

Developing Real-Time Constrained Embedded Software Using Task Models

T. Komarek
INCHRON GmbH

M. Dörfel
INCHRON GmbH

R. Münzenberger
INCHRON GmbH

Abstract

This paper presents a task-centered view of embedded software design. Analysing the interactions of tasks by simulation reveals the dynamic real-time behaviour of embedded software and ensures that the software meets defined deadlines without sacrificing efficiency in the utilisation of the electronic control units. Investigating real-time performance is essential for the design of communication between electronic control units in embedded automotive electronics applications. In the design phase, the concept of task models helps the designer to detect timing correlations between FlexRay™ frames and the execution times resulting from the distributed synchronisation algorithm and task activations.

1. Introduction

Embedded automotive electronic control applications have to react to events in real-time, as response times are constrained by other components of the vehicle, such as mechanical and hydraulic devices. It is a standard concept of automotive software that embedded applications are, firstly, organised into processes or tasks, which compete for system resources, and secondly, scheduled for task execution by a real-time-operating system (RTOS) using system calls that manage communication and synchronisation. The AUTOSAR Specification of Operating System [1] currently uses the OSEK OS (ISO 17356-3) as its basis. Telematics and infotainment systems are assumed to use propriety operating systems that are beyond the scope of this document. The AUTOSAR operating system should have features such as priority-based scheduling and be “amenable to reasoning of real-time performance.” The OSEK/VDX Operating System Specification [2], states that “complex control software can conveniently be subdivided into parts executed according to their real-time requirements. These parts can be implemented by the means of tasks.”

The computational load of embedded automotive electronic control applications has to be distributed over several electronic control units (ECUs) as processor clock frequencies are limited in order to reduce working temperature and electromagnetic emissions. The ECUs communicate via a network. Investigating real-time performance of communicating ECUs is an essential step in the design for two reasons. First, suppliers and OEMs are interested in the best possible utilisation of hardware to reduce overall system cost. Second, to avoid timing faults in automotive embedded software, which can lead to system failure. Engineers have to make high impact decisions from the beginning of the development process with almost irreversible technological and economic consequences. Late re-design cycles are costly.

Real-time simulation uses virtual prototypes, that are optimised for time budgeting of the real-time simulations. The real-time simulation of a three-ECU system, synchronised via a FlexRay bus will be taken as an application example.

2. Task Models

Task models are the representation of a run-time concept to be used as an aid to the study and design of embedded software. They do not contain the complex functional code, that would produce the kind of results to be expected from the data created by an embedded software implementation. They are designed to provide only sufficient functionality to maintain task scheduling (e.g. RTOS-API) and to specify time budgets for the allocation of run-time.

As task models contain very little or no functional code, they compute faster than their implementation counterparts. Functional code is required, where the algorithm determines the control flow in the embedded software and therefore affects the scheduling of other tasks. Task models are preferably written in the same language as the implementation (e.g. in C). They can, therefore, be freely integrated into the implementation to fill gaps, where functional behaviour is unknown or not defined.

Creating task models is simple because the coding style is optimised to allocate just run-time.

2.1 Creating Task Models

It might be preferable to write a completely new “skeleton” without any functionality. Alternatively, source code can be obtained from an existing implementation (legacy code) or code can be generated using 3rd-party tools with a graphical front-end.

Creating task models from an implementation needs some analysis of its code. Functional code is replaced with macros or functions that specify time budgets. Code with algorithm-dependent expressions that affects control flow should remain. For example, switch statements, if-then statements, case statements, for-loops, and while-loops with algorithm-dependent expressions will affect real-time behaviour. RTOS API-functions, in the task model, are identical to the corresponding functions in the implementation. It is, of course, possible to maintain the implementation as it is and add function calls that specify execution time budgets, which can be switched using pre-processor constructs.

2.2 Coding Style of Task Models

The coding style of task models is very abstract. The source code in Example 1 shows an Interrupt Service Routine (ISR) “ISR_Rot”, a task “TASK_Rot”, triggered by the ISR, and a time-triggered task “TASK_TT_5ms”. The OSEK API functions are

used to schedule the task models and “DELAY” macros define time budgets. Task “ISR_Rot” activates TASK_Rot after 30µs, and task “TASK_TT_5ms” is activated every 5ms.

Example 1 Source codes of an ISR and two Tasks

```

ISR (ISR_Rot) {
    DELAY (30, unit_us);
    ActivateTask (TASK_Rot);
}

TASK (TASK_Rot) {
    DELAY (50, unit_us);
    Schedule ();
    DELAY (80, unit_us);
    TerminateTask ();
}

TASK (TASK_TT_5ms) {
    DELAY (300, unit_us);
    Schedule ();
    DELAY (100, unit_us);
    Schedule ();
    DELAY (500, unit_us);
    TerminateTask ();
}

```

Task models can have additional attributes that are not visible in the source code. These attributes determine task scheduling and hardware architecture properties. As an example, the OSEK Implementation Language Specification [8] lists CPU-specific configuration items, such as stack size, priority, the maximum number of queued activation requests for a task, schedulability (Preemptive/Non-Preemptive) and the kind of task (TASK/ISR). The OSEK Implementation Language (OIL) provides a mechanism to configure an OSEK application inside a particular CPU [8]. For each CPU there exists an OIL file. Hence, this configuration implies a mapping of tasks on ECUs.

3. Real-Time Design Using Task Models

The embedded software of distributed embedded systems runs on several ECUs, which are connected via a bus network. Hence, real-time design of embedded software has to consider concurrently running tasks, task communication and task interactions.

3.1 Simulation of Real-Time Behavior

Tasks can suffer interference from higher priority tasks and can also suffer blocking from lower-priority tasks locking shared resources or disabling interrupts (priority inversion). In embedded systems with communicating ECUs, the algorithms rely on messages arriving other components of the vehicle, such as mechanical and hydraulic devices, and from other ECUs. Messages can be received periodically, on demand or sporadically.

As the message arrival timing is a property of the network, such things as message jitters, burst and random errors affect task scheduling. The response time of lower priority messages is difficult to predict because higher priority messages are transmitted without interruption. All of these concurrent activities lead to a variety of non-functional performance dependencies, due to scheduling and arbitration, that can result in hard-to-find timing problems and missed deadlines.

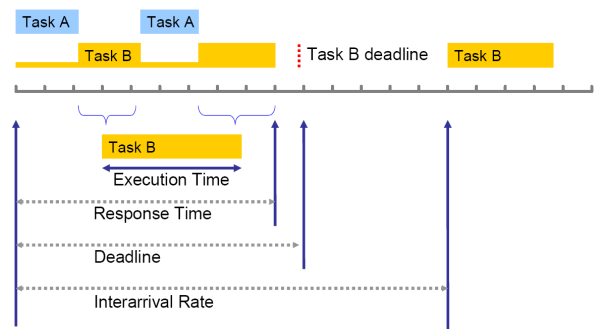
The simulation of the real-time behavior is a suitable means of investigating temporal behaviour of tasks and communication. Real-time simulations of task models must not be mistaken with functional verification. Such simulations have no functional code and cannot, therefore, produce any functional results.

3.2 Measuring Run-Time and Delay

Real-time researchers distinguish between hard and soft deadlines. Occasionally missing soft deadlines will not create major problems. Hard deadlines, however, must under no circumstances be missed – a missed deadline can result in a system failure. As an example, tasks in engine control systems can have hard type deadlines and convenience applications such as controlling indoor lights or wipers are much less critical. Because of this, all hardware resources have to have sufficient performance, to ensure that all hard deadlines are met. There has to be a trade-off - on the one hand, system failures cannot be allowed to happen, whilst on the other, suppliers of products for the high volume market might look for a compromise to reduce their hardware costs. It may be decided that scarce hardware resources have to be utilised more efficiently.

As a real-time simulation shows events at a certain point on the time line, it is helpful to look at the measurements being investigated, i.e. run-time and delay. The AUTOSAR Specification of Operating System [1] defines the timing terminology of a task (Figure 1). The response time is the time between a task being made ready to execute and generating a specified response. As such, it can be longer than the execution time. The deadline is the time, by which a task must reach a certain point in its execution, relative to the stimulus that triggered the activation. Tasks can be executed periodically or sporadically. Periodical tasks are repeated at an interarrival rate. For sporadic tasks the initial release time is an additional timing parameter which needs to be observed.

Figure 1 Definition of Timing Terminology



Measuring run-times and delays requires the observation of events because tasks react to events that occur at a certain point on the time line. A state change of any hardware or software entity can be considered as an event [4]. As an example, a hardware interrupt is triggered by a hardware event which starts an interrupt service routine. Event mechanisms are specified for operating systems. The OSEK Operating System Specification [2] defines an event mechanism as a means of synchronisation of extended tasks to initiate state transitions of tasks to and from the waiting state.

Events are communicated to the embedded system by signals. Their jitter and drift leads to a variation of events at points on the time line which in turn can change the dynamic real-time behaviour of tasks and communication. Tasks can be triggered and preempted by events that occur in other tasks and events in the vehicle's mechanical and hydraulic devices. Figure 1 shows a typical scenario that engineers have to be aware of in embedded software design. Task B is preempted, twice, by Task A, so that the response time exceeds the actual execution time.

Real-time design of embedded software considers dependencies of concurrently running tasks, communication and task interactions and not just deadlines of isolated tasks. Run-times between any two arbitrary events can be measured and used to set constraints thereby considering the effects of task preemption and randomness of events on the time line.

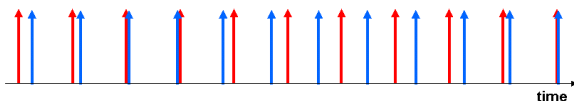
3.3 Time Bases and Scheduling

Embedded software in automotive applications reacts to events that are aligned with different time bases. First, tasks can be periodically activated according to timer periods, e.g. 5ms, 10ms, 20ms, and 100ms. Secondly, tasks react to events in the mechanical system, which depend, for example, on the pedal position or the crankshaft position of the engine. Receiving and sending messages via a bus uses a third time base. Temperature fluctuations, voltage fluctuations, and production tolerances of clock generators also affect the local time bases. The various time bases will diverge, even if all of them are initially synchronized.

If communication time bases diverge, messages will arrive too early or too late. In distributed systems with cycle-oriented communication, as is specified for FlexRay [3], message delivery has to follow a predictable schedule. As FlexRay microticks are derived from the local clocks in the ECUs, they are subject to the deviations of the local time bases, which lead to so-called offset (or phase) and rate (or frequency) differences. For this reason, the FlexRay specification [3] describes a communication clock synchronization function.

Deviations between different time bases inside each ECU will always remain as long they are synchronized with different sources. These deviations lead to a pattern of overlaid periodic events that resembles heterodyne beats. Figure 2 shows an example of two periodic interrupt signals; one is synchronous to a 5ms-FlexRay frame and the other to a local 5ms-timer.

Figure 2 Pattern of Two Periodic Interrupt Signals



Where tasks react to events that occur periodically with slightly different frequencies, a heterodyne beat pattern can be observed on task activation. This pattern is blurred by the random jitter and drift of trigger signals and interrupts. Interarrival rates or release times and execution times of tasks vary accordingly.

4. Virtual Prototypes

Virtual prototypes are suitable to simulate the real-time behaviour of embedded software on a host computer without using target hardware. They support measuring run-times between events. Real-time simulation can continue after a software error that might otherwise have led to a fatal crash of the target hardware.

Virtual prototypes contain information about delays through pipelining, registers, address modes, operation modes and target compiler settings of the processor core. Compiler settings represent target-specific optimisation modes. Clock period and frequency drift are modeled as parameters that represent the processor time base and its deviation due to temperature fluctuations, voltage fluctuations, and production tolerances.

During real-time simulation, the virtual prototype imposes measured times on the implementation code so that the simulated embedded software shows the same real-time behavior as the real target hardware. In the case of a task model, the specified time budgets are used instead.

5. Design Example

This design example demonstrates the real-time simulation of three ECUs synchronized via a FlexRay bus. The design was configured and simulated using INCHRON's chronSim® real-time simulator [5].

5.1 Hardware: Three Communicating ECUs

The hardware is modeled using three instances: ecu1, ecu2, and ecu3 from a virtual prototype. This prototype represents the real-time behavior of an Intel i386ex processor core, which is optimized for the OSEK RTOS, a clock, a timer, an IRQ source, and a FlexRay MFR4200 Communication Controller [6] from Freescale. This communication controller takes over the role of the FlexRay clock synchronization. Virtual prototypes are part of the chronSim distribution. Figure 3 shows a tree with the configuration of the hardware components in the chronSim user interface. In this configuration, the three instances ecu1, ecu2, and ecu3, are connected to each other via ChannelA and ChannelB of a FlexRay bus.

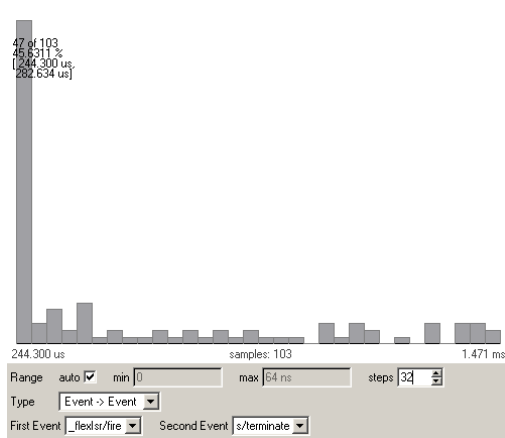
5.2 Software: Task Models

The embedded software in this design example is organized into task models. Some of them were derived from an existing implementation of a single-ECU design, and most FlexRay specific functions were created from scratch. All tasks are preemptable.

The software reacts to interrupts that are aligned with three time bases. It is triggered by the local timers Ecu_TimerFixed, the IRQ sources Ecu_RotPulse, and interrupts from the communication controllers Ecu_Flex (Figure 3). Timer ticks periodically activate time-triggered tasks. The IRQ sources generate sequences of interrupts synchronously to events in the mechanical system. The communication controller outputs an interrupt at each FlexRay cycle.

run-times and delays becomes complicated. In our design example, the run-time between the two events of firing the FlexRay interrupt and termination of the Ecu_Flex_5ms task is a characteristic size of the system that needs to be measured. The chronSim real-time simulator user interface can display histograms of the run-times distribution between any events specified. In case of the two events, firing the FlexRay interrupt and terminating the Ecu_Flex_5ms task, the histogram in Figure 4 shows a run-times distribution, where, in about 47 cases out of 103, samples belong to the class of 244.300µs to 282.634µs. The remaining samples are randomly distributed among other classes in the range up to 1.471ms. One might for example decide to enhance the priority of task Ecu_Flex_5ms in order to avoid very long run-times.

Figure 4 Histogram of Simulated Response Time



6. Conclusion

Task models support the real-time simulation of embedded software. Their coding style is optimized to allocate run-time and

to schedule tasks. Task models are fully sufficient to explore the temporal dynamics of concurrently running tasks, communication, and task interactions. Simulations of the real-time behaviour uncover the complex pattern of events adhering to different time bases and the randomness of events that affects task scheduling. Instead of target hardware, real-time simulation uses virtual prototypes. They impose run-times of the implementation or time budgets of the task models on the simulation. Real-time simulations using task models support engineers in making high impact decisions all through the development process and to minimize the risk of late costly re-design cycles.

References

- [1] AUTOSAR Specification of Operating System, <http://www.autosar.org>, AUTOSAR_SWS_OS.pdf
- [2] OSEK/VDX Operating System Specification 2.2.3, ©by OSEK, February 17th, 2005
- [3] FlexRay Communications System Protocol Specification Version 2.1, <http://www.flexray.com/>
- [4] AUTOSAR Glossary V2.0.1, <http://www.autosar.org>, AUTOSAR_glossary.pdf
- [5] INCHRON GmbH, www.inchron.com
- [6] FlexRay Communication Controllers MFR4200 Data Sheet, Rev. 0, 8/2005, www.freescale.com
- [7] J. Jasik, D. Paterson, Incorporating an MFR4300 Node in an MFR4200 Network, Freescale Semiconductor Application Note, Doc. No: AN3265, Rev. 0, 05/2006, ©Freescale Semiconductor, Inc., 2006
- [8] OSEK/VDX OSEK Implementation Language Specification 2.5, ©by OSEK

Figure 5 State Diagram of FlexRay Communication (ChannelA, ChannelB), ISRs, and Tasks for the Phase from 120ms to 170ms Simulated Real-Time

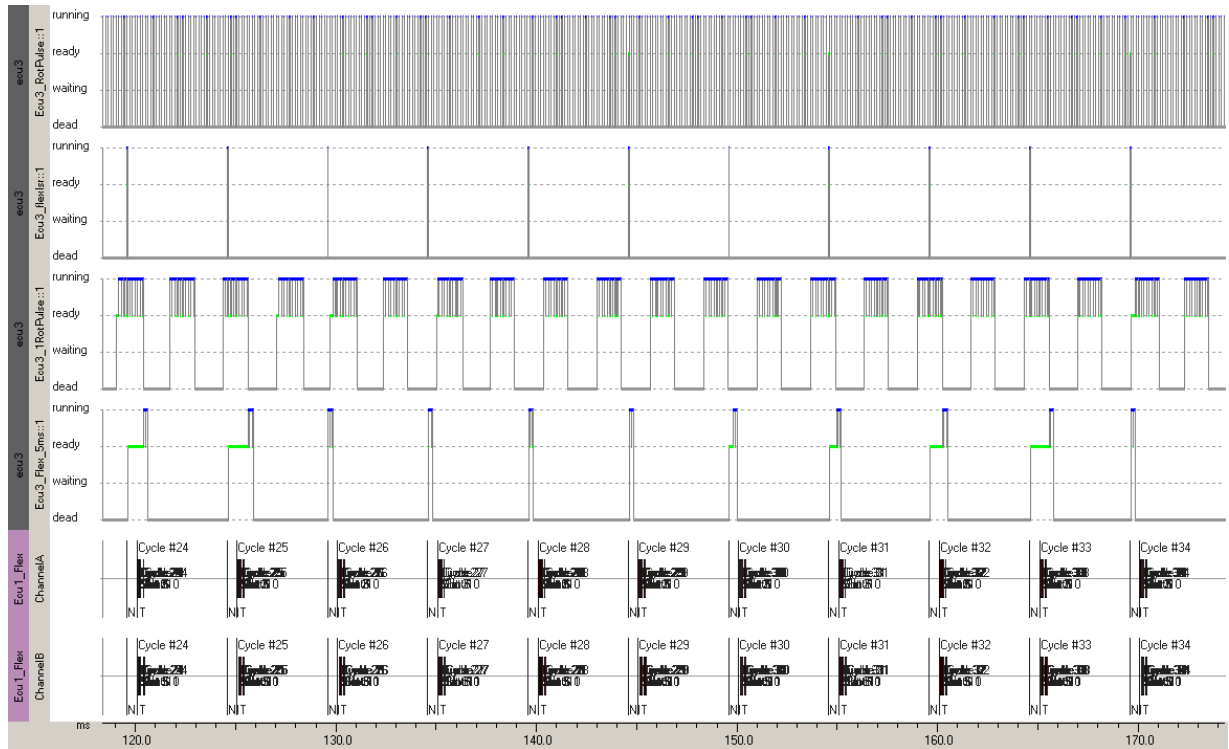


Figure 6 State Diagram of FlexRay Communication (ChannelA, ChannelB), ISRs, and Tasks for the Phase from 159ms to 169.5ms Simulated Real-Time

