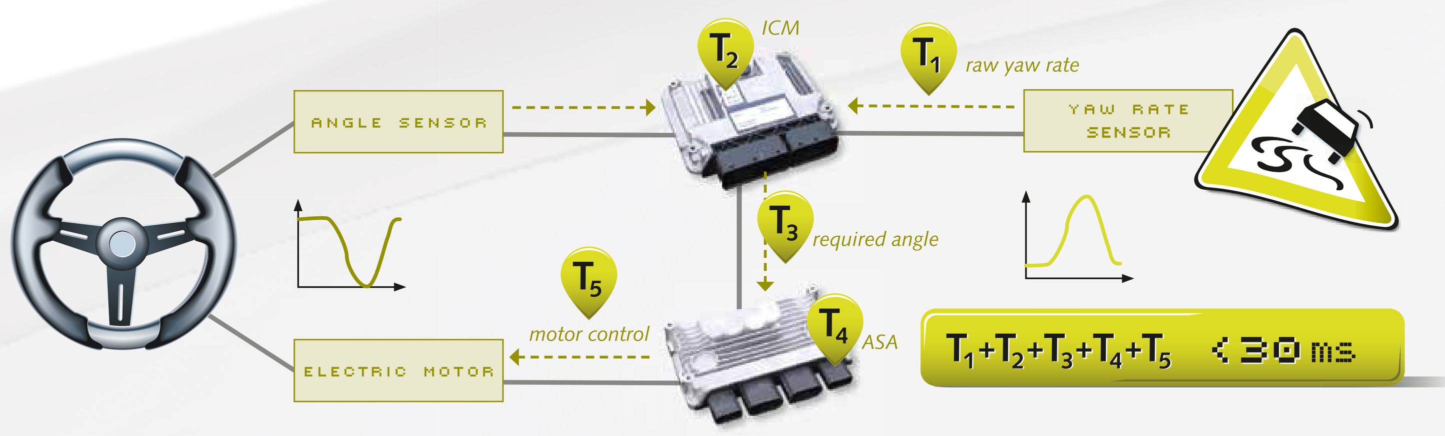
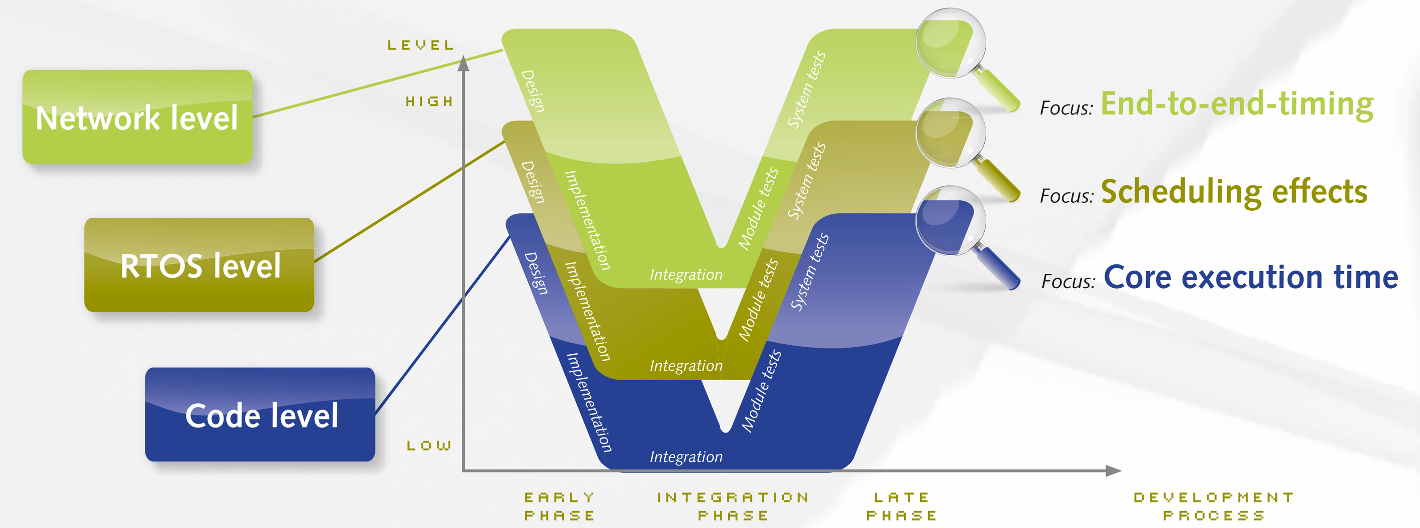
01 WHAT IS EMBEDDED TIMING AND TIMING ANALYSIS?



The active steering shown in the figure demonstrates what embedded timing is about. The system consists of sensors, ECUs, busses and an actuator. With the vehicle dynamics model of the car and the active steering function on his mind, the functional developer defined a minimum reaction time for the complete chain, here 30ms. This becomes a top level (= network-level) end-to-end timing requirement for the system. This timing requirement then gets decomposed, i.e. it gets sliced into smaller portions $T_1 \ldots T_5$, one portion for each component of the system. Obviously, ECUs and busses handle many more features with many more timing requirements, all competing for network and computation resources. On an ECU with tasks/interrupts and their runnables, the top level timing requirements are broken down into more fine grained timing requirements and the competition for resources is continued on a lower level.

Timing analysis helps planning, understanding, optimizing and securing embedded systems with respect to their timing. This poster sheds a light on the various aspects of embedded timing and timing analysis techniques.

02 TIMING ANALYSIS: LEVELS AND DEVELOPMENT PHASES



Any timing-related activity, problem or use-case can be placed in a diagram with two dimensions: the level and the development phase.

NETWORK LEVEL, RTOS LEVEL, CODE LEVEL

The **Network level** deals with inter ECU communication aspects and **end-to-end-timing**. Most Network level timing experts are found at the OEMs; they integrate several ECUs connected to various networks in one E/E platform for one vehicle.

The **RTOS level** considers only one scheduling entity (e.g. an ECU with one single core processor) and focuses on **scheduling effects**. Most RTOS level timing experts are found at the tier-1s; the tier-1 typically integrates all software components into the ECU and also configures the operating system.

The **Code level** focuses on a fragment of code (e.g. a single function) independently of scheduling. The **(core) execution time** is the most important code level result.

EMBEDDED TIMING IN THE DEVELOPMENT PROCESS

Use cases are very different depending on where in the development process they are located. In a very early phase of a project for example, most of the source code is not available yet, making it impossible to measure, trace or perform static code analysis. The key tasks for timing analysis in the various phases are:

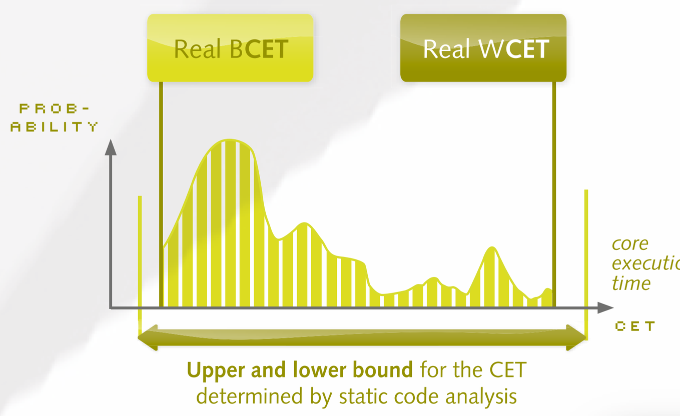
- **early phase**: determine timing requirements and design a timing layout which fulfils the requirements; define appropriate hardware (e.g. select processor); start implementation
- **integration phase**: finalize implementation; integrate components into a working environment; debug and optimize timing; measure timing and relate results to requirements; validate models
- **late phase**: measure and supervise timing; perform timing tests (can be done in parallel to functional tests); use model based approaches to cover corner cases and perform formal verification

03 USE-CASES AND TIMING PROBLEMS

USE CASE OR PROBLEM [2]	POSSIBLE SOLUTION
Sporadic system crashes	• *Use a trace solution that allows "post mortem" analysis* • *Use scheduling analysis or scheduling simulation for reconstructing the problem. However, the cause of sporadic system crashes are typically unknown so that it is unlikely to be present in the model.*
Sporadic data inconsistencies	*See "Sporadic system crashes". Differences: a) "post mortem" capability not necessary and b) choose a trace solution that allows a joint view of data accesses and scheduling events*
Model validation	• *Use scheduling tracing for scheduling analysis or scheduling simulation model verification* • *Use flow tracing for static code analysis or model verification*
Timing profiling (execution times, CPU-load, etc.)	*On network level/RTOS level:* • *Use scheduling analysis or scheduling simulation (especially in early phases)* • *As soon as code is executable, use tracing/measurement, ideally with on-target supervision* *On code level:* • *Use static code analysis for WCET determination* • *Use tracing/measurement, ideally with on-target supervision*
Analyze/optimize scheduling	*Use scheduling analysis or scheduling simulation and check results using tracing/measurement*
Code optimization (for speed)	• *Use tracing and/or scheduling analysis or scheduling simulation to find hot-spots. Very important: do not optimize code that does not cause a hot-spot!* • *Perform code reviews: look at generated/hand-written C code and for smaller fragments at generated machine code* • *use static code analysis and/or more detailed tracing for monitoring the results of the code optimization*
Optimize speed by optimizing memory usage	*Map frequently used symbols (code or data) to fast memories. Do not rely on your gut feeling when judging what symbols could be "frequently used" but use tracing/measurement*
Design space exploration	*Use scheduling analysis or scheduling simulation*
Analyze timing behavior of future SW versions	• *Inject additional load into existing software and trace/measure* • *Use scheduling analysis or scheduling simulation*
Select best CPU	*Take representative code (fragments of existing software or an automotive benchmark) and* • *use static code analysis or* • *measurement/tracing* *for benchmarking*
Design timing in the early development phase	*Use scheduling analysis or scheduling simulation*
Verify timing	• *In an early phase, use scheduling analysis or scheduling simulation* • *In a late phase, use tracing/measurement ideally with a) on-target supervision and b) corner case analysis (see next use-case)*
Corner case analysis	• *On network level/RTOS level: use scheduling analysis or scheduling simulation* • *On code level: use static code analysis*
Multicore timing analysis	*Multicore has a big impact on the RTOS level analysis. Thus, the corresponding tools explicitly need to support multicore.*
Multicore load balancing (static task allocation)	• *If a running system is available, use tracing to understand/profile it* • *See "Design space exploration"*

04 TIMING ANALYSIS TECHNIQUES

TIMING ANALYSIS TECHNIQUES [3]	INPUT (DATA)	INPUT (MODEL) OR MECHANISM	MAIN OUTPUT OR USE
Static code analysis	*Source code and/or binary*	*Processor model*	*Guaranteed BCET/WCET*
Code simulation	*Binary*	*Processor model*	*CET according to test case*
Tracing/Measurement	*Instrumented SW or probed HW*	*Events are logged into a trace buffer*	*Timing information according to test case*
Scheduling simulation	*CETs, application model (scheduler configuration)*	*Scheduler model*	*WCRT*
Scheduling analysis	*BCET/WCET, application model (scheduler configuration)*	*Scheduler model*	*Guaranteed WCRT*



STATIC CODE ANALYSIS

Static code analysis timing tools read the source code and/or the binary code of an application or part of it. They calculate a lower limit for the BCET and the upper bound for the WCET for a given code fragment, e.g. a function. Any real core execution time is guaranteed to be within this interval, as long as this fragment is not interrupted. Any data present only at run-time (e.g. upper bounds on the loop iterations and the content of dynamic function pointers) has to be provided manually in the form of additional annotations.

CODE SIMULATION

Code simulators simulate the execution of given binary code for a certain processor. A wide variety of code simulators exist. Simple instruction set simulators provide very limited information about the execution time whereas complex simulators consider also pipeline- and cache-effects. To achieve reliable WCET information from a code simulator, it has to be embedded into a test environment which actually causes the worst case to be simulated.

SCHEDULING ANALYSIS

Based on the model of a certain scheduler (e.g. a certain RTOS), scheduling analysis tools take minimum/maximum core execution times and an application model as input and provide e.g. the guaranteed WCRT. This allows checking whether any deadlines will be missed under the given conditions. For each task's or interrupt's worst case, a trace is generated allowing to analyze the run time situation under which it occurs. The execution times fed into the analysis can be either budgets, estimations, or outputs from other tools, e.g. statically analyzed BCET/WCET or traced/measured data [4].

MEASUREMENT

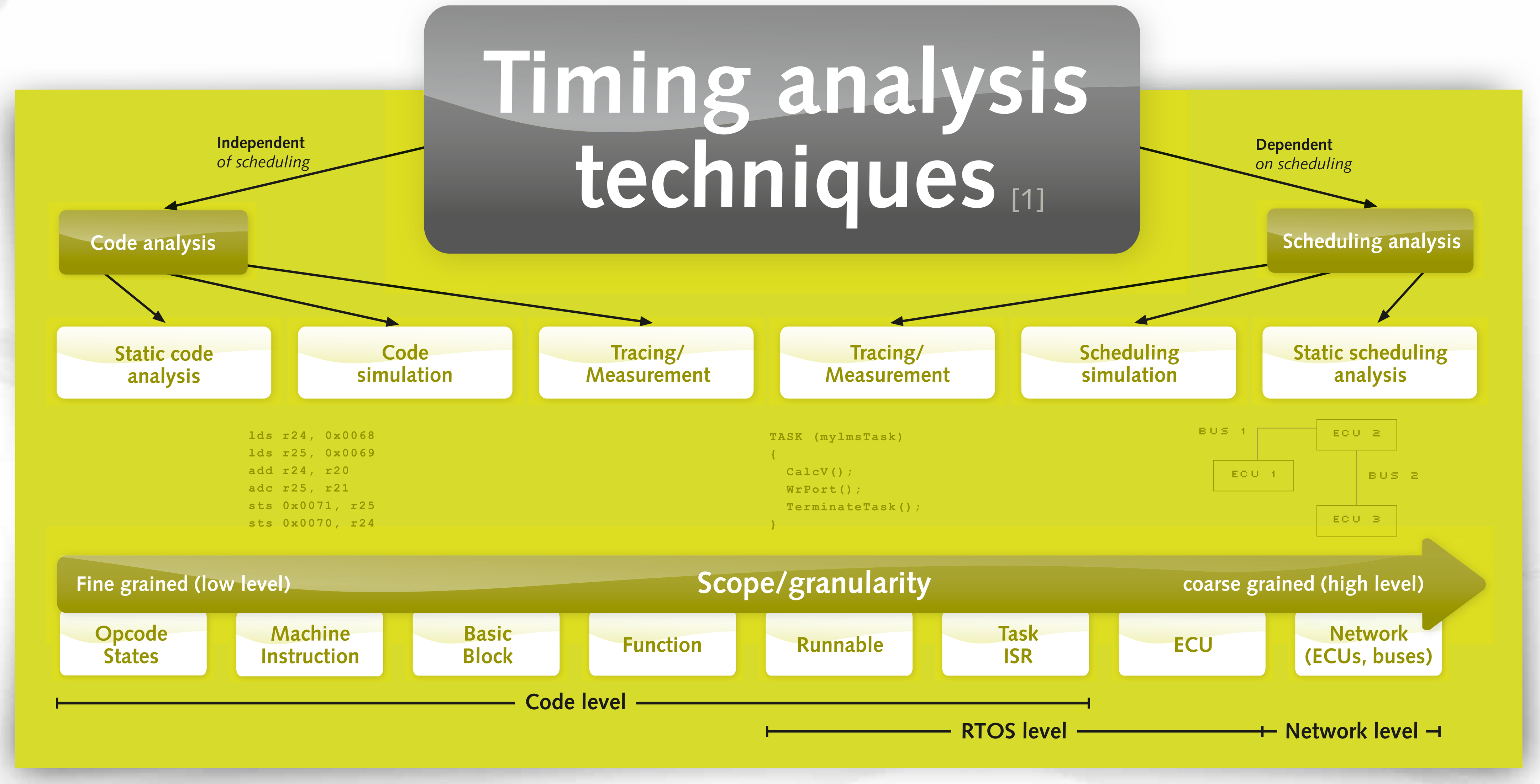
The real system is analyzed and the observed timing information is provided. Timing measurement is often based on hook routines which are invoked by the RTOS.

TRACING

Tracing observes the real system. For dedicated events, time stamps together with event information is placed in a trace buffer. The selection of events can be very fine grained like for flow traces which allow reconstructing the sequence of each machine instruction or coarse grained like when tracing scheduling related events only. Tracing can be based on instrumentation (i.e. software modification) or on special tracing hardware. Traces can be visualized and analyzed offline, e.g. for debugging purposes. All kinds of timing information can be extracted from a trace [5].

SCHEDULING SIMULATION

Scheduling simulators provide similar functionality as the scheduling analysis. Instead of calculating the results, they simulate run time behavior. The scheduling information and generated traces are the main output. If the worst case scenarios are simulated, the observed response times will equal the WCRTs. Some simulators allow Task definitions in C language so that complex applications models are supported while offering a specification language well known to automotive engineers.



05 STANDARDS

AUTOSAR TIMING EXTENSIONS

With AUTOSAR V4.0, the Timing Extensions have been introduced allowing the definition of timing requirements. In the first step, events like "start of runnable R" or "reception of data D" are defined. In a second step, requirements related to the events are formulated. Example 1: After the start of runnable A, the reception of data D must not occur later than 2.5ms. Example 2: events E1, E2, E4, E7 must always occur in this order [6].

OT1

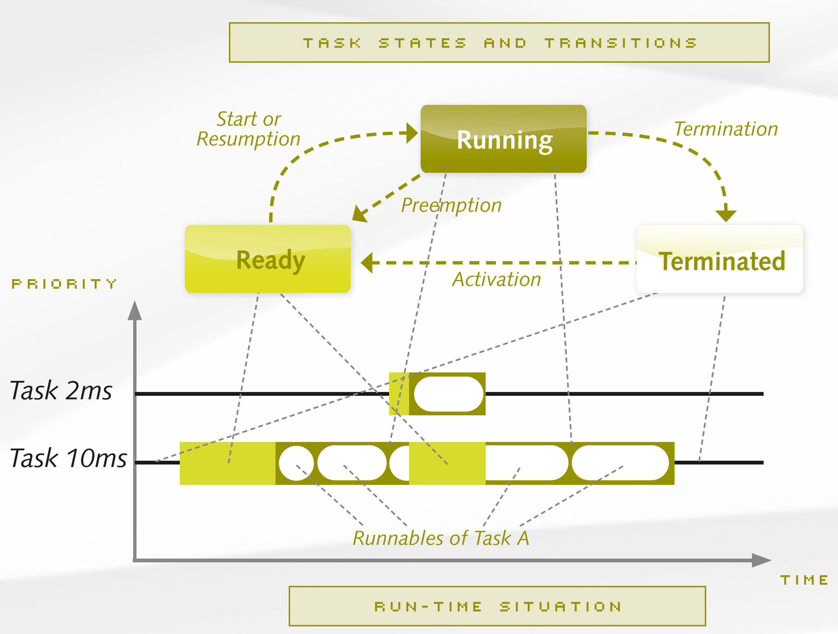
As of early 2013, there is no standard for timing information interchange between timing tools. OT1 is a new and unified data exchange format which we propose to be used by all kinds of timing related tools. OT1 comes as an XML format and allows the exchange of:

- **System configuration** (tasks, priorities, runnables, etc.)
- **Traces** (log of e.g. scheduling related events)
- **Timing information** (core execution time, response times, etc.), also referred to as "timing guarantees"
- **Timing requirements** (e.g. max. allowed response times)

All timing information related to a project is held in one big OT1 container and any related timing tool can retrieve and/or provide information. It is even possible to request absent information. A scheduling analysis tool e.g. can request the CET of a certain runnable and this request can be either answered by a tracing tool or a static code analysis tool. Since all information is tagged with its source, the handling of diverse sources for the same kind of data becomes very easy.

See *www.gliwa.com/ot1* for more details.

06 FROM SWCS TO RUNNABLES TO TASKS TO EXECUTION



AUTOSAR software components (**SWC**) encapsulate a defined functionality, e.g. idle speed control of an engine management ECU. A SWC is implemented with **runnables** which have certain scheduling, safety and timing requirements. The idle speed control e.g. could be coded in three runnables: IdleSpeedInit, IdleSpeed10ms and IdleSpeed50ms. As part of the RTOS configuration, all runnables get mapped to tasks or interrupts which match their requirements. In the run-time situation shown in the figure on the left, Task10ms holds four runnables. So runnable IdleSpeed10ms has to share its container with three runnables from other SWCs. At runtime, the RTOS schedules tasks and interrupts according to their attributes (most important: period and priority).

AUTOSAR
GLIWA GmbH is an AUTOSAR development member

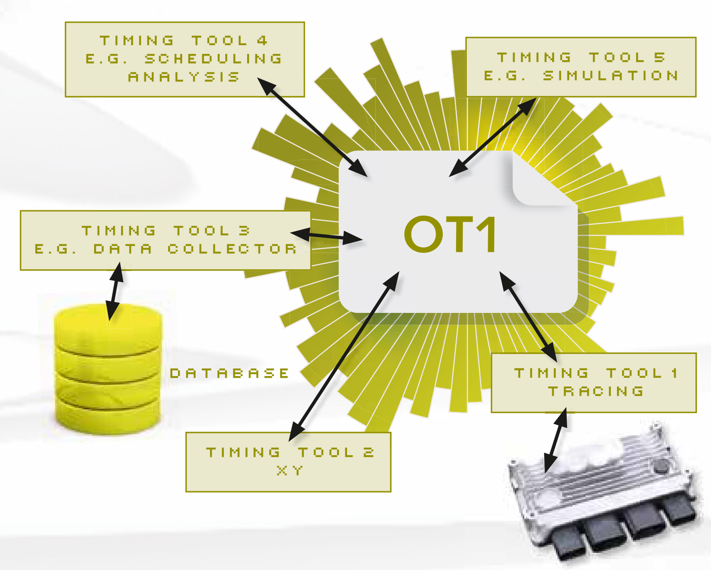
07 CPU-LOAD / BUS-LOAD

CPU-load and bus-load are the most important characteristics of timing. They compress the complex timing issue into one single number which is perfect for management reports. However, they are too simple to capture all scheduling characteristics, e.g. an ECU with 40% CPU-load can still easily violate timing constraints.

When stating CPU-load and bus-load, it has to be made clear how these are defined:

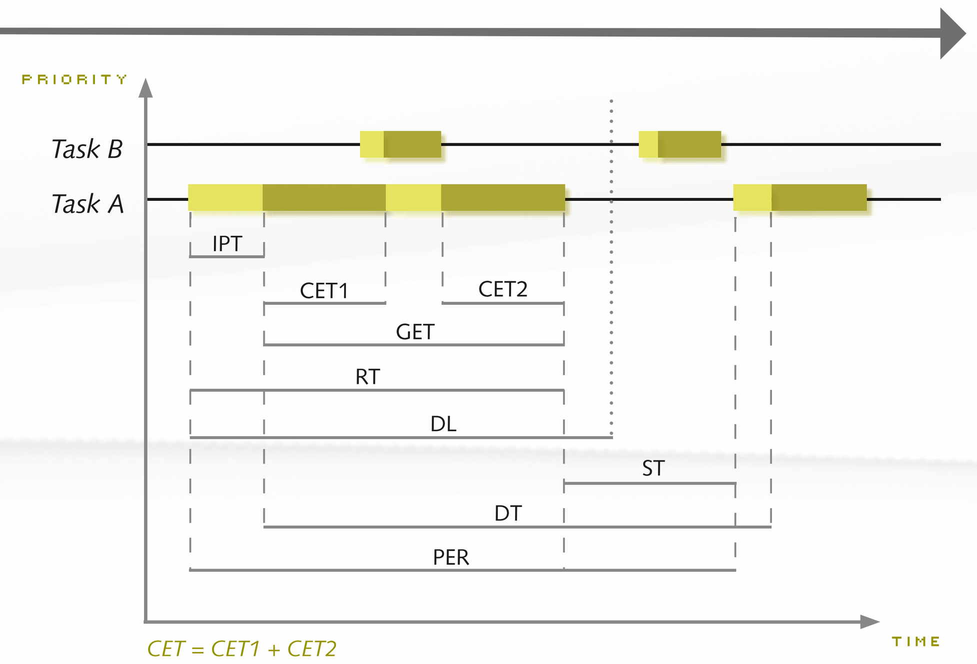
- What is the reference time frame?
- How is the background task considered (if present)?
- Is the RTOS overhead considered correctly?

Most importantly, the CPU-load / bus-load cannot replace a full set of detailed timing requirements.



08 GLOSSARY

ABBR.	EXPANSION
IPT	*initial pending time*
CET	*core execution time*
GET	*gross execution time*
RT	*response time*
DL	*deadline*
DT	*delta time*
ST	*slack time*
PER	*period*



TERM OR ABBREVIATION	MEANING
ASIL	*Automotive SIL*
Background task	*Application code which gets executed when no other task or interrupt is pending*
BCET	*Best case execution time: minimum core execution time*
BCRT	*Best case response time: minimum response time*
CPU-load / CPU-utilization	*Ratio of time not spent executing the idle task to the duration of the time frame observed. How background tasks are to be considered is undefined.*
Deadline	*By design defined point in time when a certain event must have occurred, typically the termination of a task or interrupt*
Decomposition	*Breaking down a top level attribute into its components on a lower level*
ECU	*Embedded control unit*
E/E	*Electric/electronic*
Hot-spot	*Application code that makes a particularly high contribution to CPU load, having a large core execution time or a high frequency or both.*
Idle task	*RTOS code which gets executed when no task or interrupt is pending and no background task is defined*
RTOS (or just "OS")	*Real time operating system*
Scheduling	*Deciding how to commit resources between a variety of possible tasks. [Wikipedia]*
Scheduling entity	*An entity that performs scheduling. Typically, this is one core or one bus.*
SIL	*Safety integrity level. A higher level indicates the impact of errors can be more hazardous.*
Static analysis	*Model based offline analysis*
TIMEX	*AUTOSAR Timing Extensions*
Traceability	*The ability to link certain aspects of a document (e.g. requirements in a requirements document) to the corresponding aspects in other documents (e.g. test cases in a test specification document)*
Tracing	*Logging events into a trace buffer*
WCET	*Worst case execution time: maximum core execution time*
WCRT	*Worst case response time: maximum response time*

09 SAFETY & AVAILABILITY

The safety and availability of a system are competing requirements. A system can enter a "fail safe" state by withdrawing availability. Taken to its logical conclusion, the safest system does nothing at all. However, such a system will not be commercially successful.

A failure to adhere to strict timing requirements will cause a well-protected, safe system to enter a fail safe state, meeting its safety requirements but offering little or no functionality. One task that slightly overruns might cause every function on its ECU to be withdrawn. In the context of safe systems, good timing behavior is therefore essential to maintain availability.

During the development process of such a safe system, timing errors can be very difficult to analyze and debug, since the timing protection takes over and stops operations. Only with suitable tools can timing behavior prior to such a shut-down be constructed and analyzed.

STANDARDS ADDRESSING SAFETY ASPECTS

There is no safety standard specific to embedded timing. However, the standards listed below require the identification of functional and non-functional hazards and the demonstration that the software does not violate the relevant safety goals. These standards mention explicitly three important non-functional, safety-relevant, software characteristics: Absence of runtime errors, execution time and memory consumption [7].

STANDARD	SAFETY LEVEL Lowest	SAFETY LEVEL Highest	COMMENTS
IEC-61508	*SIL1*	*SIL4*	*Depreciated general (i.e. not specific to automotive) standard for functional safety*
IEC-61508 Edition 2.0	*SIL1*	*SIL4*	*Successor of IEC-61508*
ISO-26262	*ASIL-A*	*ASIL-D*	*Automotive specific adaptation of IEC-61508*
DO-178B	*Level E*	*Level A*	*Safety standard for avionics*
DO-178C	*Level E*	*Level A*	*Successor of DO-178B*
CENELEC prEN 50128	*SIL1*	*SIL4*	*Safety standard for railway*

LEGAL ASPECTS AND LIABILITY

In the worst case, people might be killed as the result of a timing problem. It is not clearly defined how such cases are treated in court but the producer of the ECU which caused the accident will be asked whether the software was tested/verified according to the **state-of-practice**. The **State-of-the-art** is defined by research and becomes state-of-practice when applied repeatedly in production projects. So with respect to timing, projects should at least make use of the state-of-practice.

10 MULTICORE

Multicore development presents significant additional challenges. The list below outlines the key aspects.

Code level

- Shared bus/memory conflicts (e.g. n cores fetching code from the same FLASH)
- Different CETs for the same function on different cores (this becomes relevant when using dynamic task allocation)

RTOS level

- Dynamic task allocation with increased scheduling overhead due to migration costs
- Too much use of OS mutex services quickly leads to poor performance (even worse than single-core)
- Not enough use of OS mutex services is likely to result in timing-related functional defects
- Static task allocation can lead to poor performance due to poor use of some cores

11 REFERENCES

[1] Peter Gliwa, Albrecht Mayer:
Messung der Anwendungsperformance auf Mikroprozessoren für den Automobilbereich 15. Internationaler Kongress: Elektronik im Kraftfahrzeug, Baden-Baden, Germany, Oct 2011

[2] ALL-TIMES: Integrating European Timing Analysis Technology.
Research project within the European Commission's 7th Framework Programme on Research, Technological Development and Demonstration. www.all-times.org

[3] Marek Jersak, Kai Richter, Peter Gliwa:
Planung und Absicherung der Echtzeitfähigkeit von Software und vernetzten Steuergeräten Simulation und Test für die Automobilelektronik 2010, Berlin, Germany, May 2010

[4] Marek Jersak, Kai Richter, Hans Sarnowski, Peter Gliwa:
Laufzeitanalysen zur frühzeitigen Absicherung von Software ATZelektronik, 2009

[5] Nicholas Merriam, Peter Gliwa, Ian Broster:
Measurement and tracing methods for timing analysis International Journal on Software Tools for Technology Transfer, February 2013

[6] Oliver Scheickl, Christoph Ainhauser, Peter Gliwa:
Tool Support for Seamless System Development based on AUTOSAR Timing Extensions ERTS² 2012 | Embedded Real Time Software and Systems, Toulouse, France, February 2012

[7] Daniel Kästner , Christian Ferdinand:
Safety Standards and WCET Analysis Tools ERTS² 2012 | Embedded Real Time Software and Systems, Toulouse, France, February 2012

GLIWA embedded systems

www.gliwa.com

GLIWA GmbH embedded systems | Pollinger Str. 1 | D-82362 Weilheim

TIMING 1. CLASS

T1 – state of the art timing suite