A systematic approach for timing requirements

EMCC 2018, Munich

Version 1.1
Contents

• Introduction
• AUTOSAR Timing extensions
• Timing requirements
• Tracing: Timing verification
• A report from the frontline
• Conclusion
Introduction
Why care about timing?

- Proper timing of ECU software is essential for
  - Reliability / Availability
  - Safety (and often also Security)

- Timing problems are very often difficult
  - to identify as such, to debug, to solve

- ISO 26262 requires “freedom of interference”
  - Can only be guaranteed in the absence of timing problems

- Multicore
  - If your single-core timing is unstable already, multi-core will be a nightmare
Software Development Process

Apply your development processes and methodologies also to the timing of your software!
Who is GLIWA embedded systems?

- Timing analysis and embedded software expertise since 2003
  - Headquarters located in Weilheim near Munich; GLIWA Ltd. in York
  - 35 employees, many timing experts
  - Average annual growth over the past 7 years: 27.5%

- Stack Analysis combining static and dynamic methods
AUTOSAR
Timing Extensions (TIMEX)
Let's focus on one ECU ("EcuTiming")
The SW-C “idle speed control” of an engine management ECU is coded in three runnables:

- IdleSpeedInit
- IdleSpeed10ms
- IdleSpeed50ms

As part of the RTOS configuration, these get mapped to three different tasks which they share with many other runnables from other SW-Cs.

For multi-core processors, tasks get also mapped to a certain core.
AUTOSAR Timing Extension (TIMEX)

- With AUTOSAR 4.0, the *Timing Extensions* were added allowing precise timing constraint specification.

- Timing constraints can be applied to
  - Timing Description Events
  - Timing Description Event Chains
  - ordered list of Executable Entities
The views addressed by TIMEX

- **VfbTiming**
  timing related to interaction of SW-Cs at VFB level

- **SwcTiming**
  timing related to the internal behavior of atomic SW-Cs

- **SystemTiming**
  timing on system level incorporating readily configured ECUs, busses

- **BswModuleTiming**
  timing related to BSW module internal behavior

- **EcuTiming**
  timing related to everything inside one readily configured ECU

Max. latency = 2ms

Source: Specification of Timing Extensions, autosar.org
TIMEX Constraint Types I

• **EventTriggeringConstraint**
  – Example use-case: supervise jitter

• **LatencyTimingConstraint**
  – Example use-case: avoid loss and duplication of data due to under- and oversampling and/or jitter

• **AgeConstraint**
  – Example use-case: make sure, data is not too old

• **SynchronizationTimingConstraint**
  – Example use-case: establish and maintain a consistent time base for the interaction between different subsystems
• **OffsetTimingConstraint**
  – Example use-case: bound the time offset between the occurrence of two arbitrary timing events

• **ExecutionOrderConstraint**
  – Example use-case: supervise the correct execution order of runnables

• **ExecutionTimeConstraint**
  • Example use-case: specify the maximum allowed run-time budget of a process
How to use TIMEX

• Option 1: write TIMEX directly i.e. ARXML
  – Well, this certainly is no fun.

• Option 2: use tool support
  – ArTime from BMW Car IT: language for specifying TIMEX requirements. ArTime is an Artop plug-in
  – In-house tools
  – Others (not widely used though)
  – Well, there is much room for improvement, right?

• Option 3: ask experts
  – Isn’t that frustrating?

• Option 4: give up
  – Yes, it is!
Timing requirements
Requirements specification documents

- A good ECU’s requirements specification is the foundation for sound and safe timing.

- Two types of timing-related requirements should be addressed:
  - Dedicated **timing requirements** (as far as they are known)
  - Requirements regarding the **environment, methodologies and tools**
Collection of typical timing requirements

- Max. allowed CPU-load
  - Present in most specs already
  - Specify *how* it is calculated (which observation frame $T^O$ to be used)?
  - Remark: Cannot be specified using TIMEX
- Start-up time (“presence on the bus”)
- End-to-end latency
- Data-age
- Max. CET for TASKs, ISRs, runnables
  - Think of *budgeting your timing*!
    - scheduling simulation
- Response-times (at least $RT < period$)
- Jitter (max. deviation from targeted period)

**Fall-back when TIMEX not appropriate:** constraints on timing parameters

$U^o = \frac{\sum_{n=1}^{N} CET^o_n}{T^o}$

$U^{\text{max}} = \max(U^o)$
Collection of typical tool requirements

- **Feature-set**
  - Scheduling-simulation, -analysis
  - Profiling
  - Tracing
    - Synchronized traces from all cores
    - Runnables, functions, any code, data-flow, etc. without rebuilding the software
    - etc.
- **Tool availability (OEM, tier-1, both, anybody, …)**
  - Inhouse tools not appropriate
  - Make sure, all relevant data can be exchanged
- **Scope**
  - Scheduling-simulation to the detail-level of…
  - Timing verification
    - With automated HIL tests
    - In the car
BMW: TIMEX in mass-production

- In 2009, BMW used TIMEX/ArTime for a chassis ECU
  - Formal specification of timing requirements
  - Seamless interface specification ⇔ verification: timing requirements were imported and then verified by T1
  - Textual specification of requirements regarding methodologies & tools

See joint ERTS 2012 paper by BMW and GLIWA: free download at gliwa.com
Templates for requirements specifications

- Top-up result: text templates for requirements specifications
  - Reuse generic requirements in future projects

- Excel table with text templates
  - including recommendation according to ASIL level

- Word document with templates

- Some big OEMs follow this approach already. More and more follow…
Tracing: Timing verification
(Scheduling-) tracing vs. timing measurement

• Timing measurement
  – produces timing parameters (“numbers”) but no traces

• Scheduling Tracing
  – produces traces which can be viewed (and from which timing parameters can be derived)
  – different kinds exist
    • Hardware-based tracing
    • Instrumentation based tracing
    • Hybrid approaches
    • On-chip scheduling tracing (future)

• Profiling
  – The process of collecting timing parameters
ARTI

- AUTOSAR currently lacks a standardized interface to get timing data efficiently out of an ECU
  - ORTI is outdated, does not support multi-core and is not AUTOSAR
  - PreTaskHook / PostTaskHook are very inefficient and not allowed “on the road”

- In April 2016, an AUTOSAR Concept was initiated:

  ARTI ("AUTOSAR Run-time interface")

Goal: ORTI successor within AUTOSAR plus efficient support of
- instrumentation based tracing
- non-intrusive on-chip scheduling tracing
- multi-core
- runnables
- AUTOSAR AP
Tracing: End-to-end model-check

- On its way from the mind to the microcontroller, an idea can suffer from transition-errors.
- Tracing allows an end-to-end model-check.
Multi-core example of missing verification

- Customer moved code from the highly loaded core 0 to core 1
- Surprisingly, the load on core 0 went up!?!
On Chip Bus Access Times

The table describes the CPU access times in CPU clock cycles for the TC27x. The access times are described as maximum CPU stall cycles where e.g. a data access to the local DSPR results in zero stall cycles. Pls. note that the CPU does not always immediately stall after the start of a data read from another SPR due to instruction pipelining effects. This means that the average number will be below the here shown numbers.

<table>
<thead>
<tr>
<th>CPU Access Mode</th>
<th>CPU clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data read access to own DSPR</td>
<td>0</td>
</tr>
<tr>
<td>Data write access to own DSPR</td>
<td>0</td>
</tr>
<tr>
<td>Data read access to own or other PSPR</td>
<td>5</td>
</tr>
<tr>
<td>Data write access to own or other PSPR</td>
<td>0</td>
</tr>
<tr>
<td>Data read access to other DSPR</td>
<td>5</td>
</tr>
<tr>
<td>Data write access to other DSPR</td>
<td>0</td>
</tr>
<tr>
<td>Instruction fetch from own PSPR</td>
<td>0</td>
</tr>
<tr>
<td>Instruction fetch from other PSPR (critical word)</td>
<td>5</td>
</tr>
<tr>
<td>Instruction fetch from other PSPR (any remaining words)</td>
<td>0</td>
</tr>
<tr>
<td>Instruction fetch from other DSPR (critical word)</td>
<td>5</td>
</tr>
<tr>
<td>Instruction fetch from other DSPR (any remaining words)</td>
<td>0</td>
</tr>
<tr>
<td>Initial PFlash Access (critical word)</td>
<td>$5 \times \text{configured PFlash Wait States}^1$</td>
</tr>
<tr>
<td>Initial PFlash Access (remaining words)</td>
<td>0</td>
</tr>
<tr>
<td>PMU PFlash Buffer Hit (critical word)</td>
<td>4</td>
</tr>
<tr>
<td>PMU PFlash Buffer Hit (remaining words)</td>
<td>0</td>
</tr>
<tr>
<td>Initial DFlash Access</td>
<td>$5 \times \text{configured DFlash Wait States}^2$</td>
</tr>
<tr>
<td>TC1.6E/P Data read from System Peripheral Bus (SPB)</td>
<td>$4 \times \left(\frac{f_{\text{CPU}}}{f_{\text{SPB}}}\right)$</td>
</tr>
<tr>
<td>TC1.6E/P Data write to System Peripheral Bus (SPB)</td>
<td>0</td>
</tr>
</tbody>
</table>

1) $\text{FCON.WSFLASH} + \text{FCON.WSECPIF}$ (see PMU chapter for the detailed description of these parameters).
2) $\text{FCON.WSDFLASH} + \text{FCON.WSCECFD}$ (see PMU chapter for the detailed description of these parameters).
Multi-core example of missing verification

Maximum CPU stall cycles for **data** reads

Maximum CPU stall cycles for **program** reads

“Maximum” refers to a situation where there are no memory access conflicts. If these occur, the penalty can be **much** higher!

- **Core 0**
  - DSPR
  - PSPR
  - DMI
  - PMI

- **Core 1**
  - DSPR
  - PSPR
  - DMI
  - PMI

- **Crossbar**

- **System peripheral bus**

- **Data read access**
  - DSPR = data scratch pad RAM

- **Program read access**
  - PSPR = program scratch pad RAM
  - DMI = data memory interface
  - PMI = program memory interface
A report from the frontline (plus recommendations)

Mass-production projects
OSEK / AUTOSAR task states

ECC = Extended Conformance Class
TASK = container for code, e.g. runnables
Timing parameters

<table>
<thead>
<tr>
<th>Abr.</th>
<th>Explanation (EN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPT</td>
<td>initial pending time</td>
</tr>
<tr>
<td>CET</td>
<td>core execution time</td>
</tr>
<tr>
<td>GET</td>
<td>gross execution time</td>
</tr>
<tr>
<td>RT</td>
<td>response time</td>
</tr>
<tr>
<td>DT</td>
<td>delta time</td>
</tr>
<tr>
<td>PER</td>
<td>period</td>
</tr>
<tr>
<td>ST</td>
<td>slack time</td>
</tr>
<tr>
<td>PRE</td>
<td>preemption</td>
</tr>
<tr>
<td>JIT</td>
<td>jitter</td>
</tr>
<tr>
<td>CPU</td>
<td>cpu load</td>
</tr>
<tr>
<td>DL</td>
<td>Deadline</td>
</tr>
<tr>
<td>NST</td>
<td>Net slack time</td>
</tr>
</tbody>
</table>

CET = CET1 + CET2 + CET3
NST = NST1 + NST2
TASK(Task_B)
{
    EventMaskType ev;
    for(;;)
    {
        (void)WaitEvent(Rte_Ev_Cyclic2_Task_B_0_10ms |
                        Rte_Ev_Cyclic2_Task_B_0_5ms);

        (void)GetEvent(Task_B, &ev);

        (void)ClearEvent(ev & (Rte_Ev_Cyclic2_Task_B_0_10ms |
                                Rte_Ev_Cyclic2_Task_B_0_5ms));

        if ((ev & Rte_Ev_Cyclic2_Task_B_0_10ms) != (EventMaskType)0)
        {
            CanNm_MainFunction();
            CanSM_MainFunction();
        }

        if ((ev & Rte_Ev_Cyclic2_Task_B_0_5ms) != (EventMaskType)0)
        {
            CanTp_MainFunction();
            CanXcp_MainFunction();
        }
    }
}
TASK(Task_B)
{
    EventMaskType ev;
    for(;;)
    {
        (void)WaitEvent( Rte_Ev_Cyclic2_Task_B_0_10ms |
                         Rte_Ev_Cyclic2_Task_B_0_5ms );

        (void)GetEvent(Task_B, &ev);

        (void)ClearEvent(ev & ( Rte_Ev_Cyclic2_Task_B_0_10ms |
                                Rte_Ev_Cyclic2_Task_B_0_5ms ));

        if ((ev & Rte_Ev_Cyclic2_Task_B_0_10ms) != (EventMaskType)0)
        {
            CanNm_MainFunction();
            CanSM_MainFunction();
        }
        if ((ev & Rte_Ev_Cyclic2_Task_B_0_5ms) != (EventMaskType)0)
        {
            CanTp_MainFunction();
            CanXcp_MainFunction();
        }
    }
}
Previous run-time situation plus “scheduling WaitEvent”
Houston, we have a problem

• Since 15 years, GLIWA serves as a team of firefighters for timing-related problems

• More and more often we see the same problematic OS configuration:
  – non-terminating ECC task
  – a second layer of scheduling

→ unnecessary complexity
  violating the “keep it simple” rule
Recent example (there are many alike)

The user was completely unaware of the overload scenario shown in this T1 trace.
void ErrorHook(StatusType status)
{
    switch(status) {
    case E_OS_LIMIT:
        /* failed task activation
         * as a result of an overload
         * situation */
        SystemReset();
        break;
        default:
        break;
    }
}

ErrorHook: called by the OS, implemented by the user (of the OS)

User’s intention: Reset when system is overloaded

BUT: ErrorHook does not get called when an event is re-triggered
The setup with non-terminating ECC tasks adds a second layer of scheduling on top of the OS.

- No supervision through ErrorHook “E_OS_LIMIT”

- ECC in general requires more resources
  - more RAM
  - more Stack (a separate stack at least per prio, typically per task)
  - more run-time

- Difficult to analyze: ARTI will disallow a mixture of the two kinds of WaitEvent

- Increased complexity without any benefit → error-prone
Typical ECC usage by the RTE

```
TASK(Task_B)
{
    EventMaskType ev;
    for(;;)
    {
        (void)WaitEvent(Rte_Ev_Cyclic2_Task_B_0_10ms |
                         Rte_Ev_Cyclic2_Task_B_0_5ms);
        (void)GetEvent(Task_B, &ev);
        (void)ClearEvent(ev & (Rte_Ev_Cyclic2_Task_B_0_10ms |
                                 Rte_Ev_Cyclic2_Task_B_0_5ms));
        if ((ev & Rte_Ev_Cyclic2_Task_B_0_10ms) != (EventMaskType)0)
        {
            CanNm_MainFunction();
            CanSM_MainFunction();
        }
        if ((ev & Rte_Ev_Cyclic2_Task_B_0_5ms) != (EventMaskType)0)
        {
            CanTp_MainFunction();
            CanXcp_MainFunction();
        }
    }
}
```

What a mess!
Recommended configuration

```
TASK(Task_B_10ms) // BCC1
{
    CanNm_MainFunction();
    CanSM_MainFunction();
    TerminateTask();
}

TASK(Task_B_5ms) // ECC
{
    EventMaskType ev;
    CanTp_MainFunction();

    // the following WaitEvent call is a "regular" WaitEvent
    (void)WaitEvent( Can_Ev_TriggerSM_Task_B );
    (void)GetEvent(Task_B, &ev);
    (void)ClearEvent(ev & ( Can_Ev_TriggerSM_Task_B ));
    CanXcp_MainFunction();
    TerminateTask();
}

Use BCC1 whenever possible
```
Conclusion

A CONCLUSION IS THE PLACE...

WHERE YOU GOT TIRED OF THINKING.
Conclusion

- AUTOSAR TIMEX is powerful, flexible and complicated
  - Tool-support for good usability yet to come

- Rather than *not* specifying any timing requirements
  - Define min./max. constraints on timing parameters
  - Use semantics provided by tools (e.g. TA Tool Suite)
  - Use informal description (text is better than nothing)

- Reuse your requirements: build a pool of requirement templates

- Verify your timing (requirements) with the *real* system!

- Get the single-core timing right *before* addressing multi-core.

- **Keep it simple!** Complexity increases the probability of things going wrong.
Thank you