Design of Robust System Architectures for Automotive ECUs

Andreas Wolfram, Mikhail Makarov, Continental Automotive GmbH
Tapio Kramer, Wendel Ramisch, Dr. Ralf Münzenberger, INCHRON GmbH

Abstract

The Paper describes a model based approach to improve the robustness and reliability of an embedded system with respect to real-time performance. It characterizes how real-time simulation models are generated, run and analyzed to gain knowledge of the dynamic system behavior. The system's reactions to dynamic stimuli can be predicted without having to implement all hardware and software. A study with an automotive car body control unit illustrates how the timing model is developed parallel to the development progress. Findings and improvements are listed. Besides the technical aspects, the business impact for the current and future systems proves significant advantages of the chosen approach.

1 Why Real-Time Simulation?

Solving real-time problems in embedded systems with growing functionality and interaction with other devices becomes more and more complex and expensive. Raising the product quality by making such problems rare requires a robust and reliable architecture. Thorough testing of the implemented systems in realistic situations by experienced personnel supports these quality efforts. But tests and measurements with prototypes can't do the job solely. To test the dynamic real-time behavior all components have to be available or simulated in real-time using expensive hardware. Most measurement methods require instrumentation of the unit under test with additional software, that itself changes the timing. Since embedded systems by design often have limited resources, the measurements can cover only short time slices in certain system states. And when a problem has been found all changes to the system take time and money. Performing what-if-analysis to evaluate solutions or extensions are very costly.

Modeling and simulation is with no doubt the best practice to master development projects of such complex systems. Simulating the real-time behavior with the real-time simulator chronSim™ [INCH] allows a wide range of tests focused on the real-time aspects. The models can be simulated over long time, cover multiple system states and will show the reaction to various stimuli that are hard to reproduce on a Hardware in the Loop system (HiL). Finding rare real-time errors, optimizing the performance and predicting the effect of planned changes to the system is easier with real-time simulation. Overall the "demand for modeling and simulating the real-time behavior is increasing" [Rei09].

In the automotive environment the number of electronic components is more and more increasing. Reasons are a growing number of requirements as well as more integration of functions into highly integrated electrical components controlled by software. Especially in the interior area, where the realized system functions have a direct connection to the driver, the architecture and system behavior is subject to very frequent changes and a large variety of implementations. A proper method to cope with these challenges is the described use of models and simulation for the functional as well as the non-functional and timing requirements.

2 Creating a Real-Time Simulation Model

Similar to functional models with focus on continuous control applications (e.g. Simulink™) the system`s timing behavior can be modeled, simulated and analyzed with the real-time simulator chronSim.

2.1 Modelling Real-Time Systems

A real-time simulation model for chronSim is generated using an iterative approach. Fig. 1 depicts the three main steps from left to right. In the first step the system information is gathered focusing the software and hardware architecture. Experience has shown that a high abstraction level is well suited to get a first model with low effort. The data is entered using the graphical user interface and as model code in C or C++, defining the Task-Models.

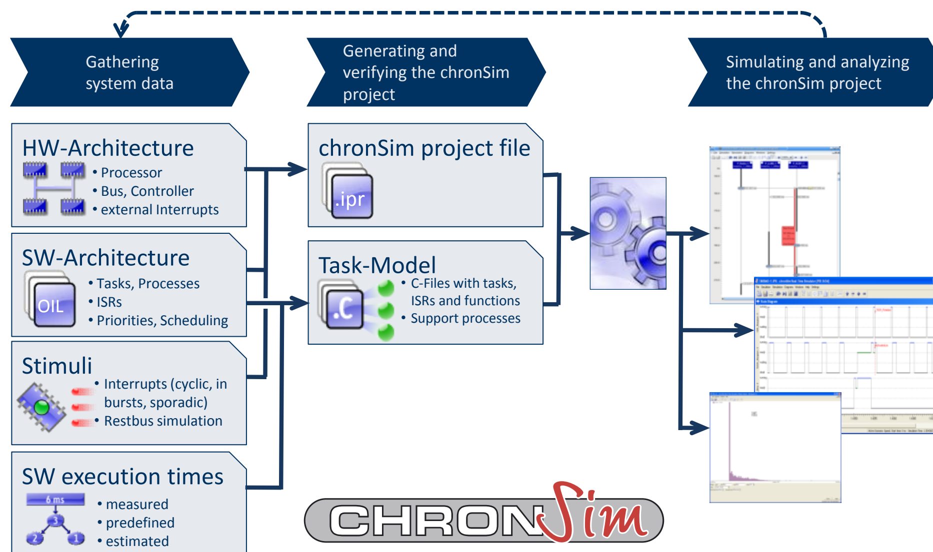

Fig. 1: Workflow to Generate a Real-Time Simulation Model

Simulating and analyzing this first model in the third step helps to understand the system behavior and identify the relevant information, that might have to be added in the next iteration. Focusing timing relevant aspects and iterating fast results in a timing model close to the real system in short time.

2.2 Task-Models

Real-time properties and requirements covering all system levels have so far not been standardized or made widely available. For a complete system description there was no versatile tool or data format in use until now. Task-Models can close that gap and enable embedded system architects and developers to cover also the timing aspects when specifying their system [Kom07].

A part of the Task-Model consists of model code in ANSI C or C++. Here the execution times of tasks, functions and modules are entered. It can be derived by abstracting the target code and annotating it with the target execution time on the level of ISRs, tasks, functions or other software modules. Or it can be the starting point of a project having a skeleton code describing the software structure. Throughout the development it will become more detailed and finally may even be replaced by the target code itself.

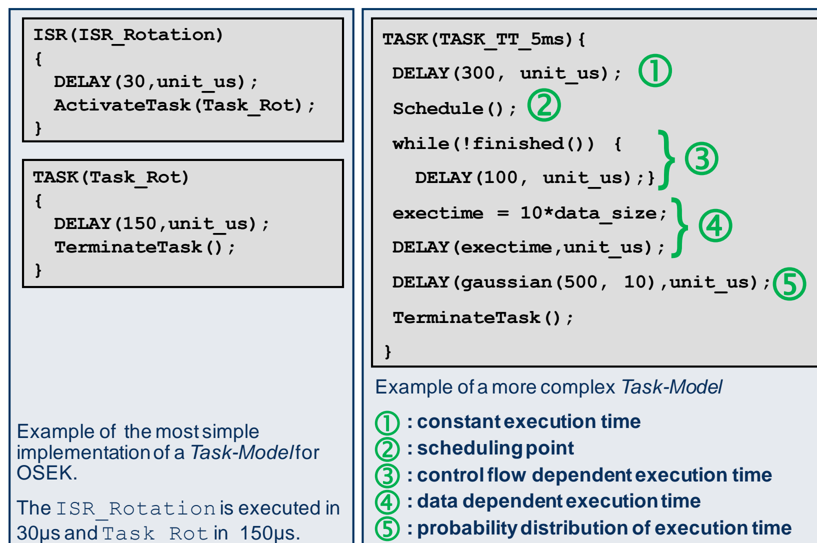

Fig. 2: Simple and Versatile Task-Models

Fig. 2 lists on the left a simple Task-Model for an OSEK OS. The macro `DELAY()` replaces the function code with its time budget. So the ISR `ISR_Rotation` will run for 30 microseconds (`DELAY(30, unit_us)`) and will then use an OS function call `ActivateTask()` to activate the task `Task_Rot`. This task will run for 150 microseconds as long as it is not suspended by a higher priority task or ISR. If the timing behavior of the

modeled system has to be defined in more detail, all possibilities of C/C++ can be used. The example on the right in Fig. 2 lists several methods to vary the execution time depending on system states, data contents or other stimuli. This enables dynamic reactions of the simulation model showing the same timing as the real system.

Event chains are specified directly in the Task-Model. By using the `EVENTCHAIN()` macro the designer marks each step in a chain of causal dependent events. The current instance of the event chain execution as well as the current step within a specific instance can be specified by free variables. By passing their values between different nodes via a communication bus (e.g. within a CAN message) even causal dependencies across ECUs (Electronic Control Units) can be analyzed. [Kra09a]

Task-Models can be adapted to the abstraction level, the development phase, the focused timing detail of all involved development partners. A network designer can model the communication across multiple nodes without having to use detailed models of the nodes' internals. Accordingly the developer of an application part within one node can model his parts very detailed. But he reduces the communication details and peripheral timing behavior models to simple stimuli sources as far as they influence his application's timing.

2.3 Gathering the Timing Information

For the correct real-time simulation and validation the net execution times of the software modules are required. Depending on the abstraction level these can be the execution times of the tasks and ISRs as fixed values. At a more detailed level the times can be split down to each function or even parts of it.

When designing a system the architect has to set certain cornerstones defining the general functionality and timing of the system. At this point he chooses what will be executed in parallel or in sequence. For every function or task the architect sets timing quotas. Throughout the implementation more details become available and can be added to refine the real-time model.

If prototypes or legacy systems are available the timing data can be derived from measurements. By adding instrumentation code at significant points (e.g. start and end of software modules) the system's timing behavior can be traced with timestamps to identify what system state occurred when. The net execution times can be derived utilizing additional scheduling information. [Kra09b]

The real-time simulator chronSim provides an optional estimator plugin chronEst™ with processor models to estimate the execution times of target C code. When the development has progressed so far that target code and compilers are available, the estimator module will use the C code, the processor model and the target compiler output to determine static execution time budgets. During the simulation the dynamic effects of e.g. pipelines and caches are applied to the static time budgets. This option has its beauty, but requires the code to be available and adds a detail level that can distract from the core problems and questions regarding the real-time behavior.

The quality of a Task-Model respective the simulation results corresponds to the input data and the timing modeling skills as shown in [Wir09]. "The simulation with a simplified model is surprisingly precise (only few percent deviation to real-time behavior, every problem found in the real system has been reproduced in simulation)."

2.4 Configuration of the System Architecture

To complete the system description the scheduling, hardware, peripherals and stimuli have to be specified (Fig. 1). The scheduling method and software architecture with the definition of processes, priorities and e.g. stack sizes can be entered or be imported from standard OS configuration files like OiL (OSEK Implementation Language).

The hardware architecture is entered into the simulator using graphical user interfaces. From a set of library components the microcontrollers, peripheral components and interfaces (I/O, memory, buses, etc) are selected and logically interconnected using memory addresses.

The stimuli to the modeled ECU(s) may be configured using the graphical user interface, added as C code to the simulation model or can be imported from standard files. Additional stimulation from a bus communication can be added by using the virtual restbus simulation plugin chronBus™ via import of Fibex or CANdb files. The simulator generates the defined bus traffic and dynamically stimulates the model accordingly.

If the development tool-chain describes the system architecture in UML the annotated models can be easily converted into Task-Models [Lok09].

2.5 The Collaboration Aspect

Good collaboration between manufacturer (OEM) and supplier(s) (Tier1 and/or Tier2) requires exchange of information and a common platform to describe the desired design goals. Describing the timing of a system using Task-Models allows to share an executable specification of the dynamic system without having to provide the source code. The `DELAY()` macros replace the actual function code and do represent its correct timing behavior. Depending on the chosen detail level a `DELAY()` can substitute a simple function call or a large application with complex internal structure. For further information about collaboration between OEM and Tier1 suppliers with Task-Models see [Mue09].

2.6 Simulation and Validation for Robustness

The real-time simulator chronSim executes the real-time models and enables the architect to experience the timing of his designed system before it has been implemented. By applying stress (stimuli) and load to the model the designer can use the simulator to perform sensitivity analysis and test the robustness of his design.

In addition the real-time requirements of the design can be tested with the same models using the real-time validator chronVal™. The analysis of the system with the validator will show the critical situations leading to requirement violations. Otherwise it reports the remaining system performance that is available for further expansions.

For an in-depth robustness analysis it is necessary to find the weak spots, where the system is sensitive to changes. Using Task-Models the real-time relevant factors can be easily varied to experience the limitations of the system's real-time capabilities.

- Changing the execution times by adding a global factor to the `DELAY()` commands.
- Adding stress by increasing the interrupt rates (stimuli) and inter arrival rates of messages on the buses.
- Applying drift and jitter to the clocks and time triggered stimuli.

Performing these tests on real hard- and software would cost immense amounts of time and money. And the real-time simulation allows an optimization on various aspects like memory need, CPU power or expandability [Mue07]. If the gathered timing data is detailed down to single functions within the tasks it is even possible to easily evaluate how remapping of functions will affect the system.

3 Real-Time Simulation of an Automotive ECU

In this study a typical automotive ECU for controlling the car body was under development. The main functionalities of a Central Body Controller are mostly

- exterior lighting
- interior lighting
- entry control and drive authorisation control
- doorlock control
- energy management

completed by a bunch of minor functions like horn, window heating, seat heating etc.

Main functionalities like interior lighting or door locking have a long term evolution. The look and feel of one feature does not change very much from one car model to another, because of two reasons: Firstly, it is the belief of the car manufacturer that his implementation is the best compared to competitors. In addition the driver should be made accustomed to the brand in the hope that he will stay on it even if a new car is needed. Secondly the functional aspects are predominantly implemented in software.

The electrical architecture of interior control function implementation changes significantly from car model to car model to realize the optimum of the wiring harness in terms of costs according to the car's assembly. For example the cost to implement an additional ECU for frontlight control in the engine bay might be less than implementing the functionality in a Central Body Controller located far away from the light bulbs with the need for a more expensive power wire connection. Therefore the hardware requirements often are specific only to one car line.

Because of this strong dependency on the car line's attributes it is obvious that the requirements for a single ECU are very specific while the requirements for a system function can be likely the same over different platforms of one OEM. This is the reason

why many OEMs created own software and system architecture departments, not only to specify the functions but implement them already in a simulation environment. These models are provided to the suppliers for implementation on the ECU platform with exactly the same behavior as the OEM specified in his model.

3.1 System Description

The ECU referenced in this paper is a central control unit for a lower class passenger car of a german car manufacturer. According to the system design the ECU is connected to the interior CAN bus as the main communication channel. Intelligent minor ECUs or sensors are connected via several, more simple LIN interfaces to the control unit.

The system can be divided in the following parts:

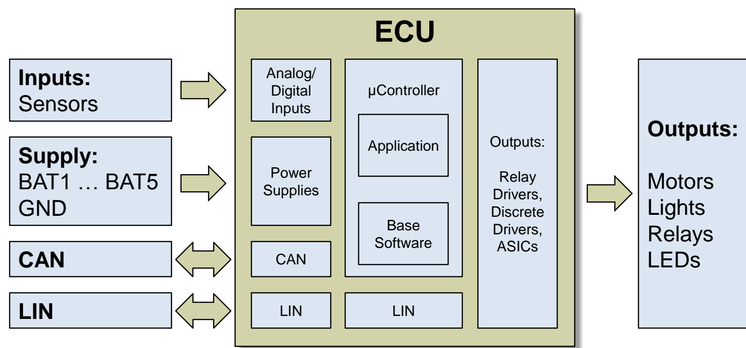

Fig. 3: Overview oft he Central Body Controller ECU

The hardware key facts are: 32-bit CPU at 64MHz, 1 MByte ROM, 64kBytes RAM, 32kBytes of NVRAM completed by 1 CAN and 3 LIN communication interfaces.

The software is structured according to AUTOSAR [AUT], with application layer, RTE and base software. The architecture is divided into four main parts:

- Application Layer: Containing the application functions, primarily model based
- Run Time Environment (RTE): Abstraction of the ECU hardware, providing a common runtime environment
- Base Software: Basic services for communication, I/O, memory and system functionality
- Flash Loader: Stand alone application allowing a flash update of the system

The application layer consists of Software Component models which are developed by the customer with Simulink™ and TargetLink™ while code generation out of the models is performed by Continental. Furthermore software modules for sensors and actuators are provided by Continental.

The main functionalities covered by the models are: exterior and interior light, wiper and washer function, power and energy management, power window control.

The applied OSEK operating system uses a mixed-preemptive scheduling concept where preemptive and non-preemptive tasks are combined. The tasks are grouped and will be scheduled cooperatively within the same group and scheduled with preemption between different groups. The following figure shows the abstract task grouping. Each group has its own priority scheme, where the left most task has the highest priority and the right most the lowest.

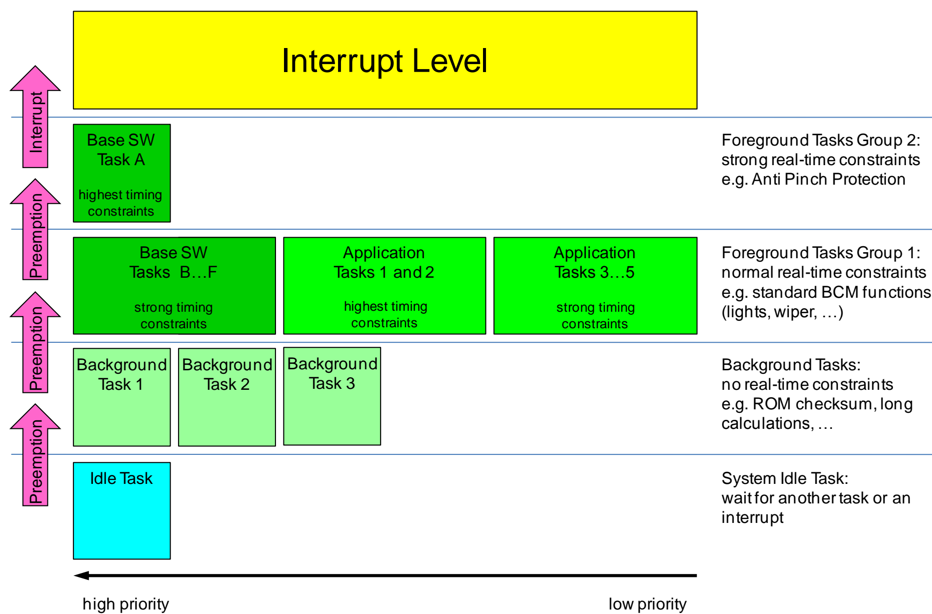

Fig. 4: Abstract Task Groups and their Prioritization

There are four groups of different priorities and scheduling policies:

- foreground 2: for functions having highest real time constraints (specific base software and Anti Pinch Protection Algorithm), covered in an 1ms task which preempts all other tasks.
- foreground 1: for normal base software and body control functions, realized in tasks with cooperative scheduling in order to secure data exchange.
- background: for extensive but low priority calculations like RAM/ROM tests.
- idle group: for platform internal purposes; the idle loop of the scheduler is implemented here

3.2 Description of the Task-Model

As described in the preceding paragraph, the Central Body Controller system is structured in a layered hierarchy. The application software is executed within a Run Time Environment (RTE) having an underlying AUTOSAR conform base software.

For the design of the Task-Model , the following goals had to be taken into account:

- Verification and optimisation of the SW architecture including task priorities, task offsets and mapping of SW modules to tasks
- Simulation of not yet implemented modules (Anti Pinch Protection)
- Investigation of influence of CAN interrupts on the real-time behavior
- Verification of system reaction time on an external trigger (time from switch pressing until lamp activation)
- Improvement of robustness regarding timing behavior

The following information was available:

- Task priorities, activation periods and offsets, scheduling policy
- Partly, execution times of the runnables and base software main functions
- Interrupt rates and ISR execution times
- System runtime requirements (average CPU load shall be < 80%) and reaction time requirements

In the context of the Task-Model creation the challenge arose to which degree to abstract the system's software structure in order to minimize effort and though receive valuable results from the simulation. It was decided to concentrate on the application software running on the top level of the hierarchy while the execution time of the RTE is implicitely included in the application software. The base software is part of the Task-Model.

Based on the collected information, altogether 11 tasks and approximately 40 ISRs were modeled. The application tasks consist of AUTOSAR runnables which were modeled as functions. The tasks were modeled in the following process groups:

- time-triggered (cyclic) SWP (software platform i.e. base software) foreground tasks (cycles: 1, 5, 10, 20, 100, 1000ms)
- time-triggered (cyclic) application tasks (cycles: 5, 10, 50, 100ms)
- sporadic (non-cyclic) application tasks, triggered by IRQ
- ISRs triggered by external interrupts e.g. from the CAN bus

The modeled hardware architecture representation consists of a controller with peripheral IRQs corresponding to the ISRs to be triggered. Task execution times were determined in an iterative process which is described in the following section. The execution times were decomposed down to function (AUTOSAR Runnable) level. According to available information concerning function properties, some tasks were given fix execution times, others consist of variable toggling execution time blocks corresponding with the configuration of the functions. Last but not least the application tasks were given scheduling points between execution time blocks (Runnables).

Fig. 5 shows some exemplary application tasks modeled in C code using chronSim extensions. Runnables have been represented within the tasks as pure time consuming modules via the `DELAY()` macro. OS schedule points `Schedule()` have been inserted between the runnables for cooperative scheduling. Instead of using fix values for time consumption, in order to ease refinement of the Task-Model a C include file approach using `#define` can be chosen.

```
TASK(Task_ApplTask_5) {

    DELAY( 171, unit_us );  // Runnable OVC_SCHED
    Schedule();
    DELAY( 182, unit_us );  // Runnable SAQ_SCHED
    Schedule();
    DELAY( 1486, unit_us ); // Runnable PNM_SCHED
    Schedule();
    DELAY( 17, unit_us );   // Runnable PNM_ACT_SCHED

    TerminateTask();
}

TASK(Task_SWP_FG1_10ms) {

    DELAY( 189, unit_us );  // Task TASK_10 w/o runnables
    TerminateTask();
}
```

Fig. 5: Excerpt from Task-Model

3.3 Simulation & Analysis

The system was modelled, simulated and analyzed in an iterative process following three phases. In the beginning only draft execution time information was available. An emulator to measure execution times was used later in the project. With the emulator results, highest quality timing information (execution times) could be incorporated into the Task-Model.

Phase One

In the initial situation the execution time was known only for 3 function modules. The remaining modules's execution times were approximated based on the model size. The execution time of the base software was derived from previous projects.

Results:

The simulation allowed to measure the start-to-start jitters of the ApplTask5ms and the SWPTask10ms over a long system run time. It revealed unexpected, seldom appearing blockings of the ApplTask5ms by the SWPTask10ms.

Phase Two

In order to improve the accuracy of the modeled execution times, a software driver was implemented which provided trace triggers at start and end of runnables and tasks allowing more detailed calculations of execution times. However, this method is time consuming and requires modification of the generated target software. In this phase, the execution time of the Anti Pinch Protection Algorithm was still unknown.[1]

The following Task-Model improvements were made:

- Execution times of the component's runnables and base software main functions were measured using the software driver
- OSEK OS schedule commands were added to the Task-Model
- CAN and ADC interrupts were added to the Task-Model

Results:

- SWPTask10ms does not last as long as assumed earlier and doesn't block the ApplTask5ms
- SWPTask5ms lasts sometimes 4ms because of the NVRAM manager causing ApplTask10ms sometimes being blocked. (See Fig. 6: varying Δt and multiple task activation)
- Interrupts don't essentially affect the software during runtime.

Phase Three

Finally, an emulator allowed fast and efficient determination of execution times along with higher accuracy. So the model could be improved by assigning the measured execution times to the functions (runnables). In addition the Anti Pinch Protection algorithm was implemented and its execution time was measured and entered into the model.

Results:

The task offsets could be optimized to obtain lower task jitter.

Measurements were made for two system states: in idle state and under load. In addition, the usage of the emulator allowed to compare the simulated with the recorded system behavior to improve the Task-Model. It turned out that the difference between simulated and measured behavior was minimal i.e. the Task-Model matches the reality very well. The single significant finding was that application and base software tasks were started by different functions and at different points in time (offsets).

In each iteration phase three different stimuli scenarios were simulated to distinguish the effects of different load scenarios. The number of vehicle window movements controlled by the ECU has a significant effect on the system behavior and therefore was stimulated here.

Since the execution times were given from an existing implementation (measurements), scheduling parameters like task activation offsets were subject to modification in order to optimize the system behavior.With the scheduling parameters priority and activation cycle the simulation showed some processes with higher priority overlapping and

[1] During execution time measurements in the real system, interrupts were disabled.

therefore suspending other processes. Due to the variant execution times of the overlapping processes, gross execution time jitters and start-to-start jitters of the suspended processes occur. The simulation diagrams also revealed that besides overlapping, intervals of low CPU load between active tasks are present which indicate unused processor resources. This results in an unbalanced load behavior.

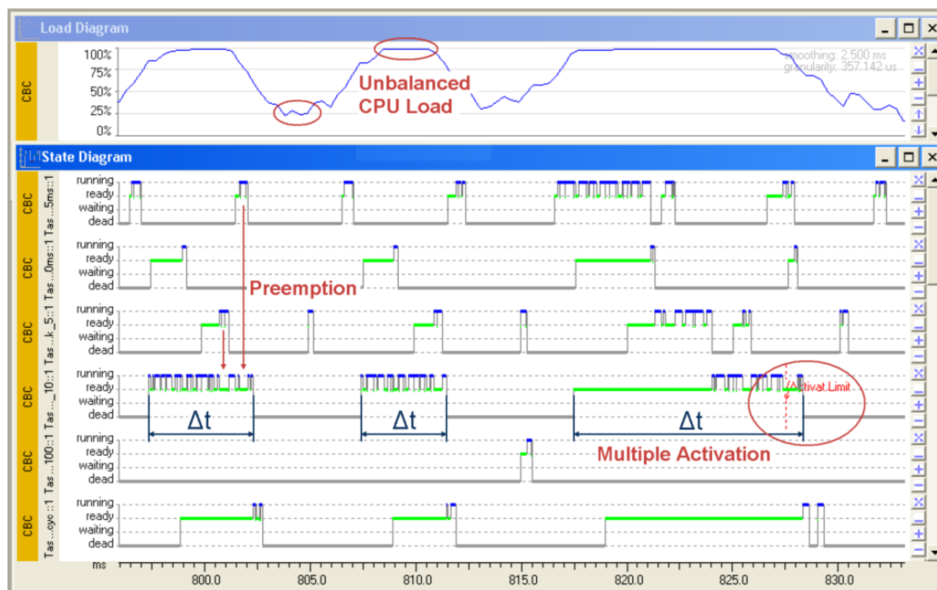

Fig. 6: Load and Task State Diagram with Multiple Task Activation

An optimization with respect to task activation offsets was performed with chronSim. This resulted in a reduction of jitters and in a smooth usage of processor resources.

The real system showed in certain load conditions that one task every now and then has a significantly longer execution time. Due to the sporadic character of this fact, there was no obvious side effect causing this situation. After enhancements to the Task-Model to show the observed behavior as well, the simulation revealed a certain load case where one application task becomes suspended too long resulting in a violation of runtime conditions: the task cannot be terminated before its next activation. (See Fig. 6)

Further analysis was performed regarding predictability of task execution times and jitters. Using the simulator's histograms, interesting results could be visualized (see Fig. 7). By optimization the task jitters were reduced from 3.1ms to 1.8ms. Additional investigations will help to judge whether or not the present degree of jitter is acceptable.

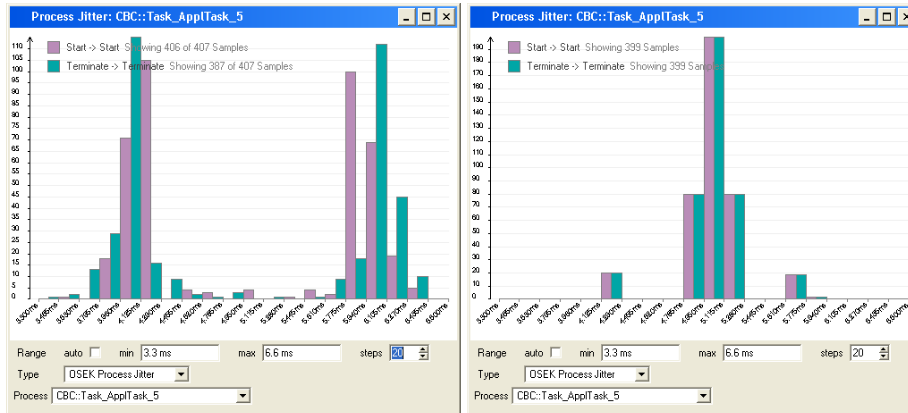

Fig. 7: Task-to-Task Jitter of 5ms-Task Before and After Optimization

3.4 Results

During the iterative generation of the Task-Model the system was modelled focusing the timing relevant parts. The factors and modules significantly influencing the system's real-time behavior were identified. Weaknesses and performance reserves became transparent. In addition components like the interrupts could be judged to have less timing influence than expected. The task offsets were adjusted to balance the CPU load and task jitters could be minimized. The real-time critical task was identified and the potential influence of the added Anti Pinch Protection Algorithm could be observed upfront. Altogether the resulting optimizations led to a more robust and reliable system with:

- No unwanted multiple task activations
- Optimized task offsets leading to less task suspensions and therefore smaller task-to-task jitters
- No missed deadlines (and the knowledge of the remaining distance to them)
- Identified performance reserves for additional functionality

Besides these details, the overall system behavior became well understood. With this gained knowledge a proposal for the next (large car) body controller was prepared having more confidence in the performance calculations.

4 Conclusion and Outlook

Using the described model based approach an automotive Electronic Control Unit for the central car body control was simulated with the real-time simulator chronSim™ to analyze and optimize the real-time performance. By revealing timing bottlenecks and root causes for missed deadlines the software architecture could be improved to result in a significantly more robust embedded system.

The complete task was performed in approx. 15 man days compared to 40 man days using conventional development methods. Change requests from customers are a regular routine in a project life cycle. The feasibility of such change requests can now be analyzed in 1/3 of the usual time. This saves time and money, allows fast feedback to the customer and gives more confidence in the modified system. Other projects will benefit by using the existing model with few adaptations. The collaboration with the customer (OEM) will base on exchange of the models and will significantly improve its efficiency and speed.

5 References

[AUT] www.autosar.org

[INCH] www.inchron.com

[Kom07] T. Komarek, M. Dörfel, R. Münzenberger; Developing Real-Time Constrained Embedded Software Using Task Models; In proceedings of the Advanced Automotiv Electronics (AAE 2007), January 2007, Gaydon; www.aae-show.co.uk

[Kra09a] T. Kramer, R. Münzenberger; New Functions, New Sensors, New Architectures – How to Cope with the Real-Time Requirements; In proceedings of Advanced Microsystems for Automotive Applications 2009, Berlin; www.amaa.de; ISBN 978-3-642-00744-6

[Kra09b] T. Kramer, R. Münzenberger; With Measurements to Real-Time Simulation; embedded world congress 2009, Nuremberg

[Lok09] M. Lokietsch; Performance Analyse von UML-Systemen; In proceedings of Fachkongress Echtzeitentwicklung 2009, Munich; www.echtzeitkongress.de

[Mue07] R. Münzenberger, M. Dörfel, C. Diederichs, U. Margull, G. Wirrer; Entwurf echtzeitfähiger Steuergerätesoftware in FlexRay-Netzwerken, DESIGN&ELEKTRONIK-Entwicklerforum Kfz-Elektronik 2007

[Mue09] R. Münzenberger, T. Kramer; Collaboration Support to Master Real-Time Challenges; embedded world congress 2009, Nuremberg

[Rei09] Dr. G. Reichart; Systemarchitektur - Logische und zeitliche Abhängigkeiten; In proceedings of Fachkongress Echtzeitentwicklung 2009, Munich; www.echtzeitkongress.de

[Wir09] G. Wirrer; Scheduling Simulation for ECS all along the Development Cycle; In proceedings of Fachkongress Echtzeitentwicklung 2009, Munich; www.echtzeitkongress.de