Connecting Simulink to OSEK: Automatic Code Generation for Real-Time Operating Systems with TargetLink

Lutz Köster, Thomas Thomsen, Ralf Stracke
dSPACE

*Translation of
"Von Simulink nach OSEK:
Automatische Codegenerierung für Echtzeitbetriebssysteme
mit TargetLink",*

*published on the occasion of
the congress Embedded Intelligence 2001,
Nuremberg, Feb 14 - 16.*

Connecting Simulink to OSEK: Automatic Code Generation for Real-Time Operating Systems with TargetLink

Lutz Köster, Thomas Thomsen, Ralf Stracke
dSPACE

.

ABSTRACT

This paper describes how one further step towards integrating the complete software development process of embedded real-time systems in Simulink and Stateflow is achieved. Automatic code generation in production quality is already possible with dSPACE's TargetLink. The next step is true integration of real-time operating systems into the code generation process. The OSEK standard is applied for this purpose and briefly described at the beginning. The article then focuses on concepts and techniques used by TargetLink to achieve OSEK-compliant code generation. It is shown, how operating system functions and properties are specified on a block diagram level, how code of different sample rates is bundled into different tasks, how tasks can communicate among each other and how all this can be simulated on a target system. The paper also gives striking evidence of how powerful standards can boost tool development and how they can help integrate different tool systems, which had been completely separated before.

INTRODUCTION

The software development of embedded systems is increasingly being done with the help of simulation tools that are programmed with executable block-diagram specifications. One of the most popular tool suites is MATLAB, Simulink and Stateflow from The MathWorks, which became the de-facto standard in the embedded systems industry. In 1999, dSPACE introduced TargetLink to the market, which extends Simulink and Stateflow with a production quality code generator. TargetLink is able to produce ANSI-C code and processor/compiler-specific C code that is highly readable and small enough for resource-limited, embedded real-time systems. Since then, automatic code generation with TargetLink has been proven in a wide range of control applications. dSPACE now widens the scope of code generation with the next version of TargetLink by supporting OSEK-compliant real-time operating systems.

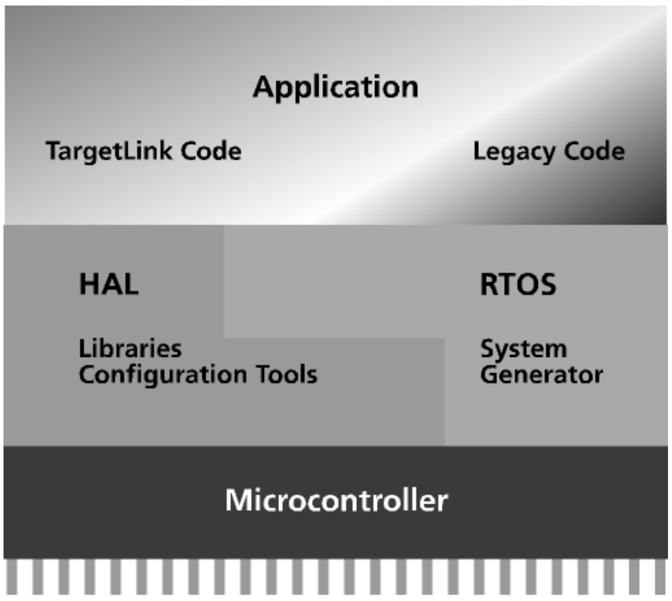


Figure 1: ECU software layers

Modern ECUs, as they are used in automobiles and aircrafts, typically contain 3 distinguished software layers (see Figure 1). The highest layer is the application code and contains signal processing, control strategies, failsafe and diagnostic functions. Simulink and Stateflow are perfectly suited to specify and simulate these functions. TargetLink then can be used to automatically generate production code.

The lowest level is the hardware abstraction layer (HAL), which contains basic I/O routines, interrupt service routines, device drivers and communication handlers. Block diagram specifications do not offer significant advantages for this layer - they are directly programmed in C or assembly. Many of these routines are readily available from external sources or are generated with the help of processor-specific configuration tools.

In-between these layers is the real-time operating system (RTOS), which is responsible for scheduling tasks and managing intertask communication and processor resources. Real-time operating systems for embedded systems are commercially available from a variety of vendors. They, too, typically come with a configuration tool and a system generator that automatically builds the real-time kernel according to the user's specification. In recent years, the OSEK standard emerged to standardize the Application Program Interface (API) and functionality of real-time operating systems. This makes it possible to connect code generators to kernel builders and to make the generated code compatible to the real-time kernel.

OSEK: GOALS AND CONCEPTS

The goal of the OSEK specification is to support the portability and reusability of application software. To achieve this goal, the OSEK working groups defined abstract and application-independent user interfaces in the areas of the real-time operating system, communication and network management. These interfaces are also independent of hardware and network. Different applications require different features of a real-time operating system. Small microcontrollers have only small amounts of memory, so ROM and RAM usage must be very economical. Bigger systems may need special features of the operating system and the limitations concerning memory usage may be less stringent. These different cases demand a scaleable and configurable real-time operating system.

The OSEK specification provides a pool of services and processing mechanisms, but leaves a certain amount of flexibility to allow optimal adaptation to different microcontrollers.

For clarity it shall be repeated that OSEK itself is not an operating system product. It only specifies a certain API and behavior. There are several commercial OS products available on the market that are compliant to the OSEK standard. Hereafter, they are referred to as OSEK operating system implementations.

The main benefits of the OSEK standard are:

- Savings in costs by reducing development time.
- Enhanced software quality because well-tested software modules can be reused in different applications.
- Because OSEK is scalable, the specification of the operating system allows optimal adjustment to the specific application and thus reduces memory usage in small systems.
- The portability and reusability of application software for different hardware platforms and possibly also for different OSEK-compliant real-time operating system implementations.

The OSEK specification describes a static real-time operating system where all operating system objects such as tasks, events, messages and resources are created at compile time. These objects are defined, and their attributes are described offline with the help of the OSEK Implementation Language (OIL). OSEK defines several mandatory standard attributes - like the priority of the object “TASK” - which must be implemented by all OSEK operating system implementations. Some of them are defined to ensure scalability of the operating system. However, the OSEK operating system developer is allowed to define additional implementation specific attributes for the OSEK objects.

The application software can be subdivided into tasks according to their real-time requirements. For memory efficiency reasons, OSEK distinguishes between basic and extended tasks. Basic tasks can assume the states ‘running’, ‘ready’ or ‘suspended’. A task is running if the processor is assigned to it. At any one point in time, only one task can be in the running state. A task is ready, if all the requirements for transition to the running state are met, but the CPU is assigned to another task. The scheduler decides when a ready task becomes a running task. The ready task can interrupt a running task if the priorities and preemptibilities allow this; otherwise, it has to wait until all tasks with higher priorities have finished. In the suspended state, the task is passive. Before it can enter the ready state, it must be activated. In addition to basic tasks, extended tasks can enter a ‘waiting’ state where they wait for at least one event before continuing execution.

The functionality of the operating system is subdivided into different conformance classes to ensure scalability. In the two basic conformance classes, only basic tasks can be used. The two extended conformance classes allow the usage of basic and extended tasks.

An OSEK operating system uses priority-based scheduling. In addition, the priority ceiling protocol is included for resource handling. The use of the OSEK object resource is similar to the use of semaphores in other operating systems, except that the priority ceiling protocol avoids deadlocks during resource occupation. Resources can be used to manage concurrent accesses of tasks and interrupt service routines with different priorities to shared resources like memory or hardware.

OSEK defines counters and alarms to process recurring events. Alarms are assigned to counters that count time ticks or other recurring signals such as angle-based encoder interrupts. An alarm triggers an event or activates a task whenever the counter reaches a predefined value. When the alarm is specified as being cyclic, the associated task is called periodically, for example, at certain sample times or certain crankshaft angles. Single alarms can be used, for example, to trigger a task some defined time after a certain event has occurred.

OSEK also defines message-based communication between tasks (intertask communication). Messages can be sent between tasks inside one ECU or they are used for communication between tasks on different ECUs.

For extended tasks the event mechanism can be used for task synchronization. An extended task can wait for an event until it is set by another extended task, a basic task, an alarm or an ISR (interrupt service routine). After the event is set, the waiting task leaves the ‘waiting state’ and enters the ‘ready state’.

The processing of interrupts is supported in OSEK by different categories of ISRs. Category 1 ISRs are not allowed to call any OSEK API function. They are absolutely independent of the operating system. ISRs of category 2 are allowed to call API functions at any place inside the ISR, whereas category 3 ISRs are divided into two sections: in the first section no calling of any API function is allowed, in the second section it is. This second section must be nested by the special macros *EnterISR()* and *LeaveISR()*.

MODELING REAL-TIME (OSEK) APPLICATIONS IN SIMULINK / TARGETLINK

GENERAL MODELING STYLES

In recent years, Matlab/Simulink/Stateflow has become a standard tool for modeling and offline simulation in control design. It is optimally tailored to the needs of the control engineer and provides a powerful set of block libraries and toolboxes. The range of applications where Matlab/Simulink/Stateflow is used is constantly growing. In addition to the classic offline simulation of dynamic systems, the tool is used as the specification basis for all phases of the electronic control unit's modern development cycle. As a graphical and executable specification, it replaces the classic, verbally-oriented approaches. With this extended usage come new requirements that formerly were not important. In conjunction with highly efficient automatic production code generation, Matlab/Simulink/Stateflow now is used to describe the complete application software. Here the real-time operating system plays an important role, and OSEK is becoming increasingly popular in the automotive field. The new version of dSPACE's code generator TargetLink provides seamless integration of OSEK-conformant operating systems into the Matlab/Simulink/Stateflow environment and the code generated from it. In contrast to many CASE tools that have their origin directly in software development, where tasks, messages, resources, etc. are a natural part of the world, the classic domain of Matlab/Simulink/Stateflow is function design, which has a completely different abstraction level. Automatic code generation for real-time operating systems out of Matlab/Simulink/Stateflow has to deal with the problem of mapping Simulink's modeling concepts onto available operating system services and providing new specification options - e.g. by new blocks - to support important operating system features. The code generator must be able to produce a proper real-time implementation for an unchanged Simulink model by a well-considered default behavior, and provide all the options necessary to optimally adjust the code to particular user needs.

A real-time application based on a (multitasking) real-time operating system (RTOS) is divided into different software units that are called by the RTOS. These tasks usually combine parts of the software with common real-time requirements such as timing, priority etc. They can be called by the RTOS automatically at certain times, or activated by hardware or software events. Simulink models initially do not contain tasks but are built up from other components. The code generator has to map these structures onto the RTOS software units.

A Simulink model usually is divided into hierarchical subsystems. The functional blocks of these subsystems are executed either cyclically with a certain sample rate or driven by certain events. While Simulink generally allows blocks with different sample rates to be mixed arbitrarily, production code generation reasonably requires that blocks with the same sample rate are combined to form system units.

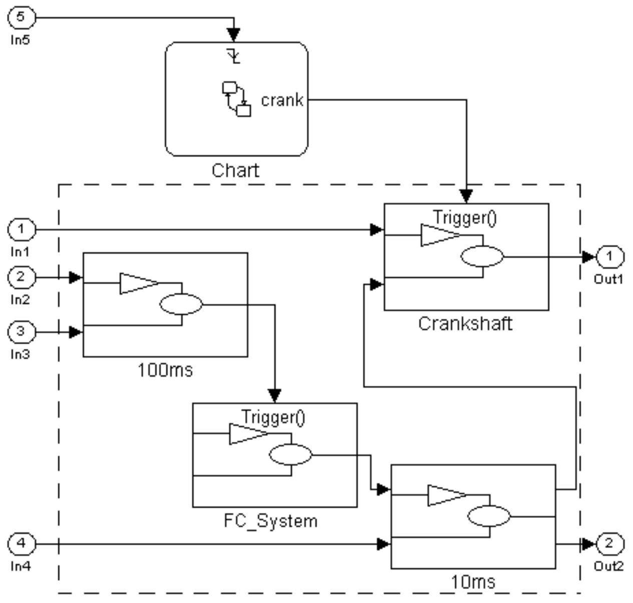


Figure 2: Typical Simulink/Stateflow diagram

Figure 2 shows a typical Simulink/Stateflow diagram that contains cyclic and event-driven parts. In an engine control application, for example, the electric throttle control algorithm executes periodically and the ignition/injection part is driven by crankshaft events. Generating code implementing this model as a real-time (OSEK) application contains the following steps.

The 10-millisecond and the 100-millisecond subsystems are implemented as separate operating system TASKS. The corresponding functions in the C code must be extended by the necessary task declarations. The RTOS must be configured in such a way that these tasks are called periodically with the specified rate. In an OSEK application, this is realized by alarms that are defined, assigned to timers and initialized properly at system startup.

The crankshaft synchronous subsystem is also implemented as a separate task. It usually is assigned directly to a corresponding hardware event. The statechart in Figure 2 only serves to generate the appropriate trigger in the offline simulation. It has to be ignored by the code generator. The user therefore places the statechart outside the focus of the code generator, thus no code will be generated for the statechart. This is an important concept. Simulink models usually contain parts that specify the controller and additional parts that are only needed for simulating the controller, e.g. plant subsystems for closed-loop simulations. The code generator only works on the controller part of the model. In Figure 2, this is marked by the dashed rectangle.

Generally, asynchronous functions are modeled as triggered subsystems. If triggered subsystems are implemented as separate tasks, they must be activated by the source which releases the trigger. *FC_System* in Figure 2 is activated by code that is automatically generated into task *100ms.* If the trigger comes from outside the scope of the code generator, as for *Crankshaft*, the user must assign task activation to the proper source manually, e.g. to a hardware interrupt.

The data connections between the subsystems/tasks must be handled carefully by the code generator. It must be considered that tasks may interrupt each other requiring that some data accesses must be protected to avoid inconsistencies. As a result, the code generator sometimes creates local copies of global data for further processing. Data connections that cross the border of the code generator's scope - e.g., to pass data to hand written code within the same ECU or to send data via a bus to different ECUs - must be established, for example, by operating system messages or by service routines that access data resources.

Finally, an offline description of the application has to be generated as the input to the tool that builds the tailored real-time kernel. This is done via the OIL file (OSEK Implementation Language), which is part of the OSEK specification.

The following sections describe the different steps mentioned above in greater detail.

TASKS

As described in the previous section, the Simulink model is partitioned into tasks. If the user makes no specific input, TargetLink uses a default partitioning. However, by means of special blocks (the task block, which is described later), the user can completely determine which subsystems are assigned to which tasks.

By default, TargetLink groups together all the model's periodic subsystems that have the same sample rate in separate tasks. Triggered subsystems with common trigger sources are combined, and their code is assigned either to the trigger source's task where this is within the code generator's scope (*FC_System* in Figure 2 is associated to task *100ms*) or to a separate task. (*Crankshaft* in Figure 2).

By placing a special *task block* in a subsystem, the code generator's default task partitioning can be overruled. The task block explicitly assigns a subsystem to a certain task. *FC_System* in Figure 3 will now be implemented as a separate task. Its trigger source in *100ms* will not directly call the code of *FC_System* any more, but will only produce the OSEK 'activateTask' command for the corresponding task; when it will be scheduled depends on task priorities and preemptibilities.

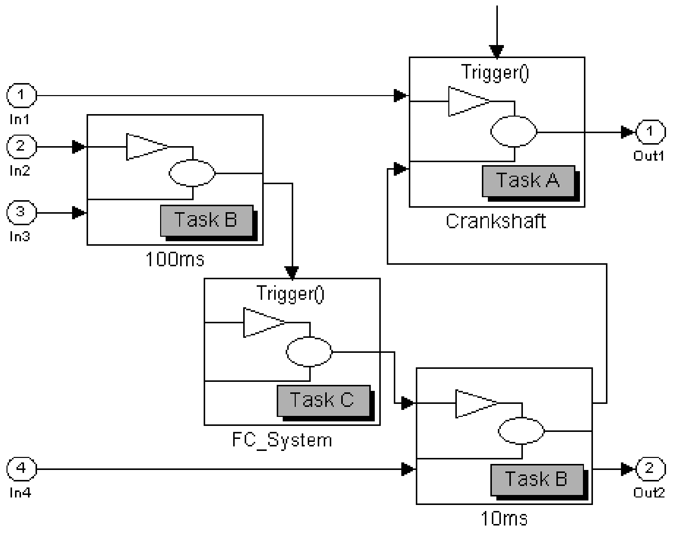
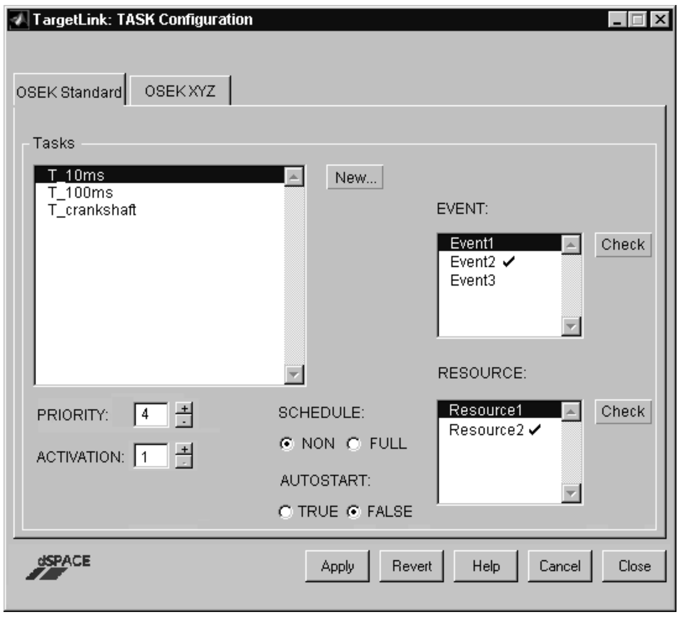


Figure 3: Task block to define separate tasks

As shown in Figure 3, it is also possible to assign several subsystems to one common task. The code for *100ms* and *10ms* will now be included in the common *Task B*. This combined task is scheduled periodically at ten milliseconds, which is the highest common denominator of both the sample rates involved. A counter encapsulates the code of *100ms* and executes it upon execution of every 10th task. Note, that the worst case execution time must be less than 10 ms. Therefore, in most cases only subsystems with the same sample rate are assigned to the same task.

The possibility of combining subsystems distributed over the model to create common tasks allows a feature-oriented modeling style. It is not necessary to partition the model at the highest level with regards to sample rates or membership in certain events. Instead, the model can be divided into parts belonging to the same high level functionality. An electronic control unit usually contains dozens of so-called *features* (e.g., throttle control, idle speed control, EGR, injection, etc.) that are developed by different teams. They form separate subsystems at the topmost model level. Any of them may contain parts that are executed at different sample rates. Bundling these parts (processes) of the same sample rate to one task avoids scheduling overhead.

The task block not only serves as a mean to assign subsystems to certain tasks, but also provides an interface to specify attributes of the corresponding task. Figure 4 shows the dialog which opens when the task block is double-clicked. The current subsystem can be assigned to an existing task, or a new task can be created. The standard OSEK attributes such as priority, number of activations, full preemptive or non-preemptive, used resources or owned events, etc. can be specified. Some of these settings influence only the generated OIL description and not the actual C source code, others - such as priority and schedule - are used by the code generator to optimize the code, e.g., for data exchange, which will be shown in a later section of this paper. Figure 4 shows two tabs in the task block's dialog: *OSEK Standard* and *OSEK XYZ*. OSEK itself requires a set of attributes that each RTOS vendor has to provide, but allows additional implementation-specific settings. For several commercially available operating systems, these specific options can also be selected from TargetLink via the *OSEK XYZ* tab of the task dialog. However, this does not limit the use of TargetLink for these supported OSEK products. A later section will show how TargetLink can interact with any other desired products and the corresponding operating system configuration tools.


TargetLink: TASK Configuration

OSEK Standard | OSEK XYZ

Tasks: T_10ms, T_100ms, T_crankshaft

EVENT: Event1, Event2 ✓, Event3

RESOURCE: Resource1, Resource2 ✓

PRIORITY: 4

ACTIVATION: 1

SCHEDULE: [✓ NON] FULL

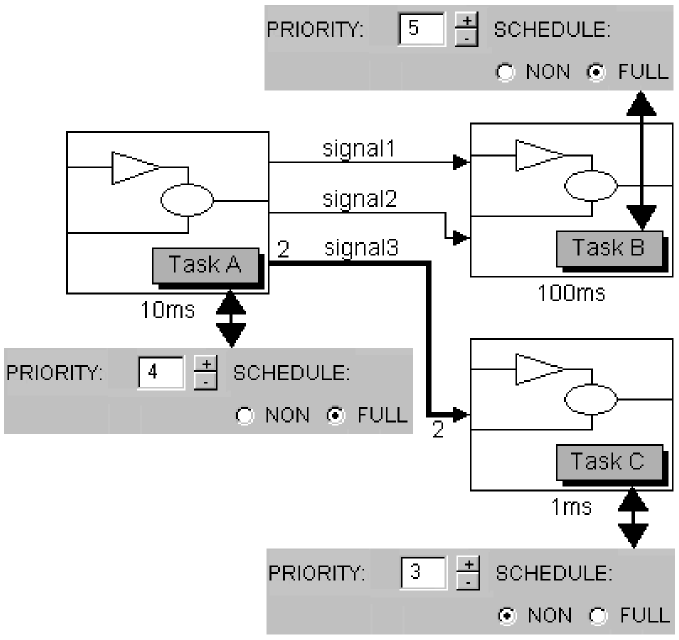
AUTOSTART: TRUE [✓ FALSE]

Figure 4: Task block dialog

When tasks are created, either by the code generator's default mechanism or by explicit user requests, the necessary declarations are generated into the C source code (e.g. `TASK(T_100ms) { ...`), and the task activation itself is initiated. Activation of periodic tasks is assigned to automatically generated OSEK alarms that are initialized at system startup. Triggered tasks are activated by code that is included in the task of the trigger source. If this source is outside the scope of the code generator, a piece of code (macro) that performs this activation is generated, and the user must assign it to the trigger source, e.g., an interrupt.

INTERTASK COMMUNICATION

In Simulink, data exchange is modeled by signal lines. In a task-driven preemptive environment, data exchange becomes a more complex issue. One important job of an RTOS is to manage the communication between tasks. Inter-ECU communication is data exchange between software distributed to different physical hardware units and usually is handled by some kind of network layer. Intra-ECU communication is connections between tasks running on the same processor. While the former always requires operating system services (OSEK messages), the latter can optionally be implemented without OS interaction.


Task B: PRIORITY: 5, SCHEDULE: NON [✓ FULL]

Task A: PRIORITY: 4, SCHEDULE: NON [✓ FULL]

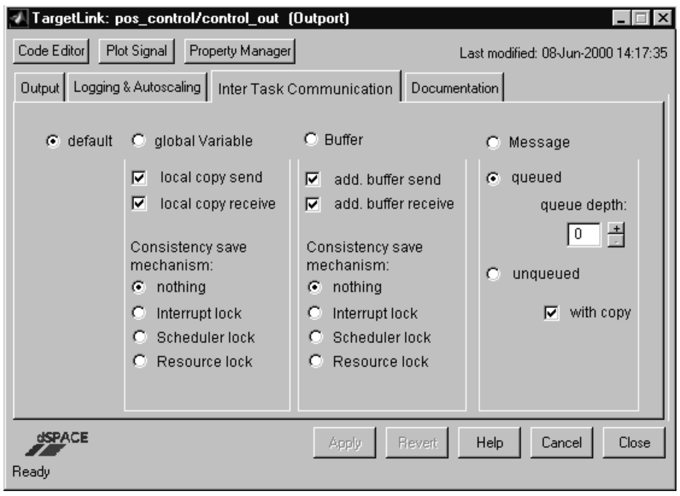
Task C: PRIORITY: 3, SCHEDULE: [✓ NON] FULL

Figure 5: Communication between Tasks

Figure 5 shows a scenario where tasks with different sample rates, priorities and preemptibilities exchange data. *Task A* sends *signal1* and *signal2* to *Task B*. Global variables that can be accessed by both tasks are usually used for this data exchange. Since *Task A* is preemptive (in OSEK terms SCHEDULE = FULL) and *Task B* has a higher priority, it may happen that *Task A* is interrupted after *signal1* has already been updated, but while *signal2* still has its old value. Consequently, *Task B* would work with values for *signal1* and *signal2* from different time steps. This inconsistency can lead to unwanted behavior and normally has to be avoided. In this example Task A is responsible for implementing a mechanism that is suited to protect against inconsistencies. One possible way is to use local copies for signal1 and signal2. At the end of Task A there is one compact block of code that copies this local representation to the corresponding global variables. It still has to be ensured that Task A is not interrupted during this copying procedure. This can be done by disabling interrupts or by using the OSEK service resource. The introduction of local working copies ensures that this time critical section is as short as possible.

It depends on the task settings and the related question 'who can be interrupted by whom' to determine if the sender or the receiver is responsible for implementation of inconsistency protection. In Figure 5, signal3 is also potentially exposed to an inconsistency. Although it is a single data line, it represents a vector, and accesses to it could be interrupted. However, in the particular case of Figure 5, the sender cannot be interrupted by the receiver (Task A has a higher priority than Task C) and vice versa (Task C is not preemptive, i.e., SCHEDULE = NON). In this case, both tasks can use the global variable for signal3 directly, which is of course the most efficient way of data exchange.

Operating systems usually provide a means to exchange protected information between tasks. OSEK defines messages as a common interface for protected intra-ECU communication and for inter-ECU communication based on a common bus. For data exchange between tasks on the same ECU, these messages are not always the most efficient implementation since the automatically implemented inconsistency protection is sometimes not necessary, as shown in the last example. Per default, TargetLink therefore automatically selects the most efficient implementation (global variables with or without copies, interrupt lock, resources, etc.) depending on the task's priority and preemptibility. This default behavior can be overruled by explicit user settings. For instance, this might become necessary if the user cannot ensure that these task settings will remain unchanged after code generation or if a signal goes outside the scope of the code generator.


TargetLink: pos_control/control_out (Outport)

Code Editor | Plot Signal | Property Manager — Last modified: 08-Jun-2000 14:17:35

Output | Logging & Autoscaling | Inter Task Communication | Documentation

[✓ default] global Variable — Buffer — Message

- global Variable: [✓] local copy send; [✓] local copy receive; Consistency save mechanism: [✓ nothing] Interrupt lock, Scheduler lock, Resource lock
- Buffer: [✓] add. buffer send; [✓] add. buffer receive; Consistency save mechanism: [✓ nothing] Interrupt lock, Scheduler lock, Resource lock
- Message: [✓ queued] queue depth: 0; unqueued; [✓] with copy

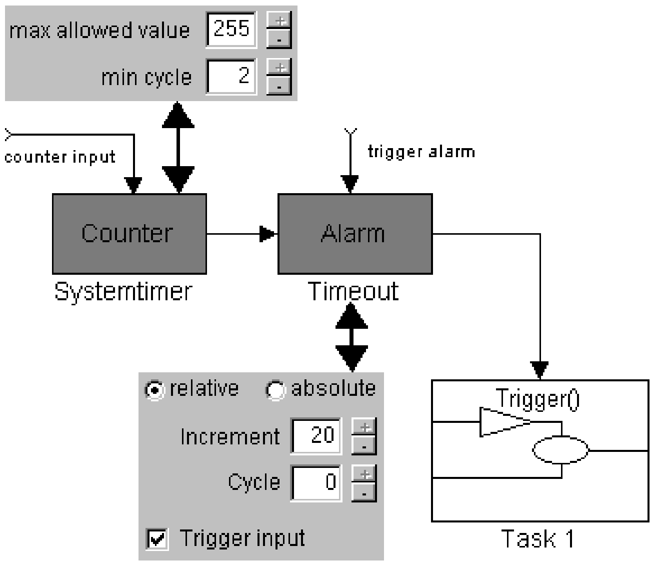
Apply | Revert | Help | Cancel | Close

Ready

Figure 6: Specification of Inter-Task Communication

SPECIAL BLOCKS

Special blocks that are not included in the normal Simulink blockset provide a specification interface to specific OSEK features and reproduce their behavior during simulation. The Alarm/Counter block is given as an example.
As described in the *Tasks* section, alarms are automatically used to set up time-periodic tasks. In addition, TargetLink provides a special block that makes the general alarm functionality of OSEK available to the user. Since each OSEK alarm is assigned to a counter, the TargetLink alarm block is always combined with a special counter block that was designed especially for this purpose (see Figure 7).


max allowed value 255

min cycle 2

[✓ relative] absolute

Increment 20

Cycle 0

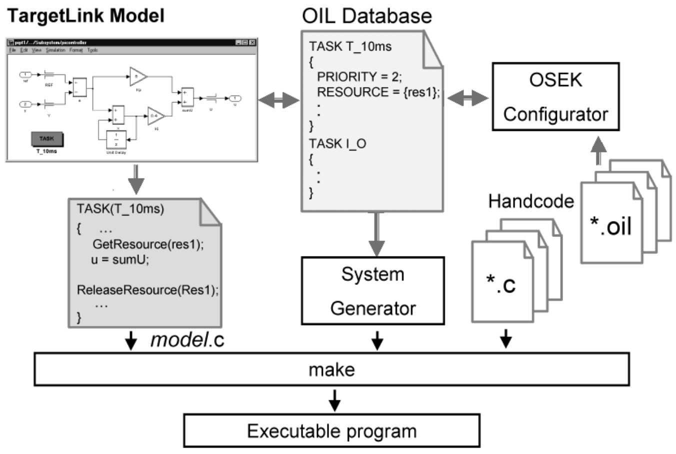
[✓] Trigger input

Figure 7: Alarm / Counter blocks

Although the alarm and the counter always form a unit, they are divided into two blocks because one counter can have several associated alarms. Code for the counter itself will usually not be generated because a hardware counter/timer or an operating system counter will be used. The TargetLink counter will be used for offline simulation only in this context. The settings in the block dialog allow simulation settings adjustment to the real counter behavior and reflect the standardized OSEK attributes for counters. The output signal of the counter is connected to one or more alarms. In the alarm block a counter value specifies when the alarm expires (the first time) and the task that is connected to this alarm by a (function call) trigger is activated. *Cycle* optionally defines the periodicity of the alarm. Alarms are set up at system initialization - e.g., for time-periodic tasks - or when a certain event occurs at runtime. The second case is modeled by a special input for a trigger that can be released by any source within the model. Absolute or relative alarms can be specified. The former expire at absolute counter values, whereas the latter expire at a certain number of time ticks after the alarm was triggered.

OSEK CONFIGURATION, TOOL INTERACTION

A typical ECU application consists of several C source files. Some of them have been generated automatically by TargetLink, others are supplied by hand programmers. As an input for the System Generator, there is one OIL file describing the complete real-time application, both the automatically generated and the handwritten parts. The System Generator is a tool provided by the vendor of the operating system that analyzes the OIL file and optimally scales the OS for the specific application. It generates C files and libraries that are linked together with the other source files to form the final executable program (see Figure 8).


```
TASK T_10ms
{
  PRIORITY = 2;
  RESOURCE = {res1};
  :
}
TASK I_O
{
  :
}
```

```
TASK(T_10ms)
{   ...
    GetResource(res1);
    u = sumU;

ReleaseResource(Res1);
    ...
}
```

Figure 8: Tool interaction

A critical path often is interaction and cooperation between the tools involved and the hand programmers. While one part of the application and the corresponding OIL description is supplied by TargetLink, the hand programmer or any other source delivers an additional part. In contrast to the C code that can be derived from several source files, the final OIL description must be one single document. Most OSEK vendors provide graphical tools to conveniently generate the textual OIL file. Such tools are especially useful since all commercial OSEK implementations have a lot of possible options in addition to the standard. Some option selections are not independent of others and without tool support, it is easy to produce inconsistencies. On the other hand, the user should be able to specify most of the attributes for those application parts that come from Simulink directly in the block diagram. It has already been pointed out that some of these attributes can be used for code optimization. TargetLink therefore works with an external OIL file as the database for the RTOS settings. It can import and export this data format. Simulink components (e.g., subsystems) can be assigned to OSEK objects (e.g., tasks) that already are in the OIL database, or new entries can be created. The user can choose the most suitable tool to enter attributes with. For example, it is possible to specify the basic, i.e., OSEK standard, attributes of tasks within the TargetLink environment and complete the description with the OSEK vendor-specific settings using the OIL configurator. By using this external database, consistency can be ensured. Changes made in any tool are automatically applied by the others.

SIMULATING PRODUCTION CODE FOR OSEK

One of the major strengths of Simulink is that it provides an executable specification. The behavior of the controller to be designed can be simulated offline on the PC, often using plant models to close the loop (see Figure 9).
Besides this Floating-Point Simulation Mode of the standard Simulink environment, TargetLink provides two further modes:

- Production Code Host Simulation Mode
- Production Code Target Simulation Mode

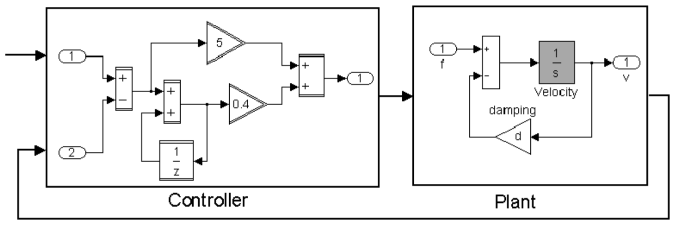


Figure 9: Floating-Point Simulation on Host PC

In Production Code Host Simulation Mode, the TargetLink-generated (fixed-point) production code is used for simulation instead of the original Simulink blocks (software-in-the-loop). Technically, the generated code is inserted into an s-function (the Simulink block that allows C code to be integrated into simulation) that replaces the original controller part of the model. Using this simulation mode, the generated code can be validated and fixed-point arithmetic effects such as quantization errors, saturation or overflows can be tested. The numerical effects that occur on the host PC are virtually the same as on the targeted microprocessor.

In Production Code Target Simulation Mode, the TargetLink-generated control algorithm is computed on a microcontroller that is equivalent to those on the production ECUs (processor-in-the-loop). Simulink still provides the stimuli, i.e., the input signals for the controller, for example, computed by the plant model, and evaluates the outputs. These signals are sent to and received from the target microcontroller by the serial interface of the PC. Again, this is accomplished by an s-function that replaces the original controller. Using the Production Code Target Simulation Mode, final verification of the generated production code can be made under realistic closed-loop operating conditions. The output of the target compiler can be verified because it is part of the test loop.

These advantages must remain unchanged when code is generated for OSEK. The original production code, now including OSEK API calls, shall be used for simulation. To achieve this TargetLink provides the following mechanisms.

SIMULATION WITHOUT AN OSEK OPERATING SYSTEM

Even if no commercial OSEK operating system is available on the target, the code must be tested in simulation without any changes. To make that possible, TargetLink creates additional code that implements - in a simple way - all OSEK macros and API functions used by the generated application code. These OSEK macros and API functions do not necessarily have to form a complete real-time operating system. They are tailored to the special needs of this non real-time closed-loop simulation.

During simulation, it is known at each time step which task is to be executed. For production code target simulation, a corresponding 'activate task' command is sent to the target (by the s-function). It is not necessary to have a complete scheduler on the target. A simple scheduler substitute can call the task like an ordinary C function. This is because the effects that task execution time can have on scheduling cannot be reproduced during offline simulation. While the task executes on the target, the host waits for a response, and the simulation time is halted. Therefore no other task becomes active and no preemptive scheduler is needed.

The Production Code Host Simulation Mode is very similar to the Production Code Target Simulation Mode. As in production code target simulation, the generated code remains unchanged. Unlike target simulation, the production code and the simulation frame consisting of the generated OSEK API functions and macros are part of the s-function. Instead of sending the input data to the target, the s-function calls the simulation frame which then activates the tasks.

SIMULATION INCLUDING AN OSEK OPERATING SYSTEM

The most realistic simulation results can be achieved if each software module in the final application is already included in the simulation. From this point of view, it is useful for the original OSEK operating system to be part of the production code target simulation. Thus correct interaction between the automatically generated code and the OS can be verified. Since the simulation remains non real-time and the simulation progress of the model part that remains on the PC - e.g., the plant model - determines the simulation time, the operating system on the target must be disconnected from its real-time inputs. The host PC, i.e., Simulink, must provide these stimuli, for example, hardware timer values, interrupts, etc., in accordance with the current simulation time (see Figure 10).
Let us assume an application with two periodic tasks; one with a sample rate of 10ms and the other with a rate of 100ms. Corresponding alarms are assigned to the system timer. Whenever the system timer reaches values that are multiples of 10ms and 100ms respectively, the corresponding task is activated. For simulation, the system timer is disconnected from its hardware timebase. Without any host intervention it does not count up and target simulation idles. When the simulation time in Simulink reaches 10ms (20ms, 30ms,...), the PC forces the target to set the system timer to the corresponding value, and the original OSEK task activation mechanism starts. After the task has finished, the computation results are sent back to the PC. The communication with the host is accomplished by a low-priority idle task.

Because the system time used on the target is simulated by Simulink and the host- PC sends input data and timer commands only after the previous simulation step was completed, the simulation does not run in real time. The main benefit of this new kind of simulation is the verification of counter and alarm handling, event and message handling, task activation and task interaction.

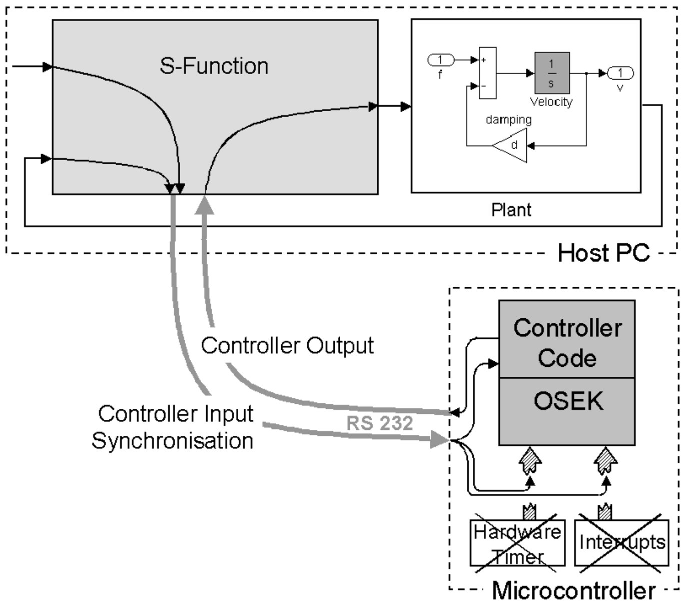


Figure 10: Target Simulation with an OSEK OS

TARGET SPECIFIC CODE GENERATION FOR COMMERCIAL OSEK IMPLEMENTATIONS

As previously mentioned, the OSEK specification leaves a certain amount of flexibility to allow optimal adaptation to different microcontrollers. As a consequence, different OSEK vendors implement some functionality in different ways, even if the implementation is for the same microcontroller.
An example is the API functions for counter access. There is no standard API function to read the system time. Therefore, the code for the same application appears slightly different for different OSEK implementations. Wherever possible, TargetLink uses standard OSEK API functions in the generated code. But when necessary, TargetLink uses implementation-

specific API functions to ensure efficient and readable application code.

Cooperation between major OSEK vendors and dSPACE ensures the best TargetLink support for the different OSEK implementations.

Another area of incompatibility is the possible OIL attributes for OSEK objects. Here the OS vendors usually provide a lot more options than described in the standard. For example, tasks have several additional attributes and the user might want to specify these attributes at the same place where an individual specifies the standard attributes, i.e., in the TargetLink Task block. TargetLink therefore provides a mechanism to identify these extensions to the standard and let the user enter the desired options. The key for this feature is again the OIL file. OSEK dictates that each vendor describes even the non-standard attributes by name and possible values in OIL syntax. TargetLink analyzes this description and provides the necessary GUI where the user can, for example, specify the stack mechanism for a task.

When the user wants to port an application from one OSEK implementation to another, TargetLink automatically replaces the implementation-specific API calls (such as the counter accesses) and the specific OIL attributes. This increases portability. Although OSEK was specified mainly to facilitate the reuse of software, these jobs would have to be done manually if no automatic code generator were available.

CONCLUSION

A steadily increasing number of development departments from different automotive areas are using Matlab/Simulink, one of the standard tools for the control engineer, as a specification basis for the entire development cycle of electronic control units. This is made possible in particular by the availability of tools such as TargetLink, which can automatically generate production-ready C code that has proven itself equal to handwritten programs. For the implementation of complete applications, the connection to a real-time operating system, in particular to the widespread OSEK standard, is of elementary importance. This paper has shown how TargetLink maps the basic modeling concepts of Simulink onto the OSEK functionality. The function developers can continue working in their familiar environment, which has proven its suitability in innumerable applications all over the world, and additionally get the flexibility and adaptability necessary for embedded software development. Some extensions to the standard Simulink blockset that make all important OSEK services and properties available in the model and that are optimally adjusted to the Simulink philosophy were introduced. It was shown, despite some divergent opinions, that Simulink together with the described environment extension is very well suited to modeling embedded software systems.

REFERENCES

[1] OSEK/VDX Operating System, Version 2.1, May 2000

[2] OSEK/VDX System Generation OIL: OSEK Implementation Language, Version 2.2, July 2000

[3] OSEK/VDX Communication, Version 2.2.1, September 2000

[4] TargetLink Production Code Generation Guide, dSPACE GmbH Paderborn Germany, May 2000

[5] Simulink User’s Guide, The MathWorks, Nattick, MA USA 1999

[6] H. Hanselmann, U. Kiffmeier, L. Köster, M. Meyer, “Automatic Generation of Production Quality Code for ECUs”, SAE 99, March 1-4, Detroit

CONTACT

Dr. Lutz Köster	Lead Engineer Software
dSPACE GmbH	Technologiepark 25
33100 Paderborn	Germany
lkoester@dspace.de	http://www.dspace.de

OSEK/VDX Homepage: http://www.osek-vdx.org/

Headquarters in Germany

dSPACE GmbH
Technologiepark 25
33100 Paderborn
Tel.: +49 5251 1638-0
Fax: +49 5251 66529
info@dspace.de
www.dspace.de

USA and Canada

dSPACE Inc.
28700 Cabot Drive · Suite 1100
Novi · MI 48377
Tel.: +1 248 567 1300
Fax: +1 248 567 0130
info@dspaceinc.com
www.dspaceinc.com

United Kingdom

dSPACE Ltd.
2nd Floor Westminster House
Spitfire Close
Ermine Business Park
Huntingdon
Cambridgeshire PE29 6XY
Tel.: +44 1480 410700
Fax: +44 1480 410701
info@dspace.ltd.uk
www.dspace.ltd.uk

France

dSPACE SARL
Parc Burospace
Bâtiment 17
Route de la Plaine de Gisy
91573 Bièvres Cedex
Tel.: +33 1 6935 5060
Fax: +33 1 6935 5061
info@dspace.fr
www.dspace.fr