REAL-TIME IMPLEMENTATION

Ulrich Kiffmeier
dSPACE GmbH, Technologiepark 25, 33100 Paderborn, Germany.

Keywords: Code generation, Deadline, Digital controllers, Discrete-time systems, Distributed systems, Embedded systems, Fixed-point arithmetic, Implementation, Interrupts, Interprocessor communication, Intertask communication, Jitter, Microcontroller, Multitasking, Real-time systems, Real-time operating systems, Safety-critical systems, Scheduling, Time-triggered systems.

Contents

Summary

Discrete-time digital controllers require real-time execution when connected to a real-world plant, i.e., the controller task must be activated at fixed points in time and finished within a given deadline. When designing such a system, the control engineer must be aware of effects resulting from the finite processing power of the controller hardware and I/O units. Computational delays and jitter must be considered carefully when designing a real-time system, because they introduce varying time delays into the system, leading to a loss of control quality or even instability, and severe physical damage of the controlled system.

In many cases, control applications require only one simple real-time task. The first part of this article describes basic implementation techniques for such simple systems, including a discussion of numerical integration algorithms suitable for real-time execution, and a comparison of fixed-point versus floating-point arithmetic.

Applications with more than one real-time task require a Real-Time Operating System (RTOS) to manage task scheduling and pre-emption. For maintainability and security reasons, the task structure of a multitasking application should be kept as simple as possible. The intertask communication must be predictable with known delays and

minimal jitter.

In advanced control systems, several dedicated controllers are interconnected by means of a communication network, thus forming a distributed embedded system. To solve the problem of reliable interprocessor communication arising in distributed real-time systems, the time-triggered architecture for safety-critical applications is described, where the communication schedule is completely known *a priori* to all nodes.

An important trend for the future will be the increasing use of code generators, which automatically convert a high-level graphical specification of a control system into an executable real-time program. Furthermore, object oriented design methods based on the Unified Modeling Language (UML) are becoming increasingly important. These novel tools help to cope with the complexity of modern control systems and to accelerate the development process for large real-time applications.

1. Introduction

The mathematical foundations of discrete-time digital controllers require that the algorithm is executed exactly at predefined points in time. When implementing a controller on a real-world computer and connecting it to a real-world plant, it is hard to fulfill this timing requirement. The developer of such a system must be aware of the fact that this algorithm may be invoked at inaccurate times, cannot be executed infinitely fast, and may be interrupted by other high-priority computations. Inaccurate timing introduces all kinds of delays into the control system that may lead to unexpected behavior, or even instability. The basic effects of real-time execution and real-time communication will be discussed in this article, including practical implementation methods to cope with problems arising in this area.

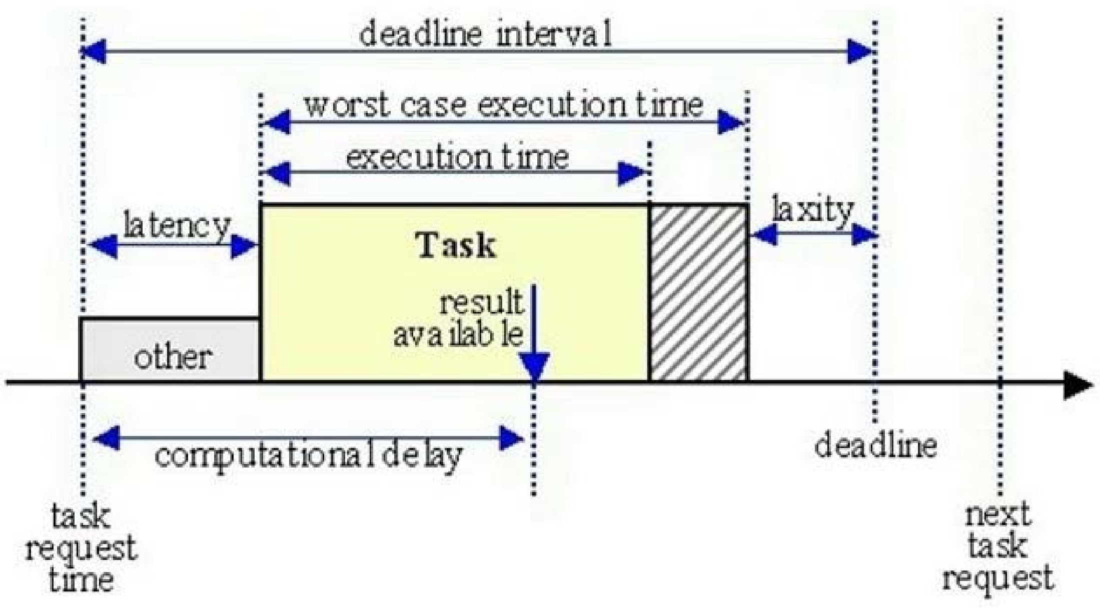


Figure 1. Notations describing a real-time task.

Figure 1 introduces some basic notations, which are commonly used to describe the timing of real-time systems. A task is the execution of a sequential program under the control of an operating system. The task request time is the point in time when the execution of a task is initiated. For a discrete-time digital controller, the task request times have a fixed period, which is equal to the sample time of the controller. The latency is the time span from the initial task request until the processor jumps to the requested task, which includes the time necessary to finish other high-priority computations and to perform the context switch. At some point of the running task, the results will be available, i.e., the actuators of the control system are updated. The time from the task request until the availability of the results to the outside world is called computational delay. The computational delay should be considered carefully when designing the control system. At the end of the task, some cleanup actions are performed. The total time required to perform all computations of a task is called execution time. The execution time may vary from one task invocation to another if some actions are performed conditionally, i.e., the program takes different execution paths. The resulting worst case execution time (WCET) is another important property characterizing a task.

The real-time execution of a task is determined by a deadline, which specifies the point in time when a real-time task must definitely be finished. This requirement is called hard real-time constraint, in comparison to soft real-time systems where deadlines must only be met on average. The following focuses on hard real-time systems only.

Note that, for a periodic task, the deadline interval must be less than the sample time to leave some time for other tasks and perform the necessary context switches. The time span from the end of a task until the deadline is called laxity. One design goal for a proper real-time implementation will be to maximize the laxity.

The first part of this article concentrates on simple applications with only one real-time task executing a control algorithm. The discussion includes the effects of execution time jitter, numerical integration algorithms suitable for real-time applications, and a comparison of fixed-point versus floating-point arithmetic. The second part of this article introduces real-time operating systems (RTOS) for multitasking applications, including interprocessor communication, and distributed embedded real-time systems. Finally, novel programming techniques, like automatic code generation and UML based software design methods are discussed, which will significantly change the development process for real-time software in the future.

2. A Simple Real-Time System

It is not always necessary to run a control algorithm in a full real-time operating system environment. For many small applications with only one real-time task, a simple implementation scheme is sufficient. The heart of such a minimal real-time program is a so-called interrupt service routine (ISR), which is invoked periodically by the timer of the controller hardware (see Figure 2). ISRs are similar to real-time tasks, but they are not under the control of an operation system. When a timer interrupt is received, the processor jumps to the ISR and computes one step of the control algorithm. After finishing the ISR, the processor returns to the interrupted background process and continues computations there.

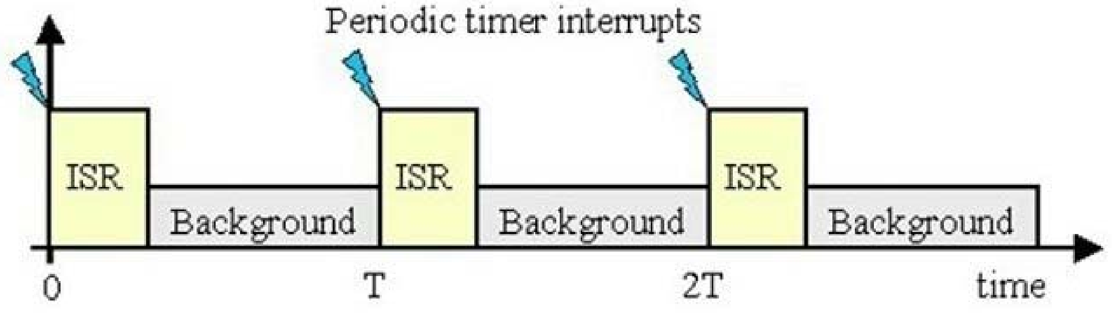
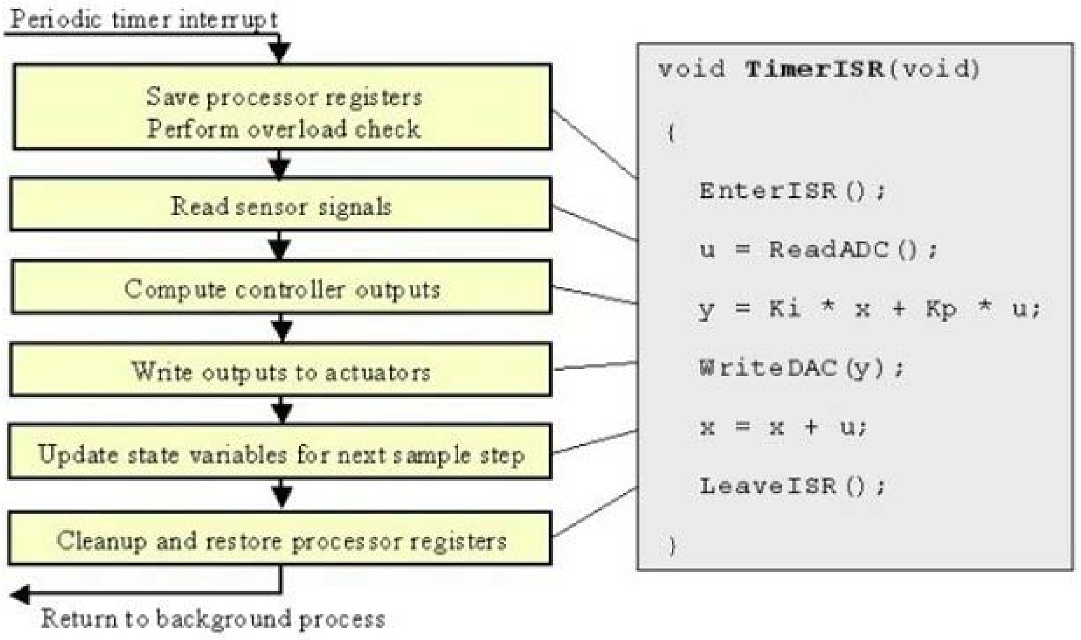


Figure 2. Timing diagram of a simple real-time program.


Periodic timer interrupt
Save processor registers
Perform overload check
Read sensor signals
Compute controller outputs
Write outputs to actuators
Update state variables for next sample step
Cleanup and restore processor registers
Return to background process

```
void TimerISR(void)
{
  EnterISR();
  u = ReadADC();
  y = Ki * x + Kp * u;
  WriteDAC(y);
  x = x + u;
  LeaveISR();
}
```

Figure 3. Typical real-time interrupt service routine (ISR).

Figure 3 shows an ISR in greater detail with the typical sequence of commands to be performed for a control application. As an example, the right side of Figure 3 shows the C code which implements a simple PI controller with the state equations

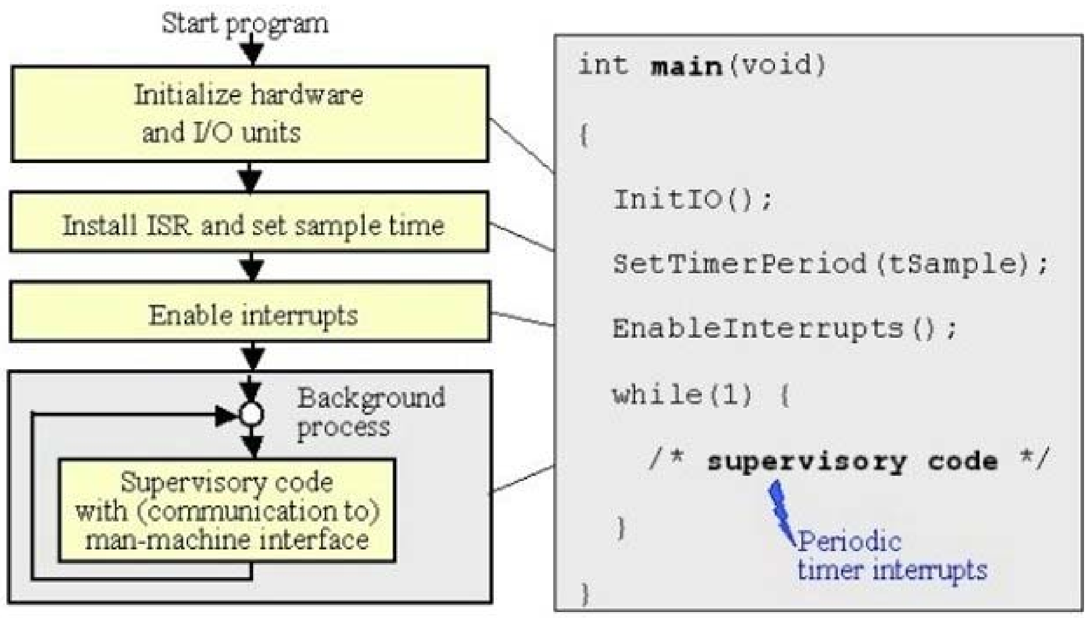
$$x(k+1) = x(k) + u(k)$$
$$y(k) = K_I x(k) + K_P u(k)$$

One of the first actions performed in the ISR is the overload check. The processor is overloaded if the ISR is still active at the time it is invoked again by the next interrupt. This means that computations could not be finished within the given deadline. Such a fatal violation of the real-time constraint must be signaled to the supervisory code by setting an overload flag.

After reading the sensor signals and computation of the control algorithm, the outputs are immediately written to the actuators. This is done as early as possible to minimize the delay between the interrupt and the availability of the results. The ISR is completed by updating the state variables of the controller for the next sample step. Finally, some cleanup actions are performed and the processor registers are restored before leaving the ISR.

So far, only the real-time portion of the application was discussed. The full program also consists of a startup procedure and the so-called background process, which is executed while the ISR is not active (see Figure 4). At program start, the main() function first initializes all I/O units and sets the period of the timer interrupts equal to the sample time. After finishing all initializations, interrupts are enabled to start the real-time execution of the control algorithm. While the background process is periodically interrupted by the ISR, it executes the supervisory code in an endless loop. The supervisory code is responsible for reporting error conditions and for exchange of non-time-critical data with the man-machine-interface, which may be located on another node in the network.

The man-machine-interface visualizes signals from the real-time system and manages user commands like stopping and restarting the controller by sending appropriate signals to the real-time system. It may also provide instruments to modify controller parameters within given limits. Note that the controller states must be re-calculated properly when the parameters are changed at run-time to enable bumpless transitions. Also, changing a parameter set at run-time must be synchronized with the controller task. Typically, the complete new parameter set is provided in a buffer, and the controller just sets a pointer to the active buffer at the beginning of the real-time task.


Start program
Initialize hardware and I/O units
Install ISR and set sample time
Enable interrupts
Background process
Supervisory code with (communication to) man-machine interface
Periodic timer interrupts

```
int main(void)
{
  InitIO();
  SetTimerPeriod(tSample);
  EnableInterrupts();
  while(1) {
    /* supervisory code */
  }
}
```

Figure 4. Typical main() function and background process.

3. Computational Delay and Jitter

On real-world hardware, a control algorithm cannot be executed infinitely fast. There will be a time delay between the beginning of a sample interval, i.e., the activation of the ISR, the reading of sensors and the final update of the actuators. This time span is called the computational delay.

The computational delay depends on the processor speed and the complexity of the control algorithm. From the control engineers perspective, it introduces a phase shift into the system, as shown in Figure 5. For some applications, the computational delay might be negligible, but for others it is crucial to take this effect into account from the beginning when the controller is designed. Otherwise the control quality suffers, or the system might even become unstable in critical cases. Normally, the computational delay will be incorporated in the actuator model and regarded as a part of the plant model when calculating the controller parameters.

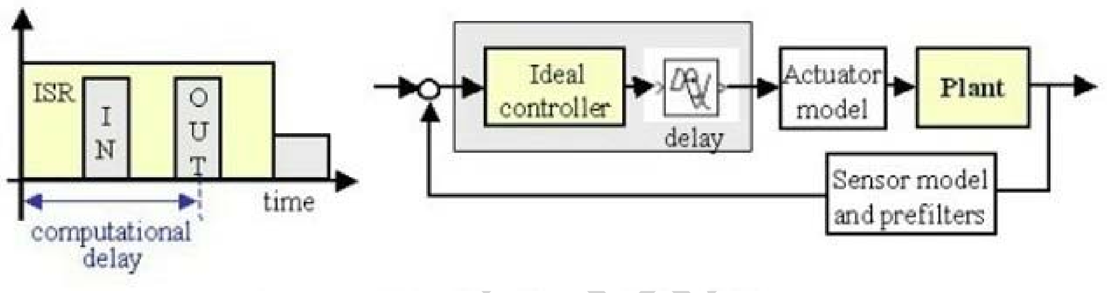


Figure 5. Controller with computational delay.

It is difficult to predict the computational delay for a control algorithm on given target hardware. It depends on many parameters, including processor speed, memory chips, caching effects, compiler optimizations, etc. Although there are some utilities available to calculate execution times for a given hardware architecture, the most practical approach is still to measure the computational delay directly, for example, by reading the timer, or using an oscilloscope. If the computational delay is significant compared to the time constants of the system, the controller parameters must be re-calculated assuming an ideal controller and regarding the delay as part of the plant model (Figure 5).

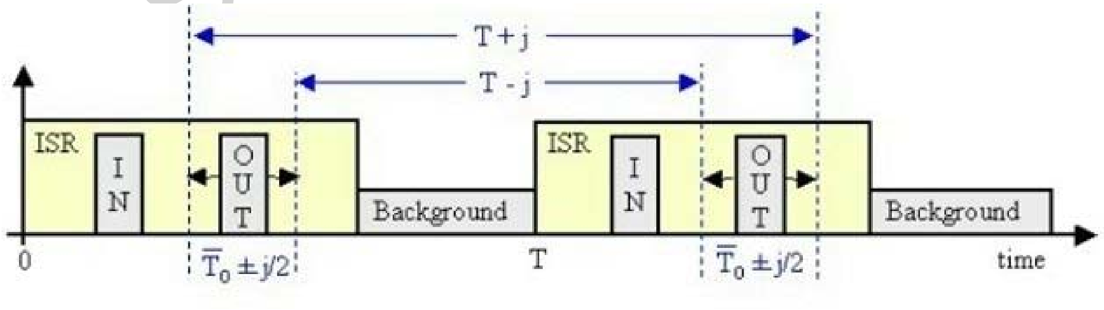


Figure 6. Timing diagram with jitter of input and output statements.

The situation becomes even worse if the program contains parts that are executed conditionally, for example, if different controller modes are selected depending on the operating point. Due to the varying execution time when the program takes different execution paths, there is a jitter in the time between two sensor or actuator updates as shown in Figure 6.

T	Sample time of the controller.
$\bar{T}_O$	Mean execution time from the beginning of the ISR until the update of the outputs.
j	Difference between the maximum and minimum value of $\bar{T}_O$.

Note that the time between two output updates varies from $T - j$ up to $T + j$, i.e. the maximum difference is $2j$. The following numerical example shows the importance of the jitter effect:

Suppose a signal with 10 V amplitude and 1 kHz frequency is sampled with a 16-bit A/D converter. The rise time of the signal is

$$\left.\frac{dy}{dt}\right|_{\max} = \left.\frac{d}{dt}A\sin(\omega t)\right|_{\max} = A\omega = 62\frac{mV}{\mu s} \qquad (1)$$

and the resolution of the A/D converter is

$$r = \frac{2A}{2^n} = \frac{20V}{65536} = 0.31mV \qquad (2)$$

which means that with a jitter of 1 μs, the uncertainty of the measurement is 200 times the resolution of the A/D converter. In other words, roughly 8 bits of precision are lost in the worst case!

For a control algorithm without direct feedthrough ($K_P = 0$), the output jitter can be significantly reduced by writing the outputs at the beginning of the next sample interval, i.e., by pre-computing the controller output for the next sample step at the end of the current step.

With this implementation policy, the output jitter is independent of the execution time of the control algorithm.

It should be mentioned here that a prerequisite for good control quality is an accurate sensor interface. If the sensors are exposed to deterministic or stochastic disturbances, analog pre-filters, anti-aliasing filters, or over-sampling methods should be applied to reduce the noise and to remove high frequency disturbances from the input signal before using it in the control algorithm. When initializing the controller, the filter states should be initialized properly. For example, if a temperature sensor is connected to a low pass filter, the low-pass state variable should be initialized with the current temperature rather than with zero.

```
void TimerISR(void)
{
   EnterISR();
   WriteDAC(y);
   u = ReadADC();
   x = UpdateEquation(x,u);        /* compute x(k+1)
*/    y = OutputEquation(x);        /* compute y(k+1)
*/
   LeaveISR();
}
```

-
-
-

Bibliography

Booch G., Rumbaugh J., Jacobson I. (1999). *The Unified Modeling Language Users Guide*. Addison Wesley, Reading, Ma. [Introduction to the Unified Modeling Language (UML) for object oriented software design. Useful for high-level system design and behavioral modeling].

Burns A., and Wellings A.J. (1989). *Real-Time Systems and their Programming Languages*. Addison Wesley, Reading, Ma. [Textbook providing an introduction into real-time systems].

Hanselmann H., Kiffmeier U., Köster L., and Meyer M. (1999) *Automatic Generation of Production Quality Code for ECUs*, SAE International Congress 99, March 1-4, Detroit, USA, (SAE Technical Paper Series 1999-01-1168). [Discusses the requirements for production-ready real-time code generation for embedded automotive control units].

Hoare C.A.R. (1985). *Communicating sequential processes.* Englewood Cliffs, Prentice Hall, USA. [Textbook on concurrent computation including a description of *monitors*].

Kopetz H. (1997). *Real-Time Systems. Design Principles for Distributed Embedded Applications.*Kluwer Academic Publishers, Boston, USA. [Standard textbook with a focus on hard real-time systems and distributed embedded systems. Provides a good introduction to time-triggered systems].

Köster L., Thomsen T., and Stracke R. (2001) *Connecting Simulink to OSEK: Automatic Code Generation for Real-Time Operating Systems with TargetLink*, SAE International Congress 2001, Detroit, USA (SAE Technical Paper Series 01PC-117). [Describes an approach for automatic code generation from Simulink block diagrams with connection to the OSEK operating system].

Mathai J. (Ed.) (1995). *Real-time Systems: Specification, Verification and Analysis*. London: Prentice Hall International.. [Collection of articles about Real-Time System design, including scheduling problems].

MISRA (1994) *Guidelines for Vehicle Based Software*, Nuneaton, UK: Motor Industry Software Reliability Association. [Contains a set of rules for software development for safety critical real-time systems. See also: http://www.misra.org.uk/.]

MISRA (1998) *Guidelines for the Use of C Language in Vehicle Based Software*, Nuneaton, UK: Motor Industry Software Reliability Association. [Contains a set of rules for C programming for safety critical real-time systems. Widely accepted as a standard].

Open Systems and the Corresponding Interfaces for Automotive Electronics. *OSEK/VDX Specification Vs. 2.1*, Karlsruhe, Germany, http://www-iiit.etec.uni-karlsruhe.de/~osek/main.html. [A scalable RTOS which has become a standard for automotive applications. Available for most relevant microcontrollers. An extension for time triggered systems is under preparation].

Sha L., Rajkumar R., and Lehoczky J.P. (1990). Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*. **39 (9)**, 1175-1185. [Introduces the *Priority Ceiling Protocol* that helps to avoid deadlocks through priority inversion in multitasking systems].

The MathWorks (2000). *Simulink User's Guide*, The MathWorks, Nattick, MA USA. [A block-diagram based tool to model and simulate control systems. Accompanied by the Real-Time Workshop, a code generator for Simulink block diagrams used for rapid controller prototyping].

Tsai J.J.P., Bi Y.., Yang S.J.H., and Smith R.A.W. (1996). *Distributed Real-Time Systems. Monitoring, Visualization, Debugging and Analysis*. John Wiley & Sons, New York, USA. [Textbook with a focus on monitoring of distributed embedded real-time systems].

Biographical Sketch

Ulrich Kiffmeier studied control engineering at the Ruhr-University Bochum, Ulrich Kiffmeier worked for 5 years as a Research Assistant at the Automatic Control Lab in Bochum, where his research interests were in the area of robust control and computer aided control engineering. After receiving his Ph.D. in 1994, he joined dSPACE where he was responsible for real-time code generation for embedded multiprocessor systems. He also headed up several hardware-in-the-loop simulation projects. Since 1997, Ulrich Kiffmeier has worked as a team leader in the development of the production code generator TargetLink, which is focused on highly efficient C code generation for automotive control units.