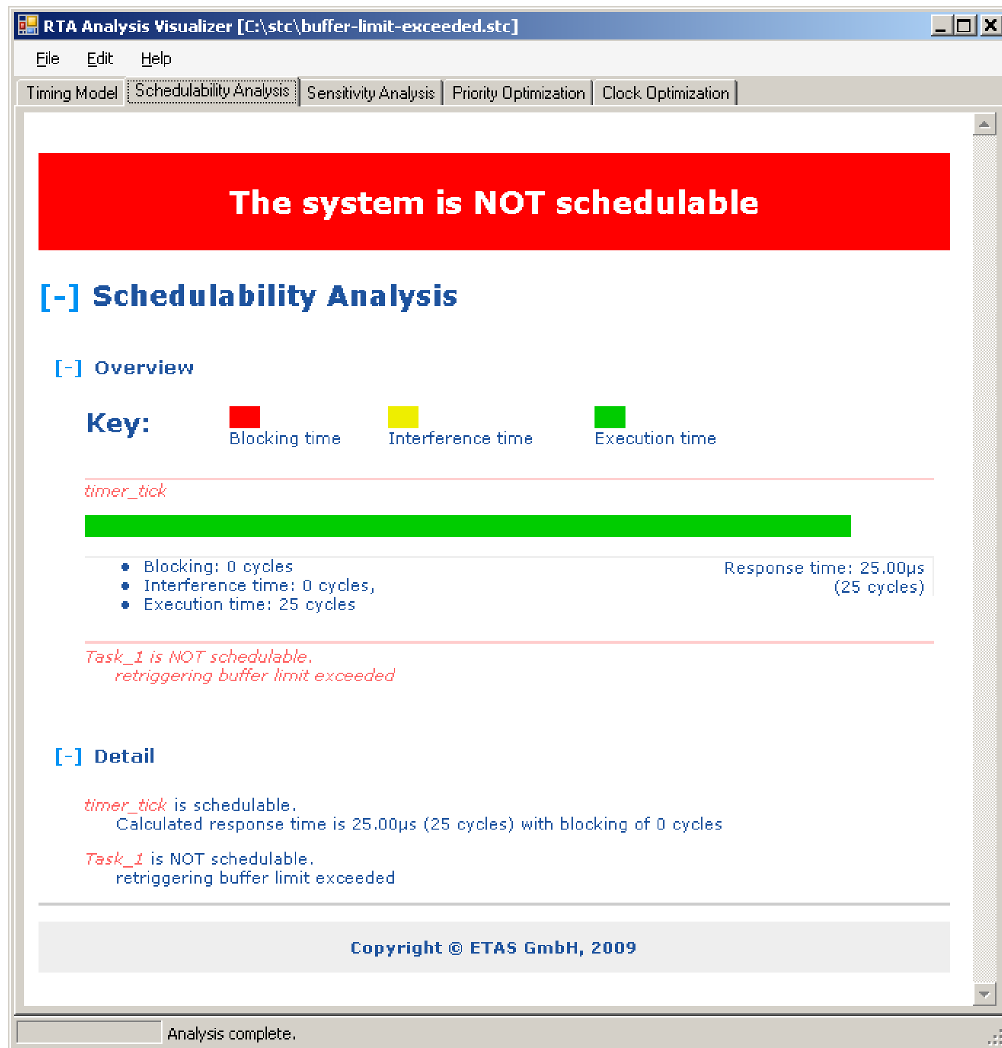


Figure 7.5: Buffer limit exceeded

There are two things that may be causing this problem:

- The tasks or interrupts are being activated more frequently than they can be handled.
- The buffer sizes are too small.

Systems, which are unschedulable for these reasons, can be made schedulable. You can try to:

- Change the priorities to ensure that the task can handle the inputs at a required rate. If you do this, try using best task priorities analysis.
- Decrease the period of the task or ISR.
- Increase the buffer size.
- Decrease the execution time of the task.

If the analysis is repeated with a buffer size larger than needed or a buffer size of unlimited, the Analysis Visualizer reports the buffer size required to make the system schedulable, as shown in Figure 7.6.

In addition to reporting the maximum buffer size, the *maximum retriggers* value is also reported. This value represents the largest number of times that an executable object will be re-triggered before the buffer is emptied. This value will always be greater than or equal to the maximum buffer size.

7.1.4 Utilization Greater Than 100%

The Analysis Visualizer may report that utilization is greater than 100%. This means that your application requires more time to execute than the time available on your target hardware.

There are a number of strategies you can use to fix this problem:

- Increase the CPU speed. This may be possible by specifying a faster part in your hardware design.
- Reduce the execution times for tasks and ISRs.
- Increase the periods for system stimuli.

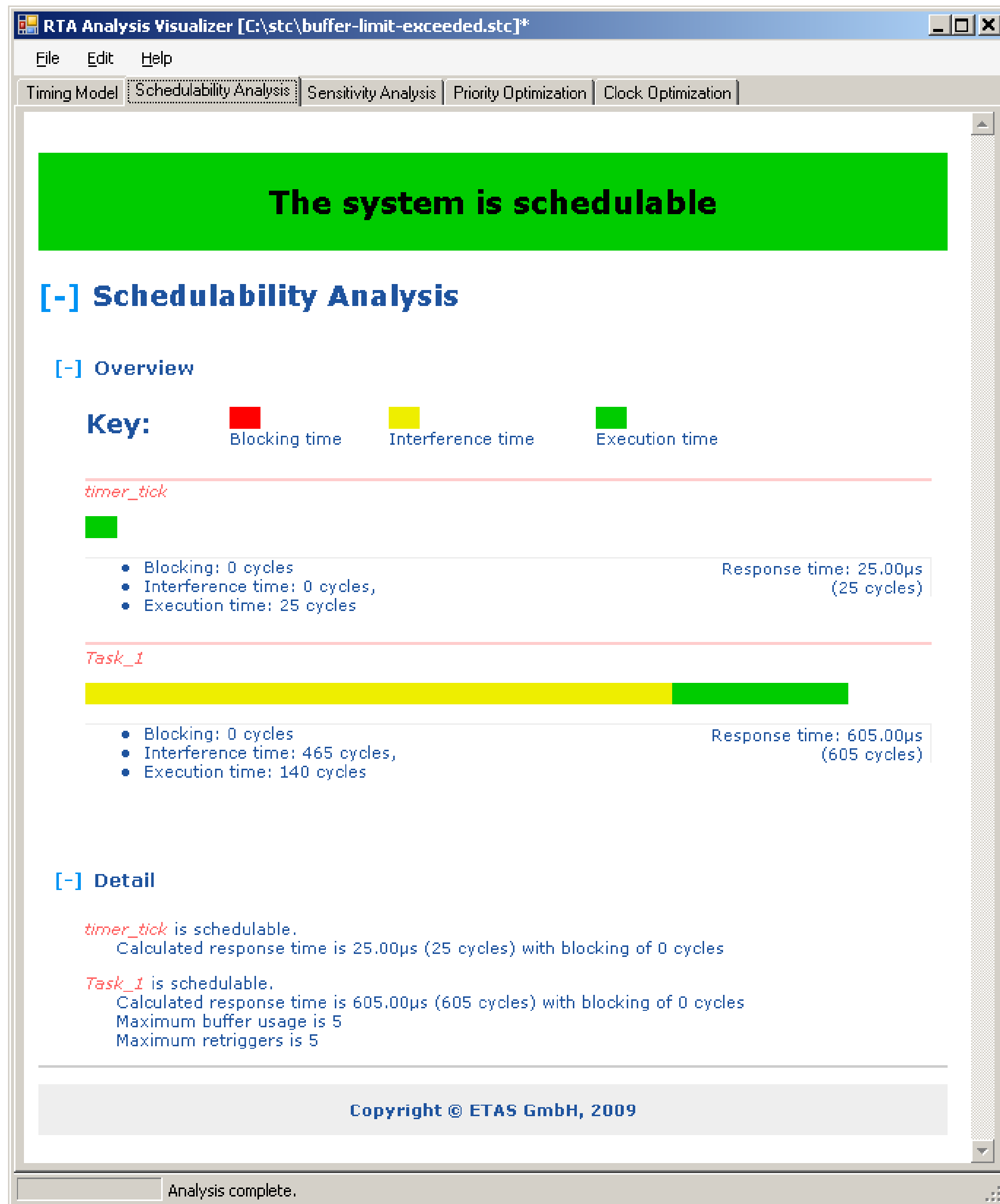


Figure 7.6: Using the Analysis Visualizer to calculate maximum buffer size

7.1.5 Indeterminate Objects

There are two reasons why the schedulability of tasks or interrupts can be indeterminate:

- The busy period for an object can be too long to analyze.
- The schedulability of the object may depend on blocking from a lower priority looping object that isn't schedulable. In this case the blocking factor for the higher priority object cannot be calculated as it depends upon the number of times that the lower priority object loops, so the system must be modified to make the lower priority object schedulable first.

Busy Period too Long to Analyze

If the sum of release delay, jitter and busy period (which is the total time that a task spends in the ready or running state) exceeds the maximum value of the stopwatch (represented as a 32-bit integer), it is considered that the object cannot be analyzed. This case will also generate an error code as shown in Figure 7.7.

It is unlikely that this situation will occur, unless especially long running tasks are defined, or tasks with large buffers allow a long time between the arrival of a task and its processing. However, in such a situation, it is possible to analyze such tasks by changing the stopwatch timebase to cause each cycle to represent multiple cycles of the processor. This does result in some pessimism (since the smallest time that can be represented is some multiple of processor cycles). Note that if you do this, be careful to scale the system timings and interrupt recognition times as these are given in *cycles*.

Indeterminate Blocking from Lower Priority Tasks or ISRs

When the Analysis Visualizer performs schedulability analysis, it attempts to establish the schedulability of the lowest priority executable object first, then progressively analyses higher priority objects. If an object is detected as not schedulable, the Analysis Visualizer will attempt to continue its analysis on higher priority objects. These objects will be marked as "NOT SCHEDULABLE" in the Analysis Visualizer results view as shown in Figure 7.8.

In this case, Task4 is the lower priority task, and is not schedulable. It is not possible to determine whether the higher priority tasks are schedulable or not.

Fixing this situation requires an incremental approach. Make the lower priority task schedulable first then iteratively apply schedulability analysis until your system is schedulable.

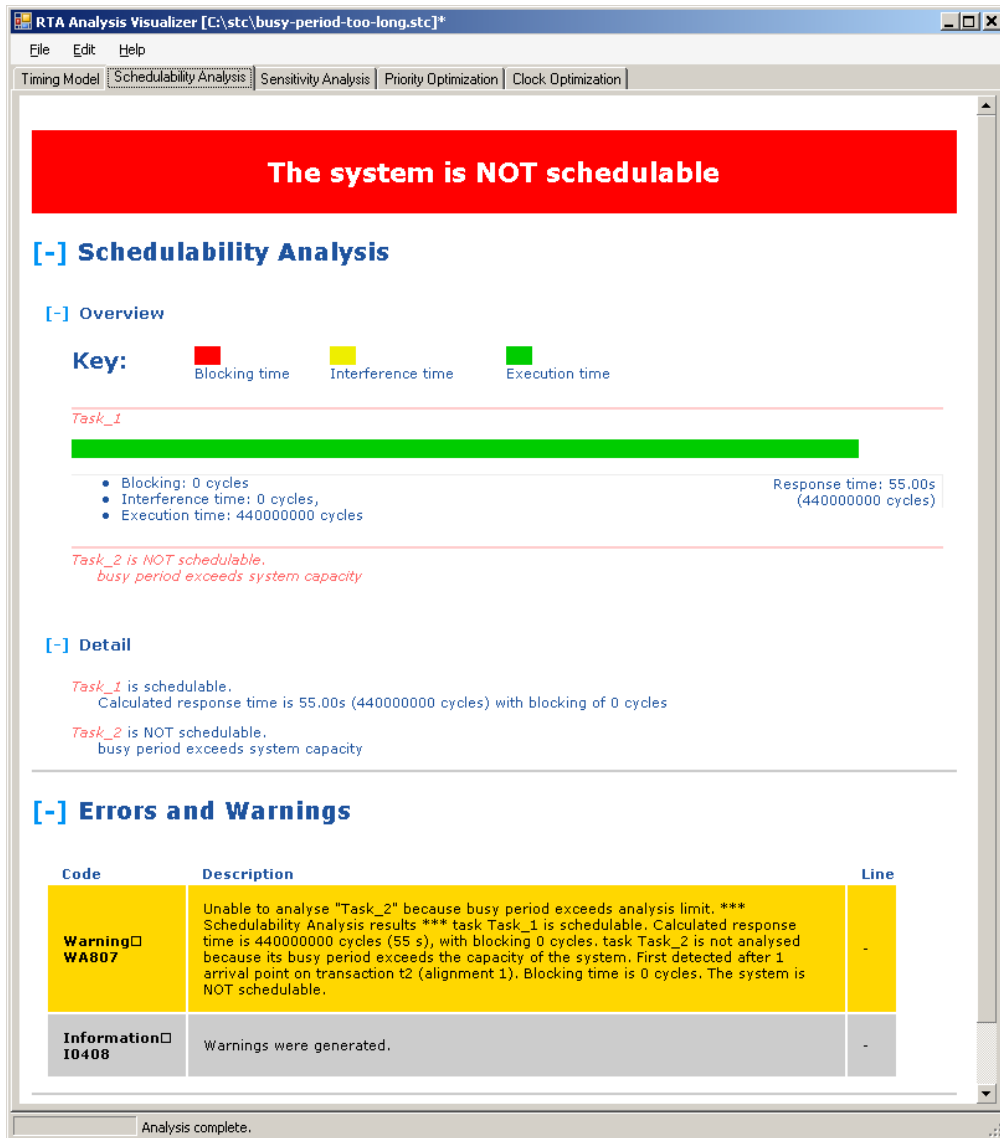


Figure 7.7: Busy period is too long to analyze

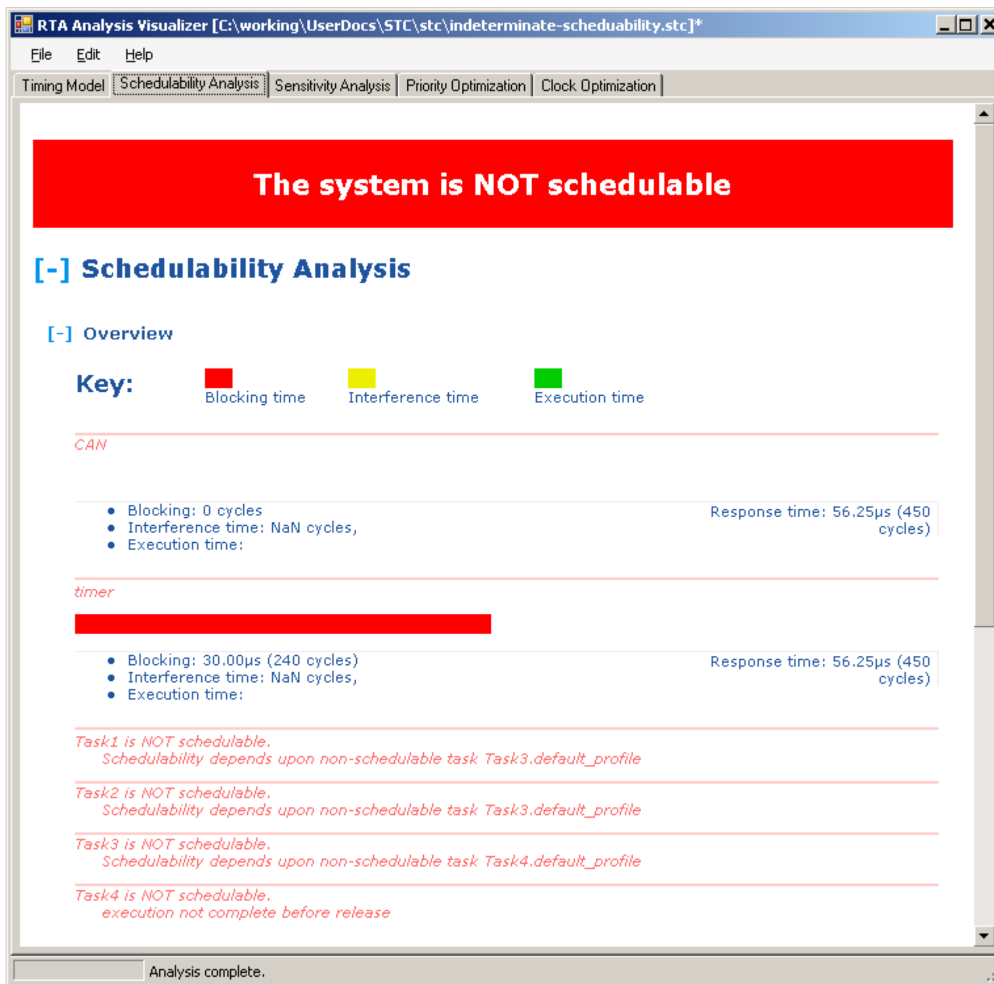


Figure 7.8: Indeterminate schedulability stops higher priority objects being analyzed

7.2 Sensitivity Analysis

Sensitivity analysis is used to explore the boundaries for your system. It allows you to answer questions like:

- What variation of clock speed is feasible?
- What is the maximum execution time allowed for a task or ISR?
- How long can I get a particular resource for?
- How long can I disable interrupts for?
- Can I vary the execution time for critical instant and still meet my deadline?

Sensitivity analysis allows you to determine what changes may make an unschedulable system schedulable. You might be able to optimize task and ISR execution times and a small reduction may be enough to make the system schedulable. Alternatively, if you want to add additional functionality to an existing application then you can use sensitivity analysis to investigate how much headroom is available on which tasks or ISRs.

The sensitivity of the tasks and ISRs is considered against the following factors:

- Sensitivity to processor clock speed.
- Sensitivity to execution times.
- Sensitivity to resource and interrupt locking times.
- Sensitivity to deadlines associated with critical instants.

7.2.1 Performing Sensitivity Analysis

Schedulability analysis is performed by selecting the “Sensitivity Analysis” tab in the Analysis Visualizer.

Alternatively, it can be run from the command line using:

```
C:\>rtaosanvis --analysis:Sensitivity Model.stc Results.html
```

7.2.2 Schedulability Analysis Reports

When the analysis is completed the results are presented in a sensitivity analysis report.

The sensitivity analysis is output in three sections:

1. critical execution time sensitivity
2. system sensitivity to execution and lock times
3. system sensitivity to clock speed

The following sections explain how to interpret the sensitivity analysis results.

Sensitivity to Critical Execution Times

An example output for the critical execution time sensitivity section is shown in Figure 7.9.

In this section, each of the deadlines within the system is considered. It is not possible to meet the deadlines on the critical instants for either task `t1` or task `t4`. However, tasks `t2` and `t3` could have execute for longer in their critical instants and still meet their deadlines.

Note that the maximum critical execution times given are complementary. If all critical values can be set to their reported maximum values the system will be schedulable.

Where it is not possible to meet the deadline you will need to either:

- Reduce the execution time for the critical instant.
- Increase the deadline for the task.
- Increase the clock speed (as recommended by *system sensitivity to clock speed*).
- Reduce the execution time (or decrease the frequency of activation) of any higher priority tasks or interrupts.
- Reduce the time of any blocking times that are applied to the task. Blocking times can arise from any lower priority task that locks a resource or changes interrupt priority levels. Lower priority tasks in a non-preemption group with the current task (or with any task with a higher priority than the current task) can also block the execution of the current task.

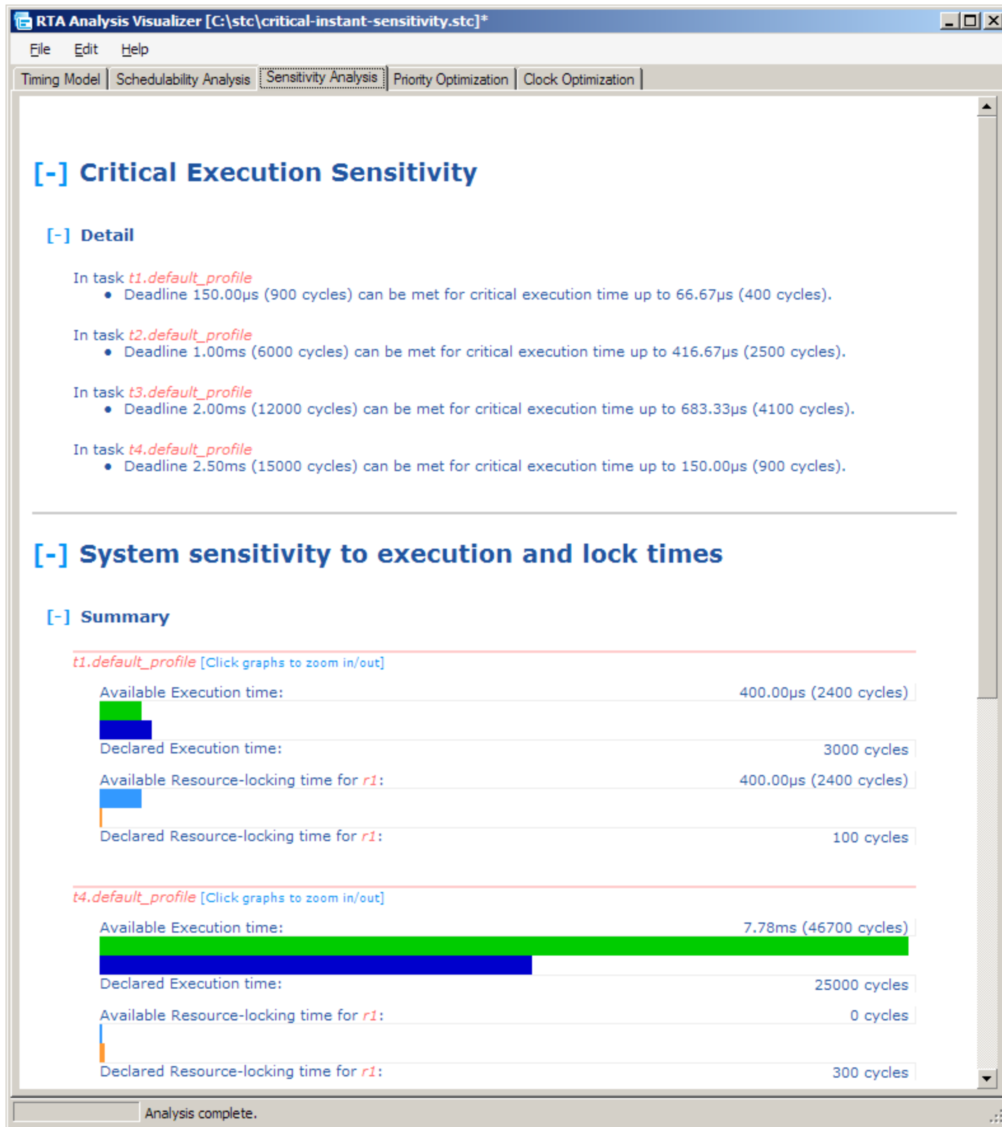


Figure 7.9: Sensitivity to Critical Execution Times

Sensitivity to Execution and Lock Times

The sensitivity to execution and lock times shows:

- The difference between the execution time declared for each task and interrupt and the longest time for which if could run. In some instances this may indicate that a particular task/interrupt can never be schedulable, irrespective of the value of the CPU clock speed.
- The difference between the time declared for each resource lock and the longest time for which it could be locked.
- The difference between the time declared for each interrupt lock and the longest time for which it could be locked.

The reported limits of maximum execution and lock times given in this section of the report are mutually exclusive for each executable object.

Thus for any one executable object, the maximum execution times, resource locking times and interrupt priority level disable times can be changed to the values given in the report for the system to remain (or become) schedulable. No other executable objects should be changed.

When the declared times are less than the maximum times then this gives you an indication of how much additional functionality can be added to a task or ISR and the system remain schedulable.

An example report is shown in Figure 7.10.

When a declared time is longer then the maximum time, then the difference shows you by how much the execution time needs to be reduced to make the system schedulable. In some cases, the Analysis Visualizer will show you that no change to a task or ISR (even it removal from the system) will cause the system as a whole to become schedulable. This means that sensitivity analysis tells you where to direct your optimization effort

An example report is shown in Figure 7.11. This shows that all the tasks are executing for too long for the system to be schedulable. Furthermore, it also shows that even deleting both interrupts would not make the system as a whole schedulable.

To make this system schedulable we could:

- Increase the clock speed (as recommended by *system sensitivity to clock speed*).

Figure 7.10: Sensitivity analysis report showing scope for extension

Figure 7.11: Sensitivity analysis report showing areas needing fixing

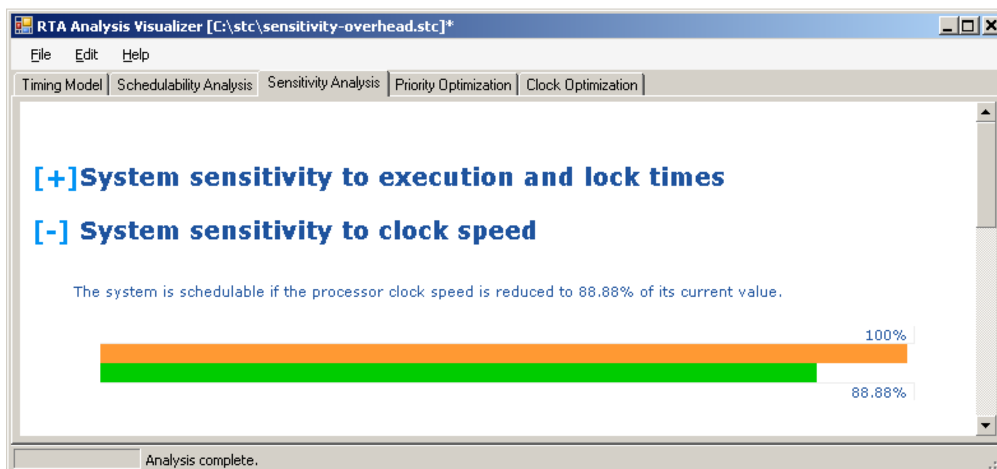


Figure 7.12: Clock speed sensitivity showing scope for reduction

- Reduce the execution time (or decrease the frequency of activation) of any tasks or interrupts with a higher priority than Task3.

Sensitivity to Clock Speed

The current speed that is displayed will always be 100% of the CPU clock frequency. The new speed will be a percentage of the clock speed required so that the system is schedulable.

If the new figure is less than 100%, as shown in Figure 7.12, then there is scope for reducing the clock speed of your target hardware. This can be useful, for instance, in the case of devices that must minimize power consumption.

If the new figure is greater than 100%, as shown in Figure 7.13, then you will have to increase your CPU clock speed to make the system schedulable. The analysis gives you the smallest increase required for your application to become schedulable.

7.2.3 Sensitivity of the Idle Mechanism

There is one case in which the results produced by sensitivity analysis might seem slightly unusual. This is the case where the lowest priority task is executed only once. So far as the schedulability of the system is concerned, the task need never terminate. Sensitivity analysis will attempt to extend the maximum execution time of such a task as far as possible. Since this execution time can be extended indefinitely, the reported maximum execution time is shown as *unlimited*. Figure 7.14 shows an example.

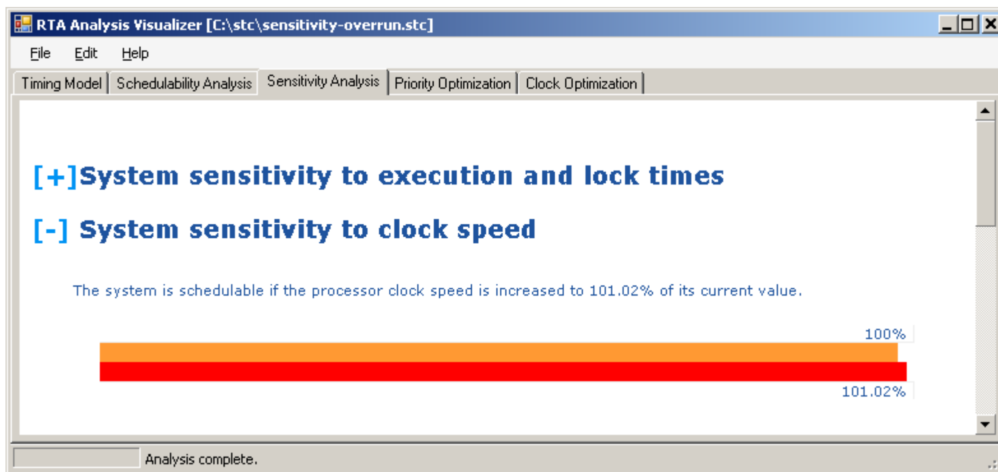


Figure 7.13: Clock speed sensitivity showing need for a higher CPU clock rate

Figure 7.14: Sensitivity analysis for the idle mechanism

7.3 Priority Optimization

Priority optimization aims to make a system schedulable by assigning priorities to tasks, and by placing tasks into soft non-preemption groups. Soft non-preemption groups are a subclass of non-preemption groups generated solely by priority allocation. When priority allocation is performed on a configuration file that initially contains user-defined non-preemption groups and soft non-preemption groups, the user-defined non-preemption groups are included in the analysis and the soft non-preemption groups are discarded.

You should not declare any non-preemption groups in the configuration file as soft non-preemption groups. If you do this, they will not be included in a priority allocation.

Maximizing the number of tasks in a non-preemption group has two key effects on the system:

1. Total required stack for the system is minimized. The worst case stack usage for an arbitrary set of tasks is normally the sum of the worst case stack usages for each of the tasks. In the case of a non-preemption group, the worst case stack usage is the single largest stack usage of any of the tasks in the group.
2. Schedulability of the system may by impacted: the additional overhead of switching from one task to another is not incurred by a non-preemption group, however one task must complete before another, even of higher priority, may begin. Generally speaking a system is expected to become less schedulable as tasks are placed in non-preemption groups, but in certain circumstances schedulability may improve.

The allocation of priorities to tasks is based upon the partial ordering implied by priority constraints and the `activates` clauses of task declarations. Within these constraints, the tool searches for the priority ordering that best allows the system to meet its schedulability requirements.

Allocation of tasks to non-preemption groups proceeds by adding tasks to a non-preemption group. The algorithm keeps adding tasks to non-preemption groups until there is no task that can be added that will result in a schedulable system.

7.3.1 Running Priority Optimization

Priority Optimization is performed by selecting the “Priority Optimization” tab in the Analysis Visualizer.

Alternatively, it can be run from the command line using:

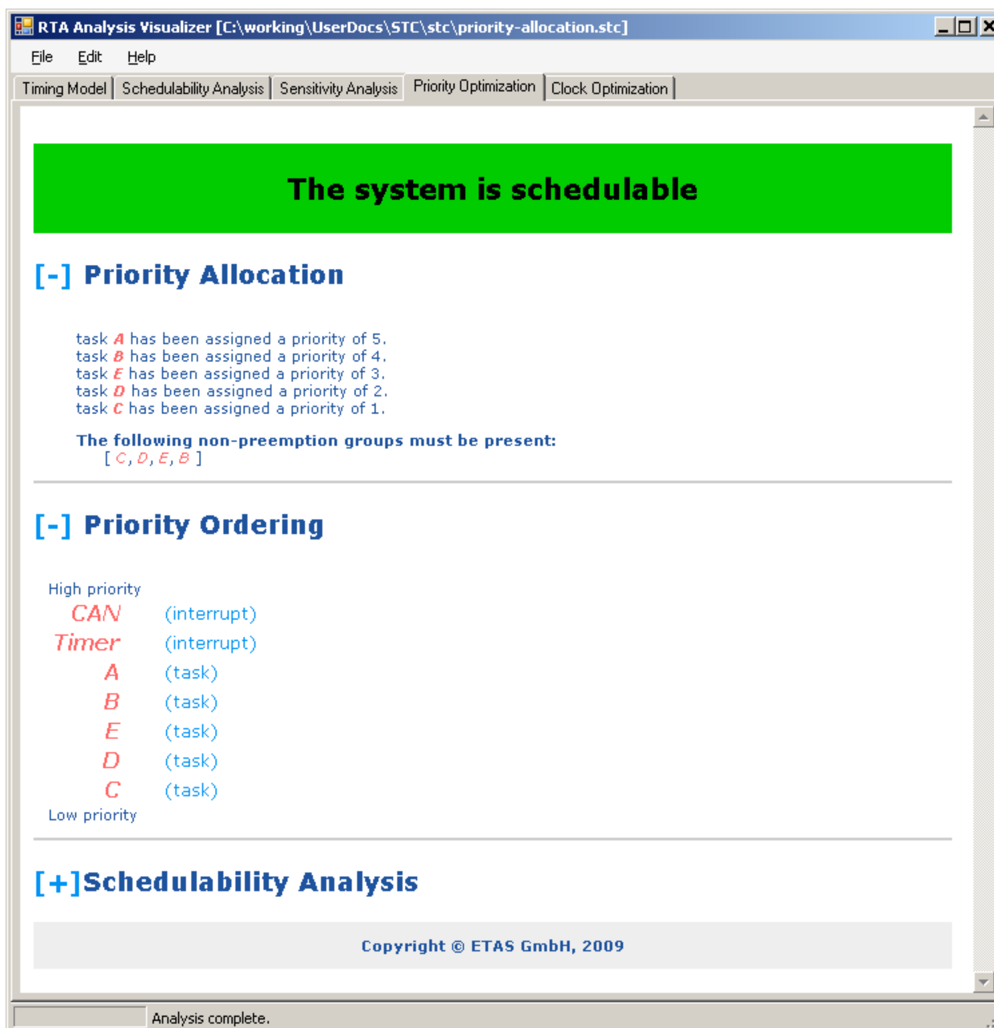


Figure 7.15: Priority optimization

```
C:\>rtaosanvis --analysis:Priority Model.stc Results.html
```

7.3.2 Priority Optimization Reports

Priority allocation will only succeed if some set of priorities that result in a schedulable system can be found. If no such priorities can be found, the Analysis Visualizer reports that the system is not schedulable. Such reports are the same as those described in the chapter on Schedulability Analysis.

If priority allocation is successful, it reports the priority levels for each task in the system and any non-preemption groups to which tasks should be allocated as shown in Figure 7.15.

It also reports the results of the schedulability analysis resulting from this allocation. For example:

In this example, the tasks A to E were declared with decreasing priority:

```
task priority order {
    task A;
    task B;
    task C;
    task D;
    task E;
}
```

After priority optimization, the Analysis Visualizer determines that the best allocation of priorities would be:

```
task priority order {
    task A;
    task B;
    task E;
    task D;
    task C;
}
```

and that all tasks excluding A could share an internal resource.

If priority allocation is not successful, the priority allocation results report which tasks are not schedulable. In the schedulability analysis results, the reasons for this are given in the usual way.

7.3.3 Controlling the Priority Optimization Algorithm

When you configure your system, you declare the relative ordering of task priorities using the task priority order clause (see Section 5.3.1).

When you use the priority allocation facility of the Analysis Visualizer the task priority order clause is not used. However, other information can be given to the priority allocation facility that does affect the eventual priority ordering. Two sources of information exist for this purpose:

1. the `activates` task clause within a task declaration; and
2. a priority constraints declaration.

The priority constraints declaration and the `activates` task clause are both used whenever analysis is performed to ensure that the specified priority ordering is consistent with the constraints.

In general, when using automatic priority allocation, the fewer priority constraints that are placed on the system, the better the priority ordering that can be defined. Therefore, only priority constraints that are absolutely necessary should be given in the priority constraints declaration.

Activates Task Clauses

When a task activates other tasks, it must include an `activates task` clause in its declaration. In addition to being used by priority allocation, this information is used by the Analysis Visualizer to confirm that analysis is possible and to ensure that tasks never activate a higher priority task.

```
task t2 {
    entry t2_main;
    activates t1;
    profile {
        this priority duration 200000 cycles;
    }
}
```

This example shows the situation where task t2 can activate t1. Because tasks can never activate higher priority tasks, providing this information ensures that in the resulting priority allocation task t2 has a higher priority than t1.

Priority Constraints

If it is known that certain tasks must have a higher priority than others (for example to ensure a specific execution order), these constraints must be provided in a priority constraints declaration.

For example, suppose that t3 and t4 are activated together, but t3 prepares some data that is then used by t4 so t3 must execute first. This requires that the following priority constraints are given:

```
priority constraints {
    task t3 higher than task t4;
}
```

7.4 Clock Optimization

Clock optimization is similar in concept to task priority allocation, but it optimizes for time rather than space. It looks for the lowest possible clock rate that gives a schedulable system.

Clock optimization can be used to determine the minimum processor clock frequency at which a schedulable configuration can be achieved. Clock optimization will rearrange task priorities if this results in a system that is schedu-

lable at a lower clock frequency than is possible with tasks at their declared priorities.

Clock optimization can be thought of as a combination of sensitivity analysis and priority allocation:

- As with clock sensitivity, it is assumed that the effect of changing the clock frequency only impacts on execution times (including critical execution times, resource and interrupt lock times). Deadlines and delays between arrivalpoints do not get scaled. Any timelines that use clocks derived from the processor clock will need to have their delays rescaled to compensate for changes.
- As with priority allocation, task priorities may get rearranged. Non-preemption groups, the `activates` clauses in task declarations and the `higher than` declarations in `priority constraints` declarations can be used to impose any necessary constraints on re-prioritization.

Designers of applications for which power consumption or heat dissipation is critical should consider using clock optimization on their final application.

7.4.1 Running Clock Optimization

Clock Optimization is performed by selecting the "Clock Optimization" tab in the Analysis Visualizer.

Alternatively, it can be run from the command line using:

```
C:\>rtaosanvis --analysis:Clock Model.stc Results.html
```

7.4.2 Clock Optimization Report

Clock optimization will only succeed if some set of priorities can be found that result in a schedulable system at some frequency. If no such priorities can be found, the Analysis Visualizer reports that the system is not schedulable.

If clock optimization is successful, it reports the priority levels for each task in the system on the standard output together with any non-preemption groups required to provide the reduction. Figure 7.16 shows the impact of clock optimization on the same configuration that priority optimization was applied in Figure 7.15.

Optimizing for clock speed gives a very different result, with the task priority ordering placing task B as highest priority and suggesting the creation of two non-preemption groups.

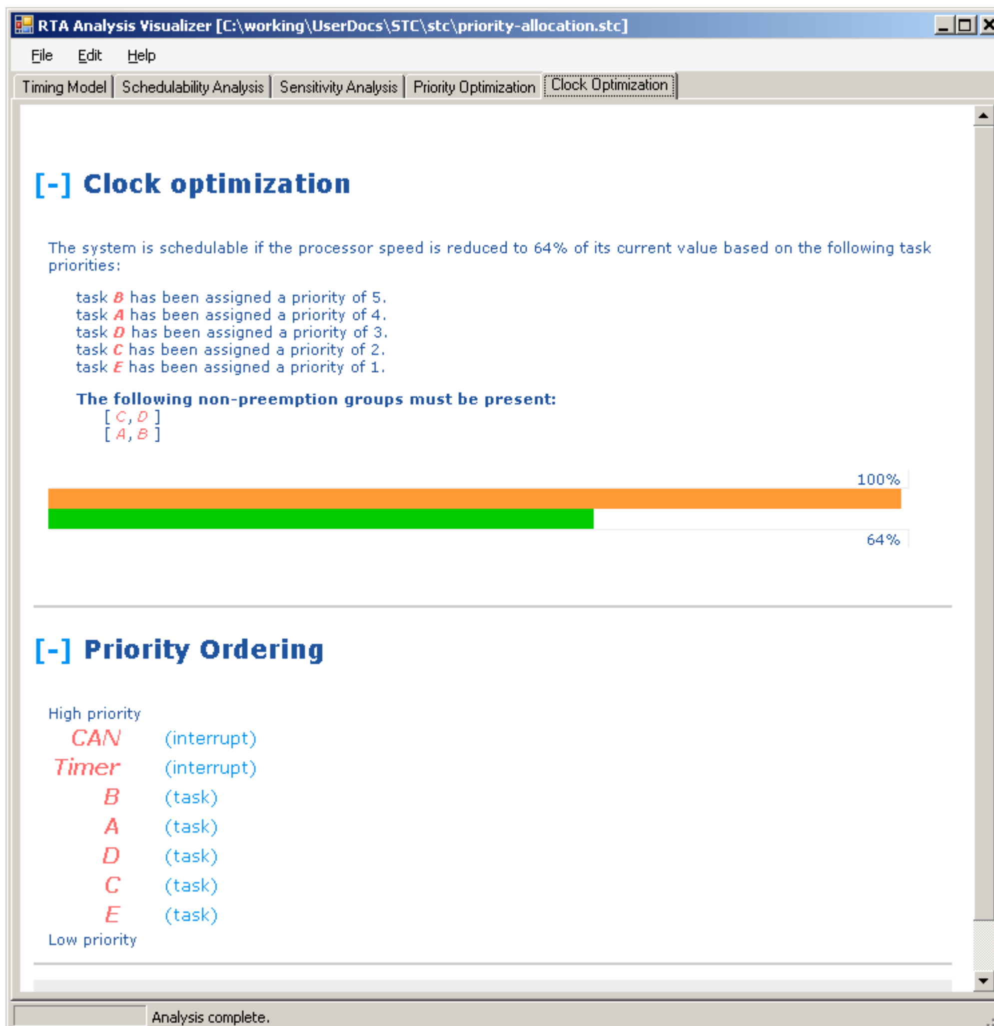


Figure 7.16: Clock optimization

7.5 Summary

- Schedulability analysis tells you whether or not every response deadline in your application will be met at run-time for all tasks and ISRs. If your application is found to be unschedulable, there are a number of approaches your can use to make it schedulable.
- Sensitivity analysis lets you explore the boundaries of your application, either to detect areas that are making your system unschedulable or to look at the scope for possible future enhancements.
- Best task priorities analysis determines the best priority allocation for your tasks, such that the system is schedulable. Required lower priority tasks can be specified for tasks whose execution ordering is important. Best task priorities analysis also determines which tasks can share an internal resource, so that stack space can be minimized.
- Clock optimization is similar to best task priorities, but optimizes for minimum CPU clock rate rather than for minimum stack space.

8 Tutorials

This chapter presents some tutorial examples using the Analysis Visualizer. All examples are fully explained including the correct syntax. However, the reader should work through this tutorial in chronological order, as some later sections are based on previous examples. Readers are also advised not simply to copy the solutions from this document but to write their own configuration files and then to check whether their results match the documented results. Complete files for each of the tutorial examples are provided in the `Documents/RTA-OS3.x Analysis Visualizer Examples` folder of the RTA-OS3.x installation.

8.1 Critical execution times and deadlines

One of the most important requirements of hard real-time systems is that tasks always have to meet their deadlines. Deadlines are usually determined during the design stage of a system and are mainly based on system requirements and hardware restrictions.

The Analysis Visualizer distinguishes between the overall computation time and the critical execution time it takes until a specific critical event has been executed. A critical event, for example, could be the acknowledgment of a message. This critical execution time can, of course, be identical to the overall computation time of a task or interrupt handler. A Analysis Visualizer deadline, therefore, specifies the maximum permitted time from the arrival of an event until a task or interrupt handler has executed for at least the associated critical execution time as shown in Figure 8.1.

Note that the Analysis Visualizer always imposes an implicit deadline on tasks and interrupt handlers: they must complete before they are released again (unless they have been declared as looping or re-triggering, see Section 6.3.2 for more details).

In order to specify a deadline for a critical execution time you have to add a `critical...has deadline` clause to the execution profile of the task or interrupt.

8.1.1 Example

To demonstrate how critical execution times and deadlines can affect the schedulability of a system we consider the following example system:

- A controlled interrupt, which is raised every 4ms, causes interrupt service task ISR_1 to run and tick a coarse activator. The activator, which follows a periodic timeline, activates a single task Task_1 every 100ms.

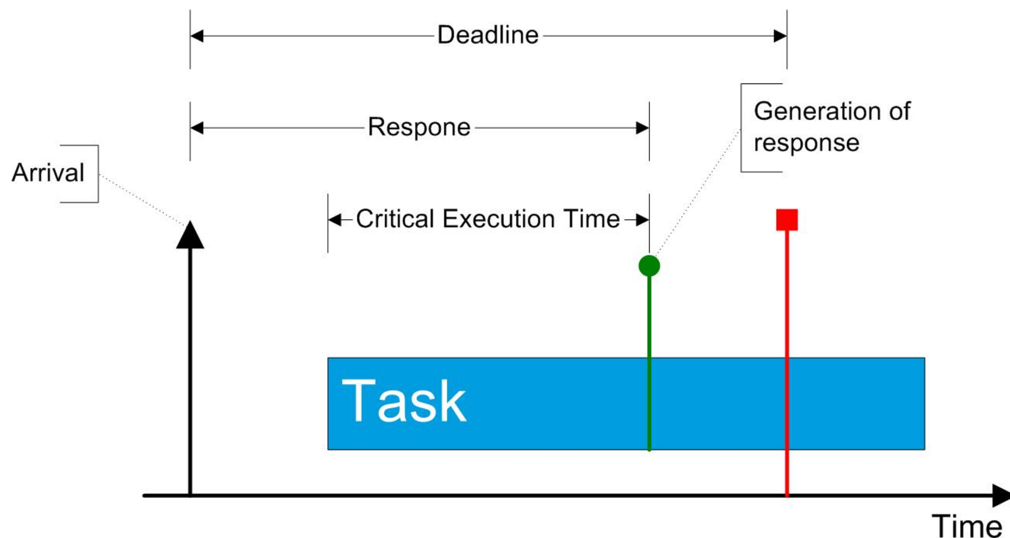


Figure 8.1: Critical execution must occur before the deadline

- Task_1 has a computation time of 1000 cycles and ISR_1 of has a computation time of 200 cycles. We know that Task_1 needs 6ms to write to a specific IO register. The system's specification requires this IO register to be written with a maximum delay of 20ms.
- In this example, we assume the stopwatch clock runs at 100 kHz. In addition, another timebase for the coarse activator runs at a frequency that has been divided down by a factor of 400 from the processor clock.
- Initially, we want to ignore any additional overheads.

You will have to create a new configuration file which contains:

1. Kernel clause
2. Two timebases (`tb_ms`, `tb_coarse_ms`)
3. One stopwatch conversion
4. One task (Task_1)
5. One interrupt (ISR_1)
6. Two timelines
7. One activator (act1)
8. Two transactions (t1, t2)
9. System timings

10. Interrupt recognition

After you have specified the kernel clause you have to declare two timebases; the first one is the stopwatch timebase at 100kHz.

```
timebase tb_ms {
    stopwatch;
    units cycles { define 1 as 1 ticks; }
    units sec { define 1 as 100000 ticks; } //100kHz
    units ms { define 1000 as 1 sec; }
    modulus 65536 ticks;
}
```

You also have to specify a coarse activator timebase for its much slower clock. This timebase is necessary because our example uses a coarse activator which is ticked each time an interrupt is raised. Since the ratio between the clock speeds is 400, the Analysis Visualizer will treat a 100ms delay as 25 ticks of the activator.

```
timebase tb_coarse_ms {
    units sec { define 400 as 100000 ticks; }
    units ms { define 1000 as 1 sec; }
    modulus 65536 ticks;
}
```

Whenever you declare more than one timebase you also have to specify a stopwatch conversion. It allows the Analysis Visualizer to convert every timing value into stopwatch cycles which is the smallest meaningful unit of analysis time and a default unit of the stopwatch timebase. Since we are using real time units on both timebases, we simply make them consistent. Note that the keywords ’on stopwatch’ are optional.

```
stopwatch conversion {
    on tb_coarse_ms 1 sec is on stopwatch 1 sec ;
}
```

In order to specify the execution behavior of `Task_1` you will have to add a `critical ... has deadline` clause to the task’s execution profile. Note that time expressed in profiles refers to the stopwatch timebase, unless you explicitly specify a different timebase.

```
task Task_1 {
    entry task1_entry ;
    profile {
        this priority duration 1000 cycles;
        critical 600 cycles has deadline 20 ms;
    }
}
```

Next, you declare the controlled interrupt. Similar to the task declaration, it contains the execution properties of the interrupt service task ISR_1.

```
interrupt ISR_1 {
    entry ist1_entry ;
    controlled;
    priority 1;
    vector 0xA;
    profile { this priority duration 200 cycles; }
}
```

You will now have to declare a sequential timeline, containing a single arrivalpoint, to model the periodic timing behavior of Task_1. As the task is activated every 100ms, you will need to specify a delay of 100ms on the coarse activator timebase.

```
timeline {
    timebase tb_coarse_ms;
    default readonly;
    sequence {
        arrivalpoint ap1 {task Task_1; delay 100 ms; }
        next ap1;
    }
}
```

Next, you have to model the periodic behavior of ISR_1. You know that ISR_1 runs every 4ms,

which can be easily expressed by creating a sequential timeline with one analysis-only arrivalpoint ap2. In this case, the referenced timebase is the stopwatch timebase tb_ms, therefore the delay is given in units defined for this timebase, i.e. stopwatch milliseconds.

```
timeline {
    timebase tb_ms;
    default readonly;
    sequence {
        arrivalpoint ap2 {
            analysis { interrupt ISR_1; delay 4 ms; next ap2; }
        }
    }
}
```

You will now have to declare a coarse activator which is used by RTA-OS3.x to follow the task's timeline, starting from the first arrivalpoint ap1. You must specify the same timebase you declared earlier in the timeline for your task.

```
activator act_1 {
    timebase tb_coarse_ms;
```

```
        coarse;
        initial ap1;
    }
```

Finally, you have to create two transactions representing the timing behavior of Task_1 and ISR_1. The task's timeline transaction starts at ap1 and runs on the coarse activator act_1, which is driven by the interrupt service task ISR_1.

```
    transaction t1 {
        start ap1;
        activator act_1 driven by interrupt ISR_1;
    }
```

Transaction t2 defines the ISR's timing behavior, using the timeline starting with arrivalpoint ap2. As this transaction describes the interrupt, it does not run on any activator.

```
    transaction t2 {
        start ap2 ;
    }
```

As before, system timings and interrupt recognition time are set to zero, thus ignoring operating system overheads.

```
    system timings { 0; 0; 0; 0; 0; 0; 0; 0; }
    interrupt recognition 0 cycles;
```

After you have saved your configuration file as `critical-instants.stc` it is time to run the Analysis Visualizer. Note, this time, you do not need a priority order clause (or automatic priority allocation), as your system only contains a single task. Open the file in the Analysis Visualizer and select the "Schedulability Analysis" tab.

Provided you typed in everything correctly, the Analysis Visualizer should report the results shown in Figure 8.2.

As you can see from the output, Task_1 is schedulable with an overall response time of 20ms (200 cycles of the stopwatch timebase). More interesting, of course, is the fact that Task_1 will comfortably meet its deadline. The worst case response time for executing the critical section is 12ms, which is well below the permitted deadline of 20ms. As a matter of fact, this system is even schedulable if the critical execution time was increased to 10ms, which means the whole task will always meet its deadline. A look at the output of the sensitivity analysis will confirm this result as shown in Figure 8.3.

This result explains precisely by how much the execution time of each item is allowed to change without making the system unschedulable.

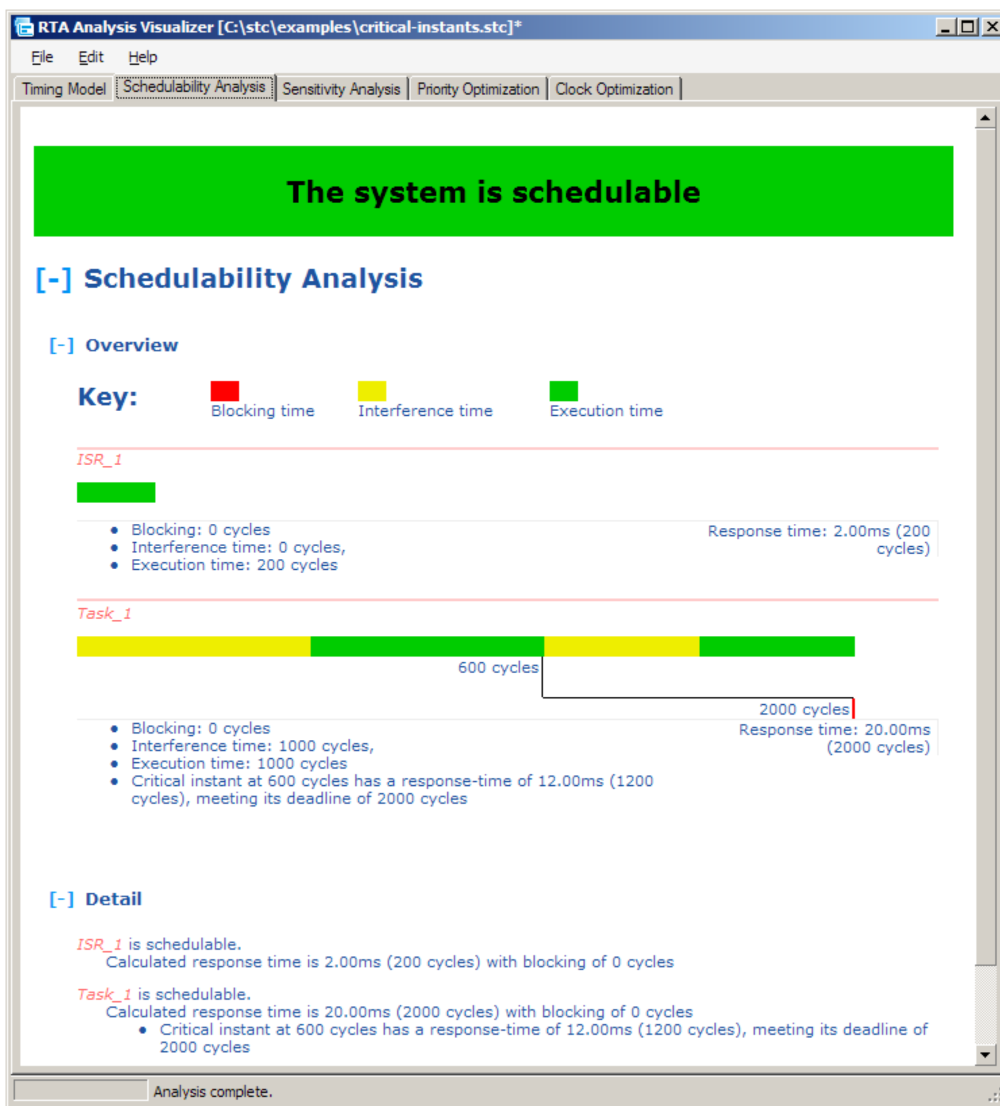


Figure 8.2: `critical-instants.stc` schedulability analysis

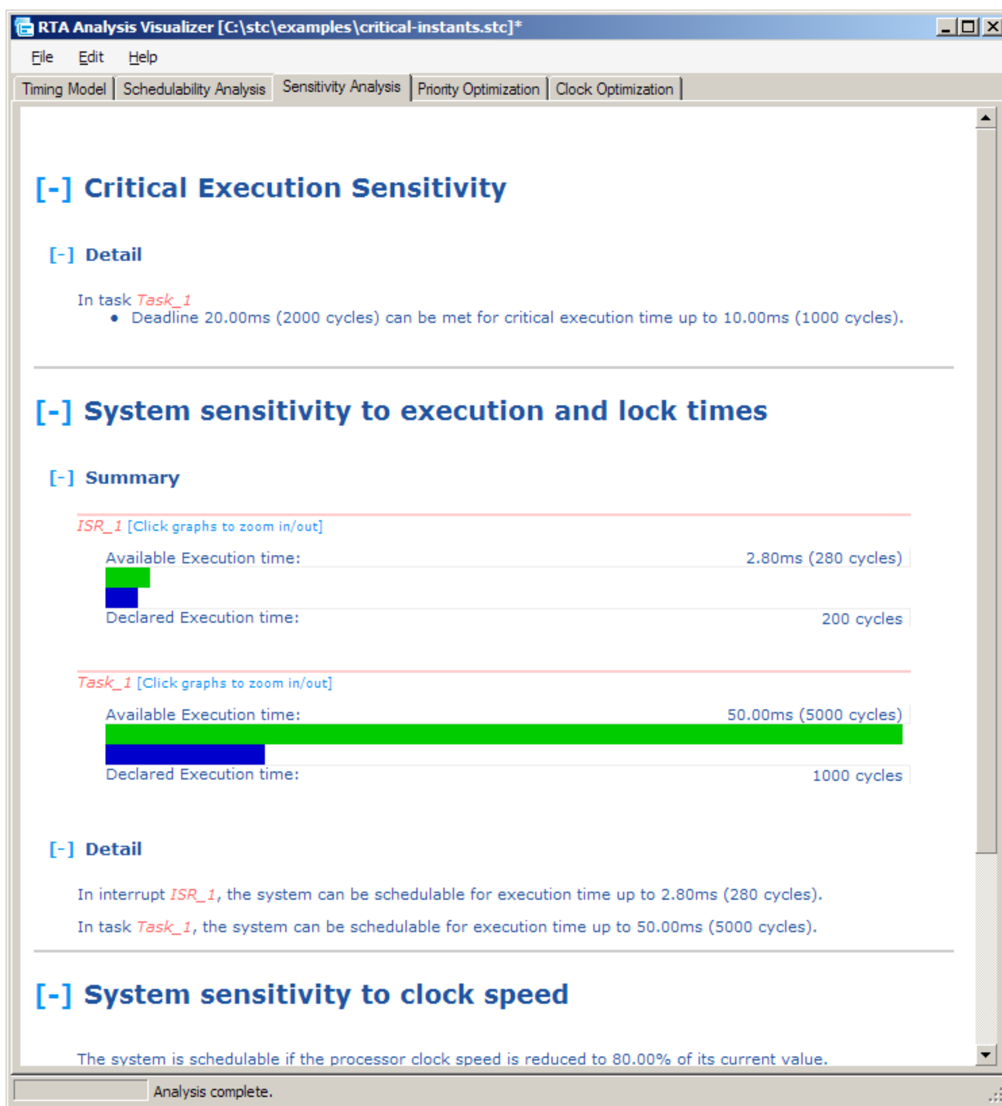


Figure 8.3: `critical-instants.stc` sensitivity analysis

8.1.2 Accounting for Overheads

In order to introduce system overheads you will have to change the system timings and interrupt recognition time values. Edit the configuration file `critical-instants.stc` and change the system timings values and interrupt recognition time as follows:

```
system timings { 10; 20; 30; 40; 0; 0; 50; 0; }
interrupt recognition 15 cycles;
```

These values represent latencies, blocking and overheads. For the interrupt recognition time you can choose 15 cycles. Note, the values in this example are not supposed to represent any realistic platform. Save the file as `critical-instants-2.stc` and rerun the analysis. The Analysis Visualizer will output the results shown in Figure 8.4.

As you can see, Task_1 will still meet its deadline quite comfortably but the response time for both task and ISR has significantly increased. Note, the different blocking times that the task and ISR experience.

Now change the critical execution time of Task_1 from 600 cycles to 1000 cycles (the maximum indicated by sensitivity analysis with system timings set to zero), and rerun the analysis. Figure 8.5 shows the results.

This time Task_1 will not be able to meet its deadline anymore. This is simply because we introduced overheads to task and interrupt execution. Note that the schedulability of ISR_1 has not changed.

The question of course is what is the maximum critical execution time for which Task_1 is still schedulable? Use sensitivity analysis to answer this question.

8.2 Execution profiles

When calculating the execution time of a task or interrupt handler it often turns out that the actual worst-case timing behavior does not occur all the time. Unless the code has been implemented without any conditional statements it is quite possible that the execution time of a task or interrupt handler will vary significantly depending on the state the system is currently executing in. This of course would make any timing analysis very pessimistic, as the analyzer would always have to consider the worst case time for every invocation of a task or interrupt handler. The following code example demonstrates this situation:

```
void Check_Data(void) {
    if (check_level == 1) {
        /* do lots of work */
    } else {
```

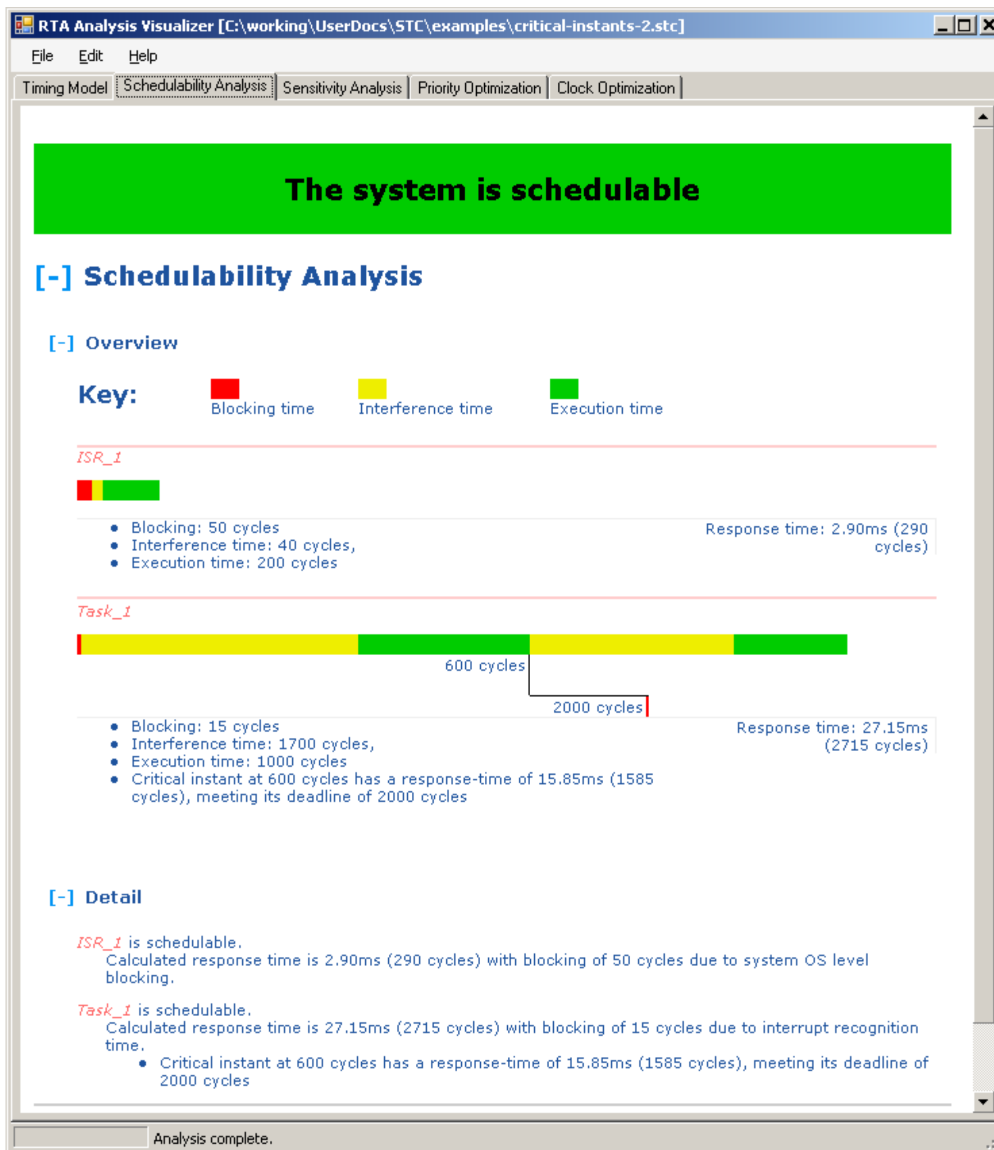


Figure 8.4: `critical-instants-2.stc` schedulability analysis

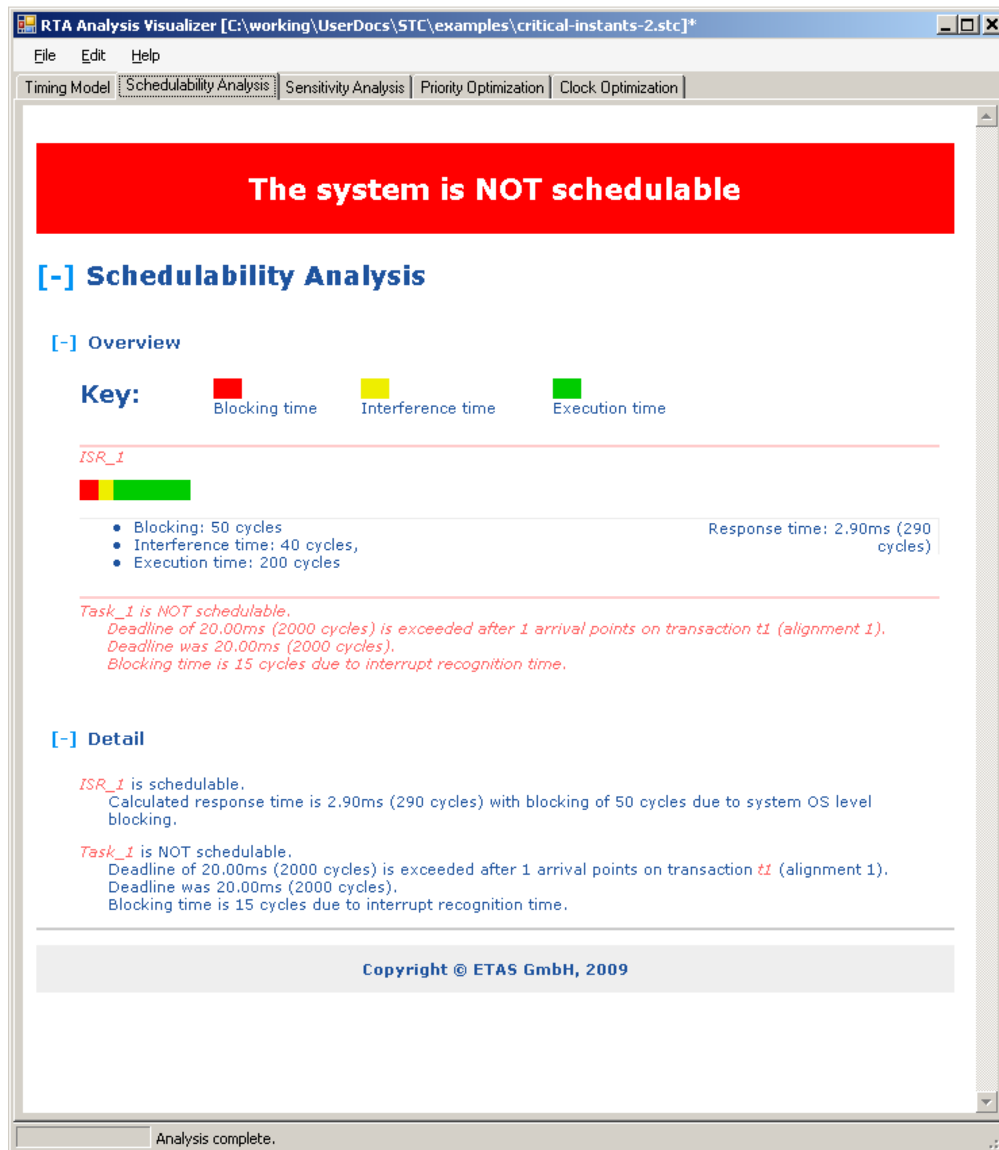


Figure 8.5: `critical-instants-2.stc` schedulability analysis - 1000 cycle critical instant

```
        /* do nothing */
    }
}
```

The timing behavior of the function `Check_Data()` depends on the value of `check_level`. If, for example, the worst case was 1000 cycles, but `check_level~==~1` could only become true every hundredth time, the overall analysis would probably report a rather pessimistic result.

The Analysis Visualizer provides execution profiles to resolve this problem. For every task or interrupt you can specify as many profiles as necessary to describe the detailed timing behavior. If you only specify one execution profile (at least one must be specified) it is not necessary to name it and you can address it by simply stating the name of the task or interrupt, otherwise execution profiles must be named. Named execution profiles are addressed by extending the task or interrupt name with the profile name, separated by a point (e.g. Task_1.profile2).

The following working example will explain how to declare multiple execution profiles in a configuration file.

8.2.1 Example

Before we explain the example system let's have a look at the following source code for an interrupt handler which is supposed to process incoming messages. Work out the number of execution profiles you will have to declare in the interrupt clause.

```
ISR(ist1_entry) {
    confirm_interrupt();
    get_message();

    if (message == START_MESSAGE) {
        /* initialize global work buffer */
        initialize_buffer();
    } else if (message == END_MESSAGE) {
        /* activate Task_1 to process message */
        ActivateTask(Task_1);
    } else {
        /* add message to working buffer */
        add_message_to_buffer();
    }
}
```

In this example, an interrupt ISR_1 is triggered by one of three possible causes: a START_MESSAGE notification, data contained within the message or an END_MESSAGE notification. When the END_MESSAGE notification has been

received, Task_1 is activated which processes the message and writes information to various log-files.

- The interrupt service task ISR_1 handles all messages. An end-to-end message sequence (START_MESSAGE - add message to working buffer - END_MESSAGE) takes place every 300 microseconds. There is a delay of 100 microseconds between each part of a message sequence.
- There are three different execution paths which all need to be analyzed separately. The specification states that the START_MESSAGE execution path has a worst-case execution time of 2 cycles, the END_MESSAGE execution path of 3 cycles and the third execution path of 7 cycles.
- The task Task_1 has an overall execution time of 10 cycles. It has to complete its critical section of 2 cycles (processing the message) before the next message is added to the buffer, giving it a maximum deadline of 200 microseconds.
- A second 'polling' interrupt ISR_2 occurs every 90 ticks. The initial design specifies that the interrupt service task ISR_2 will execute for 10 cycles .
- ISR_1 should execute at a higher interrupt priority level than ISR_2.
- A default stopwatch timebase; all overheads can be ignored.

In the configuration file, in addition to the standard clauses, you will have to declare the task, the interrupts, two analysis-only timelines to model the workflow and two transactions to follow those timelines. This is the structure the configuration file should eventually have:

1. Kernel clause
2. One timebase
3. One task (Task_1)
4. Two interrupts (ISR_1, ISR_2)
5. Two timelines
6. Two transactions (t1, t2)
7. System timings
8. Interrupt recognition

The task declaration has to specify the deadline for the critical section of the task:

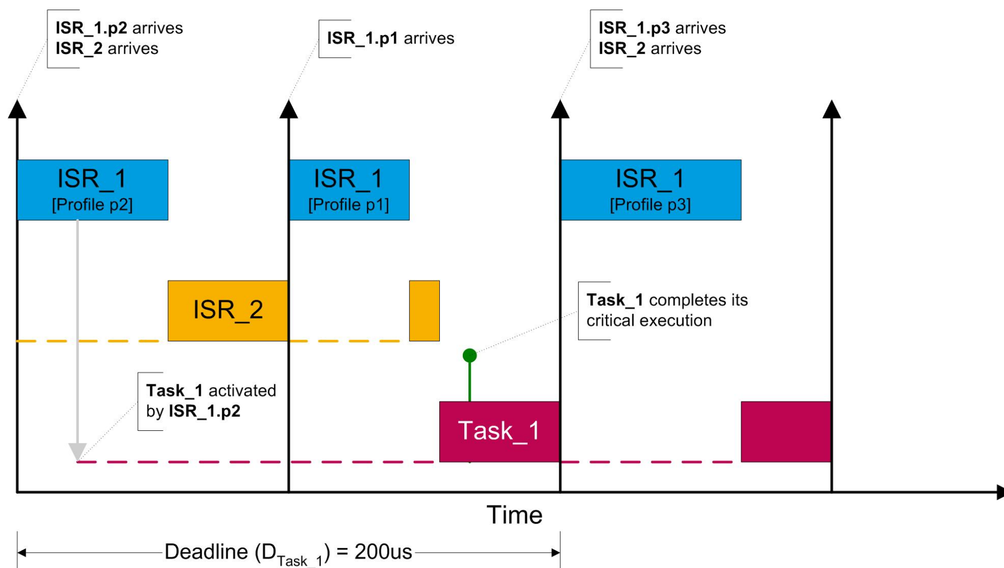


Figure 8.6: Deadlines are measured from the arrival which activates the task

```
task Task_1 {
    entry task1_entry ;
    profile {
        this priority duration 10 cycles;
        critical 2 cycles has deadline 200 us;
    }
}
```

The task's deadline of 200 microseconds may surprise you. You might have expected a deadline of 50 microseconds in order to complete before the next buffer is written, as the interrupt has already take up to 15 cycles to execute. The reason for specifying 200 microseconds is that the Analysis Visualizer measures all deadlines as from the arrival of an event, in this case the arrival of the interrupt that activates Task_1. Figure 8.6 illustrates this.

Next, you will have to declare the interrupts ISR_1 and ISR_2. You should assign ISR_1 a priority of 2 and ISR_2a priority of 1 (as larger numbers mean higher priorities).

```
interrupt ISR_2 {
    entry ist2_entry ;
    controlled;
    priority 1;
    vector 0xB;
    profile { this priority duration 10 cycles; }
}
```

The interrupt declaration of ISR_1 should contain three execution profiles; each profile must be given a unique identifier.

```
interrupt ISR_1 {
    entry ist1_entry ;
    controlled;
    priority 2;
    vector 0xA;
    profile p1 { this priority duration 2 cycles; } /* start */
    profile p2 { this priority duration 3 cycles; } /* end */
    profile p3 { this priority duration 7 cycles; } /* message */
}
```

Note that the priority and vector values are normally target dependent. Also, multiple interrupt priority levels might not be applicable to your specific platform. For a complete list of valid values you will need to refer to the *Target/-Compiler Port Guide* for your specific platform.

The timing behavior of the system is modeled in two timelines. Both timelines contain only analysis-only arrivalpoints, as Task_1 is directly activated by ISR_1 (instead of using an activator) and interrupts can only appear in the analysis clause of an arrivalpoint. The individual interrupt profiles of ISR_1 are used to model the *message sequence*.

```
timeline {
    timebase tb_sw;
    default readonly;
    sequence {
        arrivalpoint ap1 { /* START_MESSAGE */
            analysis {
                interrupt ISR_1.p1;
                delay 100 us;
            }
        }
        arrivalpoint { /* message */
            analysis {
                interrupt ISR_1.p3;
                delay 100 us;
            }
        }
        arrivalpoint { /* END_MESSAGE */
            analysis {
                interrupt ISR_1.p2;
                task Task_1;
                delay 100 us;
                next ap1;

            }
        }
```

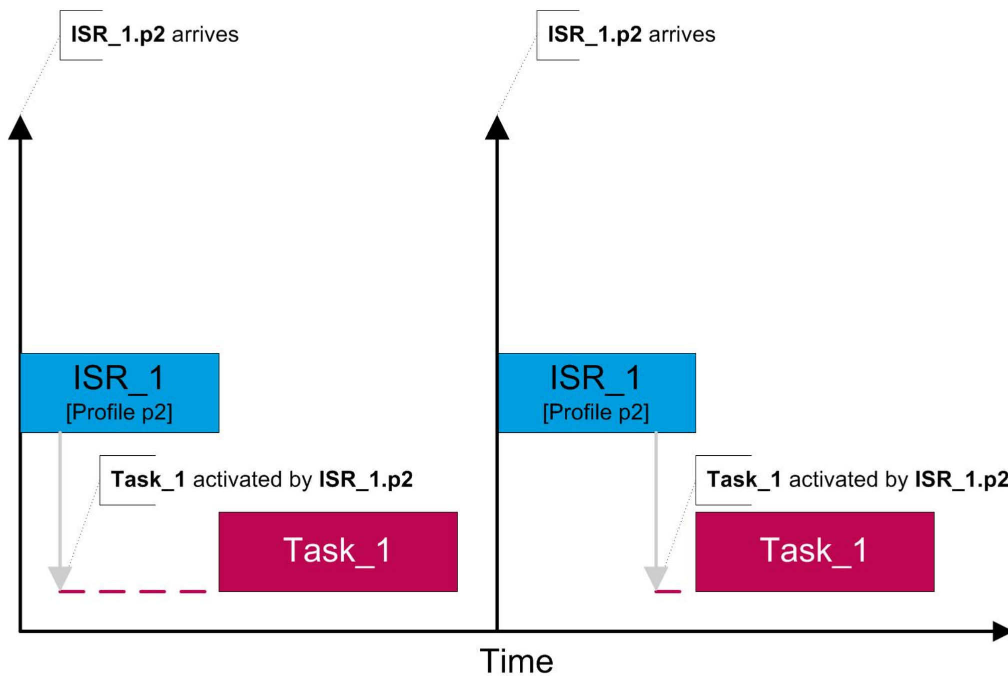


Figure 8.7: The time at which the a task is activated is not required

```
    }
}
```

Note that it is not necessary to specify the time at which ISR_1.p2 activates Task_1. As an interrupt is always of higher priority than a task, Task_1 can never start to execute until the interrupt handler has finished and in the worst case this will be for the entire duration of ISR_1.p2. Figure 8.7 illustrates this.

The second timeline you have to create describes the timing behavior of ISR_2.

```
timeline {
    timebase tb_sw;
    default readonly;
    sequence {
        arrivalpoint ap2 {
            analysis {
                interrupt ISR_2;
                delay 900 us;
                next ap2;
            }
        }
    }
}
```

You can now declare two basic transactions for both timelines, starting at ap1 and ap2 respectively. As both transactions describe interrupts, only, they do not refer to any activator.

```
transaction t1 {
    start ap1 ;
}
transaction t2 {
    start ap2 ;
}
```

As all system overheads can be ignored, you finally add the following two clauses:

```
system timings { 0; 0; 0; 0; 0; 0; 0; 0; }
interrupt recognition 0 cycles;
```

Save your configuration file as `execution-profiles.stc` and run schedulability analysis. Figure 8.8 shows the results of analysis.

Because of the interference `Task_1` experiences it will not finish in time before its next activation. The sensitivity analysis highlights the problem as shown in Figure 8.9.

Let's assume that the initial execution time estimated for `ISR_2` was too pessimistic and in reality the execution of `ISR_2` only takes 8 cycles instead of 10 cycles (as indicated by the above result). If you update the interrupt declaration and rerun the analysis then you will see that the system is schedulable.

The important information is that `Task_1` is schedulable with an overall response time of 30 cycles, just in time before its next invocation. Although the task itself is not able to finish execution before the next buffer is written its critical section is executed within the deadline specified (in fact it even finishes five cycles earlier). Note that this system would not have been schedulable if you had used the interrupt's worst case execution time instead of individual profiles.

However, if you compare the above results with the system's timing behavior illustrated in Figure 8.10, you will notice that the actual response time of interrupt `ISR_2` is only 13 cycles instead of 18 cycles reported by the Analysis Visualizer. The reason for this is that Figure 8.6 shows the worst-case response time of `Task_1`, which only occurs when `ISR_2` and `Task_1` are released simultaneously. Figure 8.10 displays the worst-case response time for `ISR_2`, which only occurs if `ISR_2` arrives at the same time as `ISR_1.p3`.

The above illustration shows that `Task_1` finishes its critical section after 10 cycles and its worst-case response time is only 20 cycles. This means that the

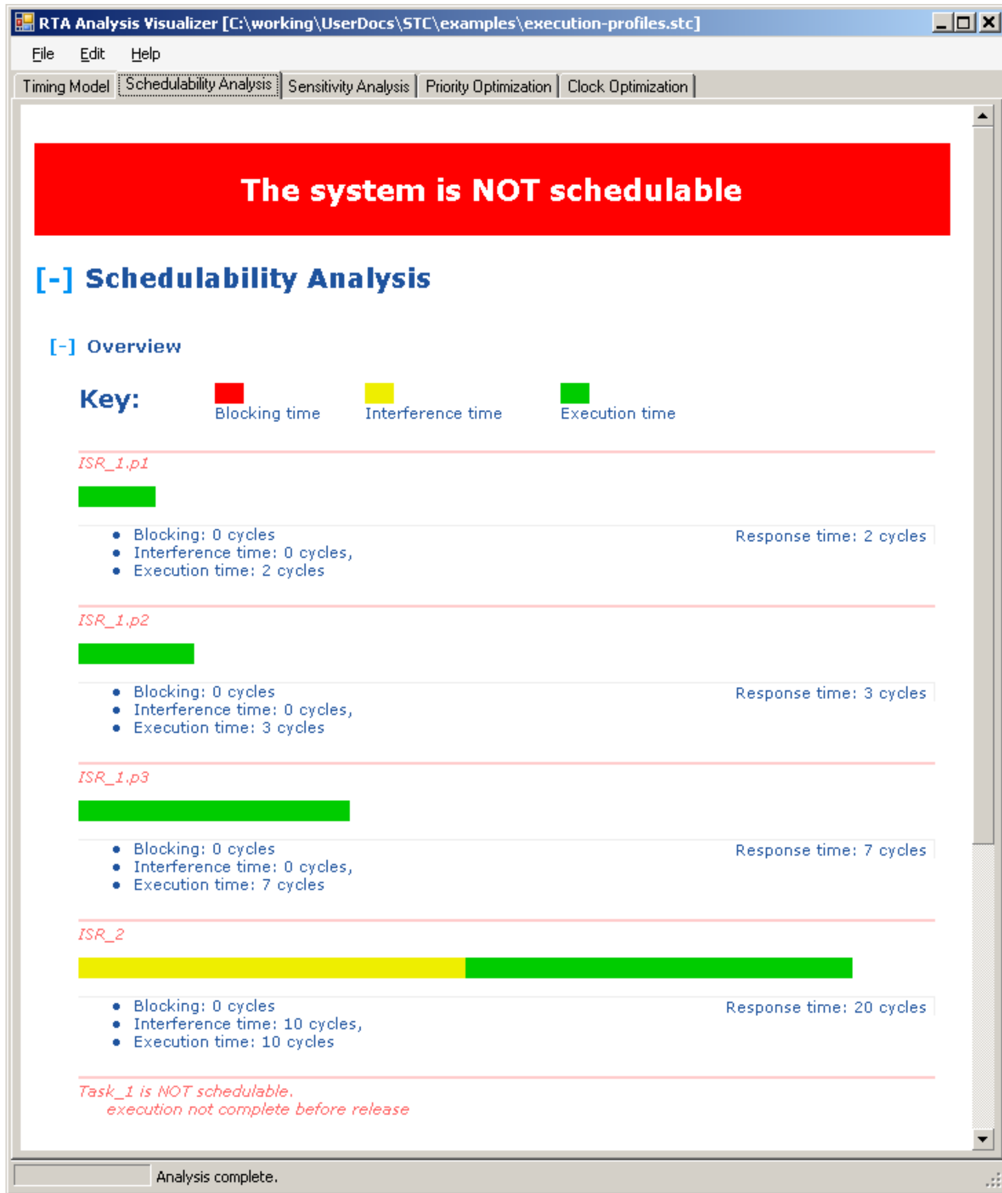


Figure 8.8: `execution-profiles.stc` schedulability analysis

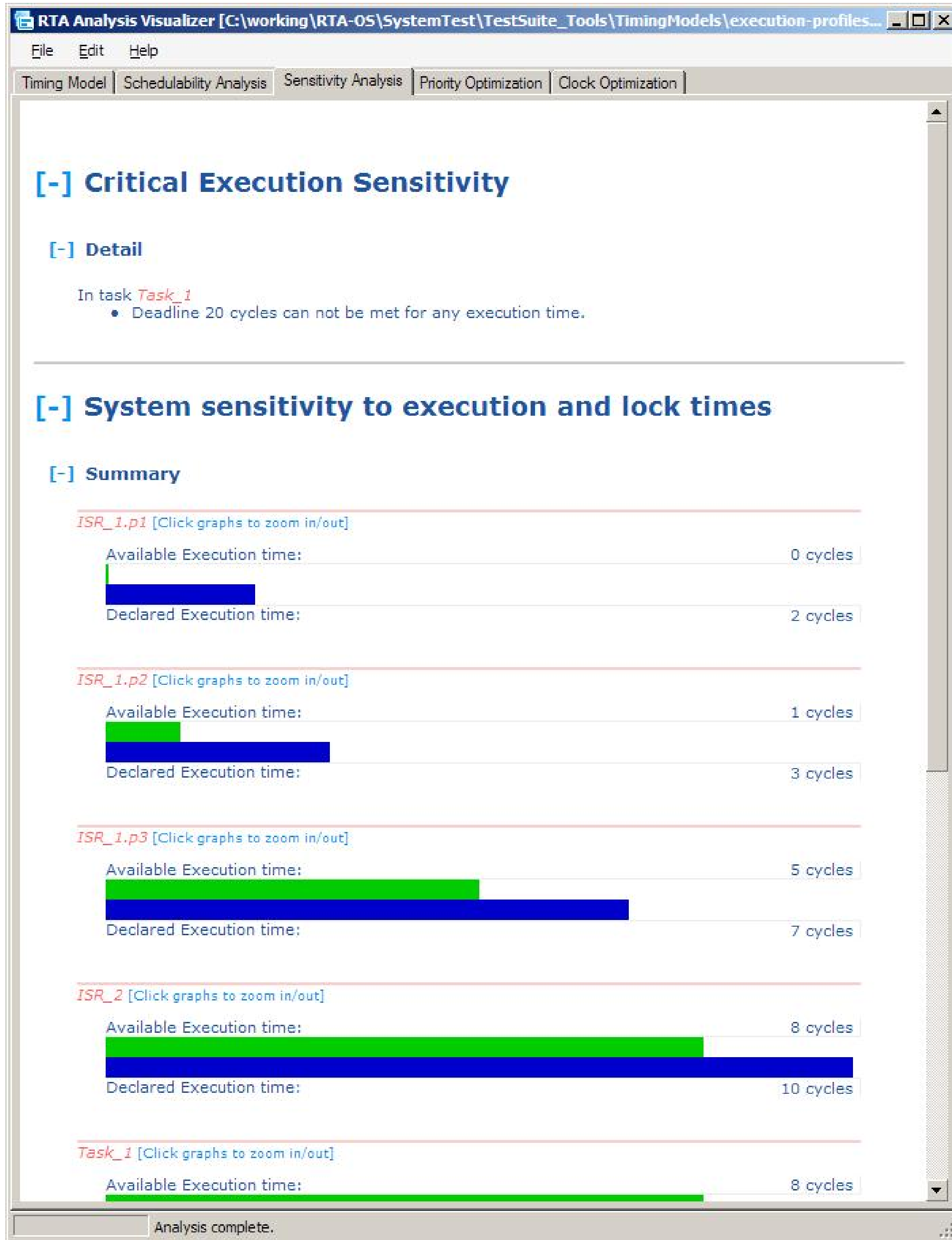


Figure 8.9: `execution-profiles.stc` sensitivity analysis

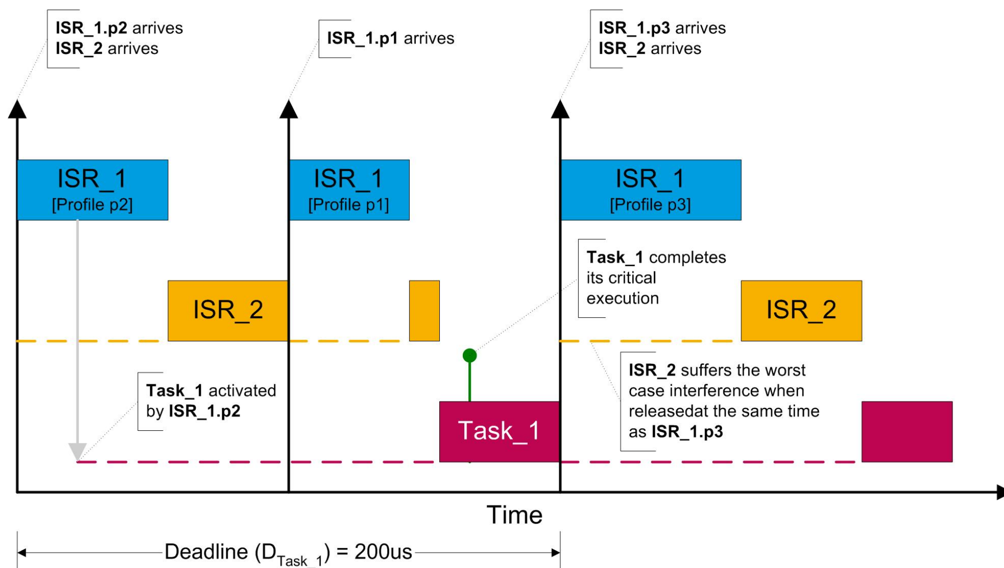


Figure 8.10: Worst case arrival for ISR_2

test system will never actually experience the worst-case behavior of Task_1 and ISR_2 at the same time. The Analysis Visualizer, however, reports both worst-case values, as the transactions for ISR_1 and ISR_2 are not synchronized and the Analysis Visualizer has to assume that either case is possible.

If you know that your system will always execute in a particular order, for example Task_1 and ISR_2 are always released simultaneously, you will have to synchronize the transactions in your configuration file by merging the two separate timeline transactions into a single timeline transaction as shown below.

```
timeline {
    timebase tb_sw;
    default readonly;
    sequence {
        arrivalpoint ap1 {
            analysis {
                interrupt ISR_1.p1;
                delay 100 us;
            }
        }
        arrivalpoint {
            analysis {
                interrupt ISR_1.p3;
                delay 100 us;
            }
        }
        arrivalpoint {
            analysis {
```

```
                interrupt ISR_1.p2;
                interrupt ISR_2;
                task Task_1;
                delay 100 us; /* time 300 us */
            }
        }
        arrivalpoint {
            analysis {
                interrupt ISR_1.p1;
                delay 100 us;
            }
        }
        arrivalpoint {
            analysis {
                interrupt ISR_1.p3;
                delay 100 us;
            }
        }
        arrivalpoint {
            analysis {
                interrupt ISR_1.p2;
                task Task_1;
                delay 100 us; /* time 600 us */
            }
        }
        arrivalpoint {
            analysis {
                interrupt ISR_1.p1;
                delay 100 us;
            }
        }
        arrivalpoint {
            analysis {
                interrupt ISR_1.p3;
                delay 100 us;

            }
        }
        arrivalpoint {
            analysis {
                interrupt ISR_1.p2;
                task Task_1;
                delay 100 us; /* time 900 us */
                next ap1;
            }
        }
    }
}
transaction t1 {
    start ap1 ;
```

```
}
```

Save your configuration file as `execution-profiles-2.stc` and run the schedulability analysis.

The Analysis Visualizer will output the following results, showing the 13 cycles response time for `ISR_2` that we'd expect, as shown in Figure 8.8.

8.3 Shared Resources and Blocking

When global data is shared amongst different tasks it is important to guarantee that this data cannot get corrupted. The recommended way to do this in RTA-OS3.x is by using resource locking: every task that wants to access shared data must first lock a particular resource. A task that wants to lock a resource will only start executing if the resource has not already been locked by another task. Of course, resource locking introduces another level of overhead, as even the most sophisticated resource locking protocol, such as the priority ceiling protocol which is used by RTA-OS3.x, cannot avoid blocking time. Blocking time occurs when a task is delayed by another task with lower priority. You must not confuse blocking time with interference, as the latter is caused by higher-priority tasks or interrupts. Figure 8.12 shows what could happen if two tasks shared the same resource (`Task_2` is released twice and has a higher priority than `Task_1`).

Note that the Analysis Visualizer also considers interrupt recognition time (the maximum time for a single instruction during which an interrupt will not be recognized) as blocking time. The Analysis Visualizer configuration language provides resource clauses in task execution profiles where you can specify for how long the task will lock a particular resource. There must be a resource clause for every resource the task locks (specified in the locks resource clause in the task declaration).

8.3.1 Example

For this example we consider a 'polling' system with two independent tasks, `Read_Data` and `Check_Data`.

- `Check_Data` should be of higher priority than `Read_Data`.
- An interrupt service task expires two fine activators; both follow separate timelines and periodically activate the tasks.
- Both tasks share a resource `Data_Guard`.
- In this example the processor clock and stopwatch clock both run at 1 MHz, so 1ms equals 1000 ticks.

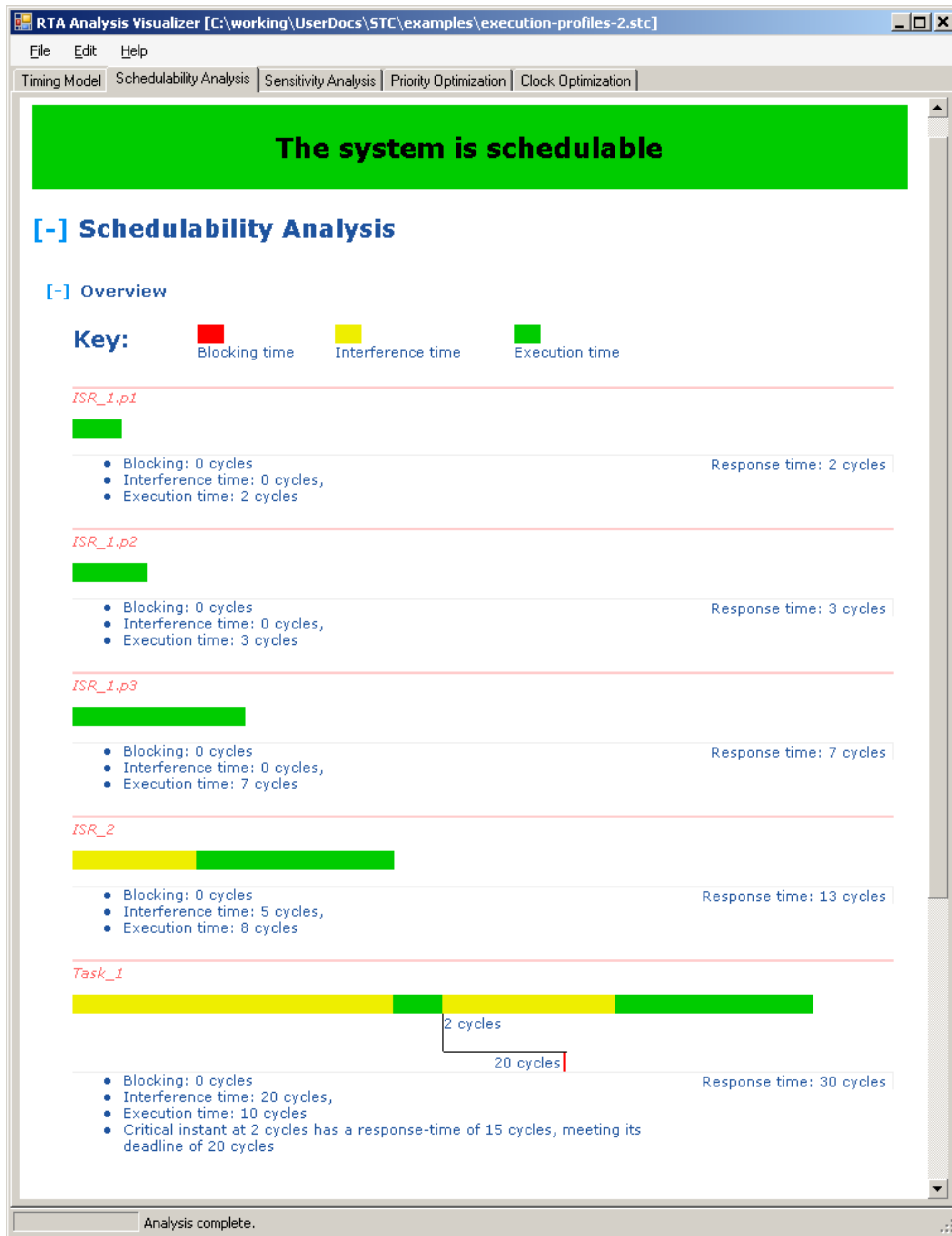


Figure 8.11: `execution-profiles-2.stc` schedulability analysis

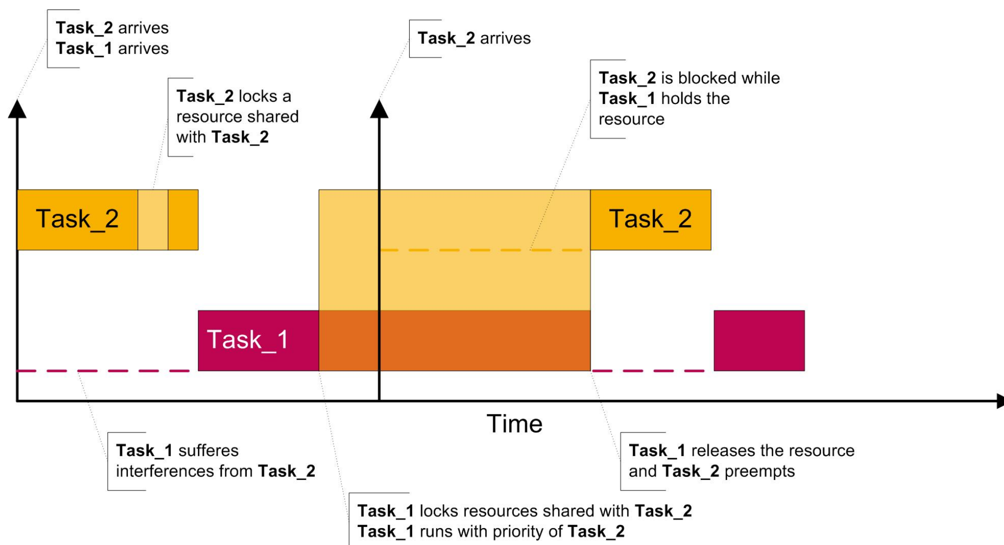


Figure 8.12: Impact of blocking on higher priority tasks

- The system timings values are: 10, 20, 30, 40, 0, 0, 50, 0 and the interrupt recognition time is 25 cycles .

The timing behavior of the system can be summarized as follows:

Task/ISR	Period [ms]	Execution Time [ms]	Data_Guai Lock [ms]	Critical [ms]	Deadline [ms]
ISR_1.p1	2	0.15	N/A	N/A	N/A
ISR_1.p2	2	0.2	N/A	N/A	N/A
Read_Data	10	4	1.8	2	5
Check_Data	5	2	1	1.4	4

In order to implement this example system you will have to declare the resource Data_Guard, both tasks and interrupt execution profiles, two timelines, two fine activators and two transactions to follow the timelines. Furthermore, you must define the timebase, system timings and interrupt recognition time.

The resource declaration is done in the resource clause.

```
resource Data_Guard;
```

As both tasks share the same resource you have to include a locks resource clause in each declaration.

```
task Read_Data {
    entry Read_Data_entry ;
    locks resource Data_Guard;
```

```
        profile {
            this priority duration 4000 cycles;
            resource Data_Guard duration 1800 cycles;
            critical 2000 cycles has deadline 5 ms;
        }
    }

    task Check_Data {
        entry Check_Data_entry ;
        locks resource Data_Guard;
        profile {
            this priority duration 2000 cycles;
            resource Data_Guard duration 1000 cycles;
            critical 1400 cycles has deadline 4 ms;
        }
    }
```

The interrupt declaration must contain two execution profiles. The Analysis Visualizer does not allow a simple ISR (completes execution before its next invocation) to appear in more than one transaction. As the two tasks require two separate transactions which are driven by the same ISR (though different profiles) you will have to declare ISR_1 to be either looping or re-triggering. In this case you also need to specify a buffer limit for each execution profile (see Section 6.3.2 for more information).

For this example system you can declare ISR_1 as re-triggering and specify a buffer limit of 1 for each profile. A buffer limit of 1 indicates that a re-triggering of the same profile actually doesn't take place but a profile can require processing before the other profile has finished its execution. Note that for looping or re-triggering interrupts buffer limits are mandatory for each profile unless you specify a fifo (first-in-first-out) ordering of profile execution and an overall fifo buffer limit.

```
    interrupt ISR_1 {
        entry ist1_entry ;
        controlled;
        priority 1;
        vector 0xA;
        retriggering;
        profile p1 {
            this priority duration 150 cycles; buffer limit 1;
        }
        profile p2 {
            this priority duration 200 cycles; buffer limit 1;
        }
    }
```

Note that for re-triggering or looping executable objects the order in which execution profiles are declared is important, as it determines the order of execution in the case where more than one invocation of the same object is ready to execute.

Next, you have to declare two separate timelines to model the periodic behavior of each individual task. Note that there is no need to declare extra timelines for the interrupts , as all transactions run on fine activators.

```
timeline {
    timebase tb_ms;
    default readonly;
    sequence {
        arrivalpoint ap1 {
            task Read_Data;
            delay 10 ms;
        }
        next ap1;
    }
}

timeline {
    timebase tb_ms;
    default readonly;
    sequence {
        arrivalpoint ap2 {
            task Check_Data;
            delay 5 ms;
        }
        next ap2;
    }
}
```

After you have declared the two fine activators act1 and act2 (one for each timeline, with initial arrivalpoints ap1 and ap2, respectively), you need to declare two transactions, indicating that two timelines are processed by fine activators driven by interrupt profiles p1 and p2 respectively.

```
transaction trans1 {
    start ap1 ;
    activator act1 driven by interrupt ISR_1.p1;
}
transaction trans2 {
    start ap2 ;
    activator act2 driven by interrupt ISR_1.p2;
}
```

Next, you need to declare the systems overheads.

```
system timings { 10; 20; 30; 40; 0; 0; 50; 0; }
interrupt recognition 25 cycles;
```

After you have specified the priority order clause for your tasks your configuration file should now have the following structure:

- Kernel clause
- One timebase
- One resource (Data_Guard)
- Two tasks (Read_Data, Check_Data)
- One interrupt (ISR_1)
- Two timelines
- Two activators (act1, act2)
- Two transactions (trans1, trans2)
- System timings
- Interrupt recognition
- Task priority order

When you have finished (do not forget the timebase) you should save the configuration file as blocking.stc and run schedulability analysis. Figure 8.13 shows the results you should get from the Analysis Visualizer.

As you can see, all tasks and interrupts are schedulable. The maximum blocking time Check_Data experiences is 1.8ms, caused by task Read_Data which, on the other hand, is only blocked for 25 cycles, due to the interrupt recognition time. Interrupt profile p1 is blocked for 240 cycles by the lower priority profile p2 (p1 is declared first and hence of higher priority than p2). Interrupt profile p2 is blocked for 50 cycles due to OS level blocking defined in the system timings values. A look at the results of the sensitivity analysis reveals the maximum time each task is permitted to lock the resource Data_Guard without jeopardizing the schedulability of the system.

Figure 8.14 shows the results you should get from the Analysis Visualizer.

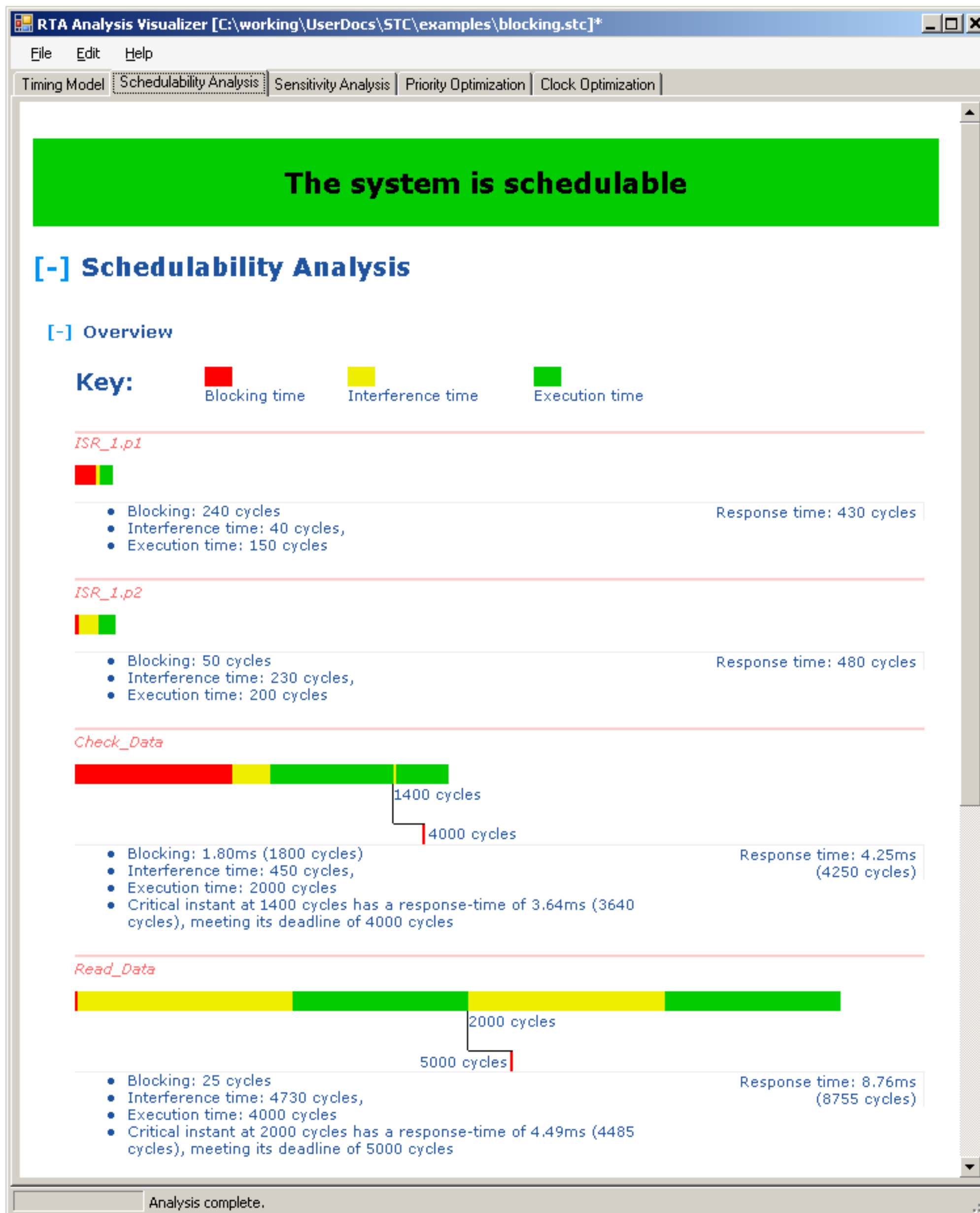


Figure 8.13: `blocking.stc` schedulability analysis

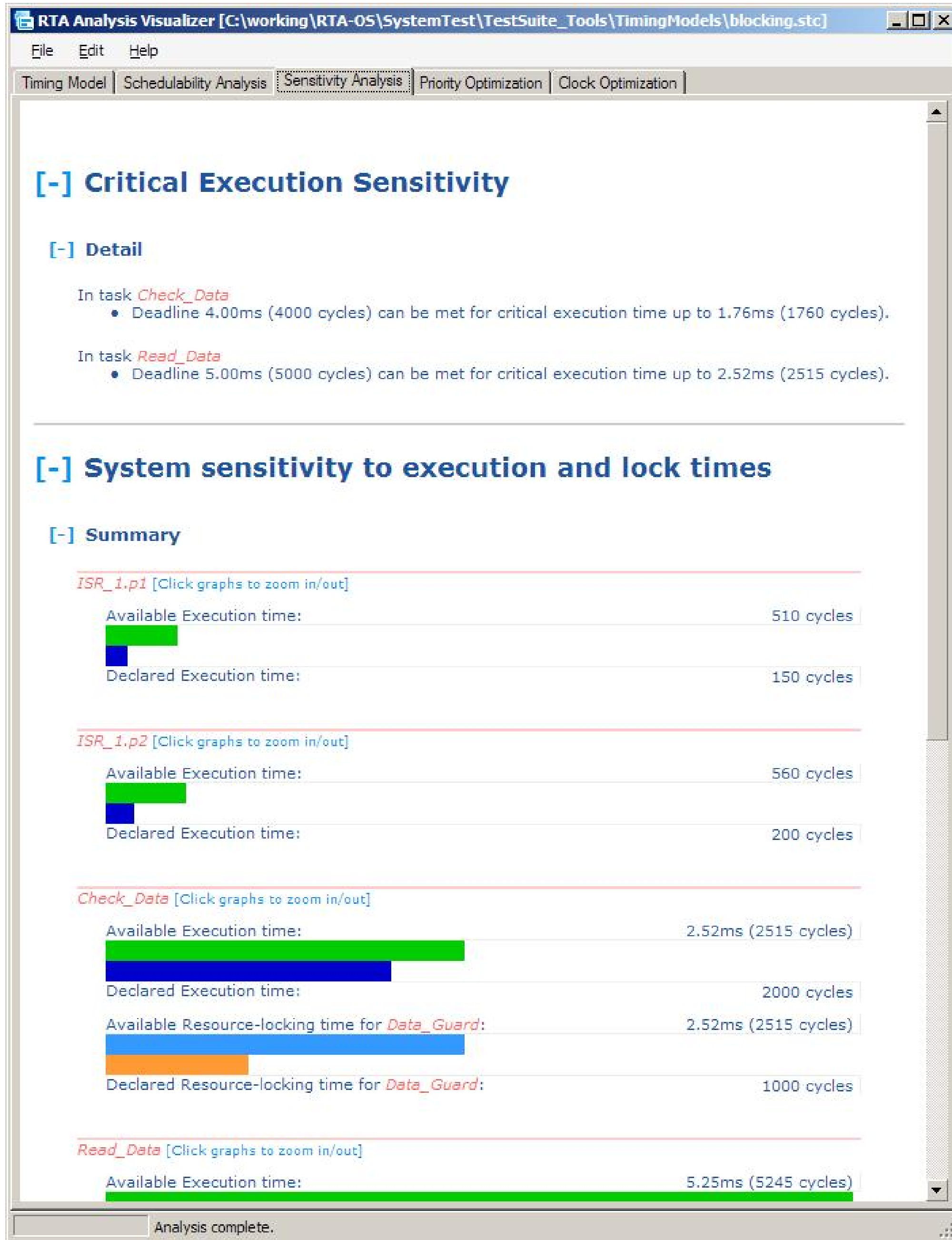


Figure 8.14: `blocking.stc` sensitivity analysis

8.3.2 Exercise

Try to enhance this example system by adding another ‘polling’ task Read_Data2 to your test system but without changing any of the existing timing properties. You will need to declare an additional task, interrupt profile, timeline, activator and transaction and add the new task to the task priority order clause.

8.4 Periodic timelines and bursting transactions

So far we have only considered sequential timelines and timeline transactions. Very often, however, it is easier to model a system by using periodic timelines and bursting transactions instead. Consider for example a system with two periodic tasks running every 3ms and 7ms respectively. If we wanted to model this simple system in a sequential timeline, we would have to specify a sequence of nine arrivalpoints:

```
sequence {
    arrivalpoint ap1 { /* 0 ms */
        task Task_1;
        task Task_2;
        delay 3 ms;
    }
    arrivalpoint ap2 { /* 3 ms */
        task Task_1;
        delay 3 ms;
    }
    arrivalpoint ap3 { /* 6 ms */
        task Task_1;
        delay 1 ms;
    }
    arrivalpoint ap4 { /* 7 ms */
        task Task_2;
        delay 2 ms;
    }
    arrivalpoint ap5 { /* 9 ms */
        task Task_1;
        delay 3 ms;
    }
    arrivalpoint ap6 { /* 12 ms */
        task Task_1;
        delay 2 ms;
    }
    arrivalpoint ap7 { /* 14 ms */
        task Task_2;
        delay 1 ms;
    }
    arrivalpoint ap8 { /* 15 ms */
        task Task_1;
        delay 3 ms;
```

```
    }
    arrivalpoint ap9 { /* 18 ms */
        task Task_1
        delay 3 ms;
    }
}
next ap1;
```

By using the periodic shorthand form of the timeline declaration (periodic timelines) we can represent this timing behavior in a much easier and far more intuitive way.

```
timeline {
    timebase ms_tb;
    default readonly;
    periodic ap1 {
        task Task_1 every 3 ms offset 0 ms;
        task Task_2 every 7 ms offset 0 ms;
    }
}
```

During processing the Analysis Visualizer will automatically transform periodic timelines into an equivalent sequential timeline declaration. Periodic timelines, however, do not allow analysis-only objects, e.g. interrupts cannot be modeled in periodic timelines. This means that if you want to model interrupts you will still need to use sequential timelines.

In order to model analysis-only objects, such as interrupts, it is often easier to use bursting transactions instead of sequential timelines. Bursting transactions do not require a timeline declaration as they only describe the arrival pattern of a particular task or interrupt. How ever, like periodic timelines the Analysis Visualizer will internally transform bursting transactions into sequential analysis-only timelines. Consider for example a system where an interrupt ISR_1 always activates a task Task_1. Let's assume the interrupt will not occur more frequently than once in every 1ms, four times in every 9ms and ten times in every 50ms. The declaration of the corresponding bursting transaction would then look as follows:

```
transaction t1 {
    bursting {
        1 times in 1ms ;
        4 times in 9ms ;
        10 times in 50ms ;
    }
    interrupt ISR_1 ;
    task Task_1 ;
}
```

This means, the Analysis Visualizer will create an internal analysis-only timeline with nine arrivalpoints to implement the following worst-case arrival pattern:

0ms; 1ms; 2ms; 3ms; 9ms; 10ms; 11ms; 12ms; 18ms; 19ms; 50ms; 51ms; 52ms; 53ms; . . .

Note that bursting transactions are used for analysis purposes only. As they don't allow the specification of arrivalpoints it is not possible to 'link' them to any existing timelines. Bursting transactions should therefore only be used to model bursting timing behavior of interrupts and tasks if they are not synchronized with any existing timelines.

8.4.1 Example 1

In our working example we consider a system with four tasks, three tasks are periodically activated by a fine activator expired from an interrupt service task ISR_1; the fourth task is activated by a second interrupt service task ISR_2 with bursting behavior.

- ISR_1 executes at priority 1, ISR_2 at priority 2.
- ISR_1 and ISR_2 both execute for 0.1ms.
- Task_1 is activated every 3ms, Task_2 every 8ms and Task_3 every 12ms.
- Task_1 has a worst-case execution time of 0.2ms, Task_2 of 0.3ms and Task_3 of 0.4ms.
- ISR_2 activates Task_4 and has a bursting arrival of 1 times in 1ms, 3 times in 8ms and 5 times in 20ms.
- Task_4 has a worst-case execution time of 0.5ms and must have the highest priority.
- The timebase should define 1ms to equal 1000 ticks.
- The system timings values are: 10, 20, 30, 40, 0, 0, 50, 0 and the interrupt recognition time is 25 cycles.

In order to model your system you will need to declare your tasks and interrupts, a periodic timeline, a fine activator, one timeline transaction which runs on that activator and one bursting transaction for the second interrupt.

In order to guarantee that Task_4 is assigned the highest priority during automatic priority allocation, you need to specify the following priority constraints clause immediately after the task declarations:

```
priority constraints {
    task Task_4 higher than task Task_1;
    task Task_4 higher than task Task_2;
    task Task_4 higher than task Task_3;
}
```

The periodic timeline you declare should have the following form:

```
timeline {
    timebase tb_ms;
    default readonly;
    periodic ap1 {
        task Task_1 every 3 ms offset 0 ms ;
        task Task_2 every 8 ms offset 0 ms ;
        task Task_3 every 12 ms offset 0 ms ;
    }
}
```

The bursting transaction should be declared as follows:

```
transaction t2 {
    bursting {
        1 times in 1 ms;
        3 times in 8 ms;
        5 times in 20 ms;
    }
    interrupt ISR_2;
    task Task_4;
}
```

All other objects are declared similar to previous examples. When you have finished (don't forget to declare the timebase and systems overheads) you should save the configuration file as `transactions.stc` and run priority optimization. Figure 8.15 shows the results you should expect to see.

As you can see, all tasks and interrupts are schedulable, and the automatic priority allocation assigned three tasks to a non-preemption group.

8.4.2 Example 2

You should now use the above information to carry out sensitivity analysis. Remember that sensitivity analysis requires a task priority clause. You also have to specify a soft non-preemption group for the suggested tasks. An example solution can be found in `transactions-2.stc` of your Analysis Visualizer tutorial folder.

Figure 8.16 shows the results you should expect to see.

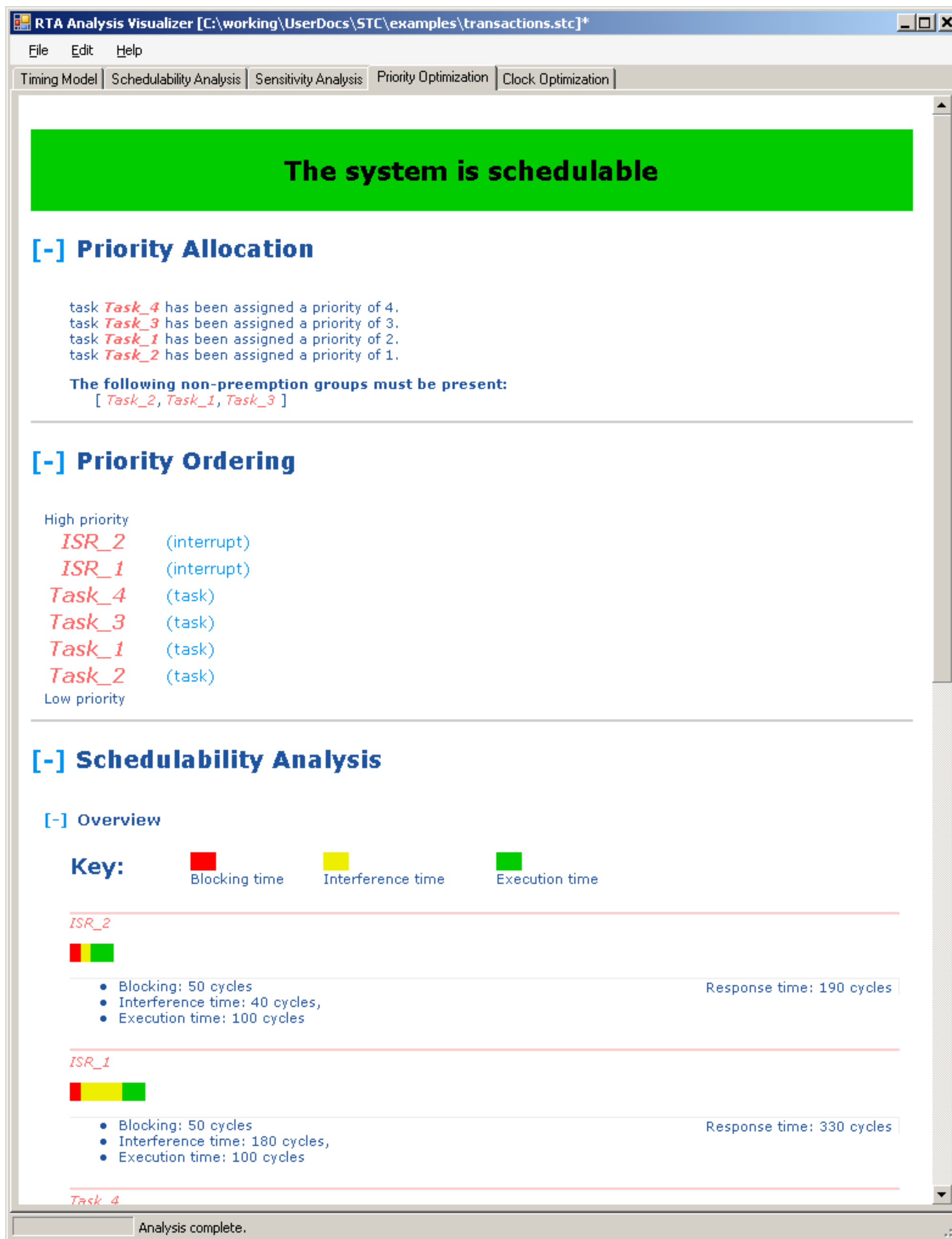


Figure 8.15: `transactions.stc` priority optimization

Figure 8.16: `transactions-2.stc` sensitivity analysis

Try now to change your configuration file (e.g. remove the soft non-preemption group clause, increase computation times, etc) and rerun sensitivity analysis. By comparing the results you will get a better understanding of how the timing behavior of your system is changing. Note that you might also have to rerun automatic priority allocation as the priority order depends on the changes you make.

8.5 Looping and re-triggering behavior

In hard real-time systems it is essential that deadlines will always be met. However, it is not always possible that tasks and interrupts finish execution before their next invocation is due. Re-triggering or looping behavior of tasks and interrupts, on the other hand, will not change the hard real-time property of a system, as long its deadlines are met. There are of course limitations as to how many instances of a task or interrupt can be buffered. Most limitations are based on hardware restrictions and have to be obtained from appropriate hardware manuals.

Where declared as re-triggering, tasks will need to chain themselves on completion while interrupts will need to either leave the interrupt pending or re-assert it. Interrupts are also allowed to be declared as looping, which means their handlers will need to loop within their entry function until all pending occurrences of the interrupt have been dealt with. Re-triggering or looping tasks and interrupts can be declared with `fifo` behavior, which means their profiles execute on a first-in, first-out (FIFO) basis as opposed to priority - based execution order. When declaring the task you will also have to specify a buffer limit, either as part of the fifo clause or explicitly in each execution profile. If you are not sure what buffer size to specify you can always declare an unlimited buffer, the Analysis Visualizer will then report the buffer size needed. But be careful when choosing unlimited buffer sizes as the Analysis Visualizer might use an unrealistic limit to schedule your system.

Re-triggering or looping execution is usually required in systems with bursting behavior. Take for example an application where an interrupt activates a task. The interrupt handler executes for 0.1ms, the task for 2ms. Let's assume a single interrupt is normally raised every 13ms but there is a bursting situation where up to six interrupts (and task activation) can occur with a delay of only one millisecond in between. We also assume that all six task's invocations must have finished their execution within 13ms after the first task invocation occurred. We, therefore, specify a deadline for the task which is the remaining time from the moment the sixth interrupt is raised; hence we declare a deadline of 8ms. The task itself executes in FIFO order and re-triggers (chains itself) until all occurrences have been processed.

Figure 8.17 demonstrates the arrival pattern and execution order of the task.

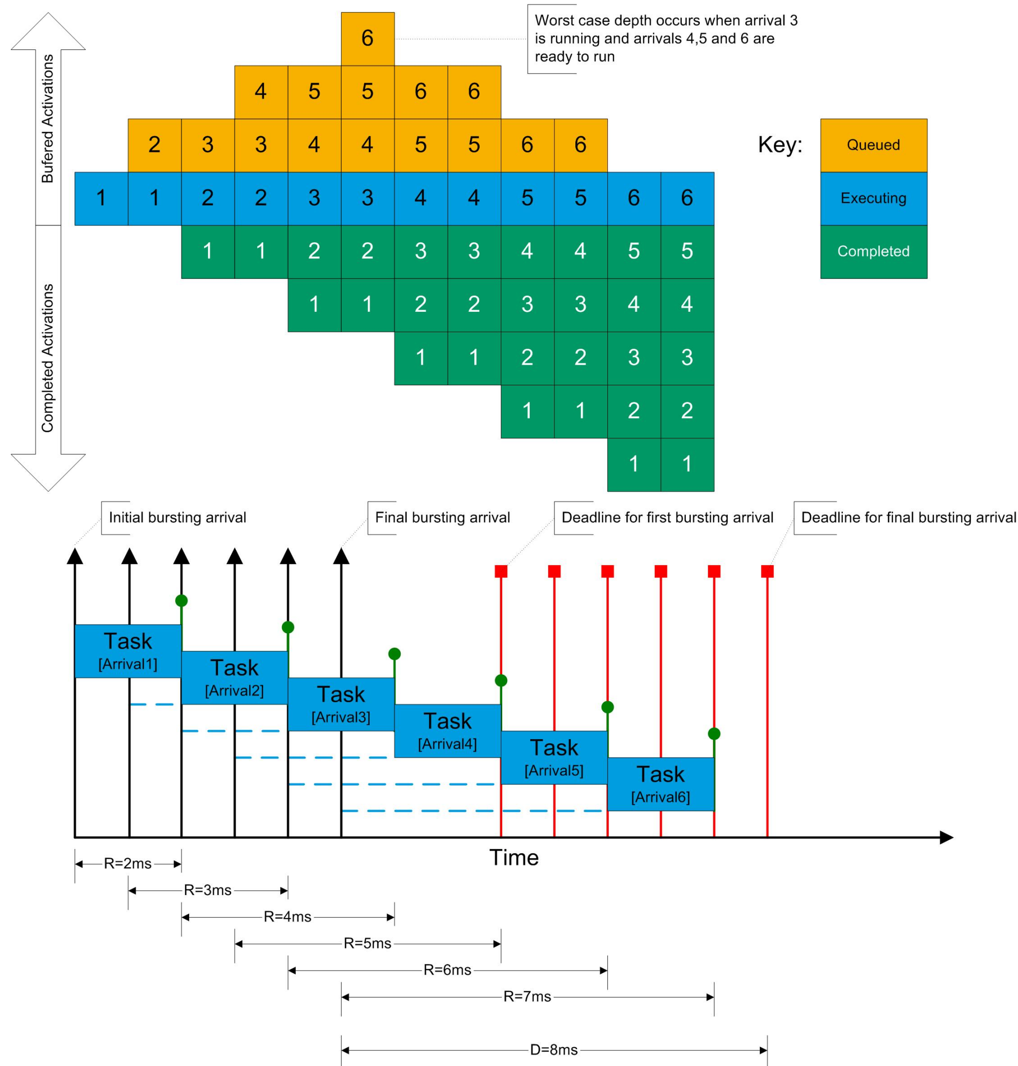


Figure 8.17: Effect of FIFO buffering on response times

You can see, for example, that by the time the sixth arrival of the task occurs the first two invocations have already finished executing. The sixth invocation also represents the worst-case response time the task will experience. It also shows that only four fifo buffer levels are required to buffer all arrivals.

8.5.1 Example 1

We can model this arrival pattern as a bursting transaction with a bursting arrival pattern of 1 times in 1ms and 6 times in 13ms. If we specify a deadline of 8ms and declare our task as `re-triggering` with `fifo buffer unlimited`, the Analysis Visualizer will output the results shown in Figure 8.18 (1ms equals 1000 ticks):.

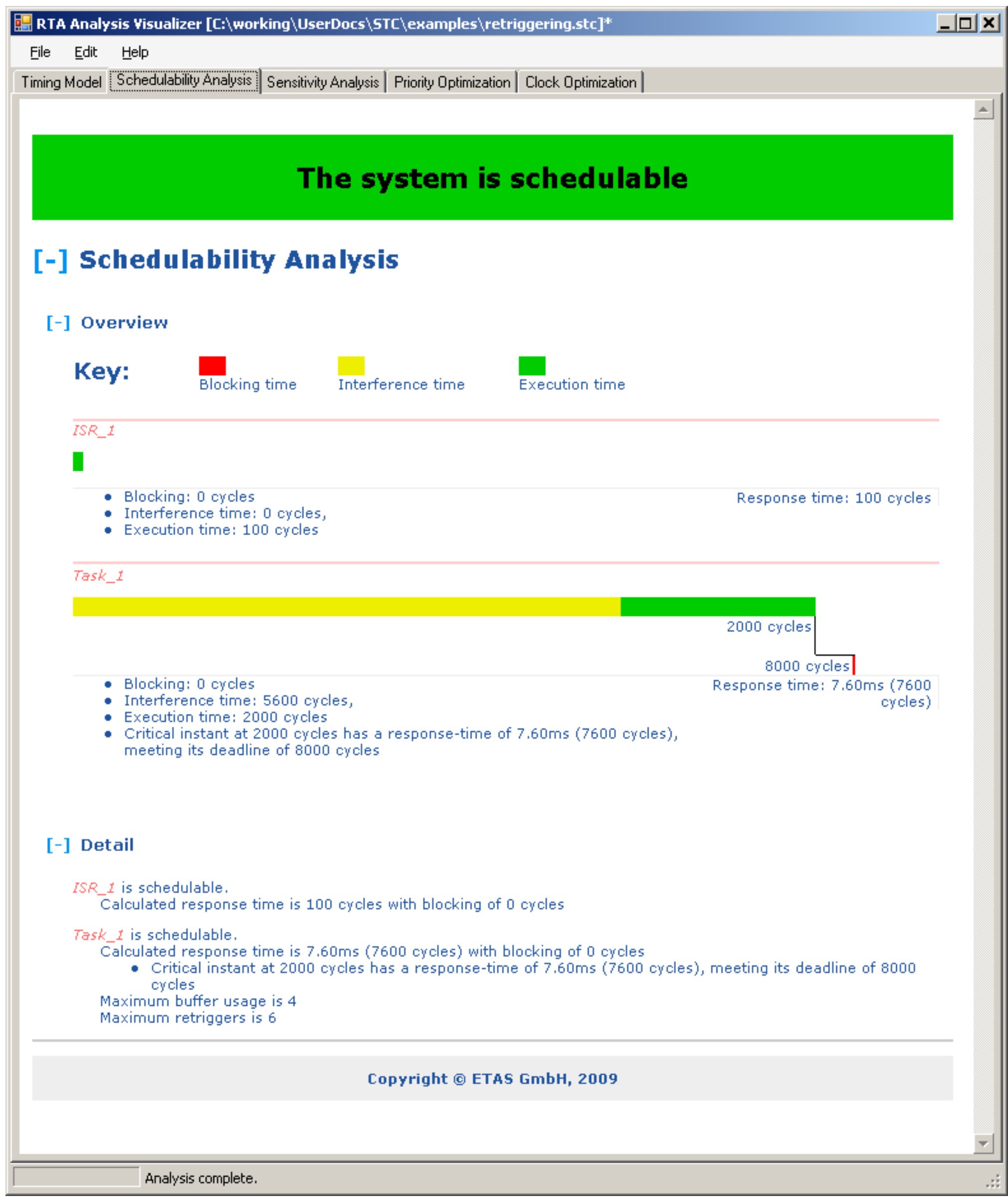


Figure 8.18: `retriggering.stc` schedulability analysis

Most importantly, our task always meets its deadline with an overall response time of 7.6ms. Furthermore, the Analysis Visualizer confirms that the system only requires a buffer limit of 4, as there are never more than three events pending at the same time (level 1 represents the execution level). You can find the implementation of the above system in `retriggering.stc` of your Analysis Visualizer examples directory.

8.5.2 Example 2

For our second example we consider a system with an ISR which activates one task.

- The interrupt service task `ISR_1` has two execution profiles, each representing a different interrupt source. The first profile executes for 0.8ms and activates `Task_1`; it has a bursting arrival pattern of 1 times in 1ms, 3 times in 5ms and 8 times in 100ms. The second profile executes for 0.5ms and runs periodically every 4ms.
- `Task_1` has a worst-case execution time of 8ms; its critical execution path takes 4ms and must be completed within 100ms after the interrupt has occurred.
- The interrupt `ISR_1` should be declared as looping, executing its two profiles in priority order (each execution profile must contain a buffer limit). The task `Task_1` should be declared as `re-triggering` with `fifo` execution behavior. All buffer limits are unknown at this stage.
- The timebase should define 1ms to equal 1000 ticks.
- The system timings values are: 10, 20, 30, 40, 0, 0, 50, 0 and the interrupt recognition time is 25 cycles.

Try to model the timing behavior of both interrupt profiles by using two bursting transactions (a periodic behavior of every 4ms can be described as 1 times in 4ms). Save your configuration file as `retriggering-2.stc` and run the analysis. Figure 8.19 shows the results you should expect to see.

If your output does not match the above result you should review your interrupt declaration, as the interrupt execution profiles might have been declared in the wrong order.

As you can see, `Task_1` is schedulable with a required fifo buffer size of 8. The interrupt `ISR_1` requires a buffer size of 2 for its profile p1, and a buffer size of 1 for profile p2.

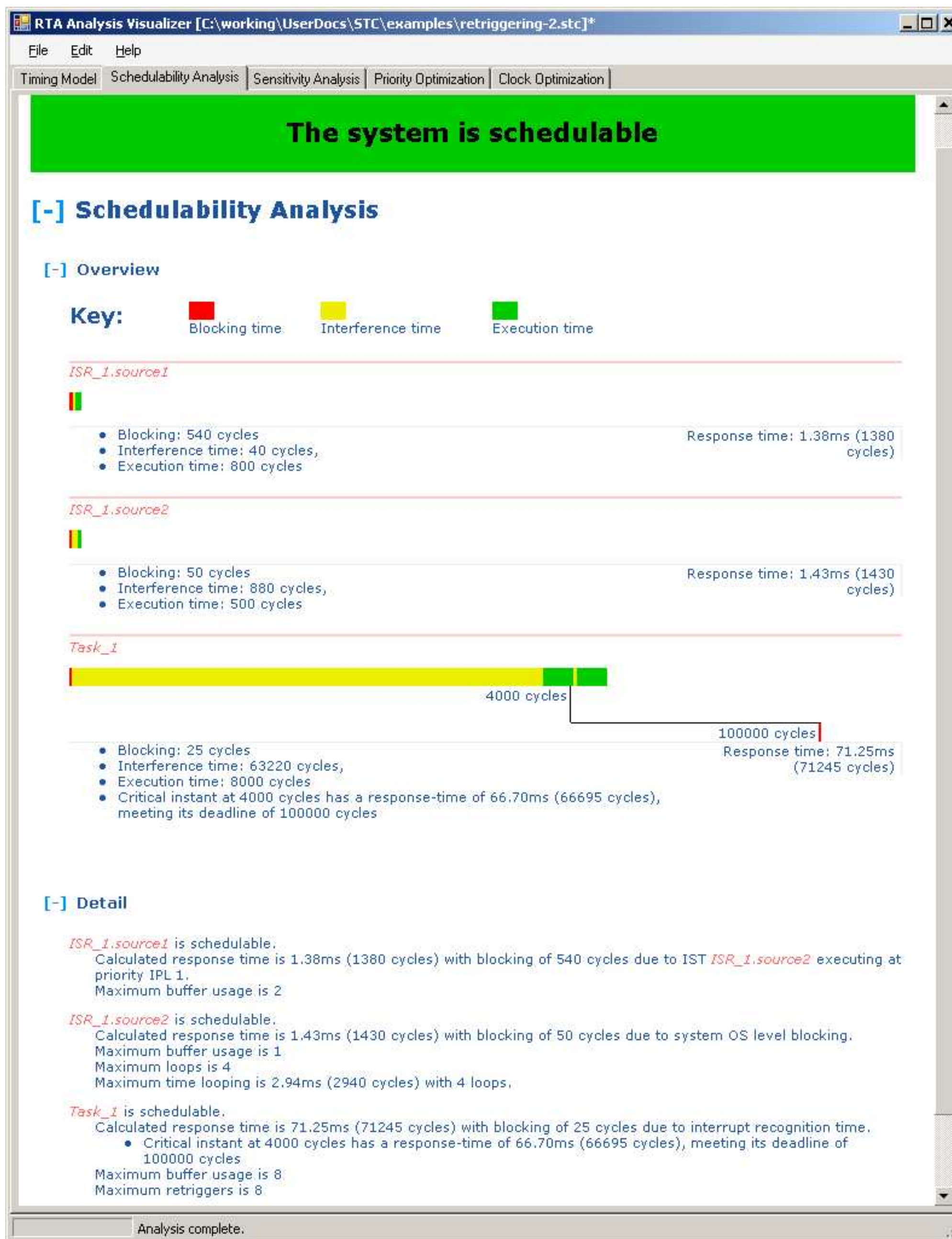


Figure 8.19: `retriggering-2.stc` schedulability analysis

8.5.3 Exercise

Let's assume the only available fifo buffer for your platform has a size of 6. This means your system would not be schedulable anymore. You should now try to alter your system (e.g. change bursting behavior, execution times, etc) to make it schedulable again. Use sensitivity analysis to obtain the required information.

8.6 Allocating priorities

The Analysis Visualizer does not require the user to assign priorities to tasks as it provides automatic priority allocation (command line option -p). Choosing this option will force the Analysis Visualizer to search for a valid priority assignment (one which allows the system to be schedulable). In some systems, however, tasks must execute in a particular order and can not be considered independent from each other. If for example two tasks are released simultaneously, but Task_2 always depends on the results of Task_1, it is necessary to take this task design into consideration and assign a higher priority to Task_1.

The Analysis Visualizer provides two different ways to describe inter-task dependencies. The first one is only suitable for systems where a specific priority ordering of tasks is desired. In this case you can specify the complete order of your tasks in the task priority order clause and skip the automatic priority allocation step. Note that if a task priority clause is given, all user-defined tasks have to be specified. If you don't use automatic priority allocation and your system contains more than one task you must specify a task priority order clause in your configuration file.

Alternatively, you can specify several task priority constraints, which will then be taken into consideration by the Analysis Visualizer during the priority allocation process. Here, you will only have to specify any known dependencies; in the above example you would have to specify that Task_1 is higher than Task_2. Specifying priority constraints, combined with automatic priority allocation, is the recommended way to prioritize your system as it can minimize memory (stack) usage and requires less maintenance. However, the results of automatic priority allocation can be explicitly specified in a task priority clause. Thus, the priority order of a system remains under the user's control and will only change whenever a new automatic priority allocation is desired.

Automatic priority allocation requires a schedulable system. If your system is unschedulable the Analysis Visualizer will not be able to determine an optimal priority order. Instead, it will output a default priority order clause which can be used to carry out sensitivity analysis. Alternatively, you can specify a reasonable task priority order yourself. A good approach is to assign task priorities according to deadline monotonic priority ordering, i.e. the task with

the shortest deadline is assigned the highest priority, the task with the second shortest deadline is assigned the next highest priority, etc. Once you have completed the task priority order clause in your configuration file you can then run the sensitivity analysis which will tell you what changes are necessary to make your system schedulable. After you applied those changes you can restart automatic priority allocation to optimize your system.

If you design a task that activates other tasks you have to declare this in your task declaration. Every task directly activated by your task must be referenced in an activates task clause. The schedulability theory implemented in the Analysis Visualizer allows a task to only activate other tasks with a lower priority. As an `activates` clause implies priority constraints, it is important that neither the priority constraints nor the priority order clauses contradict this declaration. The Analysis Visualizer will report an error, if for example Task A has been declared to activate Task B, but Task B has also been declared to be of higher priority than Task A.

Interrupts must be assigned an interrupt priority level (IPL). Where the user is free to choose the IPL, these are usually best assigned in deadline monotonic order. However, in many cases the IPL will be fixed by the microcontroller's interrupt controller. Furthermore, the hardware may place more that one interrupt at the same IPL. The Analysis Visualizer needs to know which order the hardware will process interrupts and this is specified by an `arbitration order` declaration. This allows you to specify the execution order of the interrupts when they are pending at the same time. This clause is mandatory if you declare more than one interrupt with the same IPL. Note that the arbitration order is usually defined in the processor reference manual for your target.

8.6.1 Example

In our working example we want to implement a system with four tasks and investigate how different priority allocations can affect its schedulability.

- A task `Read_Data`, which reads available data from a data buffer, is activated twice every 30ms by a fine activator which is expired by an interrupt service task `ISR_1`. On its first invocation it activates another task `Copy_Data` which copies the data to various display channels. Ten milliseconds after its first activation it will get activated again, this time simultaneously with task `Display_Data` which displays the available data. On its second invocation `Read_Data` doesn't activate any tasks.
- The execution profile of `Read_Data` that activates the other task takes 3ms to execute with a deadline of 4ms. The second execution profile requires 7ms to execute and must run to completion in 8ms.

- `Copy_Data` has a worst-case computation time of 5ms and must complete within 9ms from occurrence of the interrupt. Note, deadlines are always measured from the occurrence of an event (see Section 3.4).
- `Display_Data` executes for 6ms and has to complete in 16ms.
- An independent task `Check_Data` runs every 10ms, activated by an interrupt service task `ISR_2`. `Check_Data` has a worst-case computation time of 1ms and must complete before it is next activated.
- `ISR_1` and `ISR_2` are running at the same priority level 1, but `ISR_1` will be serviced first should both interrupts be pending simultaneously. Both interrupts have a worst-case execution time of 0.2ms.
- All tasks should be assigned priorities according to deadline monotonic priority ordering.
- The timebase should define 1ms to equal 1000 ticks; all overheads can be ignored.

In the configuration file, in addition to the standard clauses, you will have to declare the tasks, the interrupts , two timelines to model the timing behavior, one activator and two transactions which use these timelines. You also need to declare an arbitration order for your interrupts and a task priority order clause. This is the structure the configuration file should eventually have:

1. Kernel clause
2. One timebase
3. Four tasks (`Read_Data`, `Copy_Data`, `Display_Data`, `Check_Data`)
4. Two interrupts (`ISR_1`, `ISR_2`)
5. Two timelines
6. One activator (act1)
7. Two transactions (t1, t2)
8. Arbitration order
9. System timings
10. Interrupt recognition
11. Task priority order

When you declare task Read_Data you will have to add two execution profiles to model the task execution behavior. The first profile should represent the case where Copy_Data is activated,

The second profile represents the case where no tasks are activated. Don't forget to include an activates task clause as Read_Data activates another task. Note that Copy_Data must have been declared prior to Read_Data as forward references are not permitted and the Analysis Visualizer would otherwise report an error.

```
task Read_Data {
    entry Read_Data_entry ;
    activates task Copy_Data;
    profile READ_AND_COPY {
        this priority duration 3 ms;
        critical 3 ms has deadline 4 ms;
    }
    profile READ_WITHOUT_COPY {
        this priority duration 7 ms;
        critical 7 ms has deadline 8 ms;
    }
}
```

After you have completed the interrupt declarations you can now add two timelines to model the timing behavior. In your first timeline you will need to describe the periodic timing behavior of Read_Data, Copy_Data and Display_Data. Specify a delay of 10ms between the first and second activation of Read_Data, and a final delay of 20ms to create a 30ms period. Note that there is no need to declare a third timeline for the interrupt ISR_1. Where fine activators are used, the Analysis Visualizer will automatically add the interrupt execution profile(s) to every arrivalpoint of the related timelines.

```
timeline {
    timebase tb_sw;
    default readonly;
    sequence {
        arrivalpoint ap1 {
            task Read_Data.READ_AND_COPY;
            delay 10 ms;
            analysis { task Copy_Data; }
        }
        arrivalpoint {
            task Read_Data.READ_WITHOUT_COPY;
            task Display_Data;
            delay 20 ms;
        }
        next ap1;
    }
}
```

The activation of Copy_Data is modeled by adding an analysis clause to the arrivalpoint where Read_Data.READ_AND_COPY is activated. This is necessary because the activator does not activate Copy_Data directly. The Analysis Visualizer, however, requires this information to produce accurate analysis results.

An alternative design would be to remove the RTA-OS3.x activation call of Copy_Data from Read_Data and instead have the task activated directly by the activator (provided Read_Data has been declared to be of higher priority than Copy_Data). In this case you would save one RTA-OS3.x API call and you would not need the analysis clause either. This design, of course, is only applicable to systems where the activation of a task by another task is predictable, i.e. it always occurs at the same time.

The second timeline describes the runtime behavior of ISR_2 and Check_Data. It contains one arrivalpoint only.

```
timeline {
    timebase tb_sw;
    default readonly;
    sequence {
        arrivalpoint ap2 {
            analysis {
                interrupt ISR_2;
                task Check_Data;
                delay 10 ms;
                next ap2;
            }
        }
    }
}
```

Next, you will have to declare a fine activator.

```
activator act1 {
    timebase tb_ms;
    fine;
    driver callbacks {
        now act1_now;
        cancel act1_cancel;
        state act1_state;
        set act1_set;
    }
    initial ap1;
}
```

You can now declare two transactions, representing the timing behavior of the system. The first transaction runs on a fine activator driven by ISR_1; the other one runs on no activator, as it only describes the behavior of ISR_2.

```
transaction t1 {
    start ap1 ;
    activator act1 driven by interrupt ISR_1;
}
transaction t2 {
    start ap2 ;
}
```

After you added your fine activator declaration you need to declare the arbitration order of your interrupts. For this example, ISR_1 should always run first:

```
arbitration order {
    interrupt priority 1 {
        interrupt ISR_1;
        interrupt ISR_2;
    }
}
```

As before, system timings and interrupt recognition time are set to zero, thus ignoring operating system overheads.

```
system timings { 0; 0; 0; 0; 0; 0; 0; 0; }
interrupt recognition 0 cycles;
```

At the end of your configuration file you must declare the task priority order, unless you specify priority constraints and use automatic priority allocation. In this example, tasks should be assigned priorities according to their deadlines: the task with the shortest deadline is assigned the highest priority, the task with the second shortest deadline is assigned the next highest priority, etc. Note that where no deadlines are given explicitly, periods can be used, as tasks must normally complete before they are reactivated.

```
task priority order {
    task Read_Data;
    task Copy_Data;
    task Check_Data;
    task Display_Data;
}
```

Save your configuration file as `priority-allocation.stc` and run schedulability analysis. Figure 8.20 shows the detailed report that the Analysis Visualizer will output (the graphical view has been hidden).

All tasks and interrupts will always meet their deadlines and the complete system is schedulable. Note that no other priority assignment would have scheduled your system.

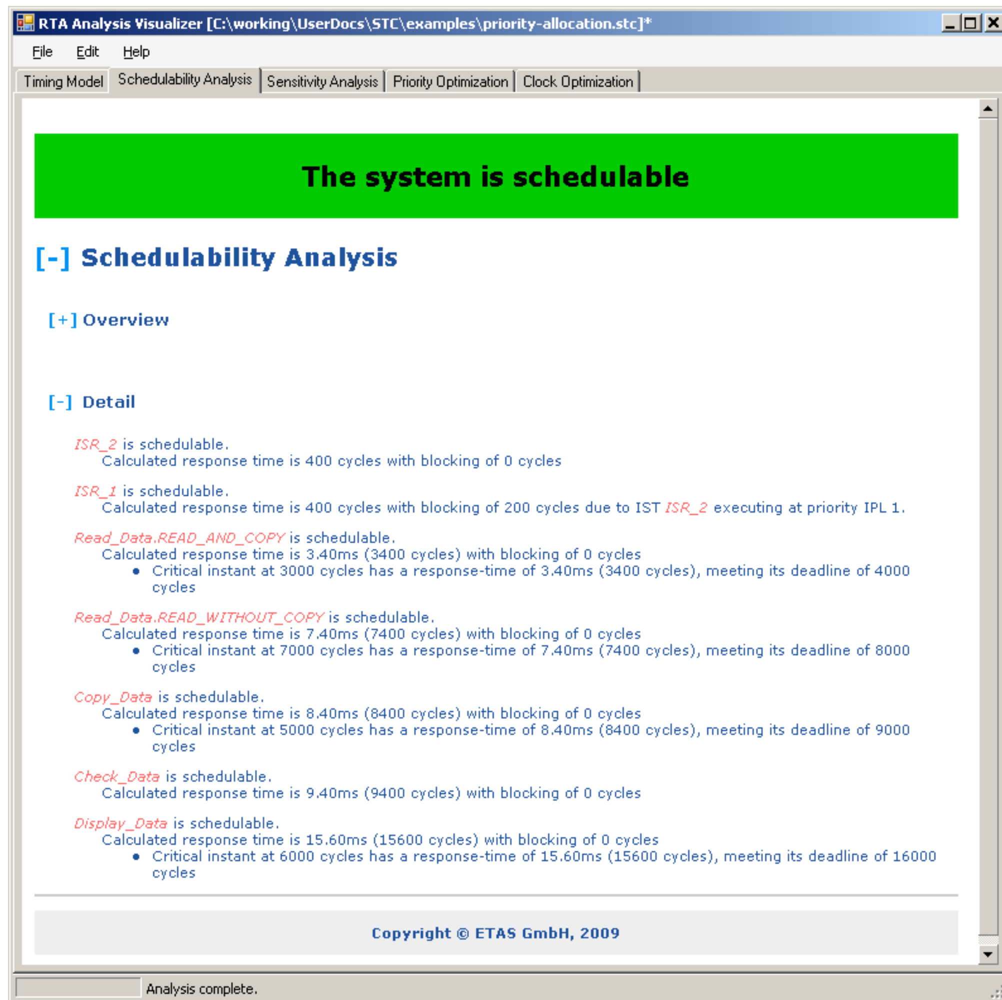


Figure 8.20: `priority-allocation.stc` schedulability analysis

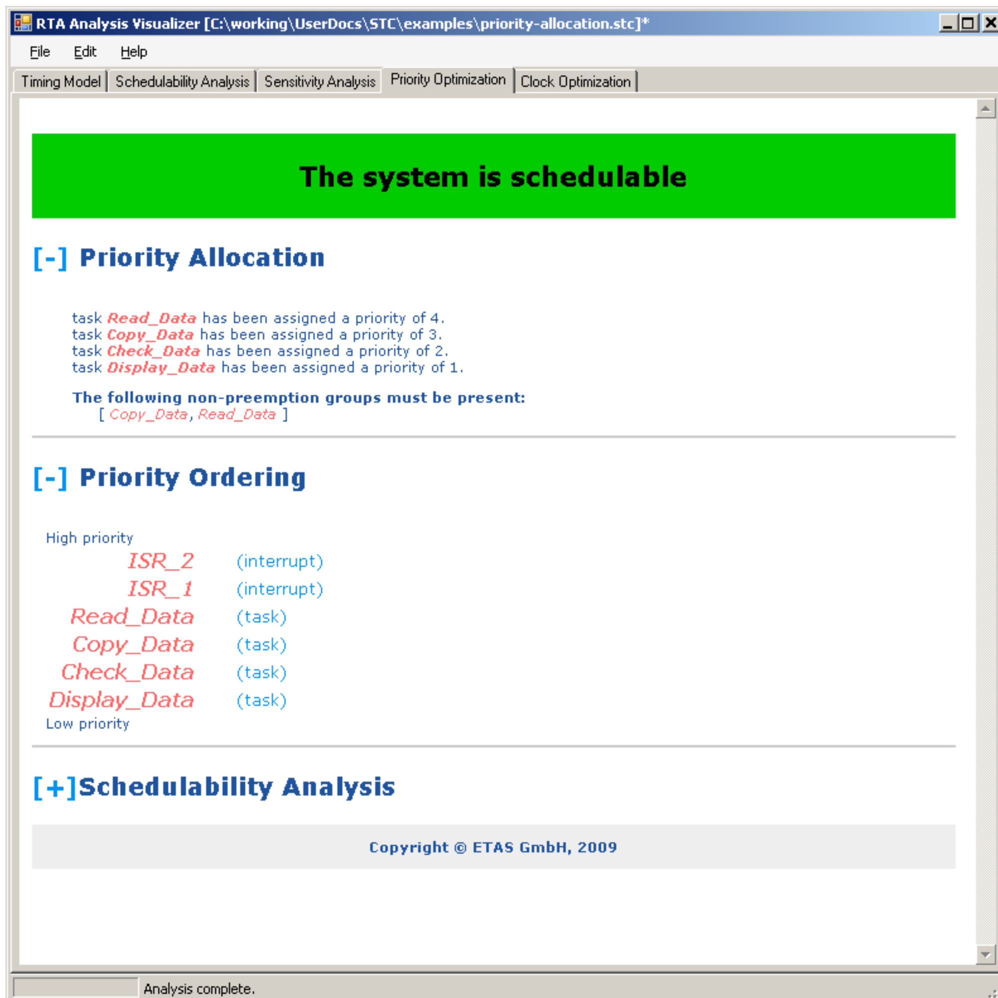


Figure 8.21: `priority-allocation.stc` priority optimization

Now run priority optimization. Figure 8.21 show the results.

The resulting priority order is identical to the one you first specified in your configuration file. In addition, the Analysis Visualizer has moved two tasks (`Copy_Data` and `Read_Data`) into a soft non-preemption group (indicated by ‘must not preempt each other’). Using non-preemption groups can reduce the number of preemption levels, minimizing the required stack size and memory usage. In some cases they are even essential to a system’s schedulability.

8.7 Changing processor frequency

In hard real-time systems it is essential that deadlines will always be met. In isolation, this requirement could lead to using the maximum approved processor frequency for all applications. However, such systems are typically embedded in environments where there may be conflicting requirements to reduce the processor frequency, for example to meet a power consumption target. Or perhaps a less expensive crystal supports a range of frequencies that does not include the approved maximum. We need to able to perform two related tasks:

- Given a processor frequency that meets all the system requirements, including being no less than the minimum possible frequency, we must be able to reconfigure the system quickly and accurately
- We must be able to determine the minimum processor frequency for which we can build a schedulable system.

The first of these can be accomplished by defining a macro to contain the processor frequency and defining all timebases in terms of this value, as we have seen in our earlier examples. The second requires us to use the clock optimization (command line option -c) function of the Analysis Visualizer. Note that sensitivity analysis tells us the minimum processor frequency given a fixed set of task priorities but clock optimization may rearrange task priorities to allow deadlines to be met at an even lower clock frequency.

8.7.1 Example

Suppose we are building a system configured with the file `clock-optimization.stc` and can easily configure our processor to run one of ten speeds: 100kHz, 200kHz, . . . , 1MHz. When we ran priority allocation and then sensitivity analysis before, the indicated minimum processor clock speed was 930kHz (93% of 1MHz). This would force us to use the 1MHz frequency. Now carry out clock optimization

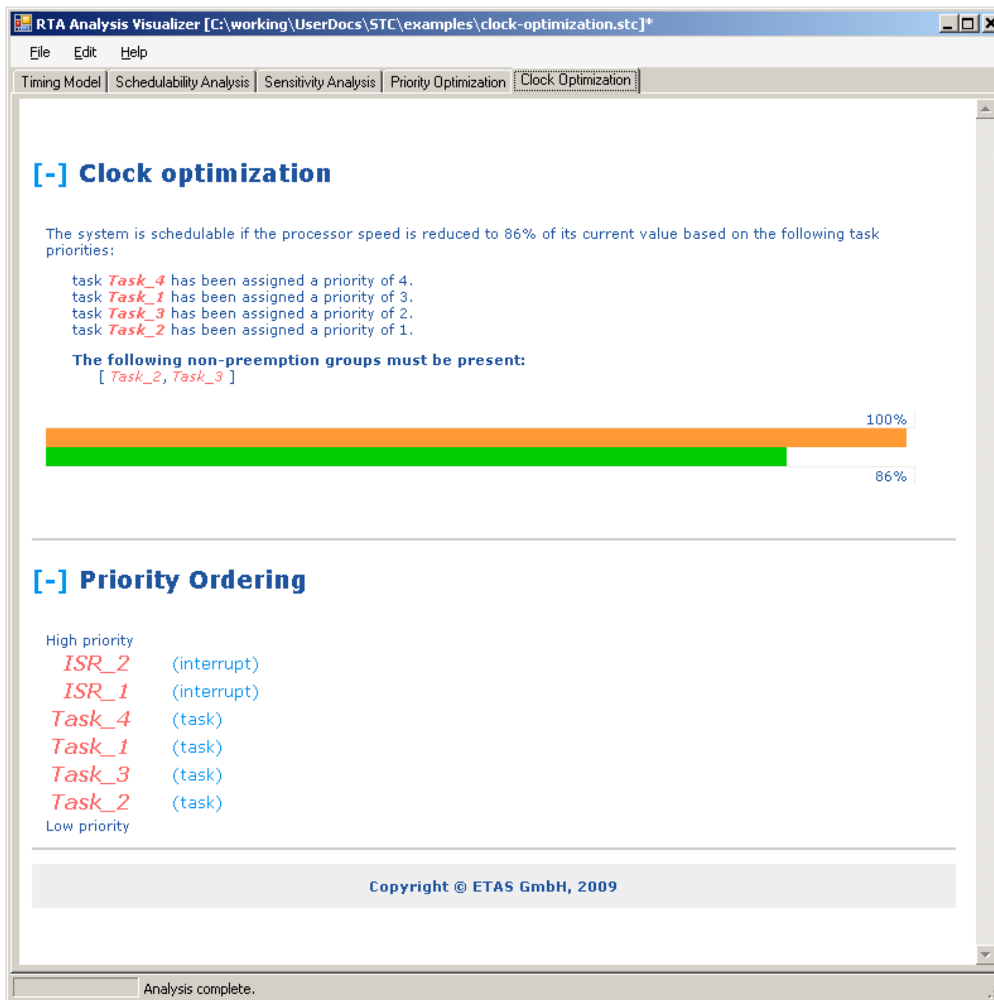


Figure 8.22: `clock-optimization.stc` clock optimization

Figure 8.22 shows that we can, in fact, find a schedulable configuration of task priorities for less than 930kHz.

Clock optimization has traded some stack usage for clock speed requirement. By allowing task Task_1 to preempt tasks Task_2 and Task_3, we are able to lower the processor clock speed to 860kHz. This means that we have a real choice of using either the 900kHz or 1MHz setting for our final system. Suppose it is decided that power consumption is a design factor and the 900kHz setting is chosen for production. Reconfigure the system by just changing the PROCESSOR_FREQUENCY macro body from 1000000 to 900000 in `clock-optimization.stc`. Now run priority allocation with this frequency to make sure we have a configuration optimized for exactly 900kHz

Notice that, because the periodic timeline is specified in real-time units (milliseconds) these are scaled correctly for the new processor clock speed. For

example Task_1 must run every 3ms, which at 1MHz was 3000 clock ticks. When we change the PROCESSOR_FREQUENCY macro to select 900kHz, this becomes 2700 clock ticks without further editing.

9 Configuration Language Reference

This chapter provides a complete reference too the STC configuration language.

9.1 Overview of syntax

9.1.1 Notation

The syntax is described using a modified BNF summarized below.

Symbol	Meaning
::=	A name on the left of the ::= is expressed using the syntax on the right
( )	Used to group together some syntax. This makes clear the boundaries of optional clauses. (See | immediately below)
|	The vertical bar indicates choice. Either the syntax on the left hand side or that on the right hand side of the vertical bar must appear.
task	The text in bold is reserved, it is either a keyword or mandatory punctuation.
[ ]	The text between the square brackets is optional. It may appear at most once.
+	A production marked with a plus is mandatory. It must appear one or more times.
*	A production marked with an asterisk is optional. It may appear zero or more times.
identifier	An identifier. Typically these are used to name objects so that they can be referred to in subsequent clauses. Identifiers follow the normal C rules for identifiers. In certain circumstances it is necessary to include characters that do not conform to the C conventions. In these cases the identifier can be enclosed in quotes "like this".
integer	An integer. Integers can be in decimal (first digit is in the range 1 to 9), hexadecimal (prefixed with 0x), binary (prefixed with 0b) or octal (prefixed with 0). For example, 123 is decimal, 0xf00d is hexadecimal, 0b10101 is binary and 0123 is octal.
`integer_or_real`	An integer or a real number. A real number is always in decimal and has an embedded decimal point. For example 1.23.

9.1.2 Declaration rules

Sections 9.2 to 9.17 describe the STC configuration file syntax for the different types of object as used by Analysis Visualizer. The following rule unites the individual parts of the syntax:

```
configuration ::=
    os_environment_definitions
    (
        timebase_declaration |
        timebase_conversion |
        resource_declaration |
        task_declaration |
        priority_constraints |
        nonpreemption_group_declaration |
        interrupt_declaration |
        timeline_declaration |
        activator_declaration |
        transaction_declaration
    )*
    [ arbitration_order ]
    system_timing_values
    interrupt_recognition
    soft_nonpreemption_group_declaration*
    task_priority_order*
```

Note objects need to be declared before they can be referenced. Objects should appear in the configuration file in the order listed above.

9.2 Base definitions

9.2.1 Time definitions

Configuration files are in free format and comments are allowed anywhere. The comment syntax is similar to that for C++ and allows anything from // to the end of a line, and anything enclosed in /* and */ delimiters is ignored.

The following syntax rule is used when an interval representing some number of counter ticks is specified.

```
units ::=
    identifier | ticks
```

Units always refer to a timebase that has been defined in the configuration file. The interval itself is specified as some number of ticks or some number of a named unit indicated by the identifier and defined within the associated timebase.

The syntax distinguishes between time specified in stopwatch units (units that are declared in the stopwatch timebase) and generic units (units that are declared in any timebase).

The following syntax rule is used to represent values expressed in stopwatch units. Note that 'on stopwatch' is optional and simply used for clarity.

```
sw_time ::=
    [on stopwatch] integer_or_real units
```

Timing values which can only be described in terms of the relevant timebase (e.g. timeline timebase) are defined by:

```
gen_time_tb ::=
    integer_or_real units
```

To refer to values expressed in generic units where time values can be specified in any unit on any timebase, the following two syntax are used:

```
gen_time_sw_def ::=
    [on (timebase_identifier | stopwatch)] integer_or_real units

gen_time_tb_def ::=
    [on (timebase_identifier | stopwatch)] integer_or_real units
```

9.2.2 Executable object profiles

The only difference between the `gen_time_sw_def` and `gen_time_tb_def` definition is the default behavior. Where no timebase is specified explicitly, the `gen_time_sw_def` definition defaults to the stopwatch timebase, while the `gen_time_tb_def` definition defaults to the timebase of the 'owning' entity - e.g. in a periodic timeline declaration it defaults to the timebase of the periodic timeline.

Note that timebase identifiers must match an existing timebase name and unit identifiers must match an existing unit on the specified timebase.

An executable object is a task, controlled or uncontrolled interrupt handler whose characteristics are described in the configuration file. Although tasks, controlled and uncontrolled interrupts have their own distinct behaviors, the Analysis Visualizer distills the features of all of these into a generic executable object so that the analysis can treat them all equivalently. Executable objects can be described by their execution profiles.

The identifier for the execution profile of a task is defined as:

```
task_profile_identifier ::=
    task identifier[.profile_identifier]
```

The identifier for the execution profile of an interrupt is given by:

```
interrupt_profile_identifier ::=
    interrupt identifier[.profile_identifier]
```

The identifier for the execution profile of an executable object is then defined as follows:

```
executable_object_profile_identifier ::=
    (task_profile_identifier | interrupt_profile_identifier)
```

Note that this form also includes execution profile identifiers for the idle task.

If a task or interrupt has more than one execution profile declared, the `profile_identifier` must be specified, otherwise it is optional.

9.3 Operating system environment definitions

The environment definitions tell the Analysis Visualizer which version of the operating system is used, the target platform and the build level.

The syntax for the OS declaration is as follows:

```
os_environment_definitions ::=
    kernel {
        version identifier ;
        target identifier ;
        build level = (standard | timing | extended) ;
    }
```

The `identifier` in the version clause is a user-defined string and must be quoted. It is common to use this to indicate which OS is being used(e.g. “RTA-OS”).

Similarly, the `identifier` in the target clause is a user-defined string that can be used to indicate the target for which is analysis is performed, or maybe the name of the project.

Example

```
kernel {
    version "RTA-OS";
    target "Babbage’s Analytical Engine";
    build level = extended;
}
```

9.4 Resource declaration

A resource declaration defines a resource which can be specified in task clauses. The syntax is as follows.

```
resource_declaration ::=
    resource identifier;
```

The identifier in the resource clause names the resource (for later reference in the configuration file) and also its handle which may be referred to in the application program.

9.5 Timebase declarations

A timebase declaration is an abstraction of the properties of a counter. The attributes captured are unit definitions and their conversion to counter ticks, the modulus, and user defined constants. The timebase declaration syntax is as follows.

```
timebase_declaration ::=
    timebase timebase_identifier {
    [ stopwatch ; ]
    units_definitions*
    modulus integer_or_real units ;
    }
```

The `timebase_identifier` after the timebase keyword is the name of the timebase. This can be used later in the configuration file when other objects need to refer to the timebase.

The `stopwatch` keyword marks this timebase for use in specifying the execution budget of tasks and ISRs. It also defines the units which are normally used to specify worst case execution times and deadlines. For the analysis, one timebase must be marked as the stopwatch timebase.

The `modulus` clause specifies the modulus of the counter represented by this timebase, e.g. if a counter goes from 0 to 9, then back to 0, its modulus is 10. The modulus can be specified in terms of ticks or some timebase specific unit (see the `units_definition` below). The `modulus` must evaluate to an integer in the range 2 to 65536 ticks.

The `units_definition` clause provides a mechanism for defining named units that can be used in other clauses to specify delays, execution time budgets, worst case execution times, deadlines, etc. Each unit must be specified as having a conversion factor to ticks or to a previously defined unit within the same timebase. The syntax for units is as follows:

```
units_definition ::=
    units identifier {
    define integer_or_real as integer_or_real units ;
    }
```

The identifier after the units keyword is the name of the new unit. Some number of these (the first `integer_or_real`) is defined as being equivalent to some number (the second `integer_or_real`) of ticks or some other previously defined unit within the same timebase.

Non-integer conversion factors are allowed. When dealing with values which are specified in defined units, the Analysis Visualizer uses a floating point format until final conversion to an integer value of ticks. If the conversion

from floating point to an integer causes too much loss of precision, or the value is too small or too large for its destination, then an error or warning message will result.

Each timebase has its own list of units. This means that different timebases may reuse the same names for units.

The stopwatch timebase uses the special unit ‘cycles’ to specify values which are shorter than a stopwatch tick. Where not explicitly defined, one cycle is taken to be one tick. Cycles should only be used where analysis-specific values have to be specified, e.g. in task execution profiles. Note, cycles are treated in a special way by the analysis, e.g. there are extra conversion routines for cycles, thus specifying another unit with the same purpose will not achieve the same result. Cycles are not allowed to be larger than stopwatch ticks.

9.6 Timebase conversion

The Analysis Visualizer converts all times represented within the configuration file to stopwatch cycles. For the analysis, a timebase conversion must be specified for each timebase which has been declared in the configuration file (except the stopwatch timebase).

The syntax of the timebase conversion is as follows:

```
timebase_conversion ::=
    stopwatch conversion {
    (on timebase_identifier integer_or_real units
    is [at worst] sw_time ;)+
    }
```

The on sub-clause lists the timebase and a time value (specified according to that timebase), and the is sub-clause lists the corresponding amount of time according to the stopwatch. Note that ‘at worst’ is optional and simply used for clarity. If it is not possible to supply an exact conversion from timebase to stopwatch, i.e. the result is non-integral, rounding may be performed by the Analysis Visualizer in order to ensure worst case values are used.

If a conversion results in a stopwatch value that exceeds its internal limit of 0xFFFFFFFE $2^{32} - 2$, the Analysis Visualizer will report an error. It is also an error if the conversion results in a timebase tick being smaller than a stopwatch cycle. All values must be greater than zero.

9.7 Task declarations

A task declaration either defines a user task and its attributes or allows the attributes of the predefined idle task to be set.

The syntax for task declarations is as follows.

```
task_declaration ::=
    idle_task_declaration |
    user_task_declaration

user_task_declaration ::=
    task identifier {
        entry identifier ;
        (locks resource identifier ;)*
        activates task identifier ;)*
        [ task_behavior ; ]
        task_execution_profile+
    }

idle_task_declaration ::=
    idle task {
        (locks resource identifier ;)*
        [ task_behavior ; ]
        task_execution_profile+
    }
```

A `user_task_declaration` declares a user defined task. The identifier after the task keyword uniquely names the task. This task name is used by the Analysis Visualizer in progress and result reporting and for generating tasks in the output file. It also is the name of the handle which may be referred to in the application program. Note that a task name must not contain the character '.'.

The user is allowed to specify certain attributes of the idle task, as indicated in the `idle_task_specification` above.

The entry clause specifies the entry point of the function that forms the body of the task. Task entry points can be shared with those of other tasks and ISRs. Typically, the identifier given in the entry clause is the name of the C function used to implement the task.

Each resource that a task locks must be specified in a locks resource clause. This clause causes the resource handles along with their static interface definitions to be output in the task's header file.

The activates clause is used to declare any tasks that can be directly activated by the defined task (this also includes task activation with

ChainTask()). Any tasks specified must have a lower priority. Also, the task itself and the idle task must not be included in an ‘activate’ clause. This information also ensures that automatic priority allocation does not allocate inappropriate levels.

The behavior of a task is classified as simple or re-triggering. A simple task must always complete before it can be made ready again. A re-triggering task permits the next invocation of the task to become ready during task execution, in which case the task needs to chain itself upon completion. Re-triggering behavior is not directly provided by RTA-OS3.x, but must be implemented by the user. Note that if a task is declared with no specified task behavior, it is assumed to be a simple task.

If task behavior is specified, it has the following syntax:

```
task_behavior ::=
    re-triggering [fifo buffer (limit integer | unlimited)];
```

A re-triggering task can be declared as fifo (first-in-first-out), in which case task arrivals are processed in the order in which they occur. Otherwise they are processed in priority order, i.e. the order in which their corresponding execution profiles have been specified. The buffer limit indicates the number of entries contained within the buffer and must be greater than zero (or unlimited). Unless the buffering of re-occurring events is handled by software the buffer size will be hardware-dependent. If during the analysis the number of simultaneously pending invocations exceeds the buffer limit, Analysis Visualizer will report the task to be unschedulable.

9.7.1 Task execution profile

The task execution profile clause is used to describe the timing characteristics of each set of task execution paths the user wishes to distinguish. The order in which execution profiles are declared is important as it informs the Analysis Visualizer which profile will be executed if there is more than one reason for the task to be running. The profile which appears first in the file is taken to have precedence. For example, a message handling task may have profiles for ‘priority ’ and ‘standard’ messages. If the code always tests for the presence of priority messages first, then the corresponding profile must appear first. These precedence rules do not apply to tasks declared as ‘fifo’ - by definition these execute the profiles in the strict order they occur in time.

The syntax of the task execution profile is as follows:

```
task_execution_profile ::=
    profile [profile_identifier] {
        this priority duration (gen_time_sw_def | undefined) ;
        (resource identifier duration [at exit]
```

```
        (gen_time_sw_def | undefined) ;)*
        (interrupt priority ( integer | OS level) duration [at
            exit]
        (gen_time_sw_def | undefined) ;)*
        (critical (gen_time_sw_def | undefined)
        has deadline (gen_time_sw_def |undefined) [max_response
            (gen_time_sw_def |undefined)];)*
        [ buffer (limit integer | unlimited) ; ]
    }
```

The `profile_identifier` after the profile keyword uniquely names the execution profile within a task declaration. If more than one execution profile is given within a task declaration, all profiles must be named. Note that a profile identifier must not contain the character '.'.

All time values in an execution profile must be specified in units defined by `gen_time_sw_def`. The keyword `undefined` is only permitted when no timing analysis is required. Where time values are specified in stopwatch units which are shorter than a stopwatch tick, the special unit cycles should be used.

The `duration` value for this priority is the worst case execution time of the task from the start of the first instruction of its entry function to the end of the return instruction of the entry function.

In the case of the resource and interrupt priority sub-clauses, the duration value represents the worst case time for which the task locks the specified resource or disables interrupts at the level specified by the first integer (or OS level). There must be at least one resource sub-clause for every resource which the task locks. Neither resource nor interrupt priority duration may be longer than the execution time as defined by this priority. Where multiple clauses within a profile describe the same resource or interrupt priority level, the longest duration is used for the analysis.

If a resource is not unlocked or the original interrupt level is not restored before the task exits, the special keyword at exit must be specified directly after the duration keyword of the resource or interrupt priority sub-clause.

The `critical .. has deadline` sub-clause is used to specify particular critical events. The critical value is the worst-case execution time from the start of the first instruction of the task's entry function to the completion of the critical event. The critical execution time must not exceed the task's execution time. The deadline value is the maximum permitted time from the arrival of the task to the completion of the same critical event. It must not be less than the associated critical execution time. Note that critical execution times do not need to be declared in a particular order. Each sub-clause can specify an

optional `max_response` that defines the response delay - the time between the response being computed and it manifesting itself in the real world.

The only situation in which the buffer limit can be declared is where the task's behavior is described as re-triggering, but for which no fifo buffer limit is given. In this case, task invocations are serviced in profile precedence order and a buffer limit must be specified for each profile and must have a value greater than zero (or unlimited). Please refer to the *User Guide* for more information.

9.8 Priority constraints

Sometimes it is useful to specify that one task must be of higher priority than another. The information is then taken into consideration by the Analysis Visualizer during analysis or when allocating task priorities.

The syntax is a follows:

```
priority constraints ::=
    priority constraints {
        (task identifier higher than task identifier ;)+
    }
```

For the analysis it is important that this clause does not contradict any existing declarations. e.g. if a task is declared higher than a task X, where the same task has already been declared (directly or indirectly) to be of lower priority than task X, Analysis Visualizer will report an error. If a task activates other tasks it must always be of higher priority than the tasks it activates. A task cannot be declared ‘higher than’ itself.

9.9 Non-preemption group declarations

A non-preemption group declaration allows tasks to be specified as executing in mutual exclusion.

The syntax of a non-preemption group declaration is as follows:

```
nonpreemption_group_declaration ::=
    nonpreemption group {
        (task identifier ;)+
    }

soft_nonpreemption_group_declaration ::=
    soft nonpreemption group {
        (task identifier ;)+
    }
```

Tasks belonging to the same non-preemption group will not preempt each other and therefore execute in mutual exclusion. It is possible to specify a task in more than one non-preemption group.

Output files generated by the Analysis Visualizer during automatic priority allocation can contain soft non-preemption groups. When building the system, soft non-preemption groups are treated like ordinary non-preemption groups. All non-preemption groups are normally retained during priority allocation, however, a non-preemption group that is defined as soft will be discarded prior to priority allocation.

Note that soft non-preemption groups are not intended to be declared by the user. A task must not appear in more than one soft non-preemption group.

9.10 Interrupt declarations

There are two classes of interrupt: controlled and uncontrolled. A controlled interrupt is handled by an interrupt service task (ISR) whereas an uncontrolled interrupt is serviced by an uncontrolled interrupt handler.

These are defined using the following syntax.

```
interrupt_declaration ::=
    interrupt identifier {
        entry identifier ;
        ( controlled | uncontrolled ) ;
        priority integer ;
        vector integer ;
        [ interrupt_behavior ; ]
        (locks resource identifier ;)*
        interrupt_execution_profile+
    }
```

The identifier after the interrupt keyword is the unique name of the interrupt and also the name of the handle which may be used in the application program.

The entry clause specifies the entry point of the function that forms the body of the interrupt handler. The execution budget clause allows the specification of an execution time budget for an ISR. It is not permitted for an uncontrolled interrupt. The timebase assumed in the units part of this clause is the stopwatch timebase. The keyword controlled or uncontrolled specifies the type of interrupt. Controlled interrupts are handled by RTA-OS3.x, which calls the associated ISR entry function. Uncontrolled interrupts are outside of the domain of RTA-OS3.x and must be handled directly by application code.

The priority clause defines (in a processor independent form) the processor interrupt priority level at which this handler will execute. The vector clause specifies the interrupt vector address which is usually target dependent.

If an interrupt is declared with no specific interrupt behavior, it is assumed to be a simple interrupt. If interrupt behavior is specified, it has the following syntax :

```
interrupt_behavior ::=
    (looping | re-triggering)
    [fifo buffer (limit integer | unlimited)];
```

This defines how the interrupt handler behaves if the interrupt become pending again while it is being processed. A looping interrupt handler will loop within the ISR until all pending interrupts that apply to this ISR have been dealt with. A re-triggering interrupt handler will either leave the interrupt

pending or reassert it, so that the handler is re-entered upon exit. A simple interrupt is defined on the assumption that another interrupt from the same source cannot be generated during the execution of the ISR. When an interrupt is defined as looping or re-triggering, it can be considered to be first-in first-out by including the fifo clause. If a fifo clause is included, the buffer limit indicates the maximum number of interrupts that can be pending. Unless the buffering of re-occurring events is handled by software the buffer size will be hardware-dependent. If during the analysis the number of simultaneously pending invocations exceeds the buffer limit, the Analysis Visualizer will report the interrupt to be unschedulable.

9.10.1 Interrupt execution profile

The interrupt execution profile clause is used to describe the timing characteristics of each execution path of an interrupt. Each interrupt declaration (controlled or uncontrolled) must have at least one execution profile specified. The order in which execution profiles are declared is important as it informs the Analysis Visualizer which profile will be executed if there is more than one reason for the interrupt to be pending. The profile which appears first in the file is taken to have precedence. For example, a UART interrupt handler may have profiles for ‘transmit buffer empty’ and ‘receive buffer full’. If the code always tests the receive buffer state first, then the corresponding profile must appear first. These precedence rules do not apply to interrupts declared as ‘fifo’ - by definition these execute the profiles in the strict order they occur in time.

The syntax of the interrupt execution profile is as follows:

```
interrupt_execution_profile ::=
    profile [profile_identifier] {
        this priority duration (gen_time_sw_def | undefined) ;
        (resource identifier duration [at exit]
          (gen_time_sw_def | undefined) ;)*
        (interrupt priority ( integer | OS level) duration [at
            exit]
        (gen_time_sw_def | undefined) ;)*
        (critical (gen_time_sw_def | undefined)
        has deadline (gen_time_sw_def | undefined) [max_response
            (gen_time_sw_def | undefined)];)*
        [ buffer (limit integer | unlimited) ; ]
    }
```

The `profile_identifier` after the profile keyword uniquely names the execution profile within an interrupt declaration. If more than one execution profile is given within an interrupt declaration, all profiles must be named.

All time values in an execution profile must be specified in units defined by `gen_time_sw_def`. The keyword undefined is only permitted when no timing analysis is required (e.g. system generation only). Note that where time is specified in stopwatch units which are shorter than a stopwatch tick, the special unit cycles should be used.

The duration value for this priority is the worst case execution time from the start of the first instruction of its entry function (in the case of an ISR) or the handler (in the case of an uncontrolled interrupt) to the end of the return instruction of the entry function/handler. In the case of the interrupt priority sub-clauses, the duration value represents the worst case time for which the ISR or uncontrolled interrupt handler disables interrupts to the level specified by the first integer (or OS level). The duration of interrupt priority sub clauses may not be longer than the execution time (as defined by this priority). Where multiple clauses within a profile describe the same interrupt priority level, the longest duration is used for the analysis.

If the original interrupt level is not restored before the interrupt handler exits , the special keyword at exit must be specified directly after the duration keyword of the interrupt priority sub-clause.

The `critical .. has deadline` sub-clause is used to specify particular critical events. The critical value is the worst-case execution time from the start of the first instruction of the entry function/handler to the completion of the critical event. The deadline value is the maximum permitted time from the arrival of the ISR or uncontrolled interrupt handler to the completion of the same critical event. The optional `max_response` defines the maximum permitted time between the response being generated and it being recognized.

The only situation in which the buffer limit can be declared is where the interrupt's behavior is described as looping or re-triggering, but for which no fifo buffer limit is given. In this case, the buffer limit must be specified for each execution profile.

9.11 Timeline declarations

A timeline is a sequence of arrivalpoints. A sequence of arrivalpoints describes the (offset) arrival pattern of a set of executable objects. Arrivalpoints perform two functions. Firstly, they can indicate which tasks the operating system will activate upon processing the corresponding arrivalpoint at run-time. Secondly, they can also contain information that is used to override some properties of the arrivalpoint. An arrivalpoint that contains only analysis information is termed an analysis-only arrivalpoint.

A timeline must be associated with a timebase to provide definitions of the units and modulus. There are two methods of specifying arrivalpoints: sequential and periodic timelines.

1. A sequential timeline is simply a sequence of arrivalpoints, one after another.
2. A periodic timeline is specified as a set of tasks to be executed with fixed periods and offsets. A periodic timeline provides a shorthand form for specifying the behavior of periodic tasks.

The syntax for sequential and periodic timelines is as follows:

```
timeline_declaration ::=
    timeline {
        timebase timebase_identifier ;
        default ( writable | readonly) ;
        ( sequential_spec | periodic_spec )
    }
```

The timebase clause specifies the timebase associated with the timeline.

9.11.1 Sequential timelines

The default is defunct and can be set to either `writable` or `readonly`. It does not affect the results of analysis.

A sequential timeline is declared as follows:

```
sequential_spec ::=
    sequence {
        (arrivalpoint | analysis_only_arrivalpoint)+
        [ next arrivalpoint_identifier ; ]
    }
```

The sequence keyword differentiates a sequential timeline from a periodic one. Contained within that clause is a collection of arrivalpoints and an optional next sub-clause. The ‘next’ property of each arrivalpoint is set to the next arrivalpoint declared within that timeline.

The next clause is optional and allows the successor to the final arrivalpoint to be specified. This must be an arrivalpoint derived from the same timebase (it may be from either the same timeline or from a previously declared timeline). If the next clause is omitted the timeline ends at that point at run-time. Note that the next arrivalpoint specified must not be an analysis-only arrivalpoint.

An arrivalpoint is defined as follows:

```
arrivalpoint ::=
    arrivalpoint [ arrivalpoint_identifier ] {
        (task_profile_identifier ;)*
        delay gen_time_tb ;
        [ analysis {
            (executable_object_profile_identifier ;)*
            [ delay gen_time_tb_def ;]
            [ next arrivalpoint_identifier ; ]
        } ]
    }
```

An analysis-only arrivalpoint is defined as follows:

```
analysis_only_arrivalpoint ::=
    arrivalpoint [ arrivalpoint_identifier ] {
        analysis {
            (executable_object_profile_identifier ;)*
            delay gen_time_tb_def ;
            [ next arrivalpoint_identifier ; ]
        }
    }
```

The optional `arrivalpoint_identifier` after the arrivalpoint keyword is used to uniquely name the arrivalpoint. The arrivalpoint may be referred to later in the configuration file (for example in a ‘next’ clause).

The optional `task_profile_identifier` identifies the execution profile of the task to be activated.

The delay clause specifies the delay before the next arrivalpoint. The units must be in ticks or a unit defined within the associated timebase. The delay must not be zero and must not exceed the modulus of the associated timebase.

The analysis clause allows the user to refer to executable objects that are activated indirectly as a result of the event described by the arrivalpoint. This

includes tasks that may are added during execution, and interrupt handlers that execute as a result of interrupts raised in response to the event. The executable objects specified within the task profiles and the other execution profiles must be unique within the arrivalpoint (each task can occur once only). If a delay or a valid next arrivalpoint is specified (one which has been previously declared and is on a timeline that is bound to the same timebase), it is used for analysis purposes only in place of the arrivalpoint's normal values. Note that a delay of zero is acceptable in this context. If the arrivalpoint is an analysis-only arrivalpoint, the delay value must be specified. The next arrivalpoint specified in the analysis clause can also be an analysis-only arrivalpoint.

9.11.2 Periodic timelines

A periodic timeline provides a shorthand form for specifying the behavior of periodic tasks. It consists of a set of task execution profiles that are activated with fixed periods. The arrivalpoints generated from the timeline do not have any analysis clauses, thus periodic timelines cannot have analysis only elements (e.g. interrupts).

The syntax of a periodic timeline declaration is as follows:

```
periodic_spec ::=
    periodic arrivalpoint_identifier {
        (task_profile_identifier every gen_time_tb (offset
            gen_time_tb)+ ;)+
    }
```

The periodic keyword is used to differentiate a periodic timeline from a sequential one. The `arrivalpoint_identifier` after the periodic keyword is the unique name of the first arrivalpoint in the timeline.

The `task_profile_identifier` identifies a task execution profile and the every sub-clause specifies its period. The period must be greater than zero and less than or equal to the modulus of the associated timebase.

The offset sub-clause specifies a new activation of the task execution profile at the offset specified into its period. This means that at run-time, processing the timeline will result in the task being activated n times per period, where n is the number of offsets specified for that task. The offset must be greater than or equal to zero and less than the task's period. It is not permitted to specify the same task execution profile more than once in a single periodic timeline clause and it is not permitted to repeat the same offset for the same task execution profile.

There must be at least one execution profile with an offset of 0.

A periodic timeline can be output in a format equivalent to the input format of a sequential timeline. This allows the generated timeline to be inspected, modified and if necessary inserted as a sequential timeline. For more details see Section 11.

Choosing task periods (as specified in the every clause) that have a large lowest common multiple can result in very long timelines when they are expanded to sequential timelines. Typically this is due to an oversight and the build process will automatically terminate if an attempt is made to generate a long timeline. Refer to Chapter 11 on how to change this default behavior.

It is also possible to choose a set of task periods such that the lowest common multiple exceeds the 32-bit representation used within the build tool. If this occurs, a fatal error message will result.

9.12 Activator declarations

The syntax for activator declarations is as follows:

```
activator_declaration ::=
    activator identifier {
        timebase timebase_identifier ;
        ( fine | coarse ) ;
        [ driver callbacks {
            now identifier ;
            cancel identifier ;
            state identifier ;
            set identifier ;
         } ]
        [ initial arrivalpoint_identifier [ autostart at
            gen_time_tb ] ; ]
    }
```

The identifier after the `activator` keyword is the name of the activator.

The `timebase` clause specifies which timebase this activator is associated with and therefore, indirectly, which timelines it may process.

The counter associated with an activator can be either `fine` or `coarse`. For `fine` activators, the driver callbacks clause must be present and must specify all four callback functions. For `coarse` activators, the driver callbacks clause must be absent.

The `initial` clause specifies the initial value of the 'next' arrivalpoint property of the activator.

The autostart at clause is only permitted if the activator is coarse and the initial clause is also present. The autostart at clause specifies the delay before the activator processes the initial arrivalpoint. The delay can be in ticks or units defined within the associated timebase. The delay must be greater than zero and less than or equal to the timebase's modulus. If the delay corresponds to 1 tick, then the activator will process the arrivalpoint on the first call to `IncrementCounter()` for the RTA-OS3.x counter used to drive the scheduling. If the delay is 2 ticks, then it will be processed on the second call to `IncrementCounter()` and so on).

9.13 Transaction declarations

Transactions are used to specify the timing relationships between executable objects. Two different types of transactions can be defined: timeline and bursting.

```
transaction_declaration ::=
    (timeline_transaction | bursting_transaction)
```

9.13.1 Timeline transaction

A timeline transaction specifies the timing relationship between a set of executable objects related to the processing of a timeline.

```
timeline_transaction ::=
    transaction identifier {
        [ release delay gen_time_sw_def ; ]
        [ jitter gen_time_sw_def; ]
        start arrivalpoint_identifier;
        [ activator identifier
             driven by executable_object_profile_identifier ; ]
    }
```

The `identifier` after the transaction keyword is the unique name of the transaction.

The `release delay` and `jitter` values, if given, describe the delay between the arrival of an event and its recognition. This delay will have a maximum and a minimum value. The `release delay` is the minimum value, and the `jitter` is the difference between the maximum value and the minimum value. If these values are not specified, they default to zero. Note that where time values are specified in stopwatch units which are shorter than a stopwatch tick, the special unit cycles should be used.

A timeline transaction follows arrivalpoints on a sequential or periodic timeline. The start sub-clause indicates the first arrivalpoint in the transaction. If the transaction is not activator driven, the arrivalpoint specified must be an analysis-only arrivalpoint.

If an activator drives the transaction, it is necessary to declare this in the activator clause. The driven by sub-clause specifies the execution profile of the task or interrupt that drives (ticks or expires) the activator.

9.13.2 Bursting transaction

Note that for a fine activator, the declared driver is added by the Analysis Visualizer to all arrivalpoints in the transaction. In the case of a coarse activator Analysis Visualizer will issue a warning if the declared driver does not appear in any transaction.

Due to the nature of activators, any executable objects that are directly released via an activator are usually ‘simple’. An executable object is normally not allowed to appear in a timeline for which it is also the activator’s driver (it’s only permitted if it can’t be reached by following each ‘next arrivalpoint’ beginning at the transaction’s ‘start’ arrivalpoint).

Analysis Visualizer will issue a warning if any executable object (other than the default idle task) does not appear in any transaction. Executable objects must not appear in more than one transaction, unless they have been declared as re-triggering (or looping) and execute in priority order (i.e. non-fifo). Usually, it is not possible for the same execution profile to appear in more than one transaction. However, where an executable object is declared as fifo, it is valid to specify more than one execution profile of that object in the same transaction.

A bursting transaction specifies the timing behavior of some event which results in the release of one or more executable objects.

```
bursting_transaction ::=
    transaction identifier {
        [ release delay gen_time_sw_def ; ]
        [ jitter gen_time_sw_def ; ]
        bursting {
            (integer times in (gen_time_sw_def | forever);)+
        }
        (executable_object_profile_identifier ;)+
    }
```

Like the timeline transaction, the identifier after the transaction keyword is the unique name of the transaction.

The release delay and jitter values, if given, describe the delay between the arrival of an event and its recognition. This delay will have a maximum and a minimum value. The release delay is the minimum value, and the jitter is the difference between the maximum value and the minimum value. If these values are not specified, they default to zero. Note that where time values are specified in stopwatch units which are shorter than a stopwatch tick, the special unit cycles should be used.

The bursting clause describes the maximum number of arrivals that can occur within a specified interval. If the interval is infinite it is possible to use the forever keyword instead. Multiple sub-clauses can be specified, however, they have to comply with certain rules , otherwise warnings will be issued stating that a clause has no effect.

Given a sorted set (in interval order) of bursting clauses, the number of arrivals and interval values for successive clauses must strictly increase, while the rate of arrival must strictly decrease.

Example of an invalid bursting clause:

```
bursting {
    2 times in 100 ticks;
    5 times in 200 ticks;
}
```

In the above example the second clause has no effect because the arrival rate does not decrease.

Example of a valid bursting clause:

```
bursting {
    2 times in 100 ticks;
    5 times in 300 ticks;
}
```

If more than one clause has the forever value specified, the clauses with the greater number of arrivals are ignored. All values must be greater than zero.

The Analysis Visualizer will report an error if a task or interrupt profile has been declared more than once in a bursting transaction. Note that the idle task can be declared in a bursting transaction.

9.14 Arbitration order

The arbitration order clause must be provided when multiple interrupts have been declared which share the same interrupt priority level. The arbitration order clause is used to specify the order in which interrupts of the same priority are serviced when two or more are pending.

The syntax of this clause is as follows:

```
arbitration_order ::=
    arbitration order {
        (interrupt priority integer {
            (interrupt identifier ;)+
         })*
    }
```

The interrupt priority clause specifies an interrupt priority level. The arbitration order of interrupts that share that level is given by the order of declaration in the sub-clause (interrupts higher up the list are taken in preference to those lower down). Where an interrupt priority level is shared, all interrupts that share this priority must be declared in the sub-clause.

The arbitration order clause can only appear once in the configuration file. When interrupts share the same priority level the arbitration order clause must be specified.

9.15 System timing values

The system timing clause contains eight target-specific values that define context switching times.

The syntax of the timing values is as follows:

```
system_timing_values ::=
    system timings  {
        integer ; // task entry latency
        integer ; // task switch overhead
        integer ; // ISR entry latency
        integer ; // ISR overhead
        integer ; // Uncontrolled interrupt entry latency
        integer ; // Uncontrolled interrupt overhead
        integer ; // minimum OS level blocking
        integer ; // UI blocking
    }
```

The system timings clause must contain eight integer values. They can be set to zero if you do not know the specific timings for your target.

9.16 Interrupt recognition

The interrupt recognition clause defines the maximum time for a single instruction during which an interrupt will not be recognized. The Analysis Visualizer includes this time as blocking time for all interrupt and task priority levels. This value is defined in terms of the stopwatch time. As the recognition time is target dependent, it is necessary to refer to an appropriate hardware manual for detailed information.

The syntax for the interrupt recognition clause is as follows:

```
interrupt_recognition ::=
    interrupt recognition sw_time ;
```

Note that where time values are specified in stopwatch units which are shorter than a stopwatch tick, the special unit cycles should be used.

9.17 Task priority order

Analysis Visualizer uses the task priority ordering clause to specify relative priorities of tasks.

The syntax is a follows:

```
task_priority_order ::=
    task priority order {
        (task identifier ;)+
    }
```

The order in which tasks are specified determines the priority order, i.e. the earlier a task is declared, the higher its priority. All user defined tasks, except the idle task, have to be specified, unless priority allocation has been requested or there is only one task in your system. Tasks must not appear more than once.

It is valid to declare more than one task priority clause as long it is still possible to identify an overall unique priority order.

Note that the Analysis Visualizer will report an error if this clause contradicts the specifications in the `priority constraints` declaration or `activates` clause of the task declaration.

9.18 Reserved words

The following words are reserved by the Analysis Visualizer configuration language:

activates activator alignment analysis arbitration arrivalpoint as at autoactivate autostart	budget buffer build bursting by bytes	callbacks cancel coarse constant constraints controlled conversion correction critical cycles	deadline debug default define delay disabled driven driver duration
entry every execution exit	fifo file fine float forever	group	has higher

idle in initial interface interrupt is	jitter	kernel	label level limit locks looping loops
max_response modulus	name next non-preemption now	offset on order OS	periodic priority production profile
readonly recognition release resource re-triggering	sequence set soft start state static stopwatch system	target task taskset than this ticks timebase timeline timepoint times timing timings transaction	uncontrolled undefined units unlimited
vector version	worst writable		

10 Configuration Language Pre-processor Reference

This chapter provides a reference for the preprocessor language that works with the Analysis Visualizer configuration language. The preprocessor language is interpreted by the Analysis Visualizer when processing configuration files. Section 10.1 describes the preprocessor syntax , Section 10.2 gives examples of the operation of the preprocessor.

10.1 Preprocessor syntax

The Analysis Visualizer provides a preprocessor that is similar in functionality to the familiar C preprocessor but uses a different syntax and a more flexible macro expansion policy.

Each preprocessor command is enclosed by matching brackets: '(' and ')'. The opening bracket must be followed by either the name of a macro or the name of a preprocessor command.

The table below gives a summary of the preprocessor commands and their syntax . In the table, a 'body' is a sequence of characters that may contain preprocessor commands. The preprocessor evaluates a body by repeatedly scanning it and interpreting preprocessor commands until it results in a sequence of characters containing no preprocessor commands. The symbols '[' and ']' are used to enclose optional syntax . White-space characters are space, tab and new-line.

10.1.1 File inclusion

Syntax

```
(include body)
```

Description

Textual file inclusion. The body must evaluate to a quoted string such as `"main_definitions.h"`. The quotes allow unusual characters in the file name (such as ')').

10.1.2 Macro definition

Syntax

```
(define name literal)
```

Description

The macro called `name` is defined to be the text in 'literal'. The name must not have any white-space characters in it. The literal starts after the first white-space character after the name and is delimited by the closing bracket that matches the one before the `define`. This allows the literal to be empty and to contain brackets and white space.

The literal is not evaluated in the definition. It is only evaluated when the macro is expanded. This allows macros to be nested. Note that the name may be generated by expansion of a body.

10.1.3 Macro undefine

Syntax

```
(undefine name)
```

Description

The macro called `name` is undefined. Note that the name may be generated by expansion of a body. See Section 3.

10.1.4 Macro expansion

Syntax

```
(name)
```

Description

The macro called name is expanded by pushing the literal (contained in the macro) back onto the input stream. The literal is then evaluated as a body.

Note that the name may be generated by expansion of a body.

10.1.5 Info

Syntax

```
(info body )
```

Description

The body is evaluated and printed on the Analysis Visualizer's standard output.

10.1.6 Warn

Syntax

```
(warn body)
```

Description

The body is evaluated and printed on the Analysis Visualizer's standard output. The global warning count is incremented.

10.1.7 Error

Syntax

`(error body)`

Description

The body is evaluated and printed on the Analysis Visualizer's standard output. The global error count is incremented.

10.1.8 Fatal

Syntax

`(fatal body)`

Description

The body is evaluated and printed on the Analysis Visualizer's standard output. the Analysis Visualizer exits

10.1.9 ifdef

Syntax

```
(ifdef condition
(then true_body)
[ (else false_body ) ]
)
```

Description

The condition is evaluated. The resulting text is treated as the name of a macro that may or may not exist.

If the macro named in the condition exists, then the `true_body` in the then clause is evaluated.

If the macro named in the condition does not exist, then the `false_body` in the else clause is evaluated.

The existence of the macro named in the condition is only tested when the condition is evaluated and not when a then or else clause is encountered.

10.1.10 ifndef

Syntax

```
(ifndef condition
(then true_body)
[ (else false_body ) ]
)
```

Description

The operation of `ifndef` is identical to that for `ifdef` except that the `true_body` is evaluated if the macro named in the condition did not exist and the `false_body` is evaluated if it did exist.

10.1.11 Compatibility with the C preprocessor

Syntax

```
# number "file name"
```

Description

The preprocessor's current file name and line number are updated. See Section 10.2.7 regarding compatibility with the C preprocessor.

10.2 Examples of Common Usage

The preprocessor works by reading input and copying it to the output until an open bracket '(' is encountered. The input enclosed by a matching set of brackets is interpreted as a command to the preprocessor. Section 10.1 defines the commands that the preprocessor recognizes.

10.2.1 Common usage

This section describes simple features of the preprocessor: textual file inclusion and simple macro definition and replacement.

Textual inclusion

Textual file inclusion works in a way similar to the C preprocessor. A file is included using the include command. For example:

```
(include "tasks.stc")
```

Causes the inclusion of the file tasks.stc from the current directory. Note that inclusions are always relative to the current directory or are absolute paths. So:

```
(include "c:\working\timers\hardware.stc")
```

includes the file `hardware.stc` from the directory `c:\working\timers`.

There is no equivalent of the C preprocessor's <filename> convention for finding files on a standard include path. See Section 10.2.6 for an alternative approach.

The text enclosed in quotes in an include is interpreted literally with no macro evaluation. This allows the use of spaces and parenthesis within the file name. Because of this, macro expansion takes place outside of include, see Section 10.2.6 for an example.

Macros

A macro can be defined on the Analysis Visualizer command line using the '-D' command line option. For example:

```
-Dname
-Dname=
```

or

```
-Dname=literal
```

In the first two cases the macro called name is defined to expand to an empty string. In the third case a macro called name is defined to expand to the text literal.

Macros can also be defined inside a file. For example:

```
(define mode extended)
```

Defines a macro called mode that expands to the text extended.

Note that the first white-space character after the macro name is stripped, it is not part of the literal text that the macro initially expands to.

A macro is evaluated by giving its name in brackets For example:

```
build level = (mode);
```

Evaluates to:

```
build level = extended;
```

A macro definition can span lines as illustrated in the following example.

```
(define Time_units // Definitions of standard timebase units
    units milli_seconds {
        define 1 as 1000 micro_seconds;
    }
    units seconds {
        define 1 as 1000 milli_seconds;
```

```
    }
    units minutes {
        define 1 as 60 seconds;
    }
)
```

Note that the C preprocessor's line continuation character ('
') is not required at the end of each line.

Assuming that the above definition of `Time_units` is contained in the file `standard.stc`, it can be used as follows:

```
(include "standard.stc")

timebase t {
    units micro_seconds {
        define 3 as 1 ticks;
    }
    (Time_units)
    modulus 65536 ticks;
}
```

The inclusion of `standard.stc` causes the macro `Time_units` to be defined. This is then expanded within a timebase clause to provide a set of unit declarations.

A macro can also appear within the definition of another macro. For example, the `standard.stc` file could contain the following macro definition:

```
(define Standard_header
    kernel {
        version "v1.6";
        target "ARM7T/ARMSDT";
        build level = (mode);
    }
)
```

To make use of this a configuration file, (called `config.stc`) might start with:

```
(include "standard.stc")
(Standard_header)
```

The `Standard_header` macro contains an embedded macro called mode. This must be defined before it is used. The '-D' command line option can be used to do this, for example:

```
stc -Dmode=debug config.stc
```

This mechanism allows the configuration file to be tailored via simple command line options.

When a macro contains a reference to another macro, the order of evaluation is strictly defined. See Section 10.2.4 for details.

If the literal text that a macro initially expands to contains brackets, then there must be a matching number of open '(' and close ')'.

Conditional evaluation

The preprocessor supports a simple form of conditional evaluation which is similar to the C preprocessor's **#ifdef** and **#ifndef** directives, for example:

```
(ifdef debug
    (then
        task debug_task {
            entry ... etc.
        }
    )
    (else
        task normal_task {
            entry ... etc.
        }
    )
)
```

Here the task called debug_task is declared only if the macro called debug is defined. If this macro is not defined then the task normal_task is declared instead.

ifdefs may nest inside one another, for example:

```
(ifdef a
    (then // This then refers to condition a
        (ifdef c // This \code{ifdef} only gets evaluated if a is
           defined
        (then d) // This then refers to condition c
        (else e)) // This else refers to condition c
    ) // terminates the outer then
    (else b) // This else refers to condition a
)
```

Note that the first white-space character after a then or an else is stripped in a similar way to macro definitions.

10.2.2 Nesting

The preprocessor commands may be nested provided that the following commands are not used when a command, name or condition is expected:

`define, undefine, fatal, error, warn, info.`

They can however be used when a body is expected - see Section 10.1 for details of when the preprocessor syntax expects a command, name or condition rather than a body. The above commands can also appear in the literal part of a macro definition, but are not evaluated until the macro is expanded.

Additionally, `ifdef` and `ifndef` may not be used in a macro name.

10.2.3 Precedence order

The precedence order for evaluation in descending order or precedence is as follows:

1. Comments
2. **`#line`** directives
3. Preprocessor commands
4. Macro expansions

Note, this means that comments are stripped first, hence, the 'Unix' path

`(include "//home/files/header.h")`

results in an error because `//` (the C++ style comment) causes the rest of the line to be discarded.

10.2.4 Order of evaluation

Macros are only evaluated at their point of use. This can be illustrated using an example from Section 10.2.1 on macros:

```
(define mode extended)
(define Standard_header
    kernel {
        version "v1.6";
        target "ARM7T/ARMSDT";
        build level = (mode);
    }
)
(Standard_header)
```

The above lines operate as follows. The definition of the macro called `Standard_header` uses the macro mode in its literal. The occurrence in the macro definition is not a point of use and so the macro `Standard_header` is defined to initially expand to:

```
kernel {
    version "v1.6";
    target "ARM7T/ARMSDT";
    build level = (mode);
}
```

When the line `(Standard_header)` is encountered it is evaluated as follows. The text

```
kernel {
    version "v1.6";
    target "ARM7T/ARMSDT";
    build level =
```

passes through unaltered, but as soon as `(mode)` is encountered it is recognized as a macro expansion and replaced with extended. This is pushed back onto the input and then passes through unaltered.

The text

```
;
}
```

is then encountered and passes through unaltered. So the overall effect of the macro is to produce:

```
kernel {
    version "v1.6";
    target "ARM7T/ARMSDT";
    build level = extended;
}
```

The order of evaluation used means that any macros embedded in the literal of another macro need not be defined prior to the definition of the enclosing macro.

Note that this order of evaluation differs from that of the C-preprocessor which expands macros when they are encountered.

10.2.5 Macro indirection

Macro expansions can be indirect. For example:

```
(define a b)
(define b c)
a
(a)
((a))
```

Evaluates to:

```
a
b
c
```

Section 10.2.4 explains why a and (a) produce a and b respectively. The final case, ((a)), works as follows. The innermost (a) is expanded first to produce b which is pushed back onto the input. However, as b is now enclosed in the outer brackets it is expanded and results in c.

Note that it is possible to define macros that are infinitely recursive. For example:

```
(define x x)
(define x (x))
(x)
```

When the preprocessor evaluates the macro x this will result in an error.

10.2.6 String concatenation

This facility is useful for combining two or more pieces of input into a single string so that the Analysis Visualizer can later interpret it as a single lexeme.

When a preprocessor command is evaluated no extra spaces are inserted. When a macro is defined the first character after the macro name is discarded. In the then and **else** clauses of `ifdef` and `ifndef` the first character after the then or else is also stripped. This character stripping is useful when concatenating strings. For example, the following input:

```
(define semi_minor_version j)
(define minor_version 4)
(define major_version 2)
version number =
    "v(major_version).(minor_version)(semi_minor_version)";
```

results in the output:

```
version number = "v2.4j";
```

If it is necessary for a macro to expand to text with leading spaces then those extra spaces can be placed on a new line. For example:

```
(define macro_with_leading_spaces
  there are leading spaces at the start of this line)
```

Note the new-line is treated as a white-space character and stripped as it is the first white-space following the macro name.

Concatenation can be used to achieve a similar effect to the C preprocessor's `-I` option. The `-I` option allows other directories to be searched for standard include files. For example, there might be a common directory that contains header files for a particular piece of hardware. Rather than build the full path name into the configuration file the following might be used.

```
(ifndef HWPATH
    (then (fatal The path to the hardware include files must be
        defined))
)
(define hardware "(HWPATH)\hardware.stc")
(include (hardware))
```

The above code attempts to include a file called `hardware.stc` from the path defined in `HWPATH`.

The Analysis Visualizer might be invoked with the following command line (from a makefile or batch file):

```
stc -DHWPATH=c:\project\hardware config.stc
```

This will result in the correct file being included. The Analysis Visualizer also understands Microsoft's UNC naming convention. This is shown in the following example:

```
stc -DHWPATH=\\Server\project\hardware config.stc
```

If the path to the hardware is omitted the `ifdef` code causes the preprocessor to generate a fatal error message.

Note, the mechanism described above only allows one directory to be searched for each file and so is less flexible than the C preprocessor's alternative. It does however have a more clearly defined behavior.

10.2.7 Compatibility with the C preprocessor

Many C compilers allow the results of the preprocessor to be output without being passed to the compiler. This allows the C preprocessor to be used as a general purpose preprocessor. The C preprocessor helps the C compiler keep track of files and line numbers by embedding the line number and file name in its output.

The Analysis Visualizer preprocessor supports the use of the C preprocessor by recognizing its line number and file directive. This is the purpose of the syntax below:

`# number "file name"`

Note that when using this compatibility feature it is assumed that the input file has already been processed by the C preprocessor and therefore contains no further directives either for the C preprocessor or this preprocessor. If this assumption is violated then the correct interpretation of the input file cannot be guaranteed.

Comments follow the C++ convention of `//` up to the end of the line, and anything bracketed by `/*` and `*/`.

The C++ style `//` comments are replaced by a new-line. C style comments `/*...*/` are replaced by a single space. This means that:

```
version/*my version numbers */20
```

becomes two tokens '`version`' and '`20`' rather than one '`version20`'.

The `/*` and `*/` style comments may not straddle more than one file.

Comments do not nest, so:

```
/*a /* b */c */
```

becomes:

```
c */
```

and a warning is generated indicating that an attempt has been made to nest comments.

11 Command Line

The following command line is used to invoke the Analysis Visualizer:

```
rtaosanvis [Options] [infile.stc] [outfile.html]
```

Optional command line arguments and parameters are indicated using [].

If outfile.html is present then the Analysis Visualizer run in command-line mode and store the results in the specified file. The Analysis Visualizer GUI will not be started.

11.1 Options

The following command line options are supported:

Option	Description
@<FILE>	Read command line parameters from <FILE>. Each command in <FILE> must appear on a separate. Quotation marks are not required to escape white space for filenames inside a command file. The @<FILE> option can itself appear multiple times inside <FILE>.
-A<TYPE>],--analysis:<TYPE>	Specify the analysis type. <TYPE> has the following options: • Schedulability • Sensitivity • Priority • Clock If the Analysis Visualizer is started in command-line mode then it will perform schedulability analysis by default.
-D<MACRO>,--define:<MACRO>	Define an Analysis Visualizer macro called <MACRO>.

Option	Description
--diagnostic	Display the diagnostic information on the standard output. Diagnostic information includes: • The version of the tool executable • The names and versions of all tool plug-ins • The names and version of all target plug-ins • The location and contents of the license file
-h, -?, --help	Display usage information on the standard output.
-S<PATH>, --stc:<PATH>	Specify the location of the STC.exe analysis engine executable.
--version	Show version information in compact form. More detailed information can be obtained using --diagnostic.

11.2 Examples

Open the Analysis Visualizer

```
rtaosanvis
```

Open the Analysis Visualizer with TimingModel.stc

```
rtaosanvis TimingModel.stc
```

Run schedulability analysis on TimingModel.stc, saving the results to Results.html

```
rtaosanvis --analysis:Schedulability TimingModel.stc Results.html
```

Schedulability analysis is the default analysis type, so the previous command line could have been written as:

```
rtaosanvis  TimingModel.stc Results.html
```

Run sensitivity analysis on TimingModel.stc, with macro CPUCLOCK defined as 42, saving the results to Results.html

```
rtaosanvis --define:CPUCLOCK=42 --analysis:Sensitivity
    TimingModel.stc Results.html
```

Run priority optimization on `TimingModel.stc`, saving the results to `Results.html`

```
rtaosanvis --analysis:Priority TimingModel.stc Results.html
```

Ask for help

```
rtaosanvis --help
```

12 Error Codes

The Analysis Visualizer reports various messages during operation. In order of decreasing severity there are:

Fatal messages. Caused where there are conditions in the input file, from which recovery is not possible. The Analysis Visualizer stops processing immediately after detecting and reporting a fatal error message.

Error messages. Generated where there are conflicts in the input file that make it impossible to perform analysis correctly or produce correct output. The tool attempts to report all the errors it can find and then exits. All errors should be removed and the operation should be repeated before attempting to use the results or output of the Analysis Visualizer.

Warning messages. Occur when the input file specifies an unusual condition, something that is redundant or a value that cannot be represented precisely and is therefore subject to rounding error. When warnings have been produced, the output files are created and the application can be built.

Information messages. Reports useful information such as the amount of memory used or the size of a data structure.

All fatal, error, warning and information messages are sent to the standard output and can therefore be easily re-directed to a file. All messages appear in the following format:

«message code»: «where the problem was detected»
«description of problem»
«possibly over more than 1 line»

«message code» This a unique five character error code which describes the error. The codes consist of one of the letters F, E, W and I (fatal, error, warning and information respectively) followed by a 4 digit hexadecimal number. The value of the number indicates the class of the error as follows:

0000 to 00FF	Configuration language preprocessor.
0100 to 7FFF	General Analysis Visualizer messages.
A000 to AFFF	Analysis-specific Analysis Visualizer messages.

«where the problem was detected» Reports a file and line number if the problem was found in a file. However, some problems can be found before any files are opened or after they have been closed. In these cases the location of the problem indicates that no line number is available as no input file is open.

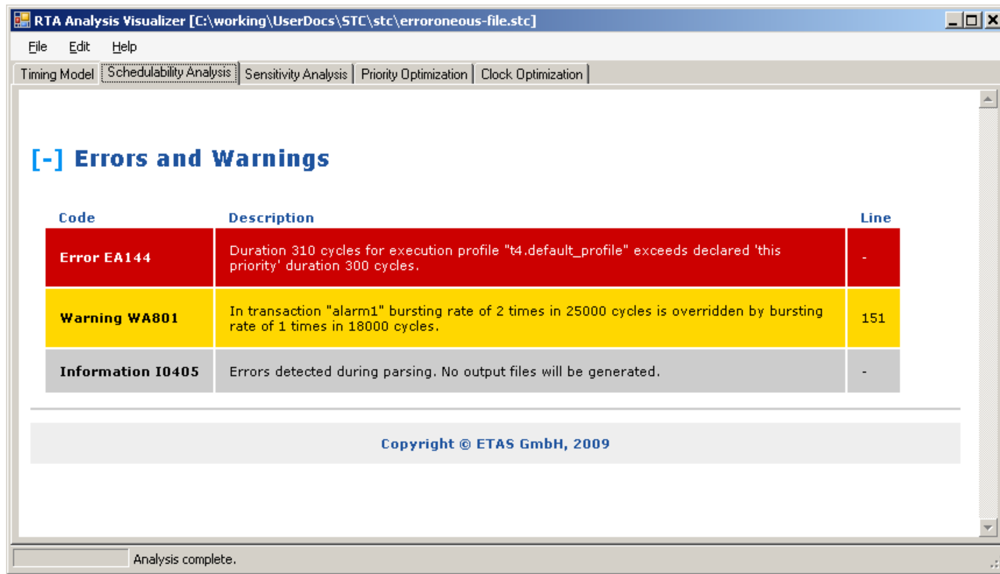


Figure 12.1: Example error reporting

«description of problem» Is text that describes the problem found.

Figure 12.1 shows an example of the message output from a run of the Analysis Visualizer on an erroneous input file.

12.1 Fatal Messages

Code	Description
F0000	`Invalid macro definition “«macro definition»”` **Cause:** This occurs when a macro defined on the command line is incorrectly formatted. The format should conform to that described in Chapter 10. **Remedy:** Inspect the command line used to invoke the Analysis Visualizer and ensure that the macro definition syntax is correct. Some command line interpreters will alter characters such as “=” and embedded spaces. Check that this is not the case. Also check that there is no space after the -D command line option.

Code	Description
F0001	`Unrecognized macro or preprocessor command “«name»”` **Cause:** The macro name or preprocessor command immediately after a ’(’ is unrecognized. Typically this is due to incorrect spelling or a macro that should have been defined but, for some reason, is not. **Remedy:** Use the location information to find the offending text.
F0002	`EOF in macro definition` **Cause:** During the definition of a macro an end-of-file was encountered before the closing ’)’. Any ’(’ and ’)’ characters must match in the macro’s literal text. So this fatal may have been caused by incorrect text. **Remedy:** Use the location information to find the offending text.
F0003	`“else” or “then” with no matching “ifdef” or “ifndef”` **Cause:** An else or a then clause was found but outside the scope of an ifdef. This might be caused because no ifdef has yet been used or because the closing ’)’ of an ifdef appeared before the then or else. **Remedy:** Use the location information to find the offending text.
F0004	`EOF in comment` **Cause:** An end-of-file has been found in a comment. **Remedy:** Use the location information to find the offending text.

Code	Description
F0005	`EOF in preprocessor directive or macro name` **Cause:** After a '(' while collecting the name of the macro or preprocessor directive an end-of-file was found. Typically this is caused by a '(' with nothing after it, or a partially completed macro name or preprocessor directive. **Remedy:** Use the location information to find the offending text.
F0006	`EOF in include file name` **Cause:** While processing the file name after an include command an end-of-file was found. Typically this is caused by a missing ')' but may also indicate a missing file name. **Remedy:** Use the location information to find the offending text.
F0008	`Missing ')' after include file name` **Cause:** The ')' that terminates an include preprocessor command is missing. **Remedy:** The ')' should be inserted.
F0009	`Failed to open input file called “«name of file»”` **Cause:** A file specified either in an include preprocessor command or on the command line could not be opened. The file name might be incorrect or the file might be protected in such a way that it can't be opened. **Remedy:** Check the name of the file and its access permissions.

Code	Description
F0010	`EOF in “ifdef” or “ifndef”` **Cause:** During the processing of an ifdef an end-of-file was found. Typically, this is caused by missing out the closing ’)’. **Remedy:** Use the location information to find the offending text.
F0011	`EOF inside fatal/error/warning/info message` **Cause:** While processing the body of a fatal, error, warning or info preprocessor command an end-of-file was found. Typically, this would be caused by missing out the closing ’)’. **Remedy:** Use the location information to find the offending text.
F0014	`«Message text defined by user»` **Cause:** A (fatal . . .) preprocessor command was used in the configuration file **Remedy:** This message was created as a result of preprocessor directives added by the user.
F0016	`Macro command used in incorrect context.` **Cause:** A fatal, error, warn, info, define or undefine command has been used in a place where only text, macro expansion, file inclusion or an if[n]def are legal. **Remedy:** Check syntax , especially for incorrect bracketing.

Code	Description

F0017	`Extraneous character («character») in “ifdef” or “ifndef” body.`
	
	Cause: After the condition in an `ifdef` the only text that may appear is the else or then clauses enclosed in parenthesis. This fatal is caused when text has been found outside the else and then clauses. Typically this might be because the closing ’)’ of the `ifdef` has been omitted, or because the opening ’(’ of a then or else clause has been omitted. Another possibility is that the condition evaluates to text with embedded spaces such as:
	
	`(define a b c)(ifdef (a) (then \ldots))`
	
	In this example the `ifdef` would evaluate to:
	
	`(ifdef b c (then \ldots))`
	
	and the c is extraneous.
	
	Remedy: Use the location information to find the offending text.

F0020	`Can’t set file and line in a macro.` **Cause:** A # line_number “file_name” directive has been seen at the point of expansion of a macro. It is legal to have a # line directive within the definition of a macro. **Remedy:** Remove the directive from the macro. See Section 10.2.7 for further information.
F0021	`EOF in macro name to undefine.` **Cause:** An end-of-file was found in the macro name within an undefine command. Typically, this is due to a missing ‘)’. **Remedy:** Check that the closing ’)’ appears in the undefine command.
F0022	`Missing ’)’ in undefine command.` **Cause:** An undefine command was not terminated with a closing ’)’. **Remedy:** Insert the missing ’)’.

Code	Description
F0024	`Unexpected “then” clause.` **Cause:** A then clause can only appear as the first clause in an `ifdef` or `ifndef`. This message is generated when a then is found elsewhere. **Remedy:** Ensure that all then clauses are enclosed by an ifdef or ifndef clause. Make sure that there is no more than one then clause in each `ifdef` and `ifndef` clause.
F0025	`EOF in body of “then” clause.` **Cause:** An end-of-file was found before the terminating ’)’ in a then clause. **Remedy:** Ensure that the `then` clause has correctly matched brackets.
F0026	`Unexpected “else” clause.` **Cause:** An else clause can only appear as the second clause in an `ifdef` or `ifndef`. This message is generated when an else clause has been found elsewhere. **Remedy:** Ensure that all **`else`** clauses are enclosed by an `ifdef` or `ifndef` clause. Make sure that there is no more than one else clause in each `ifdef` and `ifndef` clause and that it follows a `then` clause.
F0027	`EOF in body of “else” clause.` **Cause:** An end-of-file was found before the terminating ’)’ in an **`else`** clause. **Remedy:** Ensure that the else clause has correctly matched brackets.

Code	Description
F0028	`Unterminated pre-processor command at end of input. «further information»` **Cause:** After processing all input files at least one ')' was missing. **Remedy:** Check the matching of brackets. The «further information» gives details of where the opening '(' occurred.
F0029	`Macro name not terminated with closing ')'.` **Cause:** A macro expansion was not terminated correctly. There must be a ')' immediately following the macro name, instead there was white-space or other text. **Remedy:** Ensure that there are no missing brackets or extra spaces in macro expansions. If the macro name is generated by other preprocessor commands use the info command to print the name used and check that it doesn't contain white-space.
F0030	`Illegal character "«character»" in macro name "«macro name»".` **Cause:** An attempt was made to define a macro using the `-D` command line option and the name of the macro contains brackets. **Remedy:** Remove the bracket(s) from the macro name.
F0031	`Unbalanced brackets in literal text of macro called "«name of macro»". Text is «body of macro»` **Cause:** The literal text which a macro is defined as initially expanding to contains brackets that do not match either because the number of left and right brackets is not the same or because a closing bracket occurs first. **Remedy:** Ensure that any brackets balance and that a closing bracket does not appear first.

Code	Description
F0032	`Missing condition or then in ifdef or ifndef.` **Cause:** An ifdef or ifndef with no condition or then has been found. For example (ifdef) and (ifdef name) would cause this fatal error. **Remedy:** Ensure that any `ifdefs` or `ifndefs` are correctly formed.
F0033	`Extraneous characters in “include”.` **Cause:** Non-white-space characters were found in an include command, which were not part of the file name. **Remedy:** Remove the extraneous characters. Only white-space characters are allowed to separate the include file name from the command and closing bracket. If the body of the include command was generated using a macro expansion then use (info) to inspect what is generated for the body.
F0034	`: Extraneous characters in “undefine”.` **Cause:** Non-white-space characters were found in an undefine command, which were not part of the name of the macro to undefine. **Remedy:** Remove the extraneous characters. Only white-space characters are allowed to separate the macro name from the command and closing bracket. If the body of the undefine command was generated using a macro expansion then use (info) to inspect what is generated for the body.
F0101	`Run out of memory.` **Cause:** The Analysis Visualizer is attempting to allocate heap space for an identifier, but cannot do so. **Remedy:** Check that there is not a problem with your computer or that the configuration file does not contain too many objects.

Code	Description
F0102	`Maximum number of labeled objects exceeded.` **Cause:** The Analysis Visualizer generates internal labels but has a limit of 65536. This fatal indicates that the limit has been exceeded. **Remedy:** The number of labels used depends upon the complexity of the configuration file. So one solution is to simplify the file.
F0206	`Identifier too long «copy of identifier».` **Cause:** The maximum length of an identifier in the Analysis Visualizer is 63 characters; an identifier longer than this has been found. **Remedy:** Reduce the length of the identifier
F0208	`Quoted identifier too long «copy of identifier».` **Cause:** The maximum length of an identifier in the STC is 63 characters; a quoted identifier longer than this has been found. **Remedy:** Check that the identifier is correctly terminated (i.e. that the closing ” is in the right place); if it is, reduce the length of the identifier.
F0209	`End of line or end of file in quoted identifier «copy of identifier».` **Cause:** A quoted string has remained unterminated at the end of the line on which it occurs, or the end of the input file. **Remedy:** Check that the string is correctly terminated, or look for a spurious opening quote.

Code	Description
F0301	`Syntax error near «token» «message describing syntax error».` **Cause:** The syntax analyzer detected input that does not conform to the syntax of the Analysis Visualizer configuration language. A description of what the parser expected to find at that point in the configuration file follows. **Remedy:** Check the syntax and amend the file accordingly.
F0401	`Attempt to open more than 1 input file.` **Cause:** More than one input file was specified in the Analysis Visualizer command line. **Remedy:** Ensure only one input file is present on the command line.
F0801	`Cannot open assembler output file “«filename»”.` **Cause:** The specified file cannot be created. This may mean that the directory in which the file is intended to be placed does not exist or is not writable, that the syntax of the filename is illegal under the host operating system, or that a file of that name already exists but is not writable. **Remedy:** If the path and filename are correct but this error persists, contact your system administrator.
F0802	`Cannot open C output file “«filename»”.` **Cause:** The specified file cannot be created. This may mean that the directory in which the file is intended to be placed does not exist or is not writable, that the syntax of the filename is illegal under the host operating system, or that a file of that name already exists but is not writable. **Remedy:** If the path and filename are correct but this error persists, contact your system administrator.

Code	Description
F0803	`Cannot open preprocessor output file “ospp.out”.` **Cause:** The specified file cannot be created. This may mean that the directory in which the file is intended to be placed does not exist or is not writable, that the syntax of the filename is illegal under the host operating system, or that a file of that name already exists but is not writable. **Remedy:** If the path and filename are correct but this error persists, contact your system administrator.
F4007	`Timebase “«name»” not found.` **Cause:** An activator must have a timebase associated with it. The timebase specified has not been declared. **Remedy:** Ensure that the name corresponds to a declared timebase.
F4008	`Expecting “«name»” to be declared as a timebase; it is declared as «type».` **Cause:** The object referred to in the activator’s timebase clause is not declared as a timebase. **Remedy:** Ensure that the name corresponds to a declared timebase.
F4201	`Timebase «timebase name» not found in timeline.` **Cause:** The specified timebase could not be found, hence initialization of the timeline currently being described could not be completed. **Remedy:** Check that the timebase has been declared correctly and earlier in the configuration file than the timeline declaration.

Code	Description
F4202	`Expecting “«name»” to be a timebase; it is declared as a «type».` **Cause:** This occurs in the timebase clause of a timeline declaration. It occurs when the name used for the timeline exists but does not correspond to a timebase. It is the name of an object of type «type». **Remedy:** Use the name of a timebase.
F4214	`The periodic timeline named “«name of timeline»” is too complex to generate.` **Cause:** Calculation of the least common multiple (LCM) of the task periods resulted in an arithmetic overflow. The Analysis Visualizer cannot continue generating the periodic timeline. **Remedy:** Change one or more task periods in the periodic timeline clause. Try to avoid sets of periods that are co-prime. For example 7, 13 and 18 have an LCM of 1638. Whereas 8, 12, and 18 have an LCM of 72. Making task periods multiples of each other, or some common number, reduces the length of periodic timelines.
F4216	`Too many timepoints in periodic timelines.` **Cause:** A count is kept of the number of arrivalpoints that all periodic timelines create. If the set of periodic timelines specified causes a large number of arrivalpoints to be generated then this fatal error is generated. This is because long timelines consume large amounts of memory and are probably not intended. A likely cause is task periods that are co-prime. **Remedy:** Examine all periodic timelines, change one or more task periods. Making task periods multiples of each other, or some common number, reduces the length of periodic timelines. If long time lines are required, then see the “-l” command line option. The generation of long timelines may take considerable time and result in large output files.

Code	Description
FA001	`Program terminated by user.` **Cause:** User requested program termination by hitting <Ctrl+C> or <Ctrl+Break>. **Remedy:** N/A.
FA002	`Failed to open output file` **Cause:** Failed to open output file. **Remedy:** Check that your disk is not write-protected. If problem persists, contact your system administrator.
FA003	`Failed to delete output file` **Cause:** Failed to delete output file. **Remedy:** Check that your disk is not write-protected. If problem persists, contact your system administrator.
FA004	`This computer is not currently licensed for timing analysis` **Cause:** Your license file contains license features appropriate to your installation. **Remedy:** Check that your license file is installed in the correct location and is valid.
FA005	`This computer is not currently licensed for timing optimizations` **Cause:** Your license file contains license features appropriate to your installation. **Remedy:** Check that your license file is installed in the correct location and is valid.

12.2 Error Messages

Code	Description
E0013	`«Message text defined by user»` **Cause:** An (error ..) preprocessor command was used in the configuration file. **Remedy:** This message was created as a result of preprocessor directives added by the user.
E0201	`The binary number «number» must have at least one digit.` **Cause:** Binary numbers are specified in the form 0b[* 0|1 *]. The character after the b was not a zero or one and, therefore, the number does not have a digit in it. For example, the number “0bticks” was specified. **Remedy:** Correct the syntax of the number.
E0202	`Illegal binary number «number»` **Cause:** Binary numbers are specified in the form 0b[* 0|1 *]. No punctuation is supported. An illegal character was found in the number. **Remedy:** Correct the syntax of the number.
E0203	`The number «number» is too large` **Cause:** All numbers in the configuration file must lie within the range 0 to 4294967295. A number larger than this has been specified. **Remedy:** Use a number within range.
E0204	`The hex number «number» must have at least one digit.` **Cause:** Hexadecimal numbers are specified in the form 0x[* 0-9|a-f *]. The character after the x was not a valid hex digit. **Remedy:** Correct the syntax of the number.

Code	Description
E0205	`Illegal octal number «number».` **Cause:** Octal numbers are specified in the form 0[* 0-7 *]. No punctuation is supported. An illegal character was found in the number. **Remedy:** Correct the syntax of the number.
E0207	`Illegal empty quoted identifier.` **Cause:** An identifier was expected, but the empty string “” was found in its place. **Remedy:** Replace literal “”s with correct identifiers; check preprocessor macros are expanded correctly (see the “-E” command line option).
E0402	`Warnings were detected and are being treated as errors.` **Cause:** The “-e” command line option has been used and at least one warning was generated. **Remedy:** Refer to earlier Warnings and remove them individually.
E0601	`Trying to create an object of type «declared-type» named «name». The name «name» is already declared as an object of type «found-type».` **Cause:** Each object declared in the configuration file must have a unique name. An attempt has been made to create two objects with the same name. The name and type of the object currently being declared and the type of the existing object which already has this name are reported. **Remedy:** Rename one of the two objects.

Code	Description
E0602	`Trying to create an object of type «type» called “«name»”. Names prefixed by OS_, os_, _os_ and _OS_ are reserved for system use.` **Cause:** Users are not allowed to declare object with names that impinge upon the namespace reserved for the operating system. An attempt has been made to declare an object with a reserved name. **Remedy:** Rename the object .
E0701	`The timing calibration value specified («value») is out of range.` **Cause:** When the timing calibration values were being specified a value was used which was larger than 65535. **Remedy:** Enter the correct value. This can be obtained from the results of the timing calibration.
E1501	`You may only declare the idle task once.` **Cause:** Only one “idle task” clause is permitted, but more than one such clause appears in the configuration file. **Remedy:** Either remove extra declarations of the idle task or merge all clauses into a single clause.
E1502	`You may only declare the autoactivate list once.` **Cause:** Only one “autoactivate” clause is permitted, but more than one such clause appears in the configuration file. **Remedy:** Remove all but one declaration of the autoactivate list.

Code	Description
E1602	`Conversion of «floating value» to integer causes loss of precision. Converted value is «integer value»; error is «floating value» %.` **Cause:** The number of ticks specified, either directly, or when a time is given in terms of some user defined units , is converted to a whole number of ticks. If the loss of precision due to rounding amounts to more than 1% of the specified value, then this error is produced. **Remedy:** Modify the time value specified so that it evaluates to close to a whole number of ticks, or reduce the size of each tick, to increase precision.
E2001	`Too many tasks for the specified target. Maximum number of tasks is «tasks» including idle task.` **Cause:** More tasks have been declared than are permitted for the target. **Remedy:** Reduce the number of tasks.
E2002	`Illegal execution budget after addition of budget correction. The maximum allowed is 65535 cycles.` **Cause:** The execution budget has a correction value added to it to account for operating system overheads. The corrected budget is not allowed to exceed 65535 but has done so. **Remedy:** Correct the budget if it is incorrectly specified. Alternatively, the execution time of the task is too long for the timebase used for execution time measurement. Increasing the timebase tick duration would reduce the number of ticks which the budget evaluates to. However, it must be ensured that all other time values derived from the same timebase are still correct given the re-scaling. This can be achieved more easily if budgets are specified in named units related to ticks (see the unit declarations clause in Section 9.2.1) rather than in ticks. Note that timing correction values must be taken using exactly the same timebase and tick duration as that used to specify budgets.

Code	Description
E2003	`Resource list for task “«name»” can only contain resources.  Object “«name»” is a «type».` **Cause:** An attempt was made to lock an object that has been declared as some type other than a resource. **Remedy:** Ensure that only resources are locked by tasks.
E2004	`Attempt to use non-existent resource “«resource name»” by task “«task name»”` **Cause:** The named resource does not exist. The resource is not declared before it is referenced in a “locks” clause. **Remedy:** Check that the name of the resource in the “locks” clause of the task declaration and in the resource declaration is consistent. Ensure that the resource clause is before the task declarations for any tasks which lock the resource.
E2006	`Illegal attempt to use the «type» called «object name» as the entry point for the task called «taskname».` **Cause:** An attempt has been made to use an object which already exists on the namespace as the entry to a task. **Remedy:** Change the name of the object or the name of the task’s entry function to ensure name uniqueness.
E2009	`Illegal execution budget of 0 specified.` **Cause:** An attempt has been made to specify the execution budget of a task as 0. Tasks may not have 0 execution time budgets. The minimum value is 1. **Remedy:** Either remove the budget specification if timing measurement is not required for that task or replace 0 with the correct value.

Code	Description
E2010	`Attempt to assign priority 0 to task “«name»”. Priority 0 is reserved for the idle task.` **Cause:** A task has been declared at priority 0. Only the idle task can have priority 0. **Remedy:** Choose a priority greater than zero for the task
E2102	`Couldn’t find a task called “«taskname»”.` **Cause:** The named task has not been declared. **Remedy:** Ensure that the name given in the list corresponds to a declared task.
E2103	`The object “«name»” is a «type» but a task is expected.` **Cause:** An attempt has been made to add an object which is not a task to a list of tasks. **Remedy:** Ensure that the name given in the list corresponds to a declared task.
E4001	`In activator “«name»“, the specified autostart delay of «ticks» ticks is less than minimal counter resolution.` **Cause:** An attempt has been made to specify an autostart delay of less than 1 tick. **Remedy:** Increase the delay or reduce the duration of a tick so that the desired delay can be achieved. If the duration of a tick is changed, take care to ensure that all other time values derived from the same timebase are still correct given the re-scaling. This can be achieved more easily if delays are specified in named units related to ticks (see the `unit` declarations clause in Section 9.5) rather than in ticks.

Code	Description
E4002	`Zero autostart delay not permitted in activator «name».` **Cause:** A zero delay has been specified. **Remedy:** Specify a nonzero delay.
E4003	`Specified delay of «ticks» ticks exceeds valid range for counter modulus of «maxticks» in timebase “«base»”.` **Cause:** A delay longer than the modulus of the timebase being used by the activator has been requested **Remedy:** Decrease the delay or increase the duration of a tick or increase the modulus of the timebase so that the desired delay can be achieved. If the duration of a tick is changed, take care to ensure that all other time values derived from the same timebase are still correct given the re-scaling. This can be achieved more easily if delays are specified in named units related to ticks (see the unit declarations clause in Section 9.5) rather than in ticks.
E4004	`Expecting “«name»” to be declared as a timepoint; it is declared as «type».` **Cause:** The object referred to in the activator’s “initial” entry is not declared as an arrivalpoint. **Remedy:** Ensure that the name corresponds to a declared arrivalpoint.
E4009	`A coarse counter cannot have callbacks.` **Cause:** A coarse activator cannot have callbacks. However, one has been specified as having callbacks in the configuration file. **Remedy:** Ensure that the activator type is correct and that there are no callbacks specified if it is a coarse activator.

Code	Description
E4010	`A fine counter must have driver callbacks.` **Cause:** An activator marked as being associated with a fine counter must have callbacks. However, one has been specified as having no callbacks in the configuration file. **Remedy:** Ensure that the activator type is correct and that all four callbacks are present if it is a fine activator.
E4011	`Expecting timebase for timepoint “«timepoint»” to be “«correct base»”. It is declared with timebase “«wrong base»”.` **Cause:** An activator can only process timepoints which are associated with the same timebase as the activator. The specified “initial” arrivalpoint for the activator is associated with a different timebase. **Remedy:** Ensure that the activator and its initial arrivalpoint are associated with the same timebase.
E4012	`The activator “«activator_name»” has a fine counter. Only activators with coarse counters may be auto-started.` **Cause:** Only coarse activators may be auto-started. **Remedy:** Use a coarse activator or remove the autostart clause.
E4013	`Initial timepoint “«name»” not found.` **Cause:** The arrivalpoint specified as the initial entry for this activator could not be found. **Remedy:** Ensure that «name» corresponds to a previously declared arrivalpoint.

Code	Description
E4014	`A coarse counter can’t have a modulus other than 65536.` **Cause:** An attempt has been made to associate a coarse activator with a timebase that has a modulus other than 65536. A coarse activator cannot have a modulus other than 65536. **Remedy:** Change the modulus, the timebase or the type of the activator.
E4101	`Unknown unit “«unit name»” specified.` **Cause:** An attempt has been made to define a new unit in terms of some other unit that has not yet been defined. **Remedy:** Correct the name of the unit so that it refers to one that has already been defined or add a definition (earlier in the file) for the unit, if it does not exist.
E4102	`Illegal modulus «mod» specified.` **Cause:** The modulus for a counter must be in the range 1 < «mod» < «max», where «max» is a property of the hardware platform. **Remedy:** Ensure that the modulus value is within range.
E4103	`Illegal attempt to declare “«name»” as the stopwatch timebase. The timebase “«name»” is already declared as the stopwatch timebase.` **Cause:** An attempt has been made to declare more than one timebase as the stopwatch timebase. **Remedy:** Remove stopwatch keywords until only the correct timebase is marked.

Code	Description
E4104	`The stopwatch timebase modulus must be 65536.` **Cause:** The timebase that is used for timing measurements (the stopwatch timebase) must have a modulus of 65536. An attempt has been made to use a timebase with a modulus other than 65536 as the stopwatch timebase. **Remedy:** Either change the modulus or use a different timebase for the stopwatch.
E4105	`There is no stopwatch timebase defined. Therefore the only units available are ticks.` **Cause:** Execution time budgets can always be specified in ticks. However, if a stopwatch timebase is defined then the budget can also be specified in units defined for that timebase. This error is caused when units other than ticks have been used but no stopwatch timebase has previously been defined. **Remedy:** Mark the correct timebase as the stopwatch timebase. Ensure that the declaration of the stopwatch timebase precedes any budget clauses.
E4203	`Next timepoint “«timepoint name»” not found.` **Cause:** The arrivalpoint specified in the “next” clause of a timeline declaration does not exist. **Remedy:** Ensure that the “next” clause refers to a previously declared arrivalpoint.
E4204	`Expecting “«name»” to be a timepoint; it is declared as a «type».` **Cause:** The object specified in the “next” clause of a timeline is not an arrivalpoint; it has been declared as an object of another type. **Remedy:** Specify an arrivalpoint.

Code	Description
E4205	`Expecting timebase for “«name»” to be “«realbase»”. It is declared with timebase “«wrongbase»”.` **Cause:** The arrivalpoint specified in the “next” clause of a timeline is not associated with the same timebase as the timeline. **Remedy:** Refer to an arrivalpoint that is associated with the same timebase as the current timeline.
E4206	`The value specified for the period is illegal because it is greater than the timebase modulus.` **Cause:** The value specified in the “every” clause, when converted to ticks, is larger than the modulus of the timebase associated with the timeline. **Remedy:** Reduce the value specified or increase the modulus of the timebase or increase the duration of a tick. If the duration of a tick is changed, take care to ensure that all other values derived from the same timebase are still correct given the re-scaling. This can be achieved more easily if delays are specified in named units related to ticks (see the `unit` declarations clause in Section 9.5) rather than in ticks.
E4207	`The period of a periodic timeline component cannot be less than 1 tick.` **Cause:** The period is less than 1 tick. **Remedy:** Increase the value specified or decrease the duration of a tick for the associated timebase. If the duration of a tick is changed, take care to ensure that all other values derived from the same timebase are still correct given the re-scaling. This can be achieved more easily if delays are specified in named units related to ticks (see the `unit` declarations clause in Section 9.5) rather than in ticks.

Code	Description
E4210	`The value specified for the offset is too large.` **Cause:** The value specified in the “offset” clause, when converted to ticks, is larger than the modulus of the timebase associated with the timeline. **Remedy:** Reduce the value specified or increase the modulus of the timebase or increase the duration of a tick. If the duration of a tick is changed, take care to ensure that all other values derived from the same timebase are still correct given the re-scaling. This can be achieved more easily if delays are specified in named units related to ticks (see the `unit` declarations clause in Section 9.5) rather than in ticks.
E4211	`Task “«name of task»” not found.` **Cause:** The task referred to in the periodic timeline does not exist. **Remedy:** Ensure that the name corresponds to a declared task.
E4212	`The object “«name of object»” is a «type» but a task was expected.` **Cause:** The object referred to in the periodic timeline is not a task. **Remedy:** Ensure that the name corresponds to a declared task.
E4213	`The periodic timeline named “«name of timeline»” cannot be writable.` **Cause:** A periodic timeline cannot be writable but has been specified as such. **Remedy:** If a periodic timeline must be writable then use the “-t” command line option to dump the periodic timeline as a sequential one. The output from the Analysis Visualizer can then be edited to make the equivalent sequential timeline writable and included into the configuration file.

Code	Description
E4218	`The offset in the periodic timeline must be less than its period.` **Cause:** After conversion to ticks the offset in a periodic timeline "every" clause is greater than or equal to the period. **Remedy:** Reduce the offset or increase the period of the task taking care to take into account rounding errors in units. Alternatively, if an offset greater than the task's period is really required, then this can be achieved by specifying a sequential timeline as well as a periodic one. The total length of the sequential timeline should be equivalent to the offset required. Set the "next" clause of the sequential timeline to refer to the start of the periodic timeline. Care must be taken to adjust the offsets tasks have in the periodic timeline and to include them as necessary in the sequential timeline to achieve the desired offsets and periodic behavior.
E4219	`At least one task in timeline "«name»" must have a zero offset.` **Cause:** Periodic timelines must have one task whose offset is zero. However a periodic timeline has been specified where none of the tasks have 0 offset. **Remedy:** Specify a zero offset for at least one task or in the application program start processing the timeline after a delay equal to the shortest offset originally specified and subtract this from all the task offsets. Alternatively, specify a sequential timeline with a single arrivalpoint, which does not activate any tasks and has a delay equal to the minimum offset of the tasks in the original periodic timeline. Set the "next" clause of the sequential timeline to refer to the start of the periodic timeline and subtract the minimum offset from the offsets of all the tasks in the periodic timeline. Start the activator at the beginning of the sequential timeline rather than the periodic one.

Code	Description
E4220	`The task «taskname» can only appear once in the specification of periodic timeline «timeline».` **Cause:** A task appears more than once in the declaration of a periodic timeline. **Remedy:** Remove superfluous declarations. Note that it is also illegal to repeat the same offset for a single task.
E4221	`Idle task not permitted in periodic timeline.` **Cause:** The idle task has been specified in a periodic timeline. This is not permitted. **Remedy:** Remove the clause containing the idle task.
E4302	`Zero delay not permitted in timepoint.` **Cause:** A delay of zero has been specified. This is illegal because delays in arrivalpoints must be greater than zero and less than or equal to the modulus of the associated timebase. **Remedy:** Change the delay to a non-zero value, or delete the arrivalpoint and move the tasks into the next arrivalpoint.
E4303	`Specified delay of «delay» ticks exceeds valid range for counter modulus of «modulus» in timebase “«name»”.` **Cause:** A delay of greater than the timebase modulus has been specified. This is illegal because delays in arrivalpoints must be greater than zero and less than or equal to the modulus of the associated timebase. **Remedy:** Modify the delay appropriately, or add an arrivalpoint with no tasks to bridge the long delay required.

Code	Description
E4304	`Specified delay of «ticks» is less than minimal counter resolution.` **Cause:** A delay which is shorter than one tick has been specified. Typically this might be caused because any units used have evaluated to a number which has rounded down to zero ticks. **Remedy:** Increase the delay , or delete the arrivalpoint and move the tasks into the next arrivalpoint. Alternatively, if more precision is required, then reduce the duration of a tick for the associated timebase. If the duration of a tick is changed, take care to ensure that all other values derived from the same timebase are still correct given the re-scaling. This can be achieved more easily if delays are specified in named units related to ticks (see the `unit` declarations clause in Section 9.5) rather than in ticks.

E4401	`Illegal zero in conversion for unit “«name»”.`
	
	Cause: One of the following conditions has been detected:
	
	`unit bobs {`
	`  define 0 as N other things`
	`}`
	
	or
	
	`unit bobs {`
	`  define N as 0 other things`
	`}`
	
	Remedy: Correct the definition so that no zeros occur.

<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>E5001</td><td><code>Illegal attempt to use the «type» called “«name»” as the entry function for the interrupt called “«interrupt name»”.</code>

Cause: An attempt has been made to use an object that has been declared, but not as an interrupt entry function, as the entry function for an interrupt.

Remedy: Ensure that the correct unique name is being used. In particular, note that
<pre>interrupt fred {
  ...
  entry fred;
  ...
}</pre>is illegal.</td></tr>
<tr><td>E5002</td><td><code>Interrupt “«name»” is declared at priority 0. This level is reserved for user tasks and is illegal for interrupts.</code>

Cause: An interrupt has been declared as being at priority 0, which is illegal.

Remedy: Set the correct priority for the interrupt.</td></tr>
<tr><td>E5003</td><td><code>Uncontrolled interrupt “«name»” must have priority higher or equal to that of all controlled interrupts.</code>

Cause: An uncontrolled interrupt has been given a priority which is below that of a previously defined controlled interrupt. Uncontrolled interrupts are not permitted to have priorities below those of controlled interrupts.

Remedy: Reassign the priority of either the uncontrolled interrupt or the previously declared controlled interrupts.</td></tr>
</table>

Code	Description
E5004	`Controlled interrupt “«name»” must not have a priority higher than that of any uncontrolled interrupts.` **Cause:** A controlled interrupt has been given a priority that is above that of a previously defined uncontrolled interrupt. Controlled interrupts are not permitted to have priorities above those of uncontrolled interrupts. **Remedy:** Reassign the priority of either the controlled interrupt or the previously declared uncontrolled interrupts.
E5005	`Interrupt “«name»” is declared at a priority higher than the maximum for this target which is «integer».` **Cause:** An attempt has been made to use an interrupt priority that is outside the range supported on the specified target. **Remedy:** Change the priority of the interrupt. The range of permitted interrupt priorities is defined in the *Target/-Compiler Port Guide* for the given target.

Code	Description
E5006	`Illegal execution budget after addition of budget correction. The maximum allowed is 65535 cycles.` **Cause:** The execution budget has a correction value added to it to account for operating system overheads. The corrected budget is not allowed to exceed 65535 but has done so. **Remedy:** Correct the budget if it is incorrectly specified. Alternatively, the execution time of the ISR is too long for the timebase used for execution time measurement. Increasing the timebase tick duration would reduce the number of ticks which the budget evaluates to. However, it must be ensured that all other time values derived from the same timebase are still correct given the re-scaling. This can be achieved more easily if budgets are specified in named units related to ticks (see the unit declarations clause in Section 9.2.1) rather than in ticks. Note that the timing correction values must be taken using exactly the same timebase and tick duration as that used to specify budgets.
E5009	`Interrupt “«name»” is uncontrolled. Execution time budget not permitted for uncontrolled interrupts.` **Cause:** An execution time budget has been specified for an uncontrolled interrupt and this is not permitted. Uncontrolled interrupts are outside of the domain of the OS. **Remedy:** Remove the execution budget clause.
E5011	`Illegal execution budget of 0 specified.` **Cause:** An execution budget of 0 ticks has been specified. This is not permitted. **Remedy:** Specify a non-zero budget

Code	Description
E5013	`Cannot give the interrupt called “«name»” the label “«user-name»”; the system requires it to be labeled «system-name».` **Cause:** There is an internal requirement that this interrupt has a particular label. The user has inadvertently tried to subvert this requirement. **Remedy:** Remove the “label” clause from the interrupt declaration.
E5014	`Cannot share entry functions between interrupts of different types; «name1» and «name2».` **Cause:** Entry functions cannot be shared between controlled and uncontrolled interrupts. An attempt to share them has been made. **Remedy:** Ensure that interrupt entry functions are not shared between controlled and uncontrolled interrupts.
E5015	`Cannot share entry function “«name»” between a task and an uncontrolled interrupt.` **Cause:** Entry points cannot be shared between tasks and uncontrolled interrupts. An attempt to share them has been made. **Remedy:** Ensure entry points are not shared between tasks and uncontrolled interrupts.
E5016	`The default interrupt entry “«name»” cannot be shared with controlled interrupt “«name»”.` **Cause:** The default interrupt may only share its entry function with uncontrolled interrupts. It must not share an entry function with a controlled interrupt. **Remedy:** Ensure that the entry function name is either unique, or only shared with uncontrolled interrupts.

Code	Description
E6001	`Illegal attempt to add the idle task to a non-preemption group.` **Cause:** An attempt has been made to add the idle task to a non-preemption group. As the idle task never terminates, this would result in tasks which could never run and is therefore not permitted. **Remedy:** Remove the idle task from any non-preemption groups.
E7001	`No such resource “«name»”.` **Cause:** The resource referred to in the execution profile of a task does not exist. **Remedy:** Ensure that the name refers to a declared resource.
EA100	`Task «taskname» requested for sensitivity analysis does not exist.` **Cause:** Sensitivity analysis requested for a name which does not correspond to a task, ISR or uncontrolled interrupt. **Remedy:** Specify an existing identifier.
EA101	`Task «taskname» does not exist.` **Cause:** «taskname» does not identify an existing task **Remedy:** Specify an existing task identifier.
EA102	`The «objecttype» «objectname» with ‘simple’ behavior appears in transaction «transname» and «transname».` **Cause:** A simple executable object appears in more than one transaction. Simple objects are only permitted to be in one transaction. **Remedy:** Ensure that the executable object only appears in one transaction, or change the behavior of the object to ‘re-triggering’ or ‘looping’.

Code	Description
EA103	`Task «taskname» has already had priority order specified.` **Cause:** Task declared more than once in task priority order. **Remedy:** Remove duplicated declaration.
EA104	`Task «taskname» has no priority order specified.` **Cause:** Task omitted in task priority order clause. **Remedy:** Add task to task priority order or run automatic priority allocation.
EA105	`Cannot specify priority order for the idle task,` **Cause:** The idle task is not allowed in task priority order clause. **Remedy:** Remove declaration from task priority clause.
EA106	`Transaction «transname» is driven by «objecttype» «objectname», which does not have a higher priority than activated «objecttype» «objectname».` **Cause:** A task drives an activator which is capable of activating an executable object with base priority greater than that of the driving task. **Remedy:** Tasks are not permitted to activate (directly or indirectly) a higher priority task. Change the task's priorities, use different driver task or remove the activated executable object from timeline.
EA107	`Task «taskname» must be lower than task «taskname».` **Cause:** Conflicting task priority information. **Remedy:** Make sure the parts of the configuration file that can affect task priorities are consistent. Configuration file elements that can affect task priorities are the 'activates task' clause of the task declaration, the priority constraints clause and the task priority order clause.

Code	Description
EA108	`Task «taskname» cannot have higher priority than itself.` **Cause:** A task cannot be declared higher than itself, or be a driver on a timeline which it is on. **Remedy:** Remove or change declaration.
EA111	`Declared critical execution time «value» cycles for execution profile «profilename» exceeds execution time «value» cycles.` **Cause:** Critical execution time is greater than execution time. **Remedy:** Ensure that all critical execution times are less than or equal to the 'this priority' execution time of the profile in which they appear.
EA112	`Declared critical execution time «value» cycles for execution profile «profilename» exceeds deadline of «value» cycles.` **Cause:** Deadline is less than the critical execution time. **Remedy:** Ensure that all deadlines are no less than their corresponding critical execution times.
EA114	`Task entry latency of «value» cycles cannot be greater than task switch overhead of «value» cycles.` **Cause:** Task entry latency is greater than task switch overhead. **Remedy:** Change system timings values accordingly. The task switch overhead consists of entry and exit latencies , thus the value cannot be smaller than the task entry latency.

Code	Description
EA116	`Controlled interrupt latency of «value» cycles cannot be greater than controlled interrupt overhead of «value» cycles.` **Cause:** Controlled interrupt latency is greater than controlled interrupt overhead. **Remedy:** Change system timings values accordingly. The overhead includes entry latency, thus the value cannot be smaller than the interrupt entry latency.
EA117	`Uncontrolled interrupt latency of «value» cycles cannot be greater than uncontrolled interrupt overhead of «value» cycles.` **Cause:** Uncontrolled interrupt entry latency is greater than uncontrolled interrupt overhead. **Remedy:** Change system timings values accordingly. The overhead includes entry latency, thus the value cannot be smaller than the interrupt entry latency
EA120	`Nonexistent time unit in stopwatch conversion for timebase «name».` **Cause:** A non-existent time unit is mentioned in stopwatch conversion. **Remedy:** Check that the time unit has been defined.
EA121	`More than one conversion to stopwatch for timebase «name».` **Cause:** More than one stopwatch conversion clause has been specified for the same timebase. **Remedy:** Remove the duplicated stopwatch conversion clause.

Code	Description
EA122	`Timebase «name» does not have a stopwatch conversion defined,` **Cause:** A timebase does not have a stopwatch conversion. Stopwatch conversion clauses are required if analysis is needed. **Remedy:** Add a stopwatch conversion for this particular timebase.
EA123	`The stopwatch timebase «name» cannot have a stopwatch conversion factor.` **Cause:** The stopwatch timebase may not have a stopwatch conversion. **Remedy:** Remove stopwatch conversion for stopwatch timebase.
EA124	`Time «value» ticks on timebase «name» exceeds system integer capacity.` **Cause:** A time conversion results in an analysis value that exceeds the capacity of an integer cycle (4294967295). **Remedy:** Use units which, when converted into stopwatch cycles, will not result in a conversion overflow. It may be necessary to make stopwatch cycles longer.
EA126	`Interrupt priority level «value» is shared but no arbitration order for it is present.` **Cause:** No arbitration order supplied for a shared interrupt priority level. The analysis needs to know the relative precedence of interrupts at the same hardware priority level. **Remedy:** Add arbitration order for this interrupt priority level or change the interrupt priorities so that they are not equal.

Code	Description
EA127	`Interrupt «name» has priority «value». The arbitration order shows it as «value».` **Cause:** In an arbitration sub-clause an interrupt is mentioned which is not at that priority. **Remedy:** Change the arbitration order accordingly or declare the interrupt at a different priority.
EA128	`Interrupt «name» at priority «value» not listed in arbitration order.` **Cause:** In an arbitration sub-clause an interrupt that is at that priority is not mentioned. **Remedy:** Add an entry for that interrupt in the arbitration order clause, or correct the interrupt priority if it has been specified incorrectly.
EA129	`Interrupt «name» is declared more than once in the arbitration order.` **Cause:** In an arbitration sub-clause the same interrupt is mentioned more than once. **Remedy:** Remove duplicated entry.
EA130	`Interrupt «name» does not exist.` **Cause:** In an arbitration sub-clause a non-existent interrupt is mentioned. **Remedy:** Make sure all interrupts mentioned are correctly declared.
EA131	`In transaction «name», «name» is not a valid arrivalpoint.` **Cause:** The start arrival point for a transaction cannot be found. **Remedy:** Check the name of the arrivalpoint specified in the start clause of the transaction.

Code	Description
EA132	`In transaction «name», «name» does not represent an execution profile of a task or controlled interrupt.` **Cause:** The 'driven by' identifier of a transaction is not a valid task or controlled interrupt. **Remedy:** Check that the identifier has been declared.
EA133	`In transaction «name», the driver «name» has been found to activate itself.` **Cause:** A executable object is declared in an arrival point where it is also the driver. **Remedy:** Either change the driver or remove the executable object from transaction.
EA136	`In transaction «name» driven by controlled interrupt «name», another interrupt «name» has been activated.` **Cause:** The driver is an ISR and other interrupts are included in an arrival point. **Remedy:** Where a timeline transaction is driven by a controlled interrupt, the timeline must not include interrupts in analysis clauses. Remove any profiles of interrupts from the timeline processed by the activator.
EA137	`The «objecttype» «objectname» appears more than once in transaction «name».` **Cause:** A task or interrupt is declared more than once in a bursting transaction. **Remedy:** Remove duplicated declarations from bursting transaction.
EA139	`Task «name» does not exist.` **Cause:** A non existent task is specified in an 'activates ' clause. **Remedy:** Make sure the task specified has been declared.

Code	Description
EA140	`Task «name» cannot activate the idle task.` **Cause:** The idle task is not allowed in an 'activates ' clause. **Remedy:** Remove declaration.
EA141	`Buffer limits must be greater than zero.` **Cause:** The specified fifo buffer limit is less than 1. **Remedy:** Specify a value greater than zero.
EA143	`No execution profile in task «name» uses resource «name».` **Cause:** No timing information is supplied for a resource the task locks. **Remedy:** Add timing information to the execution profile for this particular resource.
EA144	`Duration «value» cycles for execution profile «name» exceeds declared 'this priority' duration «value» cycles.` **Cause:** Resource locking time or interrupt priority level disable time is longer than the task's execution time. **Remedy:** Change execution times accordingly.
EA148	`Duration «value» cycles for execution profile «name» exceeds declared 'this priority' duration «value» cycles.` **Cause:** An interrupt priority level disable time is specified which is longer than the interrupt's execution time. **Remedy:** Change execution times accordingly.
EA150	`Task name «name» contains the illegal character «character».` **Cause:** Task name contains illegal character. **Remedy:** Remove illegal character.

Code	Description
EA151	Interrupt name «name» contains the illegal character «character». **Cause:** Interrupt name contains illegal character. **Remedy:** Remove illegal character.
EA152	Execution profile name «name» in task «name» contains the illegal character «character». **Cause:** Task execution profile name contains illegal character. **Remedy:** Remove illegal character.
EA153	Execution profiles «name» and «name» detected in the same arrivalpoint. Only one execution profile per task or interrupt may be declared in any one arrivalpoint **Cause:** A task or interrupt is defined more than once in an analysis clause. **Remedy:** Remove duplicated entry.
EA157	Expecting timebase for «name» to be «name». It is declared with timebase «name». **Cause:** A “next” value points to a timeline with another timebase. **Remedy:** Make sure the timelines are declared with the same timebase.
EA158	The stopwatch timebase has not been declared yet. **Cause:** No stopwatch is declared. **Remedy:** Add stopwatch timebase declaration.

Code	Description
EA159	`Execution profile name «name» in interrupt «name» contains the illegal character «character».` **Cause:** Interrupt execution profile name contains illegal character. **Remedy:** Remove illegal character.
EA160	`Priority allocation, clock optimization and sensitivity analysis are mutually exclusive.` **Cause:** More than one of priority allocation, sensitivity analysis or clock optimization were requested from the command line. **Remedy:** Split the processing into separate phases.
EA161	`The stopwatch timebase does not have a unit named «name».` **Cause:** Units identifier for a sw_time value does not match existing unit on stopwatch timebase. **Remedy:** Make sure the specified stopwatch unit has been defined in the stopwatch timebase.
EA162	`Timebase «name» does not exist.` **Cause:** Timebase identifier does not match an existing timebase name. **Remedy:** Check that the timebase has been declared.
EA163	`Timebase «name» does not have a unit named «name».` **Cause:** Units identifier doesn't match an existing unit on specified timebase. **Remedy:** Make sure the specified unit has been defined in the particular timebase.

Code	Description
EA164	`Task «name» does not exist.` **Cause:** In an execution profile reference, the task identifier does not match the name of a task **Remedy:** Check that the task has been declared.
EA165	`Interrupt «name» does not exist.` **Cause:** In an execution profile, the interrupt identifier does not match the name of an interrupt **Remedy:** Check that the interrupt has been declared.
EA166	`«name» has more than one profile so all profiles need names.` **Cause:** Execution profile name omitted but task or interrupt has more than one execution profile. **Remedy:** Add identifiers for each profile.
EA167	`Profile «name» does not exist in «objecttype» «objectname».` **Cause:** Profile name specified does not match execution profile name of the task or interrupt. **Remedy:** Check that the profile has been declared.
EA168	`Arrivalpoint «name» not found.` **Cause:** Next identifier does not match an existing arrivalpoint. **Remedy:** Check the name of the arrivalpoint specified in the next clause.

Code	Description
EA169	`The «objecttype» «objectname» is used in transactions «name» and «name». A ‘fifo’ «objecttype» may only appear in one transaction.` **Cause:** An executable object declared with fifo behavior appears in more than one transaction. **Remedy:** Specify different behavior or remove executable object from the other transaction(s).
EA170	`Execution profile «name» declared more than once.` **Cause:** The same name has been used for more than one execution profile in the same task. **Remedy:** Make sure all profiles are named uniquely.
EA171	`In execution profile «name», a buffer limit was specified for a task that does not have re-triggering behavior.` **Cause:** Buffer limit declared in a task execution profile but the task is not re-triggering. **Remedy:** Tasks cannot declare a buffer limit if they do not have re-triggering behavior. Either make the task re-triggering or remove the buffer limit.
EA172	`In execution profile «name», a buffer limit was specified for a task declared with ‘fifo’ behavior.` **Cause:** Buffer limits can be specified in only one place: at the re-triggering declaration or within a profile. In this case, buffer limits are given in both places. **Remedy:** Remove either the fifo statement and buffer limit in the re-triggering declaration a task, or remove each buffer limit in if the execution profiles of that task.

Code	Description
EA173	`In execution profile «name», the task does not have ‘fifo’ behavior and therefore must have a buffer limit.` **Cause:** Buffer limit omitted in a task execution profile and the task is not ‘fifo’. **Remedy:** If a task is re-triggering, every profile of the task should have a buffer limit. This either comes from the buffer limit given in the profile, or from a value given in the ‘re-triggering’ declaration. Ensure that every profile does have a buffer limit, or consider defining the task as a simple task (delete the re-triggering declaration).
EA175	`Execution profile «name» declared more than once.` **Cause:** Same name declared for more than one execution profile in the same interrupt. **Remedy:** Make sure all profiles are named uniquely.
EA176	`In execution profile «name», a buffer limit was specified for an interrupt that does not have looping or re-triggering behavior.` **Cause:** Buffer limit declared in an interrupt execution profile but the interrupt is not looping or re-triggering. **Remedy:** If the executable object cannot be given looping or re-triggering behavior, the buffer limit has no meaning to the analysis and should be removed. Alternatively, the execution behavior of the executable object should be changed to re-triggering (in the case of tasks or interrupts) or looping (in the case of interrupts).

Code	Description
EA177	`In execution profile «name», a buffer limit was specified for an interrupt declared with 'fifo' behavior.` **Cause:** Buffer limits can be specified in only one place: at the retriggering/looping declaration or within a profile. In this case, buffer limits are given in both places. **Remedy:** Remove either the fifo statement and buffer limit in the retriggering/looping declaration an interrupt, or remove each buffer limit in if the execution profiles of that task.
EA178	`In execution profile «name», the interrupt does not have 'fifo' behavior and therefore must have a buffer limit.` **Cause:** Buffer limit omitted in an interrupt execution profile and the interrupt is not 'fifo'. **Remedy:** If the executable object is to be looping or retriggering it must have a buffer limit specified either after a fifo clause, or in each profile of the object.
EA179	`Execution profile «name» is used in transactions «name» and «name».` **Cause:** Execution profile appears in more than one transaction. **Remedy:** If the executable object is used by more than one transaction, declare a separate profile for each transaction (even if this results in multiple, identical profiles) and ensure that no profile is common to two or more transactions.

Code	Description
EA180	`In transaction «name», «name» is activated with multiple execution profiles and is not declared as ‘fifo’.` **Cause:** More than one execution profile in the same transaction from an executable object that is not ‘fifo’. **Remedy:** Either remove other execution profiles from transaction or declare the executable object as fifo.
EA183	`One tick on timebase «name» is less than one stopwatch cycle.` **Cause:** Conversion results in value smaller than 1 stopwatch cycle. **Remedy:** Check timebase units definition or change stopwatch conversion.
EA184	`In transaction «name», utilization is over 100% because loop takes 0 cycles.` **Cause:** A transaction was detected which contains a loop with a total analysis delay of 0 cycles, thus causing utilization > 100%. **Remedy:** This can occur if very small values are given in some timebase unit that is rounded down. Check that the ‘next’ path and delay values (in the analysis clauses of the arrivalpoints) are correct. Note that an incorrect timebase unit definition may be the cause of such very small delays.
EA185	`Illegal zero in burst declaration.` **Cause:** Zero in bursting clause. **Remedy:** This can occur if very small values are given in some timebase unit that is rounded down. Ensure that all values used in a bursting clause are non-zero.

Code	Description
EA186	`Task «name» can only be in one soft non-preemption group.` **Cause:** Task declared in more than one soft non-preemption group. **Remedy:** Remove duplicated declarations.
EA187	`Transaction «name» is not activator driven, so must reference an analysis-only timeline.` **Cause:** Timeline must be analysis -only if it is not activator driven **Remedy:** Either specify an activator or change all arrival-points to analysis-only.
EA188	`Illegal zero in stopwatch conversion for timebase «name>.` **Cause:** Zero in stopwatch conversion. **Remedy:** Change zero values in stopwatch conversion.
EA189	`Arrivalpoint «name» is analysis-only and cannot be ‘next’ on timeline.` **Cause:** The ’next’ property of a timeline must specify an arrivalpoint that will result in the generation of a time-point. This means that the specified arrivalpoint must not be an analysis -only arrivalpoint. **Remedy:** The ’next’ property should be changed to specify an arrivalpoint that is not analysis only, or the specified arrivalpoints should have non-analysis properties added to them.
EA190	`No analyzable profiles found in «objecttype» «objectname».` **Cause:** No analyzable profiles found for sensitivity executable object. **Remedy:** Check the profile name specified on the command line and make sure it is present in a transaction.

Code	Description
EA191	`In transaction «name», «name» is not a valid activator.` **Cause:** Invalid activator name. **Remedy:** Check that the activator has been declared.
EA192	`A stopwatch tick may not be smaller than a stopwatch cycle.` **Cause:** Stopwatch cycle unit is declared such that a cycle is longer than a tick. **Remedy:** Stopwatch cycles must be defined no greater than stopwatch ticks.
EA193	`Profile «name» is used in transaction «name» but has undefined timing values.` **Cause:** Attempted analysis of a profile which has a time value declared as undefined. **Remedy:** Specify the correct time value.
EA194	`System timings must be defined for analysis.` **Cause:** Attempted analysis with system timings declared as undefined. **Remedy:** Specify correct system timings values.
EA195	`Interrupt recognition time must be defined for analysis.` **Cause:** Attempted analysis with interrupt recognition time declared as undefined. **Remedy:** Specify a correct interrupt recognition time.

Code	Description
EA196	`Cannot find a task, interrupt or profile called «name» to ignore.` **Cause:** The task, interrupt or profile specified to be ignored in the analysis could not be found. **Remedy:** Make sure the identifier specified has been declared.
EA197	`Initial arrival point «name» for activator «name» may not be analysis-only.` **Cause:** The arrival point must have some non analysis-only data in it, otherwise the arrival point gets optimized away when the system is built. **Remedy:** Make sure the activator's start point is on an arrival point that will be converted in to a timepoint in the target. warnings

12.3 Warning Messages

Code	Description
W0015	`«Message text defined by user»` **Cause:** A (warn ...) preprocessor command was used in the configuration file. **Remedy:** This message was created as a result of preprocessor directives added by the user.
W0018	`Redefinition of macro «name».` **Cause:** The macro named was already defined and has been redefined. The redefinition takes place overriding permanently the older definition. The redefinition may be deliberate. **Remedy:** If the redefinition was deliberate then this warning can be ignored. If it wasn't deliberate then either the first or second definition may be in error and both should be checked.

Code	Description

W0019	`Comment embedded in a comment.`
	
	Cause: The symbols to start a comment were found inside a comment. This might be a mistake causing input not to be commented out that should be. For example, the following input:
	
	`/* some text /* more text */ no longer in a comment now */`
	
	is probably incorrect.
	
	Remedy: Examine the comments to ensure that an attempt has not been made to nest comments in this way.

W0023	`Attempt to undefine non-existent macro called “«name of macro»”` **Cause:** An attempt has been made to undefine a macro that is not defined. **Remedy:** Ensure that the name in the undefine command refers to a defined macro. If the name is generated via a macro expansion or other preprocessor command then use an info command to print the name.
W0403	`Meaningless combination of command line options, some ignored.` **Cause:** The combination of command line options supplied meant that some of them are not acted upon. **Remedy:** Formulate a correct command line by reference to Chapter 11.

Code	Description
W1601	`Conversion of «floating value» to integer causes loss of precision. Converted value is «integer value»; error is «floating value» %.` **Cause:** The number of ticks specified, either directly, or when a delay is given in terms of some user defined units , is converted to a whole number of ticks. If the loss of precision due to rounding amounts to more than 1% of the specified value, then this error is produced. **Remedy:** Modify the value specified so that it evaluates to close to whole number of ticks, or reduce the size of each tick, to increase precision.
W2005	`Multiple locks clauses for "«resourcename»" by "«taskname»".` **Cause:** It is unnecessary to state more than once that a particular resource may be locked by a particular task. Including the same locks clause more than once in a task declaration is equivalent to including it once only. However, the presence of this warning indicates that a mistake may have been made in the declaration of the task (such as a cut and paste error). **Remedy:** Ensure that the resources are named correctly or remove the duplicate locks clause.
W3001	`Resource «resourcename» is not locked by any task.` **Cause:** A resource has been declared which is not specified as being locked by any task. To use the resource in the application program would be an error, it is therefore useless and just consumes memory. **Remedy:** Ensure that this warning is not due to missing locks clauses in task declarations. If not, delete the resource declaration.

Code	Description
W3002	`Resource «resourcename» is locked by only one task.` **Cause:** There is only one task which is specified as locking the resource. A resource that is only ever locked by one task is redundant. It adds to the memory usage, and if the dynamic resource locking calls are used, then it adds to the total code size and to the execution time of the task. **Remedy:** Ensure that this warning is not due to missing locks clauses in task declarations. If not, delete the resource declaration.
W4301	`First entry in timeline is anonymous and therefore cannot be accessed.` **Cause:** A timeline has been specified and the first entry in it has not been named. This means that the first (and any other entries before the first named entry) cannot be used as the initial or repeat entries of an activator, and are hence unreachable. **Remedy:** Name the first arrivalpoint in the timeline.
WA801	`In transaction «name» bursting rate of «value» times in «value» is overridden by bursting rate of «value» times in «value».` **Cause:** Redundant burst description. **Remedy:** Make sure the particular bursting rate is not redundant by accident. See rules on specifying bursting rates in Section 5.9.1.
WA802	`Finish requested by user interrupt.` **Cause:** Finish requested by user interrupt. This is achieved by pressing the keys ESC ‘F’ ‘Y’ in that order. **Remedy:** N/A.

Code	Description
WA803	`No transaction uses execution profile «name».` **Cause:** An execution profile does not appear in any transaction. **Remedy:** Add the execution profile to either an existing or a new transaction.
WA804	`No transaction uses «objecttype» «objectname».` **Cause:** An executable object (other than the idle task) does not appear in any transaction. **Remedy:** Add the executable object to either an existing or a new transaction.
WA805	`In transaction «name» coarse activator driver execution profile «name» should appear in a transaction.` **Cause:** A coarse activator profile does not appear in any transaction.. **Remedy:** Add the driver's execution profile to either an existing or a new transaction.
WA807	`Unable to analyze «name» because busy period exceeds analysis limit.` **Cause:** The maximum value of (release delay+jitter+busy period) exceeds the maximum value. **Remedy:** Check tasks timing values and use sensitivity analysis to review the timing behavior of each task.
WA808	`Too many filenames specified.` **Cause:** More than two non-option parameters supplied on Analysis Visualizer command line. **Remedy:** Ensure that only a single input file is passed to the Analysis Visualizer

Code	Description
WA809	`Transactions «name» and «name» have different release delays but are related through driver «name».` **Cause:** Coarse activator driven transactions have different release delay. **Remedy:** Make sure both transactions have the same release delay specified.
WA810	`Transactions «name» and «name» have different jitters but are related through driver «name».` **Cause:** Coarse activator driven transactions have different jitters. **Remedy:** Make sure both transactions have the same jitter specified.
WA811	`In profile «name», deadline of «value» cycles for critical time of «value» cycles does not exceed deadline «value» cycles for shorter critical time of «value» cycles.` **Cause:** Suspicious deadline specification. **Remedy:** Check that all specified deadlines are correct and the shorter critical execution time really needs to have a longer deadline assigned to.
WA812	`Next non-analysis arrivalpoint unreachable for analysis from arrivalpoint «name».` **Cause:** A non-analysis arrivalpoint can not be reached by following the analysis 'next' clauses. This was detected at the arrivalpoint specified in the message. **Remedy:** Make sure there is a reason for leaving out the non-analysis arrivalpoint. Otherwise, it needs to be added to the analysis route, e.g. add the non-analysis arrivalpoint to the analysis next clause of the reported arrivalpoint.

Code	Description
WA813	`Next non-analysis arrivalpoint unreachable for analysis from arrivalpoint «name» because loop detected.` **Cause:** A non-analysis arrivalpoint can not be reached by following the analysis 'next' clauses because of a 'loop' condition. This was detected at the arrivalpoint specified in the message. **Remedy:** Make sure there is a reason for leaving out the non-analysis arrivalpoint. Otherwise, it needs to be added to the analysis route, e.g. break the existing analysis-loop by adding the non- analysis arrivalpoint to the analysis-next clauses of the last arrivalpoint of the loop.
WA814	`For arrivalpoint «name», cumulative analysis delay to next non- analysis arrivalpoint of «value» cycles exceeds actual delay to next non-analysis arrivalpoint of «value» cycles.` **Cause:** The cumulated delay of the analysis route is longer than the actual non-analysis delay. **Remedy:** Make sure there is a reason for a longer cumulated analysis delay. Otherwise, reduce the cumulated analysis delay or increase the main non-analysis delay value of the arrivalpoint.
WA815	`First non-analysis-only arrivalpoint on timeline is anonymous and therefore cannot be accessed on the target.` **Cause:** First non-analysis arrivalpoint in timeline has not been named. **Remedy:** Assign a unique identifier to the first arrivalpoint on the timeline which is not non-analysis.

Code	Description
WA816	`Analysis-only arrivalpoint declared (before first named arrival point on a timeline | later than arrivalpoint «name») is unreachable.` **Cause:** A specified analysis -only arrivalpoint in a timeline is unreachable. **Remedy:** Make sure the arrivalpoint can be reached by “following” the ‘next’ clause of each arrivalpoint.
WA817	`In transaction «name», «name» is activated and it does not have ‘simple’ behavior.` **Cause:** An executable object which is not analysis-only and has not been declared as simple is released in an activator-driven timeline. **Remedy:** Check that the executable object has the desired behavior and the declaration is correct. WB100 Cannot delete file “<name>”.

12.4 Information Messages

Code	Description
I0012	`«Message text defined by user»` **Cause:** An (info ..) preprocessor command was used in the configuration file. **Remedy:** N/A
I0405	`Errors detected during parsing.  No output files will be generated.` **Cause:** Errors were detected in the input language or in initial processing of data derived from this. Check the error output of the Analysis Visualizer for more details on the errors that were found. **Remedy:** N/A

Code	Description
I0406	`Errors detected during processing. No output files will be generated.` **Cause:** Errors were detected in one of the Analysis Visualizer's internal phases. Check the output of the Analysis Visualizer for more details on the errors that were found. **Remedy:** N/A
I0407	`Errors detected during consistency checking. No output files will be generated.` **Cause:** Errors were detected in the consistency checking phase where global integrity is checked. Check the error output of the Analysis Visualizer for more details on the errors that were found. **Remedy:** N/A
I0408	`Warnings were generated.` **Cause:** Warnings were generated during processing of the configuration file. This is merely a summary. **Remedy:** N/A
I4106	`Modulus for timebase "«name»" evaluates to «length» ticks.` **Cause:** As the timebase modulus can be specified in terms of arbitrary units this message provides confirmation of the resolved value. **Remedy:** N/A

13 Finding out more

Your RTA-OS3.x distribution includes the following manuals:

`<install dir>\Documents`

***Getting Started Guide*.** This guide explains how to install the product and describes the underlying principles of the operating system.

***Release Note*.** This document provides information about the release, including a list of changes from previous releases and a list of known issues.

***User Guide*.** This guide explains the concepts behind AUTOSAR OS R3.x and shows you how to use RTA-OS3.x to configure the OS and integrate it into your application

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

`<install dir>\Targets\VRTA_n.n.n`

***VRTA Port Guide*.** This guide explains implementation-specific details for the VRTA port plug-in.

***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.

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

`<install dir>\Targets\<TargetCompiler>_n.n.n`

***Target/Compiler Port Guide*.** Each port of RTA-OS3.x is supplied with a Port Guide. The Port Guide tells you specific information about the interaction between RTA-OS3.x, your toolchain and your target hardware. For example, valid compiler options, register settings, interrupt handling etc. The Port Guide also gives performance and resource usage information for the OS.

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

14 Glossary

activate To make a task or a taskset ready to execute.

activator An object which processes arrivalpoints. When an activator processes an arrivalpoint, it activates all the tasks associated with that arrivalpoint. The next and delay properties of the processed arrivalpoint indicate to the activator which arrivalpoint to process next and after what delay

active priority The current priority of a task which is set and updated by RTA-OS3.x. A task can only be preempted by tasks that have a base priority which is higher than its active priority. See also dispatch priority.

analysis-only arrivalpoint An arrivalpoint that contains only analysis properties. The Analysis Visualizer may use it for analysis.

arrivalpoint An arrivalpoint is used to model the arrival of tasks and is referenced by a transaction. When used for analysis, an arrivalpoint indicates the arrival time of specific execution profiles. The arrivalpoint also indicates the next arrivalpoint in the transaction and the time until that arrivalpoint. A variation also exists that is suitable only for analysis purposes (an analysis only arrivalpoint).

arrival time The arrival time describes the occurrence of a real-world event. This event is typically one that results in the release of an executable object. Examples of such events include switch closure or expiry of a time interval.

autoactivate tasks A collection of tasks which are made ready to execute when RTA-OS3.x is started.

autostart at The absolute counter value at which a coarse activator processes its first arrivalpoint.

base priority The priority at which a task, that has not yet started executing, competes for the processor. See also active priority, dispatch priority.

behavior An executable object is defined as having simple , looping or re-triggering behavior. This defines what happens if an object is released before it has completed the execution arising from a previous release. If an executable object is looping or re-triggering, it can also be declared as fifo.

blocking Blocking occurs when an executable object is delayed from executing by another executable object with lower base priority. This can occur for a variety of reasons. Instances of blocking can occur when a lower priority executable object raises its active priority by locking a

resource , or if it explicitly raises the interrupt priority. It can also occur when one of the lower priority tasks in a non-preemption group is executing (higher priority tasks within that group are blocked).

build level There are three build levels of RTA-OS3.x: standard, timing and extended.

bursting The arrival times of executable objects may be described as bursting within a transaction. This allows sporadic arrival times to be modeled for analysis.

coarse activator A coarse activator has a counter embedded in it and requires a driver that tells it when each tick occurs. The activator updates the counter itself. The counter always counts ticks from 0 to 65535, so the activator must be bound to a timebase that has a modulus of 65536.

configuration file An input file to the Analysis Visualizer that defines the RTA-OS3.x objects contained within the application system. It also describes the behavior of the system to allow schedulability analysis.

controlled interrupt An interrupt that is handled by RTA-OS3.x, which then calls the associated ISR entry function.

critical event Critical events are points in the processing of an executable object that must be performed within a specific time (given by a deadline) after arrival. For an executable object with simple behavior, the termination of that object is (implicitly) considered by the analysis to be a critical event.

critical execution time Critical execution time starts at the entry point of an executable object and ends when that executable object executes a critical event. Critical execution time considers only the time spent executing by the given executable object (i.e. excluding any time taken up by interruption or preemption).

deadline The deadline is the maximum allowable amount of time between the arrival time and the execution of the associated critical event.

deadlock Deadlock occurs when two or more tasks require a set of resources, each task acquires one or more resources but none can acquire all the resources it needs because some are held by other tasks. In this situation, no task can proceed. The priority ceiling protocol used by RTA-OS3.x ensures that no use of resources can result in deadlock.

delay This is a property of an arrivalpoint. The delay specifies the time until the next arrivalpoint within a given transaction. The units must be in ticks or a unit defined within the associated timebase.

dispatch priority The priority of a task when it starts executing. This value will always be greater than or equal to its base priority. See also active priority.

driver callbacks User supplied functions that enable a fine activator to access its supporting hardware (typically a timer/counter).

error messages the Analysis Visualizer outputs these messages when it detects conflicts in the configuration file that make it impossible for any resulting application program to link and run correctly or for schedulability analysis to be performed. the Analysis Visualizer attempts to report all the errors it can find and then exits. All errors should be removed and the activity repeated before continuing.

executable object An executable object is a task, ISR or uncontrolled interrupt handler whose characteristics are described in the configuration file. Although tasks, ISRs and uncontrolled interrupts have their own distinct behaviors and restrictions, it is possible to distill the features of all of these into a single 'executable object' so that the analysis can treat them all equivalently.

execution budget Used by the timing build to define a maximum execution time for a task or interrupt service task (ISR). If exceeded, the overrun hook is called.

execution profile An execution profile contains the timing characteristics of an executable object. An executable object can have more than one execution profile, so it is possible to analyze tasks or interrupts that execute different sections of code on different invocations.

execution time The execution time of a task or ISR is the amount of time from the start of the first instruction in the entry function (the entry point) to the end of the 'return' instruction. The execution time of an uncontrolled interrupt handler, is the amount of time from the start of the first instruction in the interrupt handler function to the end of the 'return from interrupt' instruction. Execution time is measured excluding any time taken up by interruption or preemption.

fatal messages the Analysis Visualizer outputs these messages when it detects a condition in the configuration file from which recovery is not possible.

fifo Executable objects with re-triggering or looping behavior can be declared as fifo (first-in-firstout), in which case their arrivals are processed in the order in which they occur. If the object is not declared as fifo, arrivals are processed in a priority order.

fine activator A fine activator makes use of an external, typically hardware, counter and requires a driver that tells the activator when a requested number of ticks have occurred. The activator uses special callback functions provided by the user to access and control the counter hardware.

idle task The lowest priority task in an RTA-OS3.x application. The idle task has a base and dispatch priority of 0 and is always present.

indeterminate schedulability An executable object's schedulability is indeterminate if the analysis can not decide whether the object is schedulable or unschedulable. This can occur for example if a lower priority looping executable object is unschedulable, so a correct blocking time cannot be calculated.

information messages the Analysis Visualizer outputs these messages which report useful information such as the amount of memory used, or the size of a data structure.

interference Interference occurs when an executable object is delayed from executing by a higher priority executable object.

interrupt priority This specifies the processor interrupt priority level at which an executable object will execute. Interrupt priority 0 is reserved for tasks and corresponds to all interrupts enabled. Higher numbers are assigned to controlled and uncontrolled interrupts and correspond to higher priorities up to some target dependent limit. The priority of any uncontrolled interrupt must not be lower than that of any controlled interrupt.

ISR (interrupt service routine) With controlled interrupts, the vector points to internal RTA-OS3.x code. RTA-OS3.x then treats the interrupt as an invocation of an interrupt service task (or ISR) bound to the interrupt. Just as for a normal task, execution starts at the specified entry point of the ISR and continues until the entry function returns.

jitter The jitter of an executable object is the difference between the minimum and the maximum delay between its arrival time and its release time. Jitter is used for schedulability analysis.

LCM The Lowest Common Multiple of the periods of a set of tasks.

looping An executable object with looping behavior is able to cope with being released again before it completes by looping within its entry function. A task cannot be declared with looping behavior

modulus The modulus of the counter represented by a timebase. The modulus must evaluate to an integer in the range 2 to 65536 ticks. The modulus is one more than the maximum value which the counter may take.

namespace RTA-OS3.x internal names comply with a simple naming convention. If you avoid names which begin with os or OS, then there will be no conflicts with the RTA-OS3.x namespace.

non-preemption group A collection of tasks that execute in mutual exclusion (will not preempt each other). The stack usage of such tasks is effectively overlaid.

notional release time The notional release time of an executable object is the earliest time at which it can be considered to have been released. This may be prior to the actual release time for the object if higher priority executable objects always occupy the period between notional and actual release time.

not-schedulable An analysis item is not schedulable if it is unschedulable, or if its schedulability is indeterminate or unknown.

offset In a periodic timeline , the activation of a task may be specified at an offset into its period.

one-shot Tasks provided by RTA-OS3.x are one-shot tasks: a task is made ready at some point, it starts executing from its entry point, perhaps being preempted by other higher priority tasks or interrupts during its execution, and then terminates. The task can be made ready again later, and the task can execute again.

OS level The highest interrupt priority level of any controlled interrupt. OS level is the processor interrupt priority level necessary to ensure mutual exclusion with any RTA-OS3.x operation.

overrun hook A user provided function that the timing build of RTA-OS3.x calls when it detects that tasks or ISRs have exceeded their execution budget.

periodic timeline A periodic timeline is declared by specifying a number of tasks with fixed periods and offsets. the Analysis Visualizer works out the minimum length of sequential timeline required to form a cyclic timeline that can be used to activate the tasks with the periods and offsets specified.

preemptive RTA-OS3.x is a preemptive operating system because it will stop executing the current task (i.e. preempt it) and run a new, higher priority task, returning later to the original task at the point of preemption.

preprocessor Part of the Analysis Visualizer that does a similar job to the C preprocessor. It lets you define symbols and include files inside other files.

priority Task priority governs the order in which tasks are executed when they become ready to execute at the same time. The highest priority task is the one that will be run first. Task priority is determined from the task priority order specified in the configuration file. Not to be confused with interrupt priority.

priority ceiling protocol The resource locking protocol used by RTA-OS3.x.

priority inversion Priority inversion describes the situation where a high priority task is ready to execute, but for some reason is unable to run. A lower priority task executes instead, in an inverse of the desired behavior.

readonly A property of an RTA-OS3.x object that indicates that it may not be modified by API calls and is therefore placed in ROM.

ready The state of a task that wants access to the processor but is not currently running.

release An executable object is released when it is first made ready to run. In the case of a task, it is released when it enters the ready state. In the case of an interrupt service task, it is released when the CPU has recognized the interrupt that causes it.

release delay The release delay of an executable object is the minimum delay between its arrival time and its release time.

release time The time at which an executable object is released. This occurs at or after the object's arrival time. The duration between arrival time and release time ranges between release delay and release delay + jitter.

resource Resources are the RTA-OS3.x objects used to guard access to data or devices shared between tasks, access to which must be in mutual exclusion. response time For an executable object, this is the actual amount of time between the arrival point and execution of some critical event. See also worst case response time.

re-triggering An executable object with re-triggering behavior copes with being released again before completion by ensuring that it gets run again when it completes. In the case of a task, it can chain back to itself on completion. A re-triggering interrupt handler can ensure that the interrupt stays pending.

schedulability analysis Mathematical analysis of task and interrupt timing behavior that can be used to predict the worst case time from an executable object being released until it terminates. For each executable object, schedulability analysis determines whether the executable object is schedulable , unschedulable or of indeterminate schedulability.

schedulable An executable object is termed schedulable if the worst case response time for each of its critical events is no greater than the corresponding deadline. Objects with simple behavior must always complete execution before their next release. Looping or re-triggering objects must not exceed their buffer limits.

sequential timeline An ordered collection of arrivalpoints declared in the configuration file. These are processed in order by an activator. Once these arrivalpoints have been processed, the activator may be stopped (if the time line is 'single shot') or it may continue running and process another timeline.

serially reentrant A serially reentrant function is one where it is OK to switch from a thread of control currently executing the function to a thread of control that is not yet executing the function but will do so later.

simple A simple executable object must complete its execution before it is next released. A simple executable object is unschedulable if it does not always complete before its next release.

static interface Some RTA-OS3.x API calls have static and dynamic interface versions. The static interface calls can be faster as they take advantage of some static property to optimize the operation. However, to use the static interface , the RTA-OS3.x object that forms the target of the call and the task or ISR that makes the call must be known at compile time.

stopwatch Refers to the timebase which is used to provide execution time measurement.

suspended The state of a task which is not ready to execute.

task An independent thread of control. (See also one-shot).

terminate A running task returns to the suspended state by terminating. It simply returns from its entry function.

tick A single increment of the counter represented by a timebase.

timebase An RTA-OS3.x object that defines the units, granularity and range of a counter. A timebase may correspond to a regular counter that marks the passing of time (for example, a millisecond timer), or to some other kind of application-specific counter (for example a count of the number of times a tooth on a toothed wheel passes in front of a sensor). A counter increment is referred to as a tick.

timeline A collection of arrivalpoints that are used to represent a transaction that will be processed in order by an activator. A timeline may be periodic or sequential.

transaction A transaction specifies the timing relationships between executable objects using a timeline or bursting clause. These executable objects inherit the same jitter and release delay specified in the transaction.

uncontrolled interrupt An uncontrolled interrupt is outside of the domain of RTA-OS3.x and must be handled directly by application code.

units User defined timebase units that are equivalent to some number of ticks. unknown schedulability An executable object's schedulability is unknown if no analysis has been done.

unschedulable The term unschedulable is used to describe an executable object that has been analyzed and found to exceed constraints on a deadline, inter-arrival time or buffer limit.

user level The processor interrupt priority level corresponding to all interrupts enabled. All tasks begin executing at user level.

vector The address that the program counter is set to when the associated interrupt is taken.

version The version number of RTA-OS3.x is used to patrol compatibility between the Analysis Visualizer, configuration file and RTA-OS3.x library.

warning messages the Analysis Visualizer outputs these messages when the configuration file specifies something that is redundant or a value that cannot be represented precisely and is therefore subject to rounding error. When warnings have been produced the output files are created and the application can be built. However, as the Analysis Visualizer has changed some values, the application may not behave exactly as expected.

worst case response time Worst case response time (WCRT) is the longest possible time between the arrival point and execution of some critical event.

15 Contacting ETAS

15.1 Technical Support

Technical support is available to all users with a valid support contract. If you do not have a valid support contract, please contact your regional sales office (see Section 15.2.2).

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

rta.hotline.uk@etas.com

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

+44 (0)1904 562624.

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

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

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

15.2 General Enquiries

15.2.1 ETAS Global Headquarters

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

15.2.2 ETAS Local Sales & Support Offices

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

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

Index

A

Activator
- Configuration, 212

Activators, 61
- Coarse, 61
- Fine, 61

Alarms, 69
Arbitration Order, 216
Arrivalpoint, 55
- Analysis Overrides, 57

Auto-started Tasks, 79

B

Blocking, 21, 48
- Standard Resources, 48

Buffering, 88
- Exceeding Limits, 118
- Looping, 92
- Re-triggering, 92

Buffering by profile, 91
Bursting Transactions, 65
Busy Period, 122

C

Clock Optimization, 11, 136
Command Line, 233
- Options, 233

Configuration
- Pre-processor, 221

Configuration Language, 190
Critical Execution, 83
Critical Execution Time, 126

D

Deadlines, 14
- Arbitrary, 83
- Difficulties in testing, 14
- Implicit, 81
- Not able to be met, 115

E

Error Codes, 236
- Errors, 249
- Fatal, 237
- Information, 293
- Warnings, 286

Execution Time, 18
Extended Tasks, 98

I

Idle Mechanism, 47, 80
- Execution Time Sensitivity, 131

Implicit Deadlines, 81
Indeterminate Schedulability, 122
Interference, 18
Interrupts, 46
- Configuration, 205
- Masking, 50
- Recognition time, 218

J

Jitter, 100

K

Kernel, 34, 194
Kernel Declaration, 27

M

Modeling Process, 12
Multiple Profiles
- Interrupts, 93
- When to use, 85

N

Non-preemption Group
- Configuration, 204

O

Offsets, 58
Operating System Constraints, 22
Optimism, 24
OS, 34
OS Overheads, 103

P

Period, 17
Periodic Timeline, 210
Periodic Timelines
- Multiple Offsets, 58

Periodic timelines, 57
Pessimism, 24
Pre-processor, 221
    Compatibility with C, 224
Priority Allocation, *see* Priority Optimization
Priority Constraints
    Configuration, 203
Priority Optimization, 11, 133
    Constraints, 135, 136
Profiles
    Configuration, 193
    Multiple, 85
    Restrictions on, 88

Q

Queueing, 88

R

Real-Time Systems, 14
Release Delay, 100
Reserved Words, 219
Resource
    Configuration, 195
Resources, 87
    Internal, 50
    Standard, 48
Response Delay, 102
Response Time, 18

S

Schedulability Analysis, 11, 15, 111
    Deadline Monotonic Analysis, 16
    Limitations, 22
    Rate Monotonic Analysis, 16
    Utilization Tests, 15
Schedule Tables, 74
Sensitivity
    Clock Speed, 131
    Critical Execution Time, 126
    Execution Time, 128
    Lock Time, 128
Sensitivity Analysis, 11, 125
Single Shot Timelines, 59
STC file, 12
Stopwatch, 36
System Timings, 217

T

Tasks
    Activation, 44
    Co-operative, 95
    Configuration, 199
    Extended, 98
    Priority Order, 44, 219
    Queued Activation, 89
    Specifying an activation, 136
Timebases, 35
    Configuration, 196
    Conversion, 198
    Non-periodic, 108
    Non-time-based, 38
    Stopwatch, 36
    Stopwatch conversions, 38
Timeline
    Configuration, 208
    Transactions, 62
Timelines, 55
    Periodic, 57
    Single Shot, 59
Transactions, 51
    Analysis Only, 105
    Bursting, 65
    Configuration, 213
    Restrictions on, 88
    Timeline, 62
Tutorials, 140

U

Units, 192
Unschedulable Objects, 113
Utilization, 15
    More that 100%, 120

W

Worst case execution time, *see* Execution Time