Master Thesis

Transformation of Hardware Traces to System Traces for Embedded Multi-Core Real-Time Systems

Author:	Felix Martin
Matriculation number:	3000126
Faculty:	Electrical Engineering and Information Technology
Primary correction:	Prof. Dr. rer. nat. Jürgen Mottok
Secondary correction:	Prof. Dr. rer. nat. Michael Niemetz
Supervisor I:	M.Sc. Andreas Sailer
Supervisor II:	Dr.-Ing. Michael Deubzer
Date of submission:	October 28, 2015

Abstract

Embedded real-time multi-core systems must adhere to strict timing requirements in order to guarantee correct execution. Timing requirements are specified to document system execution paths that are safety critical with respect to the timing behavior of an application.

Via tracing it is possible to validate the fulfillment of timing requirements in the native environment of a microcontroller. However, trace tools produce a trace on hardware or software level, whereas requirements are specified on system level. A transformation of the former to the latter is required to close this gap.

Additionally, not all trace techniques are capable of producing results suitable for the real-time analysis of embedded applications. Most techniques are not sufficient for one or several reasons: limited trace duration, inadequate number of recordable objects, and limited timing accuracy.

Therefore, this thesis examines different trace techniques and shows why hardware tracing is the most sufficient for real-time analysis. Next, the coherence between hardware, software, and system level entities is examined. Based on the results a mapping from software level to system level is introduced and validated.

The thesis concludes that it is possible to record cycle accurate system traces of arbitrary length via hardware tracing. However, this requires detailed knowledge about hardware tracing and the operating system underlying an application.

Contents

1 Introduction

Embedded applications are increasingly required to provide real-time performance [21]. This means that the correct behavior of a system is not only dependent on the logical results of a computation, but also on the physical instant in which these are produced [28]. For hard real-time applications violation of a deadline will result in damage of the system or its environment [58].

Due to the pervasive nature of embedded systems and their use for critical applications, e.g., medical devices or advanced driver assistance systems, measures to ensure the correctness of time dependent functionality must be taken [27]. Therefore, debugging and validation are a fundamental part of the development process of such applications [10].

Different techniques to debug embedded systems exist [49]. The simplest one is a classical printf statement in C (or the equivalent in another language). More sophisticated debug technologies require on-chip debug logic in the embedded processor. On-chip debug generally supports two different types of functionality: run-control debug and real-time trace [10].

The former allows it to stop and examine the state of a system at points of interest, so called breakpoints. This approach is intrusive, or in other words changes the runtime behavior of an application. This is not acceptable for time critical applications, e.g., engine control units that require continuous execution of the processor in order to control feedback loops and to maintain mechanical stability [10].

Real-time trace recording or tracing however, allows it to analyze and debug a system without stopping the execution. It works by recording processor events such as function calls and data accesses. The captured events can be used to reconstruct and analyze the runtime behavior of an application.

Since timing is an integral part in the development of safe and secure real-time applications, timing dependencies should be included in the software interface specifications [30]. One way to specify these dependencies are timing requirements, e.g., the maximum response time for a certain task [9]. Via tracing system engineers are capable of validating those requirements on target.

Timing-Architects Embedded Systems GmbH (TA) provides the TA Tool

Suite, a collection of tools for the system design, simulation, automated optimization, and target verification of embedded real-time multi-core and many-core systems [55]. These features work on the basis of system models. Consequently, requirements are defined for system entities such as tasks, runnables, signals, and semaphores.

On the contrary, trace recording produces events on software level. This means a trace contains information about function entries and exits, and data read and write accesses. As a consequence, the specified system requirements cannot be evaluated.

However, by mapping software events to the corresponding system events it is possible to transform a software to a system level trace. Best Trace Format (BTF) is a trace format on system level and is used in this thesis because of its native support for multi-core environments. To the best of my knowledge, the possibility of a software to system mapping has only been shown for a small subset of all entities specified by BTF.

In this thesis the feasibility of mapping all event actions contained in the BTF standard is discussed, evaluated and validated. Furthermore, different real-time trace techniques are discussed with respect to their versatility for the timing analysis of embedded multi-core real-time applications.

1.1 Motivation

Transformation of software events to system events is required for the timing analysis of embedded real-time systems as discussed in the previous section. Moreover, system traces can also be used for different other use cases which are covered in the following.

Simulation Validation

A simulation can be executed for a timing model by the TA Simulator. The resulting simulated trace can be evaluated to validate the compliance of an application with the specified requirements.

A simulated and a hardware based system trace will never be equal by definition because a model is an abstraction of reality. Nevertheless, simulation supports engineers in validating system behavior in early design stages. It can abstract complex problems and analyze non-deterministic system behavior [50].

However, a simulation is still a software which is vulnerable to bugs and can potentially produce wrong results. A deviation to reality due to the abstraction cannot be classified as a wrong result, on the other hand an implementation error can be.

Via tracing it is possible to validate the correctness of simulated traces. This is especially useful if a new simulation feature is implemented. In this case a system trace recorded from hardware can provide valuable insights in

the actual behavior.

OS Overhead Measurement

Another aspect that is relevant for the development of embedded applications is the overhead caused by the operating system (OS) [63]. Overheads are execution periods where the processor is not used by the actual application but by the OS for example, context switches and inter-core communication mechanisms.

Especially for applications with a high processor utilization the additional overhead caused by the OS plays a critical role. Fulfillment of timing requirements may be feasible or not depending on the overhead [20]. In order to take this into consideration a good understanding of the execution times required by OS routines is necessary. System traces recorded on hardware allow it to determine the exact execution times for these overheads easily.

Model Reconstruction

The initial creation of a timing model for an existing application is a tedious process if it must be done manually. Model reconstruction can simplify this task by creating a timing model automatically. It works by analyzing a system trace recorded from hardware. By detecting common timing patterns in the trace a model of the application can be created [47].

1.2 Related Work

The two main topics discussed in this thesis are tracing and hardware to system mapping. While the former has been an important topic in the literature over the last three decades, the necessity for the latter has only become important in recent years.

Tracing

Ferrari [12] gives an comprehensive overview of major computer performance evaluation techniques and their application to various types of performance problems. In his book *Computer Systems Performance Evaluation* he distinguishes between three trace measurement techniques: software, hybrid, and hardware based trace measurement. It is important to understand that these techniques do not directly relate to the trace abstraction levels discussed in the previous sections. The concepts described in his book which was released in 1978 are still relevant today, the implementation is outdated.

Mink et al. [38] discuss hardware based performance measurement in more detail. They argue that hardware tracing is the only sufficient trace technique for recording resource utilization information because of the high

signal speeds involved and the fact that not all signals are visible to software measurement techniques. Resource utilization is concerned with detailed information about the operation of the hardware such as cache hit ratios and access delays. Moreover, they mention that software based tracing is intrusive and thus changes the runtime characteristics of an application.

Kraft et al. [29] discuss trace measurement in the context of five industrial projects. They argue that hardware trace solutions require large, expensive equipment mainly intended for lab use. Additionally, they claim that software based trace solutions can also remain active in applications post-release. Based on this arguments they use a software based trace measurement approach in their paper. They introduce a software instrumentation approach with a very low overhead according to their measurement results.

Hardware to System Mapping

Lauterbach [18] provides a possibility to export task and runnable system events for traces recorded via hardware tracing. However, their approach is limited to a subset of the existing task and runnable events. For example, runnable preempt and resume, and task wait events are not covered by the Lauterbach export even though this information is relevant for the real-time analysis. Lauterbach uses the information from the OSEK Run Time Interface (ORTI) files and relies solely on function trace events for the export.

Kraft et al. [29] also discuss how task events on system level can be recorded. They argue that it is difficult to detect which entity blocks a task because the scheduling status of the OS only provides information about the entity type blocking the task not the entity itself. They suggest code instrumentation as a pragmatic solution to work around this problem, admitting that this approach is problematic because the instrumentation points have to be maintained by the developer.

1.3 Interrogation

Timing analysis of embedded system requires a trace, i.e., a sequence of events, with sufficient duration and timestamp accuracy. The minimum trace duration is dependent on the application and requirements that should be validated. Fundamentally, the longer the trace duration the more information for the real-time analysis of the application are acquired. However, more data requires longer processing times. Therefore, a trace duration of at least one second is demanded in this thesis to provide a tradeoff between processing time and sufficient length for the real-life use-cases discussed in section 1.1.

Timestamp accuracy is important for the real-time analysis because if the

resolution is too low no meaningful analysis may be feasible. For example, if events can only be recorded in the range of milliseconds, the analysis of requirements in the microseconds range is not feasible.

Kraft et al. [29] also state that a timestamp accuracy in the milliseconds range is too coarse-grained for embedded systems timing analysis. Especially for validation of simulation tools and model reconstruction cycle accurate timestamps would provide enormous benefits. From these requirements the first hypothesis that should be evaluated in this thesis can be derived.

Hypothesis 1 *There exists a trace technique that allows recording of cycle accurate traces for embedded multi-core real-time system with a duration of at least one second.*

Trace techniques output a trace on software level, i.e., a sequence of software events. These events provide information about the code segments executed by an application and the memory regions accessed. This information allows deep insights into the runtime behavior of an embedded system, but is not sufficient for its real-time analysis.

Traces on system level or in other words, sequences of system events are required for the real-time analysis of embedded multi-core applications. In the context of this thesis system events are defined as all events that are contained in the BTF specification and not explicitly excluded in subsection 2.2.2. With an understanding of the underlying OS mechanisms it may be possible to map software to system events.

OSEK/VDX and AUTOSAR are common standards for the development of applications in the automotive industry. These standards are discussed in more detail later. OSEK/VDX compliant operating systems feature a so-called ORTI file.

The aim of ORTI is to make OS internal data visible to external tools [43]. This means it is possible via ORTI to relate software level entities to their respective interpretation on system level. It must be examined if a mapping for all BTF entities is feasible.

Hypothesis 2 *A complete mapping from software to system entities is feasible based on the information included in the ORTI file for an OSEK/VDX compliant Operating System.*

If Hypothesis 2 does not hold other ways to achieve a complete software to system mapping must be found. An OS must keep track of the states of all relevant system objects internally. Otherwise, it would not be possible to execute appropriate actions if required. For example, if one task activates another one the OS must determine whether the corresponding task is allowed to be activated or if the maximum number of activations has already been exceeded.

By analyzing the internal data structures of an OS it may be possible to construct a mapping from software to system entities. Considering the

previous example, there might be an OS data structure that keeps track of the remaining activations for each task entity. If the field for a task is incremented, an entity of the corresponding task terminates. If it is decremented a new task instance is activated.

Hypothesis 3 *A complete mapping from software to system entities is feasible for an OSEK/VDX compliant Operating System.*

1.4 Outline

In oder to transform a trace recorded from hardware to a trace on system level an understanding of the underlying operating system mechanisms is required. An OS standard commonly used in the automotive industry is OSEK OS. It is discussed in section 2.1.

The real-time behavior of an embedded multi-core application can be represented by a system trace. Based on a system trace an application can be examined and specified timing requirements can be validated. BTF is a system level trace format and used in this thesis. It is discussed in section 2.2.

There exist different techniques to record traces of embedded applications. In chapter 3 an overview of these techniques is provided. It is then argued why hardware tracing is the only technique sufficient for the validation of embedded real-time applications. Accordingly, hardware tracing is then discussed in more detail.

On the basis of the information in chapter 2 the mapping between software entities and system entities is described in chapter 4. Mapping is done for all BTF entities that are relevant for the analysis of embedded multi-core applications as discussed in subsection 2.2.2.

In chapter 5 the mapping is validated. For that reason criteria to compare BTF traces are established in subsection 5.1.3. Based on these criteria simulated traces and traces recorded hardware are compared and evaluated. This is done in two steps. Firstly, test applications are created manually to cover all possible BTF actions in subsection 5.2.2. Secondly, applications are created randomly to avoid selection bias in the creation of test cases in subsection 5.2.3.

Finally, the results of this thesis are discussed in chapter 6 and possible topics for future work are outlined in chapter 7.

2 Fundamentals

This thesis discusses the transformation of hardware events to system events for OSEK/VDX compliant real-time OSs. Hence, the parts of OSEK/VDX that are relevant for this thesis are described in the following.

Additionally, a well-defined format is required to represent the resulting traces consisting of entities on system level. The BTF format which is discussed in section 2.2 is used within the context of this work.

2.1 OSEK/VDX OS

OSEK/VDX (Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen) [45] is an effort of the German and French automotive industry to establish common standards for the software architecture of distributed control units in vehicles. Defining a common architecture for communication, operating systems, and network management avoids problems that arise otherwise by using different interfaces and protocols. An abstraction layer between hardware and software allows OSEK/VDX compliant applications to be reused on different hardware platforms with minor modifications.

OSEK OS specifies the architecture of a real-time operating system for single processors. Based on the services offered by the OS, integration of modules from different manufactures is possible. The OS meets the hard real-time requirements demanded by automotive applications. OSEK OS can also be used in multi-core environments. In such cases a separate kernel is executed on each core. Service routines can be used to interact between multiple OS instances.

A high level of flexibility is required for an OS to support real-time systems on various target platforms. In order to support low-end and high-end microcontrollers alike OSEK/VDX conformance classes (CCs) are specified. Depending on the CC certain features, e.g. multiple task activations, multiple tasks per priority, and extended tasks are available or not.

Dynamic creation of system objects like tasks, alarms or events is not supported by OSEK OS. All objects are defined statically and created during the system generation phase [42]. Consequently, all OS entities are known before the system execution.

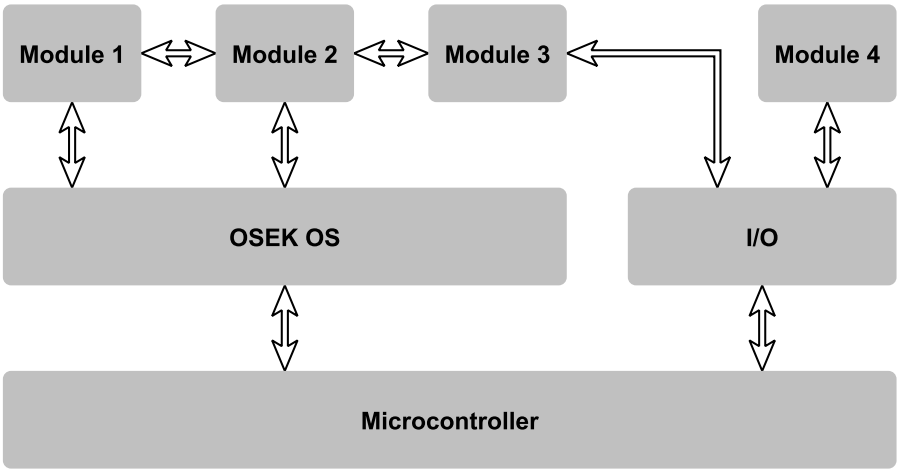

Figure 2.1: OSEK/VDX compliant OSs abstract application modules and hardware via an OS layer. A non standardized I/O module still results in hardware dependencies.

Figure 2.1 illustrates the abstraction of application modules from hardware resources. Standardized system services offer functionality that can be used by all application modules. Well-defined service calls, type definitions, and constants are specified and ensure the portability of an application to different architectures.

An Input/Output (I/O) module parallel to the OS gives access to microcontroller specific functionality like serial interfaces or analog-to-digital converters. I/O interfaces are not specified by OSEK OS which is opposing to the idea of easy portability. OSEK/VDX's follow-up standard AUTOSAR (AUTomotive Open System ARchitecture) [5] solves this problem by adding a MCAL (Microcontroller Abstraction Layer) to the AUTOSAR OS specification [4].

In 2003 AUTOSAR was established by automobile OEMs, suppliers, and tool developers pursuing the same goals like OSEK/VDX. Different parts of the AUTOSAR standard are based on OSEK/VDX and AUTOSAR OS constitutes a superset of OSEK OS. Consequently, all features discussed here are also relevant for AUTOSAR OS. Differences that are important in the context of this thesis are mentioned explicitly.

2.1.1 OSEK Architecture

OSEK/VDX provides a specification for the architecture of an embedded real-time OS. One of the main purposes of the OS is to manage the available computational resources of the CPU. Based on different factors such as priority, task group and scheduling policy, executable entities, so-called

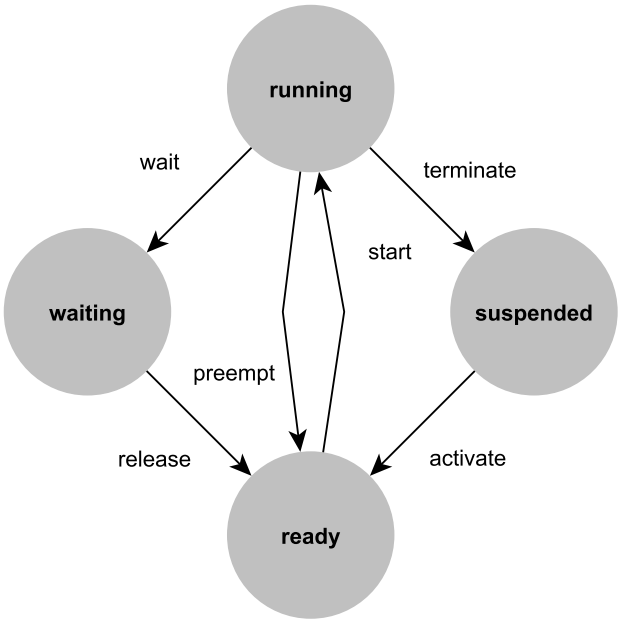

Figure 2.2: Task state model of an extended OSEK OS task. A basic task cannot enter the waiting state.

processes are given access to the processor core. The procedure of deciding which entity is executed next is called scheduling.

There are two types of process entities available: tasks and Interrupt Service Routines (ISRs). Former are scheduled on task level while for latter the interrupt level is used. Entities on interrupt level always have precedence over entities on task level. Scheduling on interrupt level depends solely on the priority of an entity and is done by hardware. For task entities scheduling is done by the OS and depends on priority, scheduling policy, and task group.

Tasks are categorized into two types by OSEK OS. A basic task has three states: ready, running, and suspended. An extended task is a basic task with the additional waiting state. Suspended tasks are passive and can be activated. A task in the ready state can be allocated to the CPU for execution which is then indicated by the running state. Only one task per core can be in the running state at a given point in time. Extended tasks can wait passively for an event. In that case they reside in waiting state. Waiting tasks are not allocated to the CPU.

Different task state transitions are possible as shown in Figure 2.2. At system initialization all tasks are suspended. If a task has to be executed it must be activated by a system service. A task can be started by the OS in order to be executed. A task is preempted if a task of higher priority is scheduled. Once a task has finished execution it terminates and switches

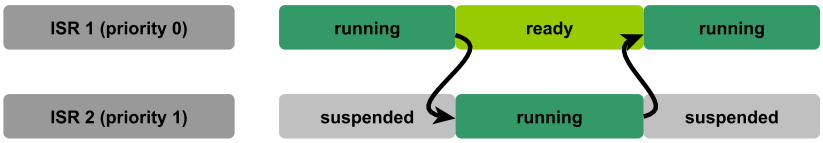

Figure 2.3: ISR scheduling is done by hardware and is solely depended on the interrupt priority. ISRs do not have a ready state because they are started by hardware.

Figure 2.4: Scheduling behavior of a non (top) vs a full (bottom) preemptive task. A non preemptive task finishes execution even though a task with higher priority is in ready state. Only for certain system services, for example, an inter-process activation, the other task may be scheduled. A full preemptive task is preempted if a task with higher priority is activated. Once this task has terminated, the task with lower priority can continue running.

to the suspended state. Extended tasks can wait for system events and are released and switched to ready once the expected event is set. The previous state of a ready task is not implicitly known.

Priorities are assigned to tasks and ISRs statically. The lowest priority is zero and greater integers mean a higher priority. If an ISR of priority zero is running and another ISR of priority one is activated, the first ISR is preempted and restarts once the second ISR is terminated as shown in Figure 2.3. For tasks the same scenario is dependent on scheduling policy and task group.

OSEK OS specifies three **scheduling policies**: non, full, and mixed preemptive scheduling. For a non preemptive tasks, rescheduling is only possible if a system routine that causes rescheduling, e.g. an inter-process activation or an explicit scheduler call is executed. A full preemptive task

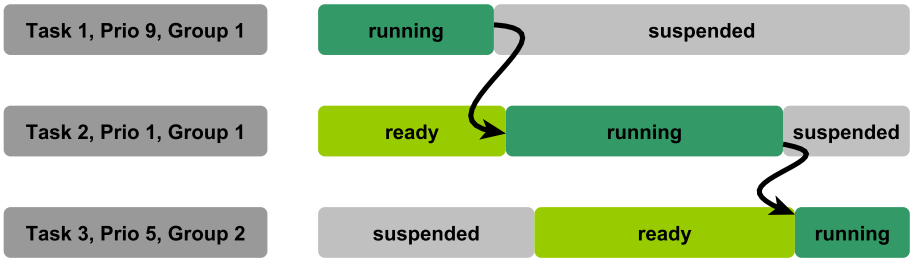

Figure 2.5: Scheduling of task entities is not only dependent on priority and scheduling priority. OSEK/VDX specifies task groups, which change the priority of tasks inside in relation to tasks outside a specific group. In this example Task 3 has a greater priority than Task 2. However, because they are in the same group, Task 2 inherits the priority of Task 1. Thus, Task 2 is not preempted by Task 3.

can be rescheduled at any point in time during its execution if another task of higher precedence is activated as shown in Figure 2.4. A mixed preemptive system contains tasks with both, non, and full preemptive scheduling policies. Otherwise the system is either non or full preemptive.

The precedence of a task is not necessarily due to its priority. OSEK OS introduces the concept of task groups which allows it to group multiple tasks into a group. A task which is not within a group has precedence over a task within a group only, if its priority is higher than the priority of the task with the highest priority within this group. This means a task acts non preemptive towards another task if the task with the highest priority within the group has a greater priority than the other task as shown in Figure 2.5.

OSEK/VDX Conformance Classes are used to adapt applications to different hardware capacities such as available memory and CPU speed. Only one CC can be active at a time and cannot be changed during runtime. Basic CCs (BCC1 and BCC2) allow basic tasks only, while extended CCs (ECC1 and ECC2) allow basic and extended tasks. Level one CCs (BCC1 and ECC1) allow multiple tasks per priority and multiple activation requests per task. For level two CCs (BCC2 and ECC2) multiple tasks can share the same priority and the same task can be activated multiple times as shown in Table 2.1. This means BCC2 and ECC2 allow Multiple Task Activation (MTAs). An active task with pending activations becomes ready again immediately after termination.

Task scheduling is done by the OS while ISR scheduling is done by hardware. ISRs can be divided into category one and category two. Category one ISRs do not run under OS control and are thus not allowed to call OS services. Category two ISRs are monitored by the OS and are allowed to execute a subset of the available OS services. Tasks are always preempted by ISRs and can only continue running when all ISRs have terminated.

	BCC1	BCC2	ECC1	ECC 2
MTA	no	yes	no	yes
Multiple tasks per priority	no	yes	no	yes
Extended tasks	no	no	yes	yes

Table 2.1: OSEK OS specifies multiple CCs to respect the computational capacities of different platforms. Depending on the CC different features are supported or not.

Tasks and ISRs serve as containers for application specific functions. These functions are not managed by the OS and must be added to the process code by the user. AUTOSAR invented the concept of runnables to solve problems related to the VFB (Virtual Function Bus) introduced by the AUTOSAR architecture [40]. A runnable is essentially the same as a function.

Events are system objects that can be set or not. Each event is owned by at least one extended task. Only a task that owns an event is allowed to clear and to wait for it. When waiting for an event a task switches into the waiting state. It is switched back to ready when the corresponding event is set.

All tasks and category two ISRs are allowed to set an event. Events are used as a binary communication technique. One task can signal another one for example, if a certain resource has been released. Events are defined and assigned to tasks before runtime. All events assigned to a task are cleared when this task is activated.

Resource management is used to manage access to shared objects. An OSEK/VDX resource is basically a mutex. Each resource gets a ceiling priority that is at least as high as the highest priority of all tasks that access this resource. When a task accesses a resource and its priority is lower than the ceiling priority of this resource its priority is raised to the ceiling priority. The priority is reset to the original value once the task releases the resource.

This technique ensures that a task that potentially accesses a shared resource cannot switch into the running state. This prevents priority inversion and deadlocks. On the downside, tasks with a priority lower than the ceiling priority may be delayed by a lower priority task.

Alarms are used to activate a task, set an event or execute an alarm-callback routine. Each alarm has an alarmtime and a cycletime that is statically defined and measured in ticks. An alarm expires the first time after alarmtime ticks and afterwards every cycletime ticks. Thus, an alarm can be used to activate a task or set an event periodically.

Each alarm is assigned to a counter object but each counter can be used by multiple alarms. Counters are responsible for triggering an alarm after the specified number of ticks have passed. Each OSEK OS offers at least one counter that is based on a hard- or software timer.

Hook routines can be used to allow user-defined code within OS in-

Define	Meaning
E_OK	Service finished correctly.
E_OS_ACCESS	Calling task is not an extended task.
E_OS_CALLLEVEL	Service called from invalid level.
E_OS_ID	Invalid OS ID.
E_OS_LIMIT	Number of activations is exceeded.
E_OS_NOFUNC	Alarm or resource is not in use.
E_OS_RESOURCE	A resource is still occupied.
E_OS_STATE	Object is in invalid state.
E_OS_VALUE	Value is not allowed.

Table 2.2: OSEK OS defines a StatusType type that can be used to return an error code from service routines. This table shows the status types that are defined by OSEK/VDX and their meaning. Users are free to define additional codes.

ternal processing. They cannot be preempted by tasks and ISRs and only a subset of the available OS services is available from their context.

The StartupHook and ShutdownHook can be used to execute user specified code at system start and shutdown. OS errors result in a call to the ErrorHook. It can be used to execute application specific error handling. Finally, PreTaskHook and PostTaskHook are called at task start and termination.

2.1.2 OSEK OS Services

OSEK OS specifies system services that can be used to interact with internal OS mechanisms and objects like tasks or resources. The internal presentation of system objects is implementation specific. Only specified system services allow well-defined interaction with OS objects. A system service may take zero or more input parameters and may return zero or more output parameters via call by reference. The return value of an OS service is of type StatusType. Table 2.2 shows defined status types.

A task can be activated via alarm or ActivateTask service routine. Latter is callable from interrupt and task level. The task to be activated must be provided as an input parameter. If this task is suspended its state will be changed to ready. If it is not suspended the pending activations counter is incremented or E_OS_LIMIT is returned if the MTA limit is exceeded.

TerminateTask is used to switch a task from running to suspended. All internal task resources are released and the service will not return if the call was successful. TerminateTask will fail with E_OS_RESOURCE if resources are still occupied by a task. ChainTask is a combination of ActivateTask and TerminateTask. It terminates the current task and activates another task which is provided via input parameter.

Schedule can be called to explicitly trigger a scheduling decision. This

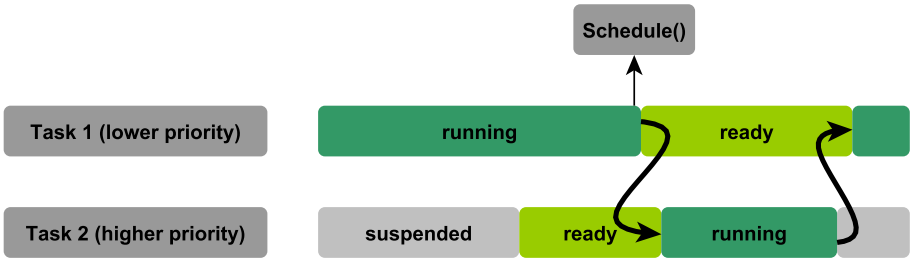

Figure 2.6: An explicit call to the scheduler can solve the problem of a delayed higher priority task.

makes sense for non preemptive tasks if a task with higher priority is ready. Normally the task with higher priority is delayed until the task with low priority has finished execution as shown in Figure 2.4. By calling Schedule the non preemptive task is preempted and the task with higher priority is executed as illustrated in Figure 2.6.

The routines GetResource and ReleaseResource can be used to request and release resources. Nested resource requests are only allowed in last-in-first-out order, i.e. the resource that has been requested first must be released last. Within a critical section that is protected via a resource no calls to services that cause rescheduling are allowed. Both methods can be called from task and ISR level. If a requested resource is already occupied E_OS_ACCESS is returned.

Interaction with event objects is done via SetEvent, ClearEvent, GetEvent, and WaitEvent service routines. SetEvent takes a mask of events that should be set for a specific task. Events can be deleted from the context of a process owning this event via ClearEvent. GetEvent returns the current status of all events related to a specified task. A task can wait for one ore more events using the WaitEvent service routine. Waiting lasts until at least on of the specified events is set.

The service routine GetAlarmBase returns the basic configuration of an alarm. The remaining ticks until an alarm expires can be retrieved with GetAlarm. SetRelAlarm increases the remaining ticks by the submitted value while SetAbsAlarm sets them to an absolute value. An alarm can be deactivated with CancelAlarm.

2.1.3 OSEK OIL and ORTI

The implementation of system objects is not specified by OSEK/VDX. Therefore, users cannot know how to create system objects because correct definition is depending on the OS. OSEK Implementation Language (OIL) solves this problem by providing a meta language for defining system objects [41]. Based on OIL configuration files code generators provided by

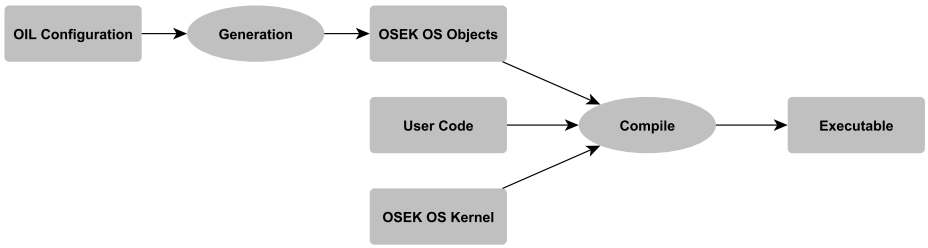

Figure 2.7: An OSEK/VDX application is compiled from three sources. The OS kernel, user created code and OSEK OS object definition files which are created via code generation based on one or more OIL files.

the OS vendor can produce OS specific source code. In combination with kernel and user code an application can be built as shown in Figure 2.7.

OSEK/VDX specifies data types for all system object types. However, the implementation of the data types is OS specific. For example, a task is identified by TaskType. TaskType could be implemented as an integer indexing a global list of task objects or as a pointer to the task object itself.

Only a minimum amount of data types necessary to interact with service routines are specified. Consequently, a lot of information is kept internally by the OS and is not available for the user. For example, there is no common interface to get data of the pending activations of a task, the current state of a resource, or the state of an event.

Application code that needs this information would need to access the OS internals directly which results in portability and security issues. Moreover, external tools like debuggers that want to provide OS aware debug information have no standardized interface to relevant internal data.

OSEK Run Time Interface (ORTI) was specified to solve this problem. Via ORTI tool vendors have a standardized interface to OS internal data and properties of relevant system objects. The Kernel Object Interface Language (KOIL) format is used to exchange relevant information via the ORTI file. This file contains mappings from OS object properties to variables that hold the respective information.

ORTI specifies a set of system properties that must be available for every OSEK/VDX compliant OS. Operating system vendors are free to add additional information. Each OS object is described in a separate section of the ORTI file. The specified sections that are relevant for this thesis are *os*, *task*, *alarm*, and *resource*.

Information about the currently running process, the system error state, and the active service routine can be found in the OS section shown in Table 2.3. The *servicetrace* attribute is written whenever a service routine is started or finished along with the ID of the corresponding routine. Task and ISR processes that are currently running in a system can be retrieved via

```
TASK T_CylinderResponser {
    priority = "osTcbActualPrio[30]";
    state = "osTcbTaskState[30]";
    currentactivations = "osTcbActivationCount[30]";
};
```

Listing 2.1: Textual representation of the ORTI attributes for a task entity.

Attribute	Content
runningtask	currently running task
runningisr2	currently running category 2 ISR
servictrace	indicates entry and exit to service routines
lasterror	contains the last error code set by the system

Table 2.3: The ORTI OS section provides information about the running task and category 2 ISR, entry and exit to service routines and the last system error.

runningtask and *runningisr2*. The attribute *lasterror* provides information about the last failure condition.

As shown in Table 2.4 the ORTI task section makes the current *priority*, *state*, and number of open activations (*currentactivations*) for each task available. Listing 2.1 shows the textual representation of the ORTI attributes for a single task.

Table 2.5 shows that alarms have an *alarmtime* attribute that contains the ticks to the next expiry time. A *cycletime* is used for periodic alarms. If a cyclic alarm expires, *alarmtime* is reset to this value. An alarm can be running or stopped which is indicated by the *state* attribute and executes a certain *action* if *alarmtime* becomes zero.

A resource can be locked or free which is indicated by the *state* attribute. In the former case the *locker* attribute indicates the corresponding process as shown in Table 2.6. The resource *priority* is also accessible.

Additional sections and attributes can be found in the OSEK/VDX ORTI specification [44]. Even via ORTI not all OS internals become available. Via *servicetrace* it can be detected that a certain event is set or cleared but no information about the event itself is available. Consequently, for cer-

Attribute	Content
priority	task priority
state	task state (Figure 2.2)
currentactivations	number of task activations

Table 2.4: The ORTI task section provides information about the current task priority, task state and number of activations. The task priority can be different to the statically defined value because of the priority ceiling protocol.

Attribute	Content
alarmtime	time till alarm expires
cycletime	alarm cycle time of periodic alarms
state	alarm state (running or stopped)
action	action at alarm expiry time

Table 2.5: The ORTI alarm section provides information about the time that is left until an alarm expires, its cycle time, the current state and the action that is executed once the alarm expires.

Attribute	Content
state	resource state (locked or unlocked)
locker	the task that has locked a resource
priority	resource priority

Table 2.6: The ORTI resource section provides information about the state of a resource. A resource can be locked or not. For a locked resource the corresponding task is made available.

tain use cases it may still be necessary to access OS specific data structures manually.

2.2 System Trace

A trace is defined as a sequence of events. Events depict a change in the state of a system and can be represented on different levels of abstraction. These are discussed in more detail in chapter 3. For the timing analysis of embedded multi-core real-time systems a trace on system level is required.

Tools that analyze or visualize traces must be able to interpret the recorded events. For example, the software that interacts with hardware trace devices must be able to understand the hardware events that are generated on-chip. Otherwise it is not possible to transform the hardware events into higher level software events. For that reason a well-defined format for events is required for further processing of recorded traces.

Depending on the goal pursued with a trace measurement, one level of abstraction can be more appropriate than another. On the one hand, a software engineer who implements a feedback control system is mainly interested in the functions and variables that correspond to that particular task. A system engineer on the other hand, who integrates a variety of different modules into a single application, is not interested in the details of each individual module. Instead the functionality of the system as a whole is of interest.

Field	Meaning
time (t)	Timestamp relative to a certain point in time.
source (Ψ)	Entity that caused an event.
source instance (ψ)	Entity instance that caused an event.
target type (ι)	Type of the entity that is influenced by an event.
target (T)	Entity that is influenced by an event.
target instance (τ)	Entity instance that is influenced by an event.
action (α)	The way in which target is influenced by source.
note (ν)	An optional field that is used for certain events.

Table 2.7: A BTF event consists of eight fields. An event describes the way in which one system entity is influenced by another one.

2.2.1 BTF Specification

A trace on system level can be used to analyze timing, performance, and reliability of an embedded system. Best Trace Format (BTF) [57] was specified to support these use cases. It assumes a signal processing system where one entity influences another entity in the system. This means an event does not only contain which system state changes but also the source of that change. For example, an observed event on system level could be the activation of a task with the corresponding timestamp. Then a BTF event additionally contains the information that the task activation was triggered by a certain alarm.

Let k be an index in $\mathbb{N}_0$ denoting an individual event occurrence then a BTF event can be defined as an octuple

$$b_k = (t_k,\, \Psi_k,\, \psi_k,\, \iota_k,\, T_k,\, \tau_k,\, \alpha_k,\, \nu_k) \tag{2.1}$$

where each element maps to a BTF field: t_k is the *timestamp*, Ψ_k is the *source*, ψ_k is the *source instance*, ι_k is the *target type*, T_k is the *target*, τ_k is the *target instance*, α is the event *action* and ν_k is an optional *note*.

A BTF trace can then be defined as a sequence of BTF events where $n \in \mathbb{N}_0$ is the number of events in the trace:

$$B = (b_1, b_2, \ldots, b_n) \tag{2.2}$$

A BTF event can be represented textually as a comma-separated list where each field maps to an element as shown in the following listing.

```
12891, TASK_200MS, 3, SIG, EngineSpeed, 0, write, 42
```

The first field (12891) represents the timestamp of the event. A BTF

trace contains the chronological order of events that occurred in a system. Therefore, for each timestamp $t_k \in \mathbb{N}_0$ in a trace it holds that $t_k \leq t_{k+1}$. All timestamps within the same trace must be specified relative to a certain point in time, that can be chosen arbitrarily. Hence, neither trace nor system start must occur at $t_0 = 0$. The time period between two events b_k and b_{k+1} can be calculated as $\Delta t = t_{k+1} - t_k$. If not specified otherwise, the unit for time is nanoseconds.

A BTF event represents the notification of one entity by another. Each entity has an unique name. In the previous example, the source entity Ψ has the name TASK_200MS and the target entity T is called EngineSpeed.

The fourth field SIG is the short representation of the target entity type ι. Table 2.8 gives an overview of all entity types and their corresponding short IDs. Entity types are discussed in more detail in subsection 2.2.2. In this example, the target entity EngineSpeed is a signal. The source entity type is not part of a BTF event.

Some entities, tasks, ISRs, runnables, and stimuli have a lifecycle. This means at a certain point in time an entity becomes active in the system and eventually it leaves the system. For example, the lifecycle of a task starts with its activation and ends when it terminates. If MTAs are allowed for an application, it is possible that multiple *instances* of a task are active at the same time. For those cases where multiple instances of an entity are currently active, it is consequently not clear to which instance of the entity the event refers.

Instance counter fields ψ and τ are used to distinguish between multiple instances of the same entity. The counters are integer values $\psi, \tau \in \mathbb{N}_0$ that are incremented for each new entity becoming active in the system. The first instance of an entity gets the counter value 0. TASK_200MS has an instance counter value of 3 which means the event refers to the fourth instance of this entity. For entities that do not have a lifecycle like signals, the counter field is not relevant and 0 can be used as a placeholder value.

The seventh field α represents the way in which the target entity is influenced by the source entity. In this example TASK_200MS writes a new value to the signal entity EngineSpeed. Depending on source and target entity type, different actions are allowed by the specification as discussed in subsection 2.2.3.

For signal write events the note field ν is used to denote the value that is written to the signal in this case 42. The note field is only required for certain events. Table 2.7 summarizes the meaning of the different BTF fields.

A BTF trace can be persisted in a BTF trace file. This file contains two parts: a meta and a data section. The meta section is written at the beginning of the file. It contains general information on the trace such as BTF version, creator of the trace file, creation date, and time unit used by the time field. Each meta attribute uses a separate line, starting with a #, followed by the attribute name, a space, and the attribute definition.

```
#version 2.1.4
#creator BTF-Writer (15.01.0.537)
#creationDate 2015-02-18T14:18:20Z
#timeScale ns
     0,      Sim,  0, STI,       S_1MS, 0, trigger
     0,   S_1MS,  0,   T,     T_1MS_0, 0, activate
   100,  Core_0,  0,   T,     T_1MS_0, 0, start
   100, T_1MS_1,  0,   R,  Runnable_0, 0, start
 25000, T_1MS_1,  0,   R,  Runnable_0, 0, terminate
 25100,  Core_1,  0,   T,     T_1MS_0, 0, terminate
```

Listing 2.2: A BTF trace file contains of two sections. A meta section at the beginning of a file includes information such as creator, creation date and time unit. It is followed by a data section that contains one event per line. Comments are denoted by a number sign followed by a space.

In the data section one BTF event is written per line in chronological order. The first event of a trace is located directly after the meta section and the last event at the end of the file. Comments are denoted by a # followed by a space. Listing 2.2 shows an example trace file.

2.2.2 BTF Entity Types

As shown in Table 2.8 BTF specifies fourteen entity types that can be classified into five categories: environment, software, hardware, operating system, and information. Some entity types are not relevant for this thesis and therefore only discussed briefly. The actions or in other words the way in which one entity can be influenced by another are defined for each entity type as discussed in subsection 2.2.3. Actions for types that are classified as not relevant are not considered.

Environment contains only the stimulus entity type. Stimuli are used to depict application behavior that cannot be represented by other entity types. A stimulus can be used to activate a task or ISR and to set a signal value. Multiple stimulus instances can exist in a system at a certain point in time. Thus, the instance counter field is required for stimulus entities.

Software contains the task, ISR, runnable, and instruction block types. Tasks and ISRs summarized by the term process are containers for application software and discussed in section 2.1.

Runnable is a term established by AUTOSAR and relates to the concept of C type functions. A runnable can be executed from the context of processes and contains application specific functionality. Multiple runnables can be active in a system at the same time for example, if the same runnable is executed by two different tasks allocated to distinct cores. Hence, an instance counter is required for runnable entities.

Instruction blocks are used to represent execution time within the con-

Category	Entity Type	Type ID	Relevant
Environment	Stimulus	STI	X
Software	Task	T	X
	ISR	I	X
	Runnable	R	X
	Instruction Block	IB	
Hardware	Electronic Control Unit	ECU	
	Processor	Processor	
	Core	C	X
	Memory Module	M	
Operating System	Scheduler	SCHED	
	Signal	SIG	X
	Semaphore	SEM	X
	Event	EVENT	X
Information	Simulation	SIM	X

Table 2.8: BTF entity types can be divided into five categories. Types that are relevant in the context of this thesis are marked by an X.

text of runnables. Since these execution times become available implicitly via the corresponding runnable events, the addition of instruction blocks to a BTF trace is optional and does not provide any immediate benefits.

Hardware contains the electronic control unit (ECU), processor, core, and memory module types. An ECU consists of one or more processors. This allows it to represent a multi-processor system. Generally, tracing only supports the recording of a single processor. Multi-processor setups require a way to synchronize the measurement between multiple trace measurement tools. The design of such a setup is not in the scope of this thesis.

A processor is composed of one or more cores and recording multiple cores on the same chip is feasible via tracing. Cores are necessary to map software and OS events to the corresponding hardware entities. Since this information is important for the analysis of embedded systems, cores are relevant for this thesis.

Memory modules model different memory sections on a chip. They allow it to represent memory related processes on the CPU such as access times to variables or cache misses. According to Helm [19], direct measurement of memory access times is not possible. Instead, dedicated code must be added to the application in order to determine the execution times for different memory access operations. Due to the intrusiveness of this approach it is not feasible for real applications. Therefore, memory modules are not supported in this thesis.

Operating System covers scheduler, signal, semaphore, and event entity types. The scheduler entity type is used to represent actions executed by

the OS that relate to the scheduling of process instances. Scheduler events become available implicitly via the respective process actions and are thus not considered in this thesis.

Signals represent access to variables that are relevant for the analysis of an application. Consequently, signal events must be added to a BTF trace that is recorded from hardware.

Semaphores entities are used to control access to common resources in parallel systems. A process can request a semaphore before it enters a critical section, e.g. a section that contains an access to a memory region that is vulnerable to race conditions. If the semaphore is free the request is accepted, the semaphore is locked and all subsequent requests fail. Once the process has left the critical section it releases the semaphore.

Events are objects for inter-process communication provided by the OS. One process can use an event to notify another one for example, when a computation finishes or a resource becomes available. Event entities do not have a lifecycle therefore, no instance counter value is required.

Information contains only the simulation entity type. This entity type has two purposes. Firstly, it can be used to provide information about errors that occurred during trace recording. Secondly, it is required to trigger stimulus events. Since stimulus events are mandatory to represent task activations by non process objects, the simulation entity must be considered in the context of this thesis. Because *simulation* does not make sense in a trace recorded from hardware *system* can be used as a more appropriate term.

2.2.3 BTF Actions

BTF specifies different actions. The available actions are dependent on the source and target entity types of the respective event.

Stimuli only support the *trigger* action. A stimulus can be triggered by process and simulation entities. Once a stimulus is triggered it can be used for the actual event: the activation of a task or ISR or to set the value of a signal.

Process entities support the actions shown in Figure 2.8. A process instance starts in the *not initialized* state. From there it can be *activated* in order to switch into the *active* state by a stimulus entity. All state transitions except *activate* are executed by core entities. An active process is changed into the *running* state as soon as it is scheduled by the OS.

A running process can *preempt*, *terminate*, *poll*, and *wait*. Preemption occurs if another process is scheduled to be executed on the core. In this case, the current process changes into the *ready* state. A ready process *resumes* running once the core becomes available again. If a process finishes execution it *terminates* and switches into the *terminated* state. This finishes the lifecycle of a process instance.

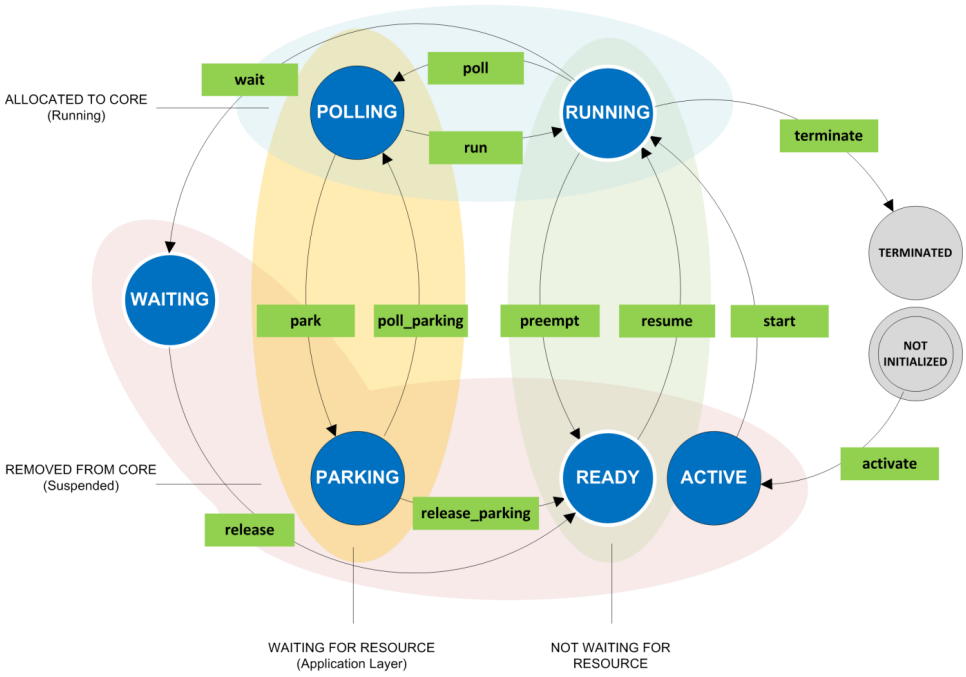

Figure 2.8: BTF [57] specifies more process states than OSEK/VDX (compare Figure 2.2). The additional states polling and parking are required to represent active waiting. Not initialized and terminated indicate the beginning and end of a process lifecycle. The green boxes between the states show the name of the BTF action for the respective transition.

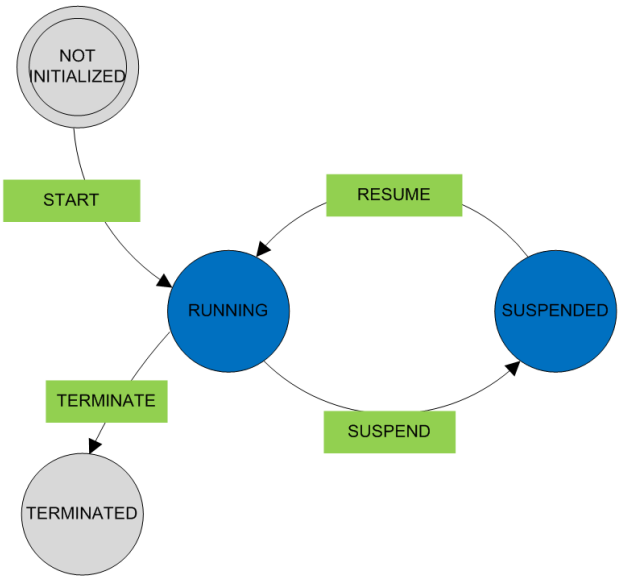

Figure 2.9: BTF runnable states and state transitions [57].

A process that *polls* a resource switches into the active waiting state *polling*. If the resource becomes available, the process continues running which is indicated by the *run* action. A process that *waits* for an event switches into the passive waiting state *waiting*. A *waiting* process is *released* into the ready state if one of the requested events becomes available.

A polling process that is removed from the core is *parked* and switched into the *parking* state. If the resource becomes available while the process is parking it is switched into the ready state. This transition is called *release_parking*. Otherwise the process continues polling, once it is reallocated to the core which is called *poll_parking*.

In addition to state transition actions, BTF specifies process notification actions. These actions do not trigger a process state change but indicate other events related to a process entity. The *mtalimitexceeded* action is triggered if more process instances than allowed are activated in parallel. If this happens, no new task instance is created. Therefore, a notification event is necessary to make the event visible in the trace.

All other process notification actions are related to migration the reallocation of a process from one core to another. OSEK OS does not support process migration since a separate kernel is executed on each core. Thus migration notifications are not relevant for an OSEK/VDX compliant OS. Additionally, migration actions become available implicitly via the respective process transition actions. If a process instance is preempted on one core and resumed on another, the resume event has a different source core than the preempt event. Consequently, the related migration event is known.

Runnable instances start in the *not initialized* state as shown in Figure 2.9. Runnables can be *started* by ISRs and tasks in order to switch into the *running* state. A runnable instance that *terminates* switches into the *terminated* stated and therefore finishes its lifecycle.

Because a runnable can only be executed from process context, it can not continue running if the respective process is preempted. In this case the runnable must be *suspended*. Once the process resumes execution the runnable can also *resume*.

Core entities are used to provide an execution context for process entities and cannot be used as a target entity themselves. Consequently, no BTF core actions are specified. Only one process can be allocated to a core at the same time and core entities do not have a lifecycle.

Signal entities can be influenced by two actions: *read* and *write*. A signal can be read within the context of a process entity. This means that the value of a variable is retrieved from memory. A signal entity does not have a lifecycle thus, the instance counter value for signals can remain constant.

Write actions can be executed by process and stimulus entities. They indicate that a new value is assigned to a variable. If this assignment is done from process context, the respective process entity is the source for the write event. Otherwise, a stimulus entity can be used to represent the source for example, if a signal is changed by the OS or a hardware module.

For signal writes, the note field must denote the value that was assigned to a variable. For read events the note field can optionally indicate the value of the variable that was accessed.

Semaphores can be categorized into different types. Counting semaphores can be requested multiple times. They have an initial counter value of zero. For every request, this counter is incremented and every time it is released the value is decremented. A counting semaphore is locked once the counter has reached a predefined value.

A binary semaphore is a specialization of a counting semaphore for which the maximum counter value is one. A mutex is a binary semaphore that supports an ownership concept. This means a mutex knows all processes that may request it. This information allows the implementation of priority ceiling protocols in order to avoid deadlocks and priority inversion. The OSEK/VDX term for mutex is *resource*, resources are discussed in subsection 2.1.1.

BTF semaphore events can represent all mentioned semaphore types. Semaphore actions can be divided into two categories: actions triggered by process instances as shown in Table 2.9 and actions executed by a semaphore entity itself as shown in Figure 2.10.

A process request to a semaphore is indicated by *requestsemaphore*. If a request is successful the semaphore counter is *incremented* and the process is *assigned* to the semaphore. The *exclusivesemaphore* action represents a semaphore request that only succeeds, if the semaphore is currently not

Action	Meaning
requestsemaphore	Process requests a semaphore.
exclusivesemaphore	Process requests a semaphore exclusively.
assigned	Process is assigned as the owner of a semaphore.
waiting	Process is assigned as waiting to a locked semaphore.
released	Assignment from process to semaphore is removed.
increment	Semaphore counter is incremented.
decrement	Semaphore counter is decremented.

Table 2.9: Processes can interact with semaphores in different ways. If a process requests a semaphore successfully, it is *assigned* to the semaphore and the counter is *incremented*, otherwise a *waiting* event is triggered. Once a semaphore is *released*, the assignment is removed and the counter is *decremented*.

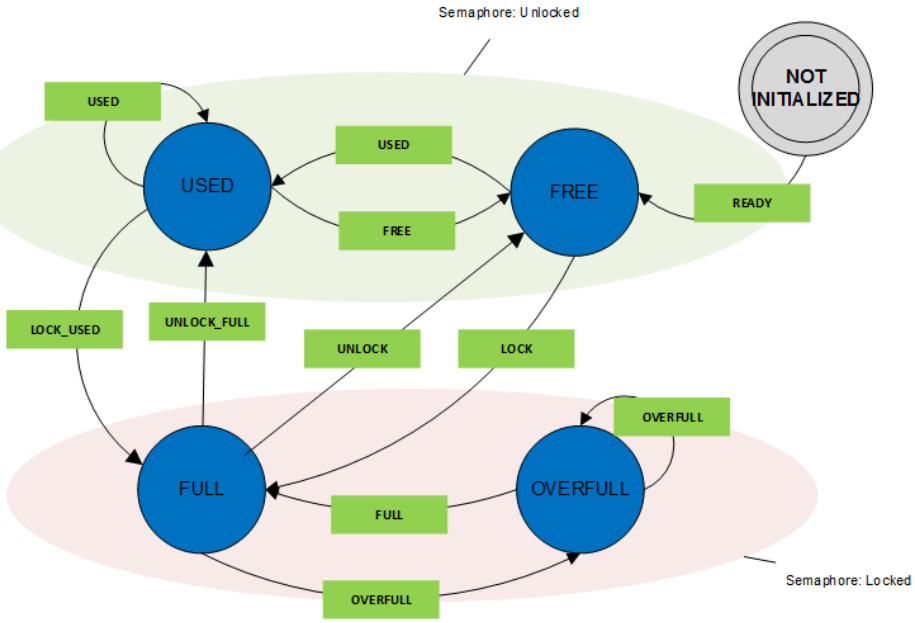

Figure 2.10: BTF [57] semaphore entities do not have a lifecycle. Nevertheless, they must be *initialized* before they are ready for the first time. A semaphore can be *unlocked* or *locked*. A counting semaphore can be requested multiple times in which case it changes into the *used* state. If there are no requests the semaphore is *free*. A semaphore that has at least as many requests as allowed is *full* and changes into the *locked* state. Further requests in the locked stated result in an *overfull* action.

requested by any other process, i.e. the counter value is zero. If a process fails to request a semaphore and switches into polling mode, indicated by the *waiting* action. A process that releases a semaphore *decrements* the semaphore counter and the respective semaphore is *released*, the process is no longer assigned to it.

Semaphores do not have a lifecycle which is why their instant counter remains constant. Nevertheless, a semaphore must be moved from the *not initialized* to the *free* state by the *ready* action before it is requested for the first time.

A free semaphore is not requested by any process. At the first request the behavior is dependent on the semaphore type. A mutex or binary semaphore is *locked* and moved into the *full* state. A counting semaphored is changed into the *used* state which is indicated by the *used* action. The used action is repeated for a counting semaphore for each further request or release as long as the counter value stays greater than zero and smaller than the maximum value. If the counter value of a used semaphore becomes zero this semaphore is *freed*. If the maximum counter value is reached the semaphore state becomes *full* which is indicated by the *lock_used* action.

When a full binary semaphore or mutex is released, it is *unlocked* and becomes free again, while a counting semaphore is changed back to the used state, indicated by the *unlock_full* action. A request to a full semaphore entity results in an *overfull* action and the state is changed to *overfull*. The overfull state indicates that there is at least one process polling a semaphore. Each additional request also results in an overfull action. Once there are no more processes waiting for a semaphore, this semaphore becomes *full* again.

Events can be influenced by three different actions. If a process starts waiting for an event, this is indicated by the *wait_event* action. Another process can set an event via the *set_event* action. For this action it is necessary to provide the entity for which the event is set via the BTF note field. An event can be cleared by the process for which the event was set which is indicated by *clear_event*.

3 Hardware Trace Measurement

Computer systems can be analyzed with measurement tools that detect events, i.e. changes in the state of a system [12, p. 28]. The same event can be interpreted on different levels as shown in Figure 3.2. A hardware trace tool can detect a voltage change in memory, e.g. triggered by the processor which is a hardware event. Accordingly, the variable that maps to the changed memory register changes too which is a software event. If this variable is related to the state of a task, a change of the variable also means a change of the task state which is then called a system event.

In many cases, the event of interest cannot be measured directly. One or more transformation steps are required to retrieve the required result. If a transformation process is executed the measurement is said to be indirect [12, p. 28]. Considering the previous example a task termination event cannot be measured directly. However, a variable that contains the current task state can be measured. If the task corresponding to the variable and the mapping from value to task state is known, a change of the variable can be transformed into a higher level event the termination of a task. After the transformation process the measurement results can be displayed to the user as shown in Figure 3.1.

During the transformation step the collected data may be manipulated which is called prereduction. Prereduction may for example be used when the actual event is not required, but rather the amount of events of a certain type that occurred. For this case the transformer would increment a counter whenever a certain event type is collected. If no prereduction is executed, the measurement process is called tracing. Tracing is the process of recording

Figure 3.1: The conceptual parts of a measurement process according to Ferrari [12]. A sensor measures data. One or more transformation steps are required if the data is not yet in the desired format. Finally the result can be presented to the user.

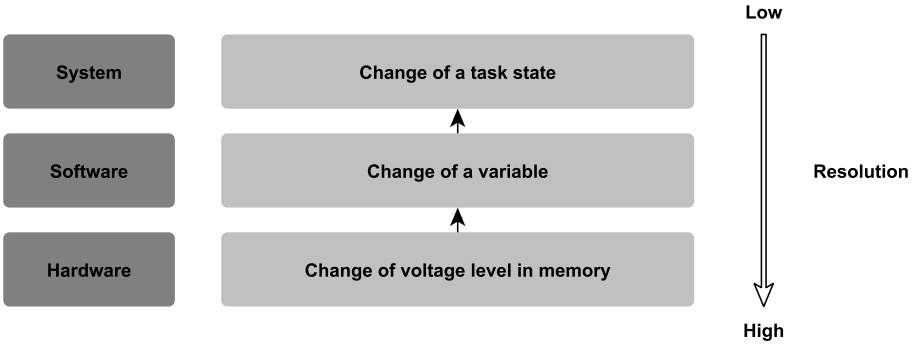

Figure 3.2: A measurement event can be interpreted on different levels. A voltage change in memory can be detected by a hardware trace tool capable of supervising the memory bus that triggers the voltage change. The memory section can relate to a variable, that changes in consequence of the voltage change, which is a software event. If the variable is related to the state of a task, a change of the variable also means a change of the task state which is then called a system event.

a sequence of events in chronological order of occurrence [12, p. 30]. The result of this process is called a trace.

3.1 Trace Tools

Ferrari [12, p. 31ff] distinguishes three trace measurement tools: software, hybrid, and hardware tools. All tools are meant to examine the behavior of a system. However, there are differences in interference, resolution, and cost as summarized in Table 3.1.

If a measurement tool uses resources of the target system it causes interference by using computational power and memory that could otherwise be utilized by the application. A tool that causes interference is said to be intrusive and may cause degradation, a reduction in performance of the target system [12, p. 29]. Consequently, intrusive trace tools change the real-time behavior of an application.

An event can be represented on different levels. A voltage level change in memory can map to a variable which can map to the state of a task as visualized in Figure 3.2. Those levels are called hardware level, software level, and system level. To clarify the level of a trace, it can be mentioned explicitly. For instance, a trace consisting of hardware level events is a hardware level trace [32, p. 29f]. Tools that can detect hardware events occurring at a microscopic level are said to have a higher resolution than tools that can detect software events only.

Different trace techniques can detect and record events with different

frequencies. The maximum frequency is usually not limited by the speed with which events can be detected, but by the available bandwidth to process and record the detected events.

The cost of different trace tools depends on several factors, the price for hardware and software licenses, the price for installing and maintaining the tool, educational costs, like training for the users of a tool, and the costs of operating the tool.

Software tools add instructions to a hardware-software system in order to detect and record events of interest. Added instructions are called instrumentation. The simplest kind of instrumentation is a classical write to the standard output interface, e.g. a printf statement in the C programming language. Instructions may be added to the application code directly, via the compiler or post compilation via dynamic binary instrumentation [60][33]. If no standard output interface is available, events are recorded into memory on target. From there they can be read out via debugger or serial interface. Instrumentation always interferes with the application. There are two components of interference, a space, and a time component [12, p. 44]. Execution of instrumentation code takes time and storing detected events uses memory space. Software tools have a low resolution because they cannot detect events on a hardware level. Event detection frequency is limited by the available computational resources. On the upside they are usually cheap and easy to implement and use.

Hardware tools do not rely on instrumentation which means that they are non intrusive and do not interfere with the application [34]. Hardware tracing works via a dedicated trace device chip that is located on the silicon of the CPU. Trace devices provide a very high resolution since they are capable of detecting events at hardware level [38]. Additionally the event detection frequency can be as high as the actual system frequency, thus it is possible to record a complete hardware-software system in real-time. Hardware tools are more expensive compared to software solutions. Installation and maintenance are more complex and require properly qualified users.

Hybrid tools rely on instrumentation and a dedicated hardware interface to record events. The boundary between software, hybrid, and hardware tools can be fuzzy in certain cases. Software tools need some kind of hardware interface to send recorded traces off-chip. In this sense, all software tools are hybrid tools. However, industry hybrid solutions often require proprietary target interfaces which justifies why these tools fit into a separate category [46]. Compared to pure software tools, hybrid tools interfere with the system to a lesser extent [39]. A dedicated hardware interface allows it to send events off-chip in real-time. Consequently, more memory becomes available on target.

As shown in Table 3.1 hardware trace tools have many advantages over hybrid and software based solutions. Hardware tracing does not interfere with the system, which is especially important for real-time systems. Hard-

	Software	Hybrid	Hardware
Interference	high	low	no
Resolution	low	low	high
Cost	low	low	high
Frequency	low	low	high

Table 3.1: Properties of different trace measurement tools [31, p. 6]. Hardware tools are superior to software and hybrid tools but come with higher expenses.

ware trace tools are capable of detecting events with a higher resolution and frequency. Additionally the trace duration of software and hybrid traces is limited to the available memory on target and to the trace interface bandwidth. When the same quantity can be measured by a hardware and a software tool, the values obtained by the hardware tool are usually to be considered more accurate because of the lower interference [12, p. 45].

3.2 Hardware Tracing

Hardware tracing is capable of recording events on hardware level. A dedicated on-chip trace device and trace interface is required to record hardware events and send them off-chip [37]. Target access hardware is connected to the trace interface to readout the trace measurement results. From there the events are forwarded to a host computer for further processing. Software that runs on the host computer in order to analyze the recorded trace data is provided by the target access hardware vendor [25]. The term host software is used to refer to such applications.

The on-chip trace device is designed to record hardware events executed by the microcontroller. It occupies a separate section on the silicon. Usually a controller is delivered in two versions, one with and one without trace device. In production the ability to execute trace measurement is not required [34]. Therefore, the trace device would only increase chip costs without providing any benefits.

Figure 3.3 shows the trace device of the Infineon TC27x microcontroller family [2]. The upper part belongs to the product chip while the lower part displays the trace device. The trace device can gather data from the product part via two interfaces. POBs (Processor Observation Block) record processor events while BOBs record bus events. All events are collected, enhanced with a timestamp and buffered in the on-chip trace memory. From there they are sent off-chip via the dedicated trace interface.

There exist different techniques to add timestamp information to a trace event. The obvious way is shown in Figure 3.4. A timestamp is added to each trace event that is sent off-chip. To save bandwidth timestamps are provided relatively to the previous event. An absolute value is computed by

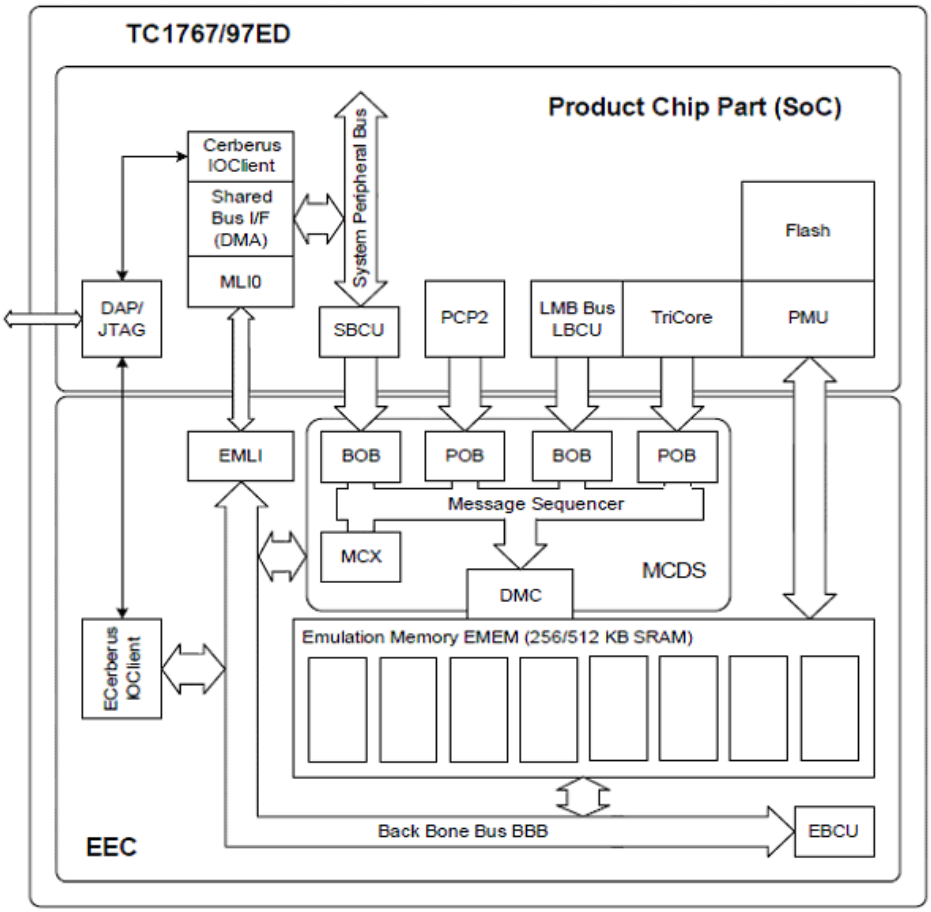

Figure 3.3: A microcontroller with hardware trace support consists of two sections. A regular product chip part and the trace device part. The trace device part can be omitted in the production version of a chip to save costs [23].

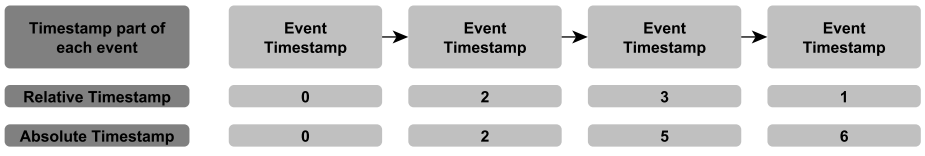

Figure 3.4: Each trace event is assigned a timestamp relative to the previous event. By summing up the relative timestamps absolute values can be generated.

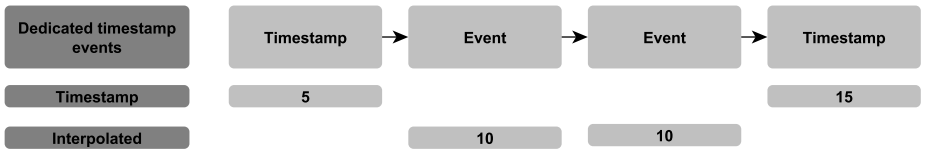

Figure 3.5: Via dedicated timestamp events, the timestamps of the other events can be interpolated. In this example two events are recorded between the previous and the next timestamp event. This is why both events get the same timestamp, based on these events. The value is calculated via Equation 3.1 as $t_i = 5 + \frac{(15-5)}{2} = 10$.

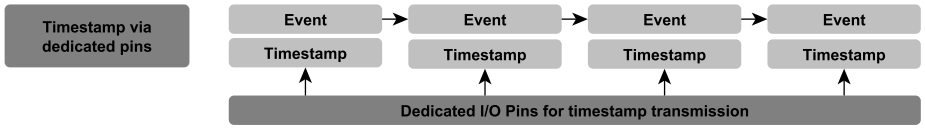

Figure 3.6: Dedicated I/O pins can be used to output a timestamp value whenever a measurement event is sent off-chip.

summing up all previous timestamp.

Another way is to send dedicated timestamp messages as shown in Figure 3.5. The timestamps for the actual trace events are then interpolated, e.g., via the equation

$$t_i = t_p + \frac{(t_n - t_p)}{2}, \tag{3.1}$$

where t_p is the previous timestamp (the latest timestamp before the event), t_n the next timestamp (the soonest timestamp after the event) and t_i the timestamp interpolated based on the dedicated timestamp events.

Finally, timestamps can also be created via dedicated I/O pins as specified by the Nexus [61] standard. This means that whenever a trace event is sent off-chip via the trace interface, the current timestamp is provided via the I/O pins as shown in Figure 3.6.

Cycle accurate timestamps are feasible with all timestamp generation techniques. However, timestamp accuracy and resolution are only partly dependent on the generation technique. More important factors are CPU and trace device clock frequency, as well as the design of CPU and trace device. For cycle accurate timestamps, trace device frequency must be greater or equal to CPU frequency. Even if this is the case, cycle accurate timestamps cannot necessarily be guaranteed.

For example, super scalar processors like the Infineon TC277 [2] are capable of executing more than one instructions per cycle. However, only one event can be processed per cycle by the trace device as shown in Figure 3.7. The processor observation block filters the instructions according to user

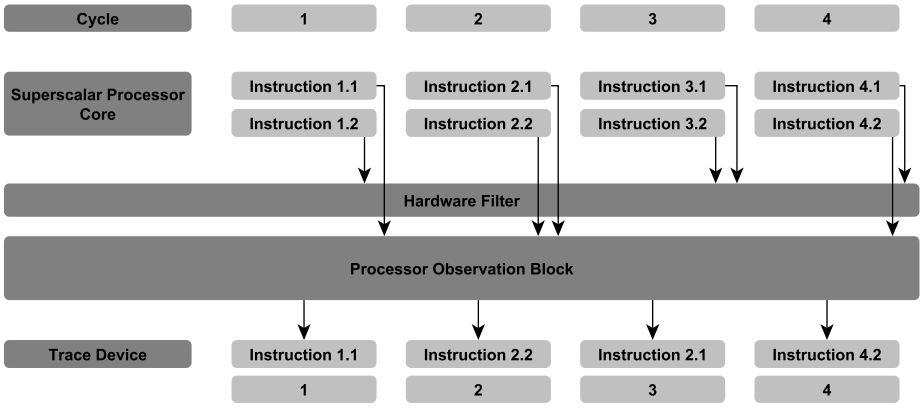

Figure 3.7: Even if the trace device runs at CPU clock frequency, cycle accurate timestamps cannot be guaranteed.

specified filter rules and forwards them for further processing. If two instructions, executed during the same processor cycle, match the filter and are thus forwarded to the trace device, one of those instructions is delayed by one cycle (in this example Instruction 2.1). For a processor running at 100 MHz this would set the timestamp off by 10 ns for this particular event.

The design of trace devices differs depending on the processor family and the processor vendor. However, the general concept and provided functionality are the same for all devices. Various standards for the implementation of trace devices are specified and used by chip vendors. Three common standards are Nexus used by PowerPC processors [61], ETM (Embedded Trace Macrocell) used by ARM processors [62, p. 476], and the Infineon Multicore Debug Solution [51] discussed here and shown in Figure 3.3.

According to Figure 3.1, a measurement process starts with the detection of an event by a sensor. In case of the trace process the sensors are the POBs and BOBs. Each POB monitors the instructions executed by one processor core. This means the complete program flow executed by a processor core can be recorded. BOBs are connected to the data busses of the microcontroller and can detect memory access events. A memory access event may be for example, writing to a variable or reading from a special function register. A typical data trace event contains in addition to the timestamp, details like address, data value, transfer size, and whether a read or write access occurred [21].

Filters can be specified by the user to reduce the amount of recorded trace events. They can be set for an address or for an address range. Different events can be executed if an address filter matches: the corresponding event can be recorded, discarded or another event can be triggered. For example, it is possible to start or stop the trace process if a specific function is accessed

or a variable is written. Filter configuration is done via the host software.

Corresponding to the two main hardware event types, instruction, and data access events, two hardware trace techniques can be distinguished, program flow trace and data trace [34]. The two trace techniques can be executed in parallel or individually as configured by the user.

A **program flow trace** (also called function trace) shows the complete execution path of an application for the duration of the trace recording. This means it is possible to detect when a certain function is called or which branch of an if statement is executed. The amount of instructions and the resulting data stream bandwidth produced by a modern CPU is too big to be transmitted via the trace interface. To solve this problem trace devices use trace compression. The most commonly used program flow trace compression technique works by detecting and recording only such instructions that cause a change in program flow such as conditional jumps and traps [21]. Using the application binary the host software is able to reconstruct the complete program flow.

A **data trace** is a sequence of data access events. Data tracing allows it to supervise and to debug the state of variables in memory. Data tracing of all active units is becoming increasingly important because not all data interactions involve a processor [35]. Thus, trace devices must also be able to detect memory accesses via DMA (Direct Memory Access) and accesses to memory of special on-chip modules like FlexRay or Ethernet. The units that are supported by a microcontroller are depended on the trace device, but all trace devices support tracing the main memory of a controller. Compression is also applied to data traces. However, those techniques are usually not sufficient to record a complete data trace of significant length since the amount of generated data is too big. The best way to solve this problem is to apply filters to avoid detecting and recording data events in memory sections that are not of interest [21].

A recorded hardware trace event is buffered into an on-chip trace memory. From there the events can be read via the trace interface. On-chip trace memories can be operated in different modes [34]. In continuous mode the trace data is streamed of chip in real-time. This technique is limited by the bandwidth of the trace interface. If it is high enough the trace duration is only depended on the available memory on the host computer and traces of arbitrary length can be recorded. If the bandwidth is too small to process the recorded trace stream *buffer mode* must be used. This means the recorded trace is written into trace memory and read out by the target access hardware post tracing. Buffer mode can be used in pre- and post-trigger mode. In pre-trigger mode the trace buffer is filled like a circular buffer. The oldest events are discarded for new events. The trace process can be stopped at an arbitrary point in time and the latest trace events become available. In post-trigger mode the trace process is stopped as soon as the buffer has been filled for the first time.

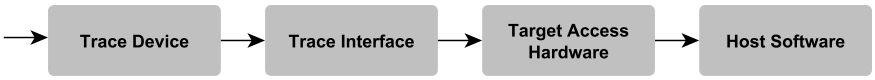

Figure 3.8: Recording a hardware trace and making it available to the user requires multiple steps. Hardware events must be measured on target via a trace device. Using a trace interface the recorded data can be readout by the target access hardware and transmitted to a host computer. Target access hardware vendors provide special software to analyze and visualize the recorded trace.

Standard	Architecture	Function Trace	Data Trace
Nexus	PowerPC	Branch Trace Messaging	Data Trace Messaging
ETM	ARM	Program Trace Macrocell	Embedded Trace Macrocell
IMDS	TriCore	Processor Observation Block	Bus Observation Block

Table 3.2: Trace devices exist for different CPU architectures. All solutions provide methods for recording program flow and data traces.

A trace device operated in buffer mode is limited by the available trace memory. The trace memory size of an Infineon TC275 microcontroller (Figure 3.9 a)is 2 MB which allows for approximately 33 ms of unfiltered function and data trace of a single processor core running at 200 MHz [34]. Depending on the measurement use case this may be sufficient or not. If the trace duration should be increased tracing in continuous mode is mandatory. Continues tracing requires a high bandwidth interface such as AGBT (Aurora Gigabit Interface).

3.3 Hardware Trace Toolchain

Multiple steps are required from recording a hardware trace on target to presenting it to the user on a personal computer as shown in Figure 3.8. Many different solutions exist for each of those steps. Nevertheless, the basic functionalities provided by all solutions is comparable to each other.

The basic prerequisite for executing a hardware trace is the availability of an on-chip trace device. All major chip vendors provide trace devices for their microcontrollers that support program flow and data trace. Table 3.2 gives an overview of the state-of-the-art trace solutions.

Events that have been recorded by the trace device are sent off-chip via a dedicated trace interface. If the bandwidth provided by an interface is lower than the transfer rate of created events continuous tracing is not possible. However, this use case is often required. There are two ways two solve this

Interface	Pros/Cons	DAQ rate $[MB/s]$
JTAG	+ Reuse of existing interface + Small chip area − Low bandwidth	1.2
DAP2/SWD	+ High bandwidth with few pins + Small silicon area − Proprietary	10
AGBT	+ Very high bandwidth with few pins − Large silicon area − High cost	30
CAN	+ Robust and well known standard + Low cost − Very low bandwidth	0.05

Table 3.3: Commonly used trace interfaces and their DAQ (Data AcQuisition) rates. AGBT (Aurora Gigabit Interface) is the only interface capable of recording continuous hardware traces of a complete system.

problem. The amount of created trace data can be reduced using filters or the available bandwidth can be increased. If an entire application must be analyzed as a whole the first way is not an option.

Mayer et al. [36] give an overview of trace interfaces used in the automotive industry as shown in Table 3.3. JTAG (Joint Test Action Group) is a common debug standard [22], suitable for regular debugging. It can be used to read out a buffered traced post tracing, but for continuous tracing it is not sufficient due to its low bandwidth of 1.2 MB/s. Because of that DAP and DAP2 were developed by Infineon and SWD by ARM. Both protocols are based on JTAG but use a higher frequency and improved communication protocols to provided more bandwidth.

AGBT is currently the fastest trace interface. It was specified by XILINX and adopted by the Nexus standard. AGBT is the only interface which is theoretically capable of recording a continuous trace of a complete application running on a processor with a frequency of 200 MHz. CAN is used by some hybrid trace tools but is only mentioned for completeness since its bandwidth is too low to be considered for hardware tracing.

Target access hardware is connected to the hardware interface to readout recorded trace events. From the target access hardware the data is transmitted to a host computer for further analysis via USB 3.0 or Ethernet. Examples for target access hardware are the iC6000 by iSYSTEM [24] (Figure 3.9 b) and the PowerTrace-II by Lauterbach [59] (Figure 3.9 e).

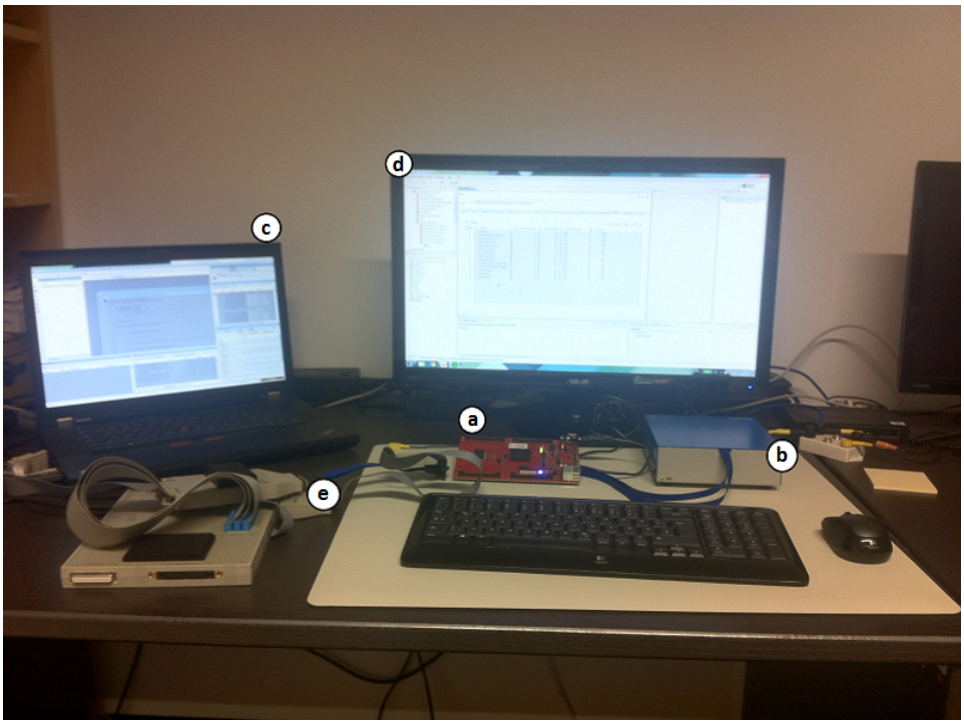

Figure 3.9: A complete trace workbench. An Infineon TriCore evaluation board (a) can be traced by the iSYSTEM iC6000 (b) or the Lauterbach PowerTrace-2 (e) via the highspeed AGBT interface. Host software is used to control the hardware and to analyze the recorded trace, for example WinIDEA (c) by iSYSTEM and TRACE32 (d) by Lauterbach [20].

Both devices support different architectures and trace interfaces by using architecture specific debug cables. Besides reading hardware traces those devices also support all functionalities provided by a regular debugger such as step wise debugging, reading of memory content, and manipulation of CPU configuration registers.

Dedicated software on the host computer is used to configure and control the target access hardware and the trace device itself. After recording, this software transforms the recorded hardware trace into a software trace (see Figure 3.2). For this process the host software must have access to the ELF file of an application. This is required to map the addresses of hardware trace events to the corresponding software entities. Based on the software trace, different analysis techniques such as metric evaluation, performance analysis, and code coverage are supported. Gantt charts are provided to examine the trace visually. Via export functions a software level program flow and data trace can be made available for external tools. Figure 3.9 shows the toolchain described in this section.

4 Mapping

Systems are analyzable on different levels of abstraction as shown in Figure 3.2. Depending on the use case, one or another level is more sufficient to perform the required analysis. For example, a hardware designer does not care about task states while a system engineer is usually not interested in voltage levels of transistors in memory.

For the timing analysis of an embedded system a trace on system level is required because timing requirements are usually specified for system entities such as tasks or signals. Hence, system level traces contain the information necessary to validate an application with respect to its timing behavior.

A trace long enough, so that all relevant entities appear with sufficient frequency for the timing analysis, is required. For example, at least two task instances must be activated in one trace to calculate the activate-to-activate time. Additionally, it is important not to influence the timing of an application by trace measurement. Consequently, the only sufficient trace technique for the timing analysis of embedded systems is hardware tracing according to Table 3.1.

Hardware tracing records events on hardware level. As stated above this level is not sufficient for the timing analysis of an embedded system. Thus, it is necessary to transform hardware events to system events as shown in Figure 4.1. Two steps are required for this transformation. Hardware level events must be transformed into software level events which are then further processed into system level events.

The first step is done by the trace software. It is capable of analyzing and interpreting the hardware events that are recorded from the processor. Via the application binary files it is possible to map the raw memory addresses contained in the hardware events to the corresponding symbols of the real application as depicted in Figure 4.2.

Depending of the trace device further steps may be required. For example, some trace devices produce timestamps, relative to the previous event which must then be transformed into absolute timestamps. Another example are program flow traces. Hardware level program flow events are usually only recorded for instructions that change the flow of an application as described in section 3.2. Only with the application binary it is possible for the software to reconstruct a complete program flow trace.

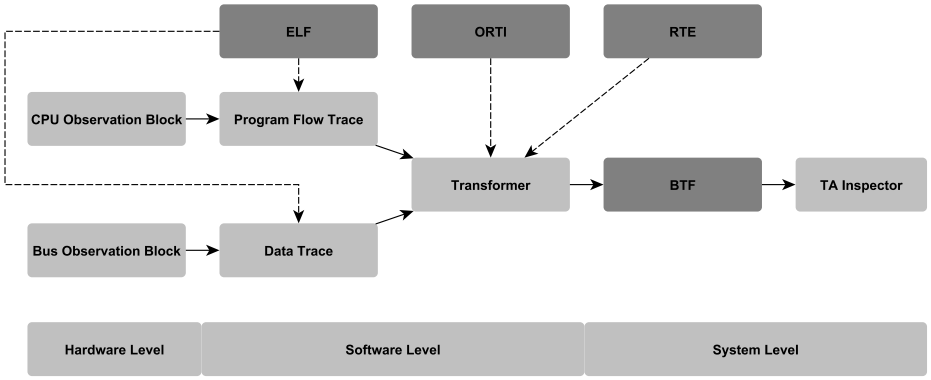

Figure 4.1: Hardware tracing records events on hardware level. This is not sufficient for the timing analysis of an embedded system. Thus, it is necessary to transform the hardware events to system events. This requires two steps. In the first step hardware events are transformed to software events. This step is done by the trace software and requires the application binary. The next transformation step produces a trace on system level, e.g. in the BTF format. An ORTI file as well as additional information that can for example, come from a timing model file (RTE) are required for this step.

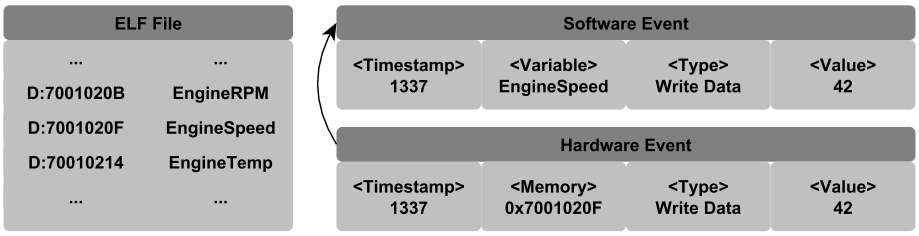

Figure 4.2: The trace software is capable of transforming a hardware level event to a software level event. This involves for example, changing memory addresses with the actual symbol names based on the application binary (ELF file). Further actions may be required depending on the trace device. Note that the displayed hardware event is just a generalization, the actual structure can be different depending on the trace device vendor.

Based on the software level trace, a system level trace can be generated in the next step. A suitable system level trace format is BTF which is described in section 2.2. It is capable of representing the behavior of an application in a way that is eligible for its timing analysis. Different additional information, e.g. the ORTI file is required to execute the transformation from software to system level trace.

4.1 Mapping Proceedings

Transformation from hardware to software level is done by the trace software. Corresponding to the composition of an on-chip trace device it creates two types of traces on software level: a data trace and a function trace.

Let i be an index in $\mathbb{N}_0$ denoting an individual event occurrence. Then a data event can be defined as an octuple

$$d_i = (t_i, \pi_i, a_i, v_i, c_i) \tag{4.1}$$

where $t_i \in \mathbb{N}_0$ is the timestamp in nanoseconds, π_i is the name of the accessed variable, $a_i \in \{R, W\}$ is the way in which the variable is accessed either R for read or W for write, $v_i \in \mathbb{N}$ is the value that was read or written and c_i is the core name on which the access has occurred.

Consequently, a data trace can be defined as a sequence of data events where $n \in \mathbb{N}_0$ is the number of events in the trace.

$$D = (d_1, d_2, \ldots, d_n) \tag{4.2}$$

Let j be an index in $\mathbb{N}_0$ denoting an individual event occurrence. Then a function event can be defined as a quadruple

$$f_j = (t_j, \pi_j, \theta_j, c_j) \tag{4.3}$$

where $t_j \in \mathbb{N}_0$ is the timestamp in nanoseconds, π_j is the name of the accessed function, $\theta_j \in \{A, \Omega\}$ indicates whether the function has started (A) or terminated (Ω), and c_j is the core name on which the function event has occurred.

After that a function trace can be defined as a sequence of function events where $m \in \mathbb{N}_0$ is the numbers of events in the trace.

$$F = (f_1, f_2, \ldots, f_m) \tag{4.4}$$

Based on Equation 2.1, Equation 4.2, and Equation 4.4 the goal is to describe a function g so that

$$g : (D,\, F) \to B, \tag{4.5}$$

where the timestamps t of the events in D, F, and B are relative to the same point in time. However, D and F alone are not sufficient for the transformation from software to hardware level because of three reasons.

Firstly, the events on software level do not provide enough information to decide which variable maps to a certain entity on system level. For example, the state of each task is stored in a certain variable. Whenever the state changes, this variable changes too and a data event is generated. However, the transformation function does not know that the variable maps to the state of a task. Because of that the ORTI file described in subsection 2.1.3 is required. Via this file it is possible to relate variables to the corresponding system objects.

Secondly, not all entity types specified by BTF for example, runnables and signals are included in the ORTI file. The former are included in the function trace, the latter in the data trace. But if the transformation function is not able to distinguish regular functions from runnables and regular variables from signals this information cannot be used. Thus, it is necessary to provide a list of those entities to the transformation function.

Finally, it is necessary to keep track of the internal state of an application. If the ORTI file is available it can be detected that a certain task has changed its state. Consequently, a BTF event must be generated. Without the knowledge about the previous task state however, it is not possible to decide which task action has occurred. If the task changes into the running state, this could mean that the task has started for the first time resumed from ready state or continued to run after polling a resource.

Because of this reasons the function g must be redefined as

$$g' : (D,\, F,\, o,\, l,\, S) \rightarrow (B,\, S') \tag{4.6}$$

where o is the ORTI file of the traced application, $l = (l_r,\, l_s)$ is a tuple that contains a list of runnables l_r and a list of signal names l_s, and S and S' are the system states before and after the transformation. The information must be part of the system state S is discussed in the next sections.

4.2 ORTI Mappings

Task entities are capable of executing twelve actions according to Figure 2.8 plus the additional notification event if the MTA limit is exceeded. The lifecycle of a task entity starts with its activation.

An **activation** can be detected via the ORTI *task status* attribute. If no other task instance of the same task entity is active in the system, a task whose state changes to ready is activated. However, this does not work if a task instance of the same task is already active in the system. This can happen if multiple task activations are allowed by the OSEK Conformance Class. In case of a MTA the corresponding OSEK/VDX *task status* attribute

Action	ORTI attribute	System state
trigger (ipa)	servicetrace (ActivateTask)	running task
trigger (alarm)	alarmtime	-

Table 4.1: In BTF, a stimulus must be triggered so that it can activate a task. On target a task can be triggered via an IPA or by an alarm. The first can be detected via the *servicetrace* attribute, while the latter is indicated if the *alarmtime* attribute reaches the value zero.

already indicates an active state (any state that is not suspended) and will not change to ready again.

Consequently, another way to detect task activations is required. Via the ORTI *currentactivations* attribute, the number of open activations for each task can be detected. Whenever this attribute is incremented, a new task activation BTF event must be created. Therefore, it is necessary to keep track of the number of activations for each task entity in the system. Only if the previous number of activations for a task is known, it is possible to decide whether the value is incremented or decremented when a new data write event occurs. Thus, the number of current activations for each process is a relevant information and must be part of the system state s.

Since tasks have a lifecycle it is necessary to keep track of the instances for each task entity. Whenever a new task is activated the instance counter must be incremented and the counter value is assigned to the task. The same procedure is necessary for all other entities that have a lifecycle. The latest instance counter value for each entity must be available in the system state s to create correct BTF events. Additionally, it is necessary to add newly created tasks to a list of task instances active in the system. When a task's lifecycle ends, i.e., the task terminates, it is removed from this list.

A **stimulus** is required to activate a task. Stimuli can be **triggered** by process and by simulation entities. A stimulus triggered by another process represents an Inter-Process Activation (IPA). An IPA is implemented via the ActivateTask service routine. The ORTI *servicetrace* attribute can be used to detect when this routine is executed. Whenever the ActivateTask routine is entered and a task is running on the same core a stimulus event is created with the task as the source entity.

Alarms are the second way to activate tasks. The *alarmtime* attribute indicates how many ticks are left until an alarm expires. The ORTI file also contains the action that is executed by an alarm. Thus, a stimulus can be triggered whenever an alarm that activates a task reaches an *alarmtime* value of zero.

A triggered stimulus must be added to the system state. Later, when the actual task activation is executed by the OS the latest stimulus is removed from the system state and used to create a correct BTF event. Table 4.1 summarizes how stimulus events are detected.

```
TASK(EngineManager) {
  /* Wait actively until EngineResource becomes available. */
  while(GetResource(EngineResource) != E_OK);
  engineRPM = calculateEngineRPM();
  ReleaseResource(EngineResource);

  TerminateTask();
}
```

Listing 4.1: The BTF polling state indicates that a process is actively waiting for a resource. This listing shows how this might be impolemented in C.

A **task start** event occurs if a task which was previously active changes to running. There are two cases for which preempt and resume actions must be created. The first case is a normal state change that can be detected via the *task status* attribute. A task is **preempted** if the state changes from running to ready and **resumed** if the state changes from ready to running.

However, the task state is not updated by the OS when a task is preempted by an ISR. Consequently, a task preempt event must also be created, if the *runningisr2* attribute indicates that a new ISR is running on the core and a resume event must follow once the ISR terminates execution.

A task **terminate** event occurs if a running task changes into the suspended state. The previous state must not be known because a task can only be terminated from the running state.

However, there is a special case for task terminate events. As mentioned in subsection 2.1.1, a task with pending activations switches directly into the ready state, after the current instance terminates. To work around this problem it is necessary to detect when a certain task instance executes the TerminateTask service routine via the *servicetrace* attribute. If this happens a flag in the system state must be set to indicate that the respective task instance has been terminated. Whenever a task changes from running to ready this flag must be checked to decide whether the corresponding event is a preemption or a termination.

A **wait** event occurs if a running task waits for an event that is not set. In this case the OS will change the task state to waiting and the task is removed from the core. A **release** event occurs once the event is set and the OS changes the task state to ready.

Poll actions are more difficult to detect, since they are not directly related to a concept specified by OSEK OS. The idea of the BTF polling state is to indicate that a task is actively waiting for a resource. In code this can be implemented via a loop in which a resource is requested repeatedly until it becomes available as shown in Listing 4.1.

Via *servicetrace* and *lasterror* it can be detected that a process has requested a locked resource: the *servicetrace* attribute indicates when the

GetResource service routine is called and E_OS_RESOURCE is written to the *lasterror* attributed in case the resource is locked.

However, a single request does not necessarily mean that a change into the polling state is happening. Instead a task might just execute one code segment, if the resource is available and a different one, if it is not. Therefore, it is necessary to set a *previous request* flag for a task instance that has requested a locked resource once. If another request follows in the same running interval a poll event is generated. Once there are no more requests, the last request must have been successful and a run event is created to indicated the state change from polling to running. Then the previous request flag must be cleared.

A **park** action must be created if a task that is in polling state is changed into the ready state. Next, it is necessary to detect resource state changes of the resource which the parking task has been polling. If the respective resource changes into an unlocked state, a **release_parking** event is created. On the other hand, if the resource stays locked and the task changes back into running state, a **poll_parking** event is required.

The **mtalimitexceeded** notification event is the last task event that must be detected. This event is created, if a task activation gets triggered, but no actual task instance is added to the system. An OSEK/VDX compliant OS writes an E_OS_LIMIT error into the *lasterror* attribute, if a task activation is triggered, but the maximal MTA value is already reached. To create a valid BTF event it is necessary to know for which task entity the error is created. Since ORTI does not provide this information the creation of *mtalimitexceeded* events is not feasible. Table 4.2 gives an overview of the task mapping.

ISRs and tasks share the same BTF state model. However, OSEK/VDX does not specify a detailed state model for ISRs as it does for tasks. Consequently, the basic process actions activate, start, resume, preempt, and terminate are detected differently compared to task actions as shown in Table 4.3. ISRs are not allowed to wait for events. Therefore, waiting related process state transitions must not be considered. The detection of semaphore polling events works equally to task events and is therefore not discussed again.

An Interrupt Service Routine is triggered by a hardware interrupt. This means if the hardware detects a certain condition, e.g., an I/O pin state changes from high to low, the program flow is interrupted and a certain code section that is mapped to this interrupt is executed. Depending on the trace device, it may or may not be feasible to detect the activation of an interrupt via the corresponding ISR control register.

In the former case, it is possible to create a stimulus and the resulting activate event by detecting when the interrupt activate bit is set in the corresponding control register. Otherwise, the **activate** event must be created when the ISR changes into the running state for the first time. In this case

Action	ORTI attribute	System state
activate	currentactivations	currentactivations, last stimulus
start	state (running)	state (active)
resume	state (running)	state (ready)
resume	runningisr2	running task
preempt	state (ready)	task not terminated
preempt	runningisr2	running task
terminate	state (suspended)	active tasks
terminate	state (ready)	task terminated
wait	state (waiting)	-
release	state (ready)	state (waiting)
poll	lasterror	servicetrace, previous request
run	servicetrace	state (polling)
park	state (ready)	state (polling)
poll_parking	state (running)	state (parking)
release_parking	resource state	state (parking)
mtalimitexceeded	lasterror	entity cannot be detected

Table 4.2: Different pieces of information are required to detect all possible task actions. The states in the ORTI attributes column are OSEK/VDX task states while the states in the system information column are BTF process states. The previous state is necessary to create correct events. For example, a task state change to running could mean a BTF start, resume or run event.
For some actions, it is necessary to use multiple approaches to detect them. For example, a task terminate event happens if the OSEK/VDX state of changes to suspended. However, if another entity of the same task is already activated, a change to suspended does not occur. To catch this case it is necessary to set a *task terminated* attribute for a task instance when it calls the TerminateTask service routine.

Action	ORTI attribute	System state
activate	-	-
start	runningisr2	ISR stack
resume	runningisr2	ISR stack
preempt	runningisr2	ISR stack
terminate	runningisr2	ISR stack

Table 4.3: The *runningisr2* attribute is used to detect basic ISR actions. Because ISRs are not allowed to wait for events, waiting state related actions must not be created. All other actions can be detected in the same way as for task instances as shown in Table 4.2.

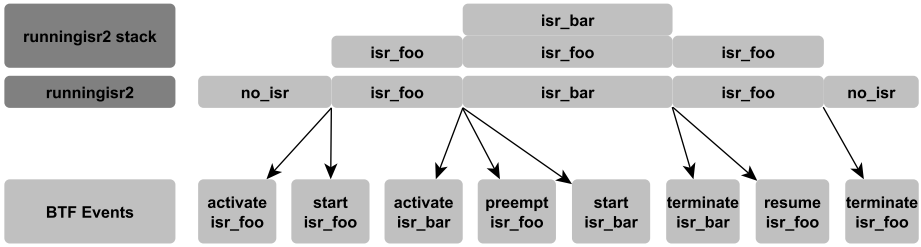

Figure 4.3: A stack can be used to track the active ISRs in a system. This is necessary to create appropriate BTF events. For example, the event when *isr_foo* is set as the running ISR, is different, depending on the current state of the stack. If the ISR is already on the stack, a resume event must be created, otherwise a start event.

trigger, activate, and start event are all created with the same timestamp.

The currently running category two ISR is indicated by the *runninngisr2* ORTI attribute. Each ISR has an unique ID that is written into the variable, if the respective entity is running. Otherwise runningisr2 is zero which indicates that no ISR is active. Mapping from ID to name is included in the ORTI file. If *runningisr2* changes to the ID of a certain ISR, it is not possible to decide whether this instance runs for the first time or whether it is resumed, after it has been preempted by an ISR with higher priority as shown in Figure 4.3.

Therefore, it is necessary to keep track of the active ISR instances in the system, e.g. via a stack. Whenever the value of *runningisr2* changes it is checked whether the corresponding ID is already on the stack. If so, the ISR was already running and has been **preempted**. Consequently, the ISR that caused the preemption has terminated and must be popped off the stack. The ISR that has been preempted must be **resumed**.

The other case is that the new ISR has not been running yet, i.e. is not on the stack. This means that the ISR on top of the stack, if there is one gets **preempted** and the new ISR is **started** and pushed on the stack. If *runningisr2* becomes zero the last ISR is popped of the stack and **terminated**.

As the name indicates, *runningisr2* is only written for category two interrupt routines. Regular ISRs are not managed by the OS and therefore not detectable via ORTI attributes. Instead function trace must be utilized to detect when a category one ISR is started or terminated. To map the function names to actual ISR entities, a list of category one ISRs is required. If such a list is available, the proceeding is the same as described above.

Runnable actions are detectable via function events. Start and terminate events must be created for function entry and function exit events. A program flow trace contains the information about all functions in the

Action	ORTI attribute	System state
start	-	running process
terminate	-	running process
suspend	task state	running process, process runnables
resume	task state	running process, process runnables

Table 4.4: Runnable start and stop events can be detected via function tracing. The source entity for a runnable event is the process in whose context the runnable is executed. A runnable is suspended when the corresponding process is preempted. If the process resumes, the runnable is resumed, too.
One runnable can be called in the context of another runnable. This means multiple runnables can be running within the same process context at the same point in time. If this is the case, all running runnables must be suspended and resumed.

Action	ORTI attribute	System state
write	-	running process
read	-	running process

Table 4.5: Signals can be read or written. To create valid BTF signal events, it is necessary to know which process is currently running on the core, i.e., which process executed the read or write.

system. A list of runnable entity names is thus required to check whether a function is a runnable or not.

Suspend events must be created, if the process context in which a runnable is running is preempted and a resume event is required if the corresponding process resumes. This means that whenever a process is deallocated, a potentially active runnable must be suspended. Once the process is reallocated the runnable also resumes.

Additionally, runnables can be nested, i.e. one runnable can be executed by another runnable. If this happens it is important to suspend and resume all running runnables, if the corresponding process is preempted and resumed.

Signal events are detectable via data events. To decide which data event corresponds to a signal event a list of signal names must be available. With this list it can be decided if a certain data event results in a signal event or not. The source entity for signal read events is the currently running process as shown in Table 4.5. If no process is running an entity of type simulation can be used to set the value of the signal.

Event actions are easily detectable via the *servicetrace* attribute. Via this attribute it is possible to create set, wait, and clear event actions. However, in order to create valid event actions, it is also necessary to know the event entity that relates to the respective action. ORTI does not specify event related attributes. Because ORTI does not specify OS event related attributes, it is not possible to create valid actions for this entity type.

Action	ORTI attribute	System state
ready	resource object	-
lock	resource locker	-
unlock	resource locker	-
full	resource locked, servicetrace	-
overfull	resource locked, servicetrace	-

Table 4.6: OSEK/VDX resources can only be locked or unlocked which means they do not support all semaphore actions. Lock and unlock actions can be detected via the ORTI locker attribute.
Full and overfull events are created if an already locked resource is requested again. This is detectable via the *servicetrace* attribute. The resource for which the *resource locked* attributed was read the last time is the resource for which the error has occurred.

Action	ORTI attribute	System state
requestsemaphore	resource locker	-
assigned	resource locker	-
waiting	resource locked	-
released	resource locker	previous locker

Table 4.7: Via the resource locker attribute it is possible to detect if a resource has successfully requested a semaphore.
The *resource locker* attribute changes to the no task ID if the resource is no longer locked. For this case it is necessary to know the task that has previously locked the resource in order to create the correct release event.
Waiting actions can be created by detecting data read events to the *resource locked* attribute.

Resource entities must be initialized via the ready action before they can be used in a BTF trace. This can be done at the beginning of a trace with the timestamp zero. The ORTI file contains a list of all resource objects that are part of the application.

Since resources can only be locked or unlocked, they cannot change into the semaphore used state. Consequently, only the state transition actions shown in Table 4.6 can occur for resource events. Additionally, only a subset of the process semaphore actions are required to represent the behavior of resources.

Via the ORTI *resource locker* attribute it is possible to detect by which task entity a resource is locked. This means a lock event can be generated whenever the ID of a certain task is written to this attribute. On the other hand, an unlock event is created when *resource locker* indicates that the respective entity is currently not locked by any task. Moreover, it is necessary to assign a process to the locked resource once it is locked by the task and to release it when the resource is released as shown in Table 4.7.

Full and overfull actions are created when a locked resource is polled by a process. The semaphore waiting action is used to indicated the identity of the polling process. As shown above, it is possible to detect whether a process is polling a resource via the *servicetrace* and *lasterror* ORTI attributes. *Lasterror* is set to E_OS_ACCESS in case a resource is already locked. The resource for which the polling occurs is detectable via the *resource locked* attribute. Whenever a certain resource is requested the OS will read this attribute to decide whether a request is allowed or not.

4.3 OS Specific Mappings

It is not feasible to create all BTF events relying solely on the ORTI file. For example, it is necessary to have a list of runnable and signal names in order to create valid events for those entity types. But even for entities that are supported by the ORTI interface not all events can be generated. It is possible to detect if the activation limit of a task is exceeded however, it is not possible to determine for which task entity this happens.

Nevertheless, even though not all events are detectable via ORTI alone, an OSEK OS stores the information of interest internally. During a task activation the OS must decide whether the MTA limit is reached or not. To do so it is necessary to compare the current amount of pending activations to the value of maximal allowed activations. Consequently, the OS has to read certain information from memory which results in data trace events.

Based on this argument all other events can be reconstructed, if the corresponding OS specific operations are known. On the downside, it is no longer possible to rely on a standardized interface like ORTI. This means the algorithm that does the transformation must be customized depending

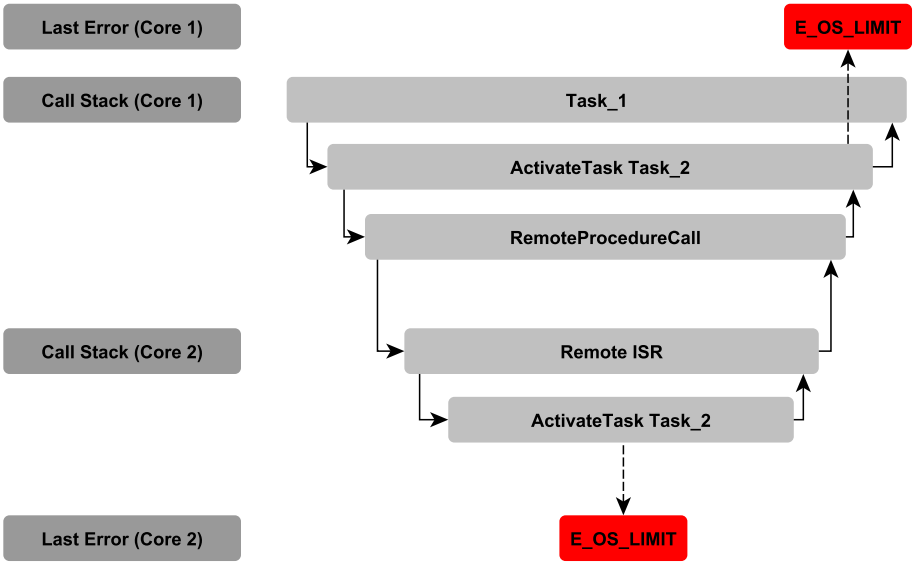

Figure 4.4: A *mtalimitexceeded* event must be created if the E_OS_LIMIT error is set via the *lasterror* ORTI attribute. However, this is not correct for Erika Enterprise multi-core applications. For a failing inter-core inter-process activation the error code is written two times, once on the source and once on the target core. Therefore, special care must be taken, so that the BTF event is created only once.

on the OS. In this section the adaptations required to create a BTF trace for the OSEK/VDX compliant Erika Enterprise (EE) Operating System [53] are shown. In section 5.1 the reasons for choosing EE are discussed.

Task *mtalimitexceeded* events cannot be created based on ORTI alone because the task entity for which the event occurs is not detectable. One way to get this information is to remember which task's *currentactivations* attribute was read the last time. The OS has to decide whether a task instance can be created once an activation is triggered. To do so it compares the maximum allowed activations with the current number of activations of a task. In other words, the OS reads the *currentactivations* attribute for the task that should be activated. If the MTA limit is exceeded an error code is written.

As it turns out this approach is not sufficient for multi-core systems. Activation of a task entity by a task on another core via ActivateTask is implemented by a Remote Procedure Call (RPC) as shown in Figure 4.4. The RPC triggers an ISR on the other core which performs the required action. In case of an inter-process activation the ActivateTask routine is executed again, but this time on the core the target task is allocated to. If the MTA limit of the task is exceeded an E_OS_LIMIT error event is written and a mtalimitexceed event is created.

```
if ( EE_th_rnact[TaskID] == 0U ) {
  ev = E_OS_LIMIT;
} else {
  /* Do activation. Code removed for clarity. */
  ev = E_OK;
}
if (ev != E_OK ) {
  EE_ORTI_set_lasterror(ev);
  EE_oo_notify_error_ActivateTask(TaskID, ev);
}
```

Listing 4.2: Erika Enterprise keeps track of the remaining activations that are allowed for a task entity. If the value is zero and another activation occurs an E_OS_LIMIT error is set.

Action	Variable	Additional Information
mtalimitexceeded	lasterror	previous data read event
trigger (alarm)	alarm action type	ORTI

Table 4.8: Via ORTI it is not possible to detect for which task an E_OS_LIMIT event has been created. However, the data read event before this error can be used to get this information.
Additionally, alarm trigger events cannot be created via the *alarmtime* attribute in Erika Enterprise, because it is not implemented in an OSEK/VDX compliant way. Instead, read events to the ActionType attribute of an alarm can be used to detect when a stimulus event must be created.

However, the remote procedure call is notified by the remote ISR once the service routine has finished. The corresponding error code is also returned back to the initial core and written to the *lasterror* attribute. The resulting problem is that the transformation algorithm would create another *mtalimitexceeded* event based on the last read from the pending activations variable on the initial core which is not correct.

A way to work around this problem can be derived by looking at a part of the source code of the ActivateTask implementation shown in Listing 4.2. It shows that EE keeps track of the remaining activations of each task in an array called EE_th_rnact. If the field for a specific task becomes zero, an E_OS_LIMIT error is written. This means if a task should be activated on one core and this activation fails due to too many pending activations this will become clear by a data read event to EE_th_rnact directly followed by a write event to the *lasterror* attribute. For a remote activation there are multiple other data events between the error and the previous read to EE_th_rnact. Therefore, no incorrect *mtalimitexceeded* event is created.

Stimulus events must be created for inter-process and alarm activations as shown in Table 4.1. An alarm activation stimulus is created if the ORTI *alarmtime* attribute becomes zero. However, EE OS does not update this

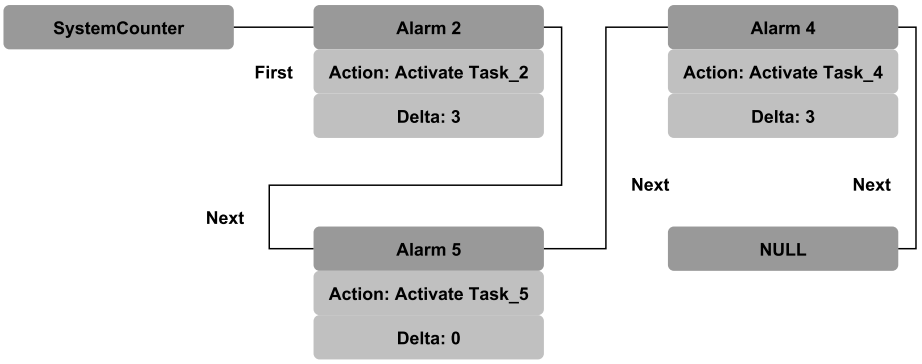

Figure 4.5: EE implements alarms via a delta queue. There is one queue, containing of the corresponding alarms, for each counter. Each alarm has a delta value that indicates after how many ticks in relation to the previous alarm it must be executed. Only the delta of the first alarm in the queue must be decremented for each counter tick. If an alarm expires it is removed from the queue, and inserted again in case it is cyclic.
In this example Alarm 2 expires after three ticks. Since Alarm 5 has a delta of zero it expires at the same counter cycle. Alarm 4 expires after six cycles, i.e. the sum of its own and all previous deltas.

attribute in compliance with the OSEK/VDX specification [52]. Hence, another technique is required to detect alarm events.

EE keeps track of all active alarms in a delta queue as shown in Figure 4.5. There is one queue for each counter. Whenever a counter is incremented the delta of the first element in the queue is decremented. If the delta of the first alarm in the queue becomes zero this alarm and all following alarms with a delta of zero expire and the corresponding actions are executed.

For an expiring alarm the OS is required to execute the corresponding action. As shown in Table 4.8 each alarm has an ActionType attribute. Via this attribute the OS determines the correct action for an alarm. In other words, if an alarm expires this attribute must be read and a data read event is generated. Consequently, a BTF stimulus event is created whenever the action type attribute of an alarm is read. The exact action executed by an alarm, e.g. which task is activated for a process activation is read from the ORTI file.

Event actions must include the information about the affected event. For example, if a task sets an event it is necessary to know the target task and event for this action. ORTI allows it to detect when an event related service routine is executed however, no information about the event itself is made available.

Erika Enterprise uses two arrays to keep track of the event related state

```
if ( EE_th_status[TaskID] == SUSPENDED ) {
  ev = E_OS_STATE;
} else {
  /* Set the event mask only if the task is not suspended */
  EE_th_event_active[TaskID] |= Mask;

  /* Check if the TASK was waiting for an event we just set */
  if ((EE_th_event_waitmask[TaskID] & Mask) != 0U)
  {
    /* Activate task here */
  }
}
```

Listing 4.3: Erika Enterprise uses the EE_th_event_active array to keep track of the events set for each task. If a new event is set the mask is updated by connecting the previous events and the new event via bitwise or. It is not possible to set an event for a suspended task.

Action	Variable	Additional Information
wait_event	EE_th_event_waitmask	previous wait mask
clear_event	EE_th_event_active	previous active mask
set_event	EE_th_event_active	previous active mask
all actions	-	event bit from eecfg.h

Table 4.9: Erika Enterprise uses two arrays to keep track of the event states for each task entity. Via write events to these arrays and the previous event state for a task instance correct BTF events can be generated.

of a task: In EE_th_event_active the events currently set for a specific task instance are stored and EE_th_event_waitmask includes the information about which events a task entity is waiting for. Each field in the array corresponds to one task and each bit of a field is related to a certain event. Whenever a task is terminated both event masks are cleared.

Using these arrays it is possible to create correct events as shown in Table 4.9. Whenever an OS event related service routine is executed the corresponding event mask is updated. For example, if an event is set for a specific task, the event mask is updated based on the new event. This means the events which are currently set for a task and the new event are connected via the bitwise *or* operation as shown in Listing 4.3.

Hence, a data write event to one of those arrays is created whenever a event service routine is executed. However, only the new state of the bitmask becomes available. To determine the event ID it is necessary to remember the previous state of the mask. By executing a bitwise *exclusive-or* operation on previous and current mask, the bit of the current event is computed.

Unfortunately, this information is still not enough to create a valid BTF

```
/* if this is a global resource, lock the others CPUs */
if (isGlobal) {
  EE_hal_spin_in((EE_TYPESPIN)ResID);
}
```

Listing 4.4: In case a global resource (a resource used on multiple cores) is requested, Erika Enterprise uses a spinlock mechanism to lock the CPU until the resource becomes available.

event. For each bit it is necessary to know the corresponding entity name. ERIKA Enterprise Operating System defines the bitmask for each OS event in the *eecfg.h* file which is created during the code generation process. By parsing the event defines the mapping between bit and event name is retrieved.

Resource events or in BTF terms semaphore events, can be created based on the information provided by ORTI as shown in Table 4.6. However, certain semaphore events like waiting can only occur in multi-core systems. In a single-core system it is not possible that one task polls a resource that is already occupied because of the priority ceiling protocol.

Erika Enterprise implements inter-core resource requests via spinlocks. If a task requests a resource that is locked by a task on another core, the service routine does not return an error code but starts spinning as shown in Listing 4.4. As a consequence, the mapping for full, overfull, and waiting actions introduced in the previous section does not work.

To solve around this problem, it is necessary to understand how spinlocks are implemented in Erika Enterprise. The state of each spinlock is stored in the EE_hal_spin_status array where each field corresponds to a separate spinlock. A value of one indicates that the spinlock is locked otherwise the value is zero. The EE_hal_spin_in method is implemented via the atomic compare-and-swap operation. This method is used to write a one into a certain spinlock field, but only if the spinlock is currently free. Compare-and-swap returns a value that indicates whether the operation was successful or not. In the latter case the operation is executed again until it succeeds.

Compare-and-swap operations result in a data access to the variable for which the operation is executed. Therefore, it is possible to detect when a spinlock is polled based on data access events to EE_hal_spin_in. This information can then be used to create correct semaphore events as shown in Table 4.10.

Whenever the *resource locker* attribute is read within the context of the GetResource service routine, the corresponding resource entity must be stored in the system state. If the resource is free, a write event to the *resource locker* attribute follows and the corresponding BTF events can be created as described above.

If there is no write event to the *resource locker* attribute the resource

Action	Variable	Additional Information
waiting	EE_hal_spin_status	running task, requested resource
full	EE_hal_spin_status	requested resource
overfull	EE_hal_spin_status	requested resource

Table 4.10: Not all BTF semaphore actions can be created based on ORTI alone for an Erika Enterprise multi-core application. This is because inter-core resource requests are implemented via spinlocks. Spinlock operations can be detected via the EE_hal_spin_status array.

is currently locked and the OS starts spinning which is detectable by continuous data access events to the field of EE_hal_spin_status relating to the requested semaphore. Consequently, the running process is assigned to the semaphore via the waiting action and an overfull action must be created. The process is now in polling mode. Once there are no further accesses to EE_hal_spin_status the request was successful, the task state changes to running and the resource state to full.

5 Validation

In this chapter the software to system mappings are validated as depicted in Figure 5.1. A timing model of an application is created and a BTF trace is generated from this model via discrete event simulation. The simulated trace represents the expected result for the trace recorded from hardware.

Next, C code is generated from the model. The code is compiled, executed on hardware, and the runtime behavior is recorded via hardware tracing. The resulting software level trace is transformed to system level according to the respective mappings. The BTF trace recorded from hardware is then compared to the simulated trace. Since both traces result from the same timing model they are expected to represent the same system behavior.

Nevertheless, two kinds of deviations are expected. Firstly, timestamps of otherwise identical events might differ. This is unavoidable because simulation is an abstraction of reality and is not capable of taking all subtle effects influencing the timing on real hardware into consideration. Secondly, events may indicate a different software behavior. For example, a task starts a runnable in one trace but not in the other. In this case, the deviation must be examined because it might point to a mapping error.

5.1 Evaluation Test Bench

To make the results of the validation process comprehensible and reproducible for others it is important to document the hardware and software setup, the configuration of all tools in use, as well as the ways in which the traces are compared.

5.1.1 Software Setup

Simulation is used to validate the BTF traces obtained from hardware via tracing and transformation. It allows analyzing of embedded real-time systems by generating an event trace. A simulation is easy to configure and executable without hardware. This is an advantage in the early design stages of an application when the final target platform is not yet defined.

Advanced simulation tools allow it to take platform dependent timing behavior into account. It is possible to select the OS and processor platform

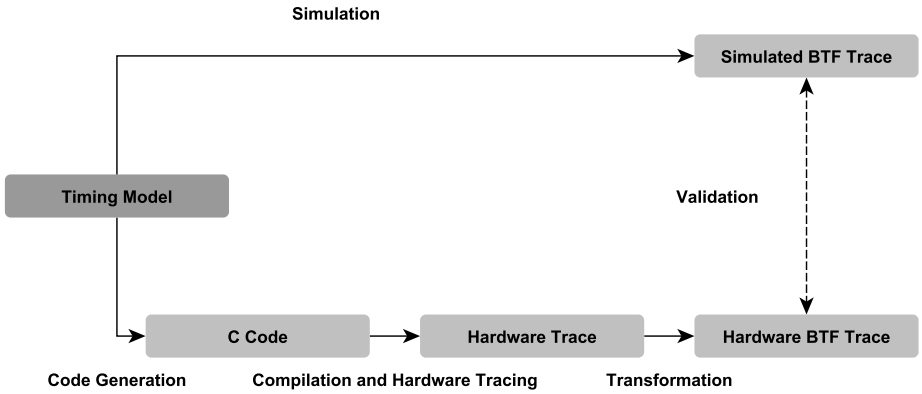

Figure 5.1: The general idea for the validation of the software event to BTF event mapping. A model that represents a certain system is created. Based on the model, a simulation, and a hardware trace are generated. By comparing those traces errors in the transformation process can be detected.

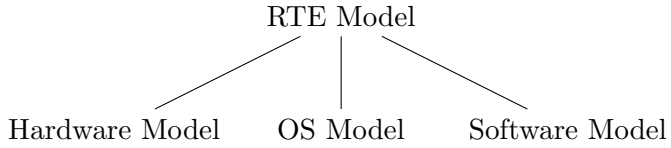

Figure 5.2: A RTE model consists of a hardware, an OS and a software part.

in use. Therefore, more accurate simulation results can be achieved. For example, memory access times [19] and timing overheads caused by OS service routines [20] can be taken into consideration.

Timing-Architects Embedded Systems GmbH provides the simulation software used for validation [56]. The TA Simulator is based on a discrete-event system simulation approach [8, 7]. It has already been used successfully in research projects to evaluate scheduling algorithms in multi-core systems [9], to examine synchronization protocols [3], and to validate optimization algorithms for embedded applications [48]. In this thesis version 15.02.1 of the TA Simulator is in use.

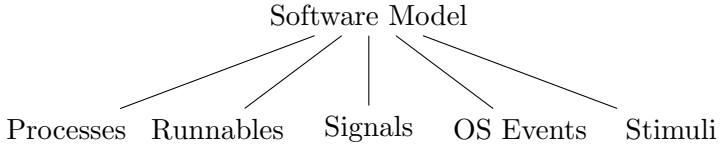

Figure 5.3: The software model represents the entities of an application that are executed by the OS and the hardware.

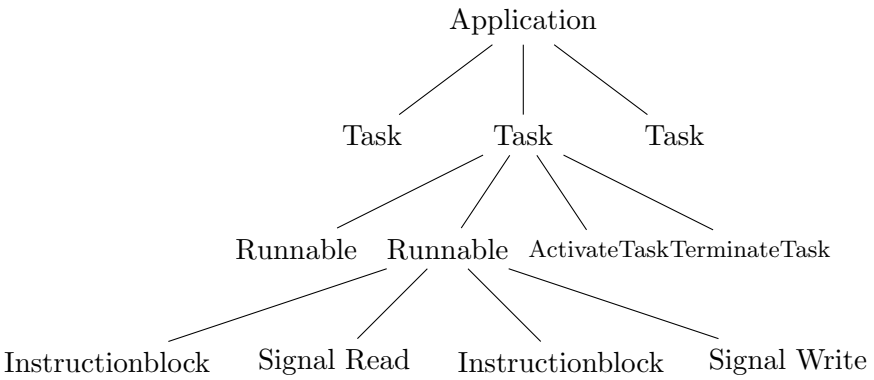

Figure 5.4: The software model allows it to represent the runtime behavior of an application. All relevant software entities are part of the system model and stand in relation to each other. For example, a task can call a runnable which itself writes a signal value and runs for a certain amount of processor instructions which is represented by an instruction block.

Timing Models describe the architecture and timing of an embedded system. Model based development is a software development paradigm where the design of an application is created in form of a timing model. This can be done before the actual application software is implemented. Based on the timing model requirements and constraints can be specified and validated via simulation.

Timing models can provide different levels of granularity depending on the use case. TA uses the Real Time Evaluation (RTE) model format which consists of three parts as shown in Figure 5.2.

The hardware model includes the processor with all cores, quartzes, and memory modules. Quartzes are used as a clock source for cores and memory modules. Memory modules can be connected with each other and to the processor cores to represent the architecture of the real chip. Vendor specific hardware models are available for certain processor families for example, the Infineon Aurix and the Freescale Matterhorn.

The OS model defines the scheduling policy for an application as well as OS related timing overheads. Implementation of service routines varies depending on the OS vendor. Consequently, the timing overhead resulting from this routines is also different which makes it necessary to take their runtime into account in order to get more accurate simulation results. Vendor specific OS models are available for certain OSs for example, Elektrobit Autocore OS [11].

The software model represents how hardware and OS are used by an application. Hardware and OS model remain the same for all tests and only the software part is changed depending on the different test scenarios. Figure 5.3 depicts the system entities that are part of the software model.

Processes and runnables are ordered in a hierarchical structure as shown in Figure 5.4. Processes can call system routines and runnables, while runnables can access signals, request, and release semaphores and execute instruction blocks. The latter represents a certain number of clock cycles required to execute a code section. It is required to mimic the runtime behavior of a real application. The number of instructions taken by an instruction block can be configured to be static or to vary depending on a specific distribution, e.g., Weibull distribution.

Stimuli are used to activate process entities. Similar to alarms they can activate processes periodically or only once. Additionally, it is possible to trigger stimuli to represent more complex activation patterns for example, arrival curves. Since runtime and activation patterns based on random distributions are tough to represent in C code instruction blocks and stimuli with constant values are used for the test models.

Code Generation is used to create C code based on the timing model of an application. A template based model export was specified and implemented in the context of this thesis. The solution is already in production and allows it to create C code and the corresponding OIL files automatically.

The idea is to iterate over all software entities and create the appropriate code dependent on the entity type. Transformation of most model entities is straightforward. Runnable calls map to function calls in C. A signal read access occurs if one signal is assigned to another variable. Accordingly, a write access is represented by assigning a value to a signal. Task, event, and semaphore actions are created based on the respective OSEK/VDX service routines discussed in section 2.1.

An instruction block is the only software model entity that cannot be mapped to C code straightforwardly. As discussed before, an instruction block represents a certain amount of clock cycles required to execute a code section. Normally, this value is set based on measurement results or empirical values from other applications. For code generation it is necessary to create code whose execution takes the same amount of clock cycles as specified in the model.

The obvious way to do so is a for loop however, the exact code is dependent on compiler and hardware. Listing 5.1 shows the code necessary to get the desired behavior for the hardware used in this thesis. It works because the Infineon Aurix processor family features zero overhead loops. This means a for loop with one nop instruction takes exactly one clock cycle because loop condition check, loop incrementation, and loop content are executed in parallel.

It is important to add multiple nop instructions per loop cycle. The Aurix trace device implements a compressed program flow trace. This means trace messages are only created for certain function events. Since the loop assembly instructions is one of the commands that cause the creation of a trace message, a loop with a single nop would cause the trace buffer to over-

```
void executeInstructionsConst(int clockCycles) {
    int i;
    clockCycles /= 2;
    for (i = 0; i < clockCycles; i++) {
        __asm("nop");
        __asm("nop");
    }
}
```

Listing 5.1: The function takes the specified amount of clock cycles to be executed. This code is dependent on hardware and compiler in use and must therefore be adapted to other platforms.

```
EE_OPT = "EE_EXECUTE_FROM_RAM";
EE_OPT = "EE_ICACHE_ENABLED";
EE_OPT = "EE_DCACHE_ENABLED";
REMOTENOTIFICATION = USE_RPC;
CFLAGS = "-O2";
STATUS = EXTENDED;
ORTI_SECTIONS = ALL;
KERNEL_TYPE = ECC2;
COMPILER_TYPE = GNU;
```

Listing 5.2: Subset of the EE OIL OS attributes used for validation. Attributes that are not mentioned are set to the default value described in the EE RT-Druid reference manual.

flow if the value of clockCycles exceeds a certain value. By adding additional nop commands less trace messages are created per time unit and the function events can be transmitted off-chip without overflowing.

ERIKA Enterprise Operating System is an OSEK/VDX compliant real-time operating system. It is free of charge and open-source which makes it an excellent choice for this thesis. Without access to the OS internal code creation of many BTF events would not have been feasible. The EE software packet contains the complete OS source code as well as RT-Druid, the code generation tool to create OS specific source code from the OIL file. In this thesis the ERIKA Enterprise Operating System and RT-Druid 2.4.0 release is used.

Listing 5.2 shows the OIL attributes set for validation. All attributes that are not mentioned take their default value as documented by the RT-Druid reference manual [54]. The test applications are executed from RAM, instruction and data caching is enabled, and the O2 optimization level is configured. Inter-core communication is implemented via remote procedure calls. All ORTI attributes and extended error codes are logged by the OS. The configuration is created in a way that allows maximum traceability combined with decent performance. Consequently, a similar configuration

```
MCU_DATA = TRICORE {
  MODEL = TC27x;
};
BOARD_DATA = TRIBOARD_TC2X5;
```

Listing 5.3: EE ECU config to support the Infineon TC27x microcontroller family and the TC2X5 evaluation board. Source code changes are necessary to support the hardware used in this thesis.

could also be used in a production system.

The **Hightec Compiler** [15] is used to compile the C code generated by code generation and RT-Druid. It is based on GCC and EE generates appropriate makefiles automatically if GNU is set as compiler. For the tests Hightec Compiler v4.6.5.0 is used.

TRACE32 [17] is used as the hardware trace host software. Configuration of this part of the test setup is the most complex. Different vendor specific properties, like the number of processor observation blocks, must be taken into consideration to create a setup that produces optimal results. The used hardware and the corresponding configuration is discussed in the next section.

5.1.2 Hardware Setup

An **Infineon TriBoard TC298** evaluation board assembled with the Infineon **SAK-TC298TF-128** microcontroller is used for evaluation. This board provides an Infineon Multicore Debug Solution together with an Aurora Gigabit Interface. According to Table 3.1 and Table 3.3 this setup allows for optimal trace performance.

EE provides support for the Infineon TC27x processor family which can be activated in the OIL file as shown in Listing 5.3. TC27x and TC29x are quite similar. Nevertheless, it is important to adapt the configuration to the TC298TF processor. This is done by changing the includes in ./cpu/tricore/inc/ee_tc_cpu.h from <tc27xa/Ifx_reg.h> to <tc29xa/Ifx_reg.h>. The layout of the evaluation board is the same.

Based on MCU_DATA EE configures the controller in the correct way during system initialization. The OIL CPU_CLOCK attribute can be used to set the desired CPU frequency. The configuration done by EE is sufficient to put the controller into a usable state. However, there are problems regarding the frequency of the Multi-Core Debug System (f_{mcds}). The TC298TF can run at a frequency up to 300 MHz. EE does not configure the MCDS clock divisor at all and consequently f_{mcds} is equal to the system frequency. However, the TC29xA user manual states that the maximum allowed value for f_{mcds} is 160 MHz [1].

Incorrect clock configuration may result in data and function events be-

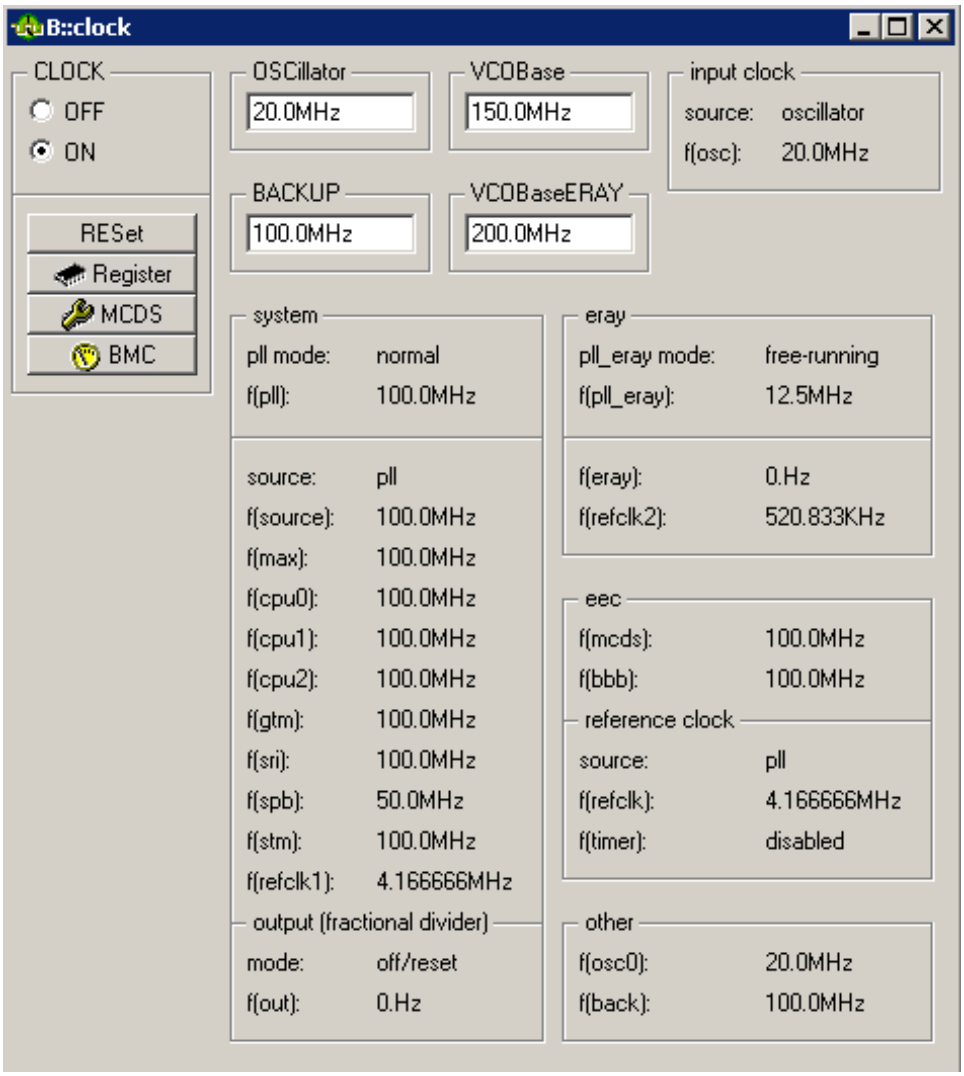

Figure 5.5: Correct clock settings are essential to record valid hardware traces for the Infineon TC298TF microcontroller. The multi-core debug system frequency must be lower or equal to 160 MHz and the ratio between CPU and MCDS frequency must be 1 : 1.

```
SYStem.CPU TC298TF
trace.method.analyzer
trace.mode.stream

mcds.source.cpumux0.program on
mcds.source.cpumux0.readaddr on
mcds.source.cpumux0.writeaddr on
mcds.source.cpumux0.writedata on

break.set symbol.begin(foo)--symbol.end(bar) /r /w /tracedata

Go
wait 1.s
break

printer.filetype csv
printer.file data.csv
winprint.trace.findall , cycle readwrite /list %timefixed \
                        ti.zero varsymbol cycle data

trace.export.csvfunc func.csv
```

Listing 5.4: Script to configure TRACE32 and the on-chip trace device. The setup allows for continues function and data trace.

ing dropped randomly. According to the manual it is necessary to set the f_{system} to f_{mcds} ratio to 2 : 1 to avoid this problem. Despite using the proclaimed configuration event dropping still occurred during the validation. After consultation with the hardware experts from Lauterbach GmbH it turned out that a ratio of 1 : 1 between system and MCDS clock is the only way to guarantee the reception of all trace events. Thus, the EE clock configuration must not be changed, but the system frequency must be smaller or equal to 160 MHz. Figure 5.5 shows a configuration with a system frequency of 100 MHz as used in this thesis.

The **PowerTrace II** by Lauterbach is used for trace recording. EE creates so called Lauterbach PRACTICE Scripts [16] also called cmm scripts during the compilation process. These scripts can be used to operate the TRACE32 software automatically. The generated scripts by EE are inadequate for the requirements in this thesis. Thus, it is necessary to improve the scripts in a way that allows continues data and function trace as shown in Listing 5.4.

Firstly, it is necessary to select the correct CPU (line 1). This is important because otherwise the TRACE32 trace decoder is not able to interpret the hardware trace events in the correct way. The trace method analyzer is required for real-time tracing and trace mode stream means that the trace data is sent to the host computer in a continuous way (lines 2 and 3).

Next, the processor and bus observation blocks are configured to detect

all function and data events (lines 5-9). This is done via the multi-core debug system. Setting the program attribute to on activates the function trace. The other three attributes are necessary to record all data events.

A complete data trace may still overexert the bandwidth of the setup. Via break.set filters as described in chapter 3 can be created (line 10). The trace device is configured to record data read and write events for all variables in the memory range defined by symbol.begin(foo)−−symbol.end(bar). Here foo is a variable that has a lower address than the variable bar. Using the configuration described in this section, the compiler allocates the array EE_as_rpc_services_table at the beginning of the OS memory section and EE_th_status at the end. So those two variables provide a convenient boundary to detect all OS data events of interest.

Trace recording is started via the Go command (lines 12-14). The wait command waits for an eligible amount of time and recording is stopped by the break command.

Now the data and function traces can be exported (lines 16-21). For the data export it is first necessary to configure the desired output file type (csv) and output filename (data.cvs). Via the winprint command the data export process is started and trace.export.csvfunc exports the function trace.

TRACE32 creates multiple graphical user interfaces one for each core of the target platform. Accordingly, the export commands must be executed for each core or in other words for each GUI. The resulting files data.csv and func.csv contain one event per line. The following listing shows a data event.

```
−0083448136,0.0004372600,"EE_ORTI_servicetrace","wr−data",43
```

A Lauterbach data event consists of five comma separated fields. In Equation 4.1 the elements of a data event are defined. The second field is the timestamp t_i in seconds, the third field is the name of the accessed variable π_i, the fourth field specifies in which way a_i the variable is accessed (a data write in this case), and the fifth field contains the value of the data access event v_i. Since one trace data trace file is exported per core, the core name c_i is the same for all events from one file. Accordingly, the next listing shows a Lauterbach function event consisting of three fields.

```
+437050; EE_as_StartCore; fentry
```

In Equation 4.3 the elements of a function event are defined. Analogous to data events, the core name c_j is the same for all events within a file. The first field maps to the timestamp t_j, the second field is the name of the function π_j that is affected by the event, and the third field indicates the way θ_j in which the function is affected.

5.1.3 Validation Techniques

Traces can differ in two ways. A temporal difference exits for two traces B^1 and B^2 with the same length n if there is at least one event pair with

the index $i \in (1, 2, \ldots, n)$ for which $t_i^1 \neq t_i^2$. As discussed before, the TA Simulator is capable of taking hardware and OS specific behavior into account. Nevertheless, simulating a trace for which all timestamps are equal to the corresponding hardware trace is not feasible by definition [6].

This problem is bypassed in two steps. At first the general accuracy of the trace setup is validated by tracing events whose timing characteristics are precisely known in advance. Secondly, for the actual test models, a plausibility test based on certain metrics such as task activate-to-active and task response time is conducted.

The second way in which two traces can differ is called semantic difference. It exists for two traces B^1 and B^2 with the same length n if there is an event pair with the index $i \in (1, 2, \ldots, n)$ for that at least one of the following cases is true: source or target entity differ ($\Psi_i^1 \neq \Psi_i^2 \vee T_i^1 \neq T_i^2$), source or target instance differ ($\psi_i^1 \neq \psi_i^2 \vee \tau_i^1 \neq \tau_i^2$), target type differs ($\iota_i^1 \neq \iota_i^2$), event action differs ($\alpha_i^1 \neq \alpha_i^2$), or note differs ($\nu_i^1 \neq \nu_i^2$).

If two traces B^1 and B^2 have a different length $|B^1| \neq |B^2|$ they also differ semantically. Assuming the trace and simulation setup is correct a difference in length can have two reasons: either the trace times differ or one trace includes entities that do not occur in the other trace. In the former case, the disparity can be fixed by removing the events at the end of the longer trace until both traces have the same length. In the latter case, events for entities that are not contained in both traces may be removed in order to achieve semantic equality.

5.2 Test Cases

As discussed in the previous section traces can differ in a temporal and in a semantic way. To exclude the appearance of temporal discrepancies due to a wrong trace setup, the timing accuracy is tested based on code with known event-to-event durations. Next, the semantic correctness of the trace mapping is validated based on manually created test models. Finally, randomized models are generated in order to detect semantic errors that may not be detected by the manually created models due to selection bias [14].

5.2.1 Timing Precision

In Listing 5.1 code to execute a fixed number of instructions is introduced. This code is now used to evaluate the timing precision of the trace setup. According to section 3.2 the setup should allow for cycle accurate trace measurement.

The Infineon Aurix processor family provides performance counters [1]. Once started, these counters are incremented based on the CPU core frequency. A frequency of 100 MHz is used for the validation, consequently an

```
EE_UINT32 i;
EE_UINT32 ccntStart;
EE_UINT32 ccntEnd;
EE_UINT32 n = N / 4;

__asm("nop");
ccntStart = EE_tc_get_CCNT();
__asm("nop");
for (i = 0; i < n; i++) {%
  __asm("nop");
  __asm("nop");
  __asm("nop");
  __asm("nop");
}
__asm("nop");
ccntEnd = EE_tc_get_CCNT();
```

Listing 5.5: Code to validate the timing precision of the trace setup.

increment occurs every 10 ns. The counter can be started at an arbitrary point in time for example, at program start. By reading the counter value at the beginning and at the end of a critical section the clock cycles that expired between these two points can be determined.

Listing 5.5 shows the code that is used to check the timing precision. EE provides the API function EE_tc_get_CCNT to read out the performance counter register. As described above, the performance counters are read out before and after the critical section.

The critical section is guarded with two additional nop assembly instruction to avoid compiler optimization. Additionally, the generated assembly code was examined manually to verify that no unwanted instructions were added by the compiler. A for loop is used to execute a predefined number of instructions. The number of repetitions is depended on the define N which should be a multiple of four.

The code is now executed for different values of N. For each event the expected number of clock cycles c_e, the actual number of clock cycles c_a, the expected time difference t_e in nanoseconds, and the actual time difference t_a in nanoseconds between the writes to ccntStart and ccntEnd are listed in Table 5.1.

The expected number of clock cycles is calculated by $c_e = N + 2$. The value two is added because of the additional nop instructions. The expected time is calculated by $t_e = c_e * \frac{1}{f}$ where f is the processor frequency.

The actual number of clock cycles is calculated by $c_a = ccntEnd - ccntStart$. The actual time is calculated by $t_a = t_j - t_i$ where j is the index of the write event to ccntEnd and i is the index of the write event to ccntStart.

Four different values for N, 128, 1024, 4096, and 65536 are chosen and for each value 101 measurement samples are taken. The results for all samples

N	128	1024	4096	65536
c_e [1]	130	1026	4098	65538
c_a [1]	134	1030	4102	65542
t_e [us]	1.300	10.260	40.980	655.380
t_a [us]	1.340	10.300	41.020	655.420
samples	101	101	101	101

Table 5.1: Experiment to validate the accuracy of the trace setup. A code snippet that takes a known number of instructions c_e is executed. Based on the number of instructions the expected execution time t_e can be calculated. If cycle accurate measurement is supported, the actual execution time t_a should be equal to t_e. The execution times differ by 40 ns because the expected number of instructions is off by four cycles. If this deviation is taken into consideration t_e and t_a coincide.

with the same value of N are equal. It can be observed that for all values of N the execution of the critical section takes four ticks more than the expected value e_c. This is because the additional instruction executed by the second call to EE_tc_get_CCNT are not taken into consideration.

Consequently, the expected and the actual execution time differ by 40 ns. Besides this differences, the result is as expected and the conclusion that the setup is in fact able to measure hardware events on a cycle accurate basis can be drawn.

5.2.2 Systematic Tests

In this section test models are created systematically to validate the complete software to BTF event mapping discussed in chapter 4. For each test application a simulated and a hardware based BTF trace is generated as shown in Figure 5.1. The traces are then compared in three steps.

- A basic plausibility test based on the Gantt chart of the TA Tool Suite is conducted.
- The semantic equality is validated.
- Different real-time metrics are compared and discussed.

Five test models as shown in the following list are required to cover all BTF actions for which a mapping has been provided.

- task-runnable-signal
- task-event
- task-resource-release-parking
- task-resource-poll-parking

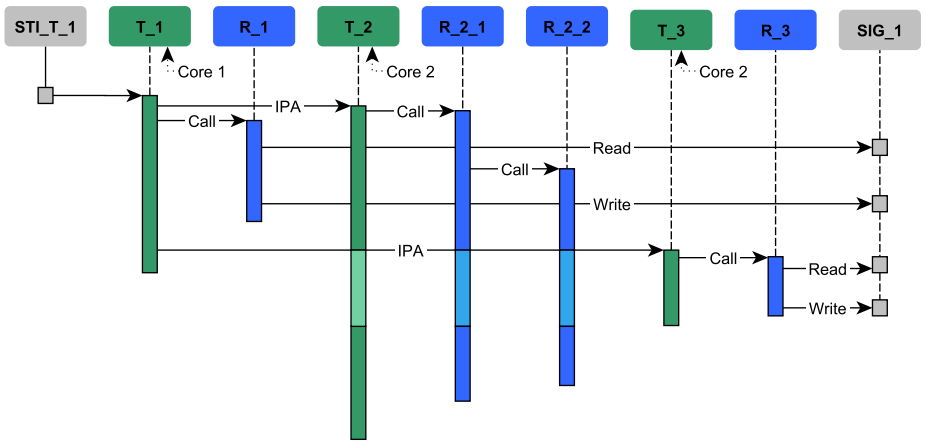

Figure 5.6: Test application to validate basic task and signal read and write events.

- task-MTA

Each model represents a periodic system where a defined sequence of events is executed every 10 ms. UML sequence diagrams [13] are used to illustrate the behavior of the test applications during one period.

Task-Runnable-Signal Test

The task-runnable-signal application is depicted in Figure 5.16. Task T_1 is activated periodically by the stimulus STI_T_1 every 10 ms. T_1 activates T_2 on another core via IPA and then executes runnable R_1. T_2 executes a runnable R_2_1 which executes another runnable R_2_2. Once execution of R_1 has finished, T_1 activates another task T_3 on the second core which has a higher priority then T_2. Consequently, T_2, R_2_1, and R_2_2 are preempted as indicated by the light green and light blue colors. T_3 calls a runnable R_3. The runnables R_1 and R_3 both read and write the signal SIG_1. Once T_3 has terminate, T_2 and the corresponding runnables resume execution. The purpose of this test application is to cover the following BTF actions:

- Stimulus: trigger by alarm and IPA
- Task: activate, start, preempt, resume, terminate
- ISR: activate, start, terminate
- Runnable: start, resume, suspend, terminate
- Signal: read, write

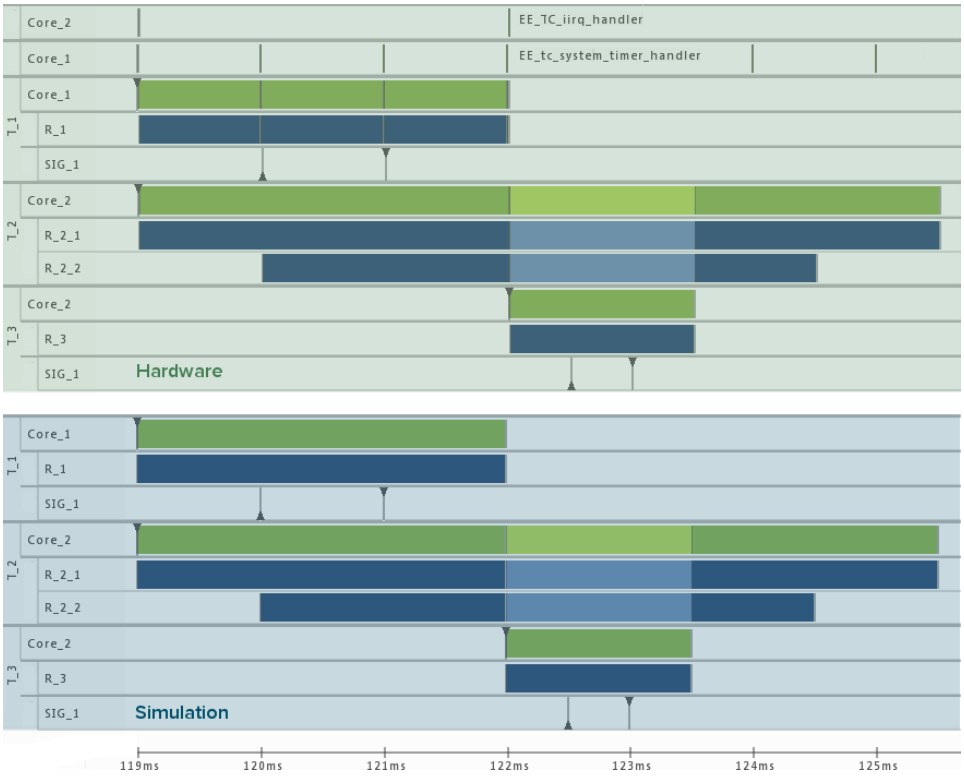

Figure 5.7: Hardware and software trace for the task-runnable-signal test model. Attention must be directed to the signal read and write accesses to SIG_1. Additionally, the nested runnables must be suspended when the respective task T_2 is preempted.

Based on the Gantt chart of the TA Tool Suite the BTF trace can be compared visually. The hardware trace is shown in the upper part and the simulated trace in the lower part of each picture. Both traces use the same time scale so that semantic and temporal comparison is feasible.

Figure 5.7 shows one period of the task-runnable-signal test application in the Gantt chart of the TA Tool Suite. The figure depicts that R_2_2 is called from the context of R_2_1. When T_2 is preempted, both runnables must be suspended too, indicated by the light blue color in contrast to the stronger blue when a runnable is running. Runnable entities are not shown in the traces for the other test models for clarity. A running task is colored in dark green, while preempted tasks are shown in light green.

A separate row in the Gantt chart is used to depict signal accesses from the context of tasks. Whenever a horizontal line is drawn the corresponding signal is read or written. The former is indicated by an arrow pointing up at the bottom of the row. The latter is indicated by an arrow pointing down at the top of the row. It can be seen that the signal accesses are recorded on hardware as expected.

The hardware trace shows two additional ISRs that are not part of the simulation trace. EE_tc_system_timer_handler is a timer interrupt which is executed every 1 ms and serves as clock source for the system counter. EE_TC_iirq_handler is used for remote procedure calls.

Two traces can not be semantically identical if entities exist in one trace that are not part of the other trace. There are two ways two solve this problem. Either the ISRs are added to the system model and therefore considered during simulation or all BTF events related to the ISRs are removed from the hardware trace.

A script that checks the semantic equality of two traces based on the criteria established in subsection 5.1.3 is used for the second validation step. However, semantic equality could not be shown for the test cases in this and the next section. The reason for this is discussed in subsection 5.2.3.

The TA Inspector is capable of calculating a variety of real-time metrics based on BTF traces. Selected metrics are shown to discuss the similarities and discrepancies between hardware and simulation trace. Common metric types are activate-to-activate (A2A), response time (RT), net execution time (NET), and CPU core load. The upper part of each metric table shows the hardware trace metrics abbreviated by *HW* and the lower part shows the simulation trace metrics abbreviated by *Sim*.

Table 5.2 shows selected real-time metrics for the task-runnable-signal application. In the first approximation all values seem identical so the basic configuration of the complete setup is likely to be correct. Nevertheless, the activate-to-activate times between hardware and simulation differ by almost 6 us which is non-negligible.

The reason for this deviation can be found by examining the activate-to-activate times of the timer ISR EE_tc_system_timer_handler. The average A2A

		A2A [ms]	RT [ms]	Load Core_1 [%]	Load Core_2 [%]
HW	T_1	10.005998	3.025510	30.124423	0.000000
	T_2	10.005990	6.516440	0.000000	49.950032
	T_3	10.005987	1.506300	0.000000	15.000495
	Sum	-	-	30.12	64.95
Sim	T_1	10.000000	3.000100	30.000000	0.000000
	T_2	10.000000	6.500200	0.000000	50.000000
	T_3	10.000000	1.500100	0.000000	15.000000
	Sum	-	-	30.00	65.00

Table 5.2: Metrics of the task-runnable-signal test application. Activation-to-activation (A2A) and response time (RT) are average values calculated over all instances of the respective entity.

time for the ISR is 600 ns greater than expected. Since T_1 is activated every 10 ms or in other words for every tenth instance of the timer ISR, the expected deviation can be calculated as $d_{A2A} = 10 \cdot 600\,ns = 6\,us$.

To detect why the A2A times of the timer ISR diverge, it is necessary to read the corresponding source code. Whenever the timer ISR is executed the time delta to the next instance is calculated based on the current number of counter ticks in the timer register. There is a time delta between the point where the last counter ticks value is read and the point where the newly calculated value is written. This is the delta that causes the delay of 600 ns. By doubling the frequency the delta reduces to 300 ns by halving the frequency it increases to 1200 ns as expected.

Task-Event Test

Figure 5.8 shows the task-event test case. T_1 is activated in the same way as in the first test case. Again, it activates T_2 on a second core via IPA. T_2 executes a runnable R_2. After execution of the runnable T_2 waits for the event EVENT_1. Since the event is not set it changes into the waiting state indicated by the orange color. After activating T_2, T_1 executes a runnable R_1 and sets the event EVENT_1. T_2 returns from the waiting state, calls R_2 again, and clears the event EVENT_1. The purpose of this test application is to cover the following BTF actions:

- Process: wait, release
- Event: wait_event, set_event, clear_event

Figure 5.9 shows the Gantt chart for the task-event test case. As before T_1 is interrupted by the timer ISR multiple times. A separate row in the Gantt chart is used to indicate the current state of the event entity. An

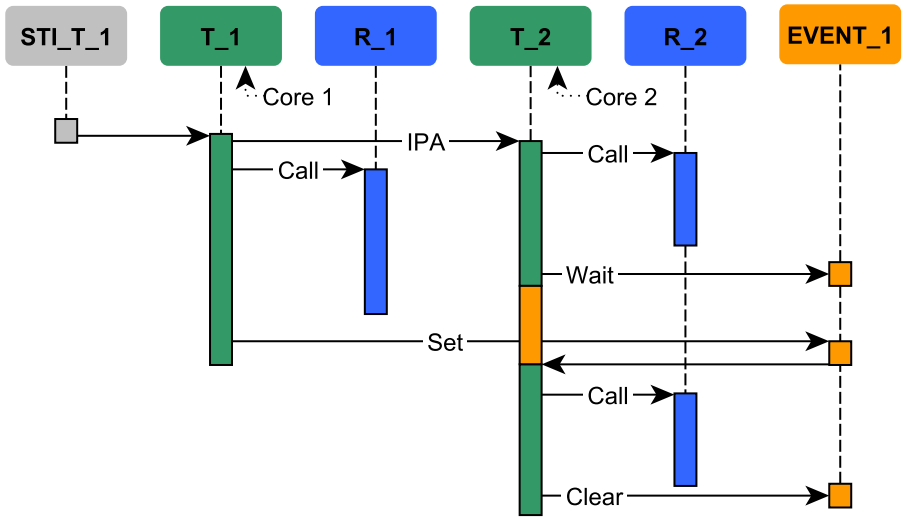

Figure 5.8: Test application to validate BTF event actions.

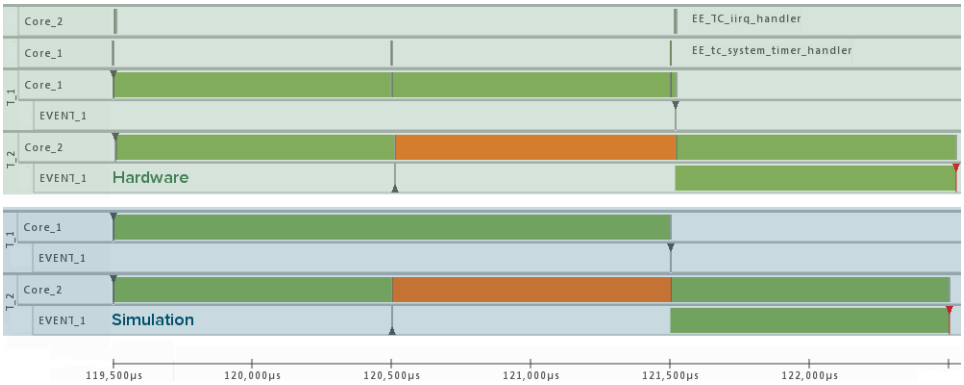

Figure 5.9: Comparison of hardware (top) and simulated (bottom) trace of the task event test application.

		A2A [ms]	RT [ms]	CPU Waiting Core_2 [%]
HW	T_1	10.006198	2.023460	0.000000
	T_2	10.006189	3.018570	10.046955
Sim	T_1	10.000000	2.000100	0.000000
	T_2	10.000000	3.000100	9.999000

Table 5.3: Metrics of the task-event test application.

upward pointing arrow indicates that a process starts waiting for an event. The waiting period is colored in orange. A downward pointing arrow indicates that a process sets an event. Finally, the event is cleared which is indicated by an downward pointing arrow in red.

Table 5.3 shows the resulting metrics for the task-event test case. The activate-to-activate times depict the same behavior like the previous test application as expected. The relative waiting time on hardware is greater than it is for the simulated trace.

A possible reason might be the longer runtime of the set_event routine on-target. The task on core Core_1 sets the event for the task on the second core. Therefore, a Remote Procedure Call is necessary to set the event. Since the RPC via EE_TC_iirq_handler is not taken into consideration in the simulation, the time in the waiting state is longer on hardware.

Response times are also significantly longer on real hardware compared to the simulated trace. The response time measures the period from task activation to termination of a task instance. The difference in response time sums up from different factors.

Firstly, the initial ready time, i.e. the period from task activation to start is longer on hardware. It takes about 2 us. Secondly, T_1 is preempted by the timer ISR two times. Category two ISRs require a context switch which costs additional task execution time. Finally, the IPA and TaskTerminate routines take longer on real hardware. By measuring the execution times of the respective system services it could be shown that the response times are equal if the measured overhead is taken into consideration. As mentioned before, these effects could be respected for the simulation by adding the execution times of the routines to the OS part of the timing model.

Task-Resource Tests

The third and fourth test case are similar except for one difference as shown in Figure 5.10 and Figure 5.11. As before, T_1 is activated by a periodic stimulus and activates T_2 on another core via IPA. T_1 executes the runnable R_1_1 which requests the semaphore SEM_1. T_2 tries to request the same semaphore which is now locked and changes into the active polling state indicated by the red color. As soon as R_1_1 finishes, T_1 activates the task T_3 which has a higher priority than T_2, on the second core. Consequently,

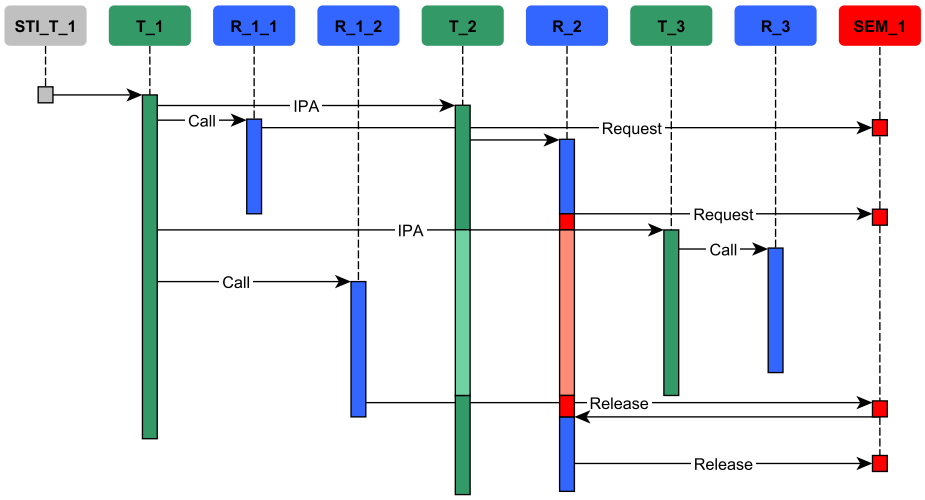

Figure 5.10: Test application to validate semaphore events, especially the poll_parking action.

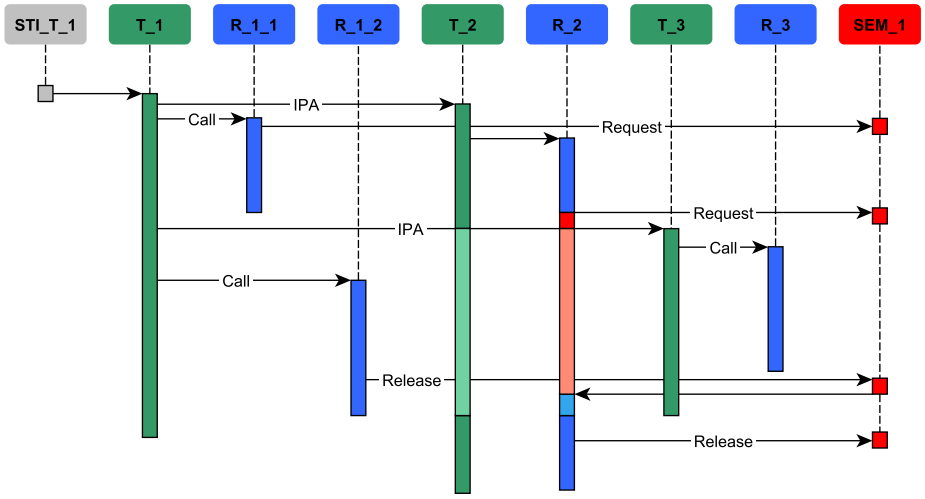

Figure 5.11: Test application to validate semaphore events, especially the release_parking action.

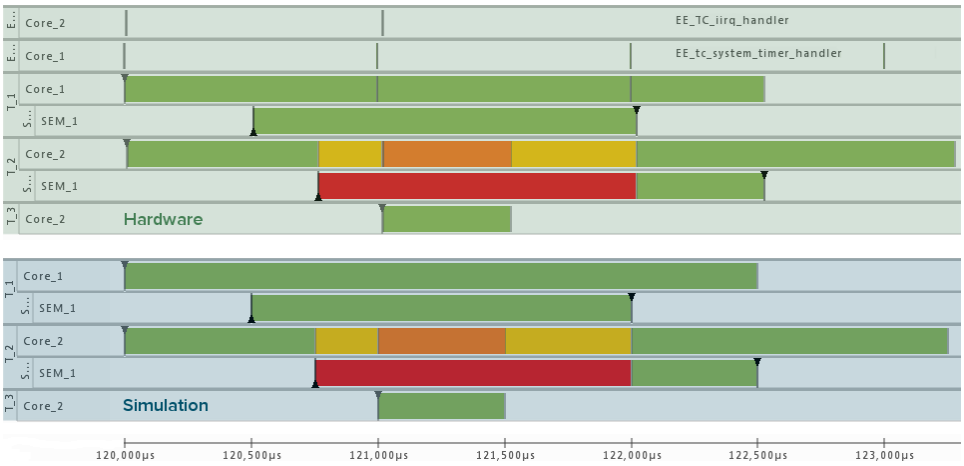

Figure 5.12: Comparison of hardware (top) and simulated (bottom) trace of the task-resource-poll-parking test application.

		RT $[ms]$	Polling Time $[ms]$	Parking Time $[ms]$
HW	T_1	2.524897	0.000000	0.000000
	T_2	3.269190	0.751730	0.508011
	T_3	0.506321	0.000000	0.000000
Sim	T_1	2.500140	0.000000	0.000000
	T_2	3.250040	0.749800	0.500100
	T_3	0.500100	0.000000	0.000000

Table 5.4: Metrics of the task-resource-poll-parking test application.

T_2 is deallocated and changed into the parking state.

At this point the two models differ. In first model *task-resource-poll-parking* T_3 has a shorter execution time than in the model *task-resource-release-parking*. Consequently, in the former model T_2 is resumed while SEM_1 is still locked and a poll_parking action takes place.

In the latter case when T_3 has a longer execution time, SEM_1 becomes free while T_2 is still preempted. This results in a release_parking action and T_2 changes into the ready state. Once T_3 has terminated T_2 continues running immediately. The purpose of these applications is it to test the following actions.

- Process: park, poll_parking, release_parking, poll, run
- Semaphore: ready, lock, unlock, full, overfull
- Process-Semaphore: requestsemaphore, assigned, waiting, released

Figure 5.12 and Figure 5.13 show the comparison of the traces for the two resource test applications. For both test cases T_1 requests SEM_1 as

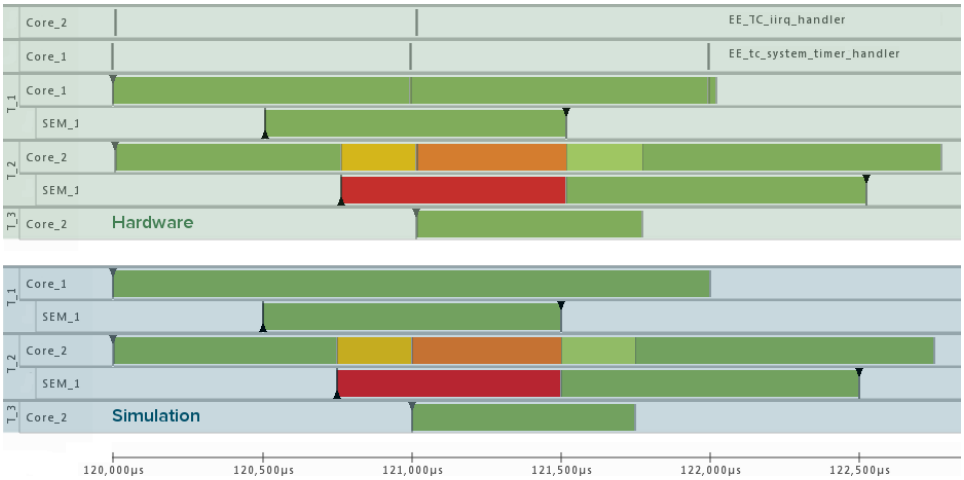

Figure 5.13: Comparison of hardware (top) and simulated (bottom) trace of the task-resource-release-parking test application.

		A2A $[ms]$	RT $[ms]$	CPU Parking Core_2 [%]
HW	T_1	10.005997	2.026420	0.000000
	T_2	10.005989	2.772670	4.984965
	T_3	10.005984	0.756450	0.000000
Sim	T_1	10.000000	2.000140	0.000000
	T_2	10.000000	2.750240	4.949010
	T_3	10.000000	0.750100	0.000000

Table 5.5: Metrics of the task-resource-release-parking test application.

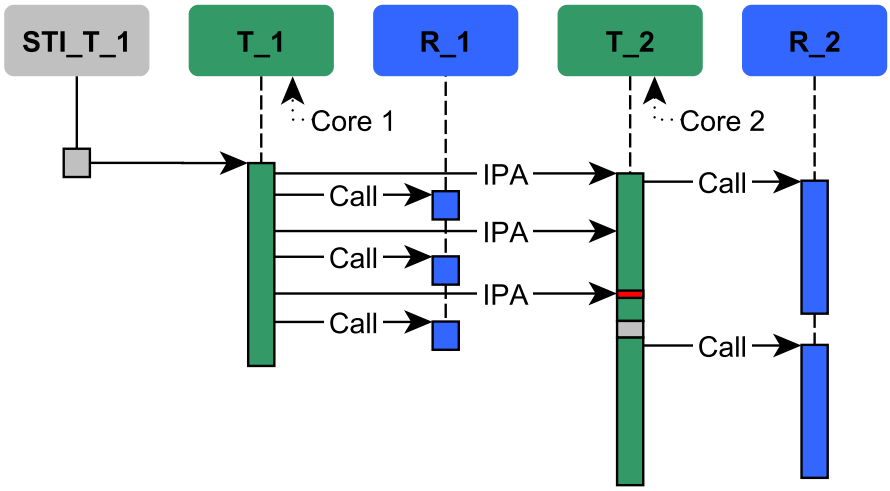

Figure 5.14: Test application to validate mtalimitexceeded events.

indicated by an upward pointing arrow. The semaphore is now locked and T_2 changes into the polling mode when requesting it. This is indicated by the yellow color. Once T_3 is activated T_2 changes into the parking mode indicated by the orange color.

In Figure 5.12 T_3 has a runtime of 500 us and resumes running before the semaphore is released. Thus, it returns into the polling state until the semaphore is released. The release event is depicted by a downward pointing arrow. In Figure 5.13 the execution time is longer and T_1 releases the semaphore earlier. Consequently, SEM_1 becomes free while T_2 is still deallocated from the core and changes into the ready state.

For both resource test applications the BTF traces recorded from hardware match the simulated traces as shown in the previous figures. The metrics in Table 5.4 and Table 5.5 show similar results compared to the previous tables and are therefore not discussed again.

Task-MTA Test

The purpose of the last specified test application is to validate the correctness of MTA and mtalimitexceeded events. Figure 5.14 shows the sequence diagram of the respective test model. In this example T_2 is allowed to have two activations. This means two instances of the task may be active in the system at the same point in time.

Like in the previous tests T_1 is activated by STI_T_1 periodically. T_1 then activates T_2 three consecutive times via inter-core IPA. The runnable R_1 is executed to consume some time between the activations. After the

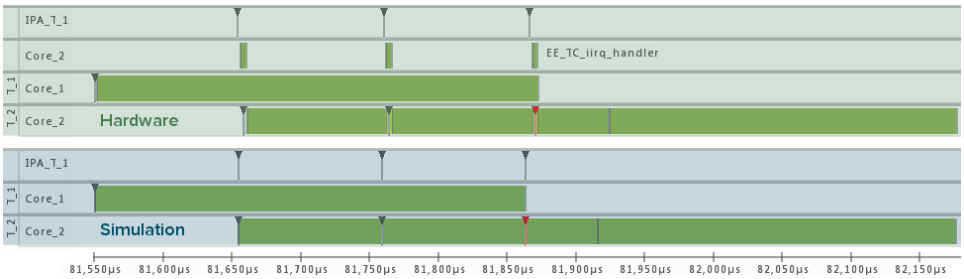

Figure 5.15: Comparison of hardware (top) and simulated (bottom) trace of the task-MTA test application.

first activation the task starts running as expected. The second activation is stored by the OS. Once T_2 terminates, it changes into the ready state and starts running again. The third activation is not allowed by the OS as indicated by the red box. An error message is created and a mtalimitexceeded event must be added to the BTF trace.

Figure 5.15 shows the comparison of the BTF traces created by simulation and from hardware for the task-MTA test model. The hardware traces illustrates the procedure for an inter-core process activation really well. At first the activation is triggered on Core_1 as shown in the row IPA_T_1. This results in the execution of the inter-core communication ISR EE_TC_iirq_handler.

The ISR then activates T_2 which changes into the ready state indicated by the gray color. During the second activation T_2 is already in the running state. Consequently, the activation is only illustrated by a downward pointing arrow. In the simulated trace the task keeps running during the activation process. In the hardware trace the task is preempted by the inter-core ISR and the activation takes place while the task is in the ready state.

During the third activation two instances of T_2 are already active in the system. Thus, no further activations are allowed and a mtalimitexceeded event is created. This is indicated by a downward pointing red arrow. At around 81925 us the first instance of T_2 terminates and the next instances becomes ready immediately. Shortly after that the next instance starts running.

5.2.3 Randomized Tests

Randomized tests are used to avoid insufficient test coverage due to selection bias in the creation of the test applications. A tool for generating random models automatically with respect to predefined constraints has been developed in previous research projects [47]. It allows the creation of an arbitrary number of test models and works with respect to user-defined distributions for example, for the number of cores, tasks, and runnables. Based on these

Entities	min	max	average	distrbution
Cores [1]	2	-	-	const
Tasks [1]	9	22	15	weibull
Runnables/Task [1]	6	13	-	uniform
Instructions/Runnable [10^3]	10	50	30	weibull
Activation [ms]	1	20	1000	weibull
Signals [1]	3	11	17	weibull
Signals/Runnable [1]	3	7	-	uniform

Table 5.6: The configuration used for creating test models randomly.

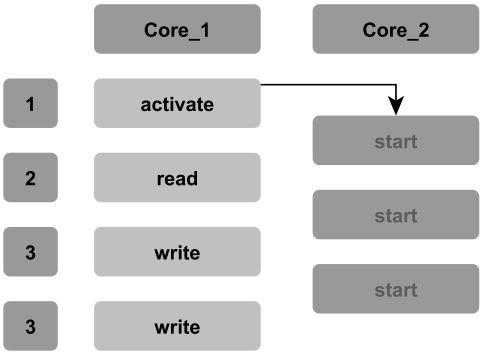

Figure 5.16: Semantic comparison of multi-core systems is not feasible if the execution time of service routines varies between hardware and simulation.

values models can be generated randomly.

Table 5.6 shows the distributions for the number of entities that should be created for each entity type. This configuration is used for each of the ten models that are tested in this section. The distributions for *cores* and *tasks* represent the number of entities of the respective type in the system. The metric *runnables per task* determines how many runnables are called from the context of each task. Each task is activated by a periodic stimulus with a period depending on the *activation* value. *Signals* specifies the number of signal entities in the system and *signals per runnable* the accesses to these signals within the context of each runnable. Event and resource entities cannot be generated by the random model generator and are therefore not covered by randomized tests.

Validating these models manually is not feasible. Therefore, only the semantic equality is tested because this can be done without user interaction. In previous work a closed loop model based development process was created to conduct the proceeding shown in Figure 5.1 automatically [32]. This process was extended to support the model generator and semantic comparison of two traces.

As mentioned before semantic equality could not be shown for any of the test applications. The reason for this is depicted in Figure 5.16. Assuming that one task activates another task on a different core and executes multiple other actions afterwards. The position in which the start event of the second task is added depends on the time that vanishes between activation and start. This means two traces may be semantically different even though they show the same behavior. Consequently, the definition of semantic equality used in this thesis is not sufficient for the comparison of multi-core systems. Nevertheless, by randomized comparison of the traces the correctness of the mappings could be validated manually. However, this fallback solution is not sufficient for validating a wide range of test cases.

6 Conclusion

Cycle Accurate Tracing

Hypothesis 1 asks whether there is a trace technique capable of recording cycle accurate traces with a duration of at least one second. There exists three general measurement techniques. Hybrid and software based trace tools rely on instrumentation. Thus, they change the runtime behavior of an application and do not allow cycle accurate trace recording.

Additionally, an on-chip memory to buffer the recorded trace events is required. Hence, the trace duration is strongly limited by the available memory. An application with 28 tasks can only be traced for 350 ms using the Gliwa T1 hybrid trace tool [26] providing events solely on task level. Runnables were not even considered.

Hardware tracing is the only trace technique that allows cycle accurate traces with a duration of at least one second. Actually, durations of over ten seconds are possible with the correct hardware configuration.

However, there are certain limitations for the hardware platform used in this thesis. Depending on the clock configuration not all data events are recorded. This can be avoided by using a CPU core clock frequency smaller or equal than 160 MHz. Therefore, Hypothesis 1 is true.

ORTI Based Software to System Mapping

Hardware trace tools create traces on software level. This level is not sufficient for the real-time analysis of embedded systems. A transformation from software to system level is therefore required. ORTI was designed to give third party tools additional information for the trace recording of applications that use an OSEK/VDX compliant OS. Hypothesis 2 asks if ORTI is sufficient to create a complete mapping from software to system level.

It has been shown that ORTI can be used to cover only a subset of the OS entity types specified in the BTF standard. Even for those entities covered by ORTI no complete mapping is feasible. For example, information about task entities is included in the ORTI file, but it is not feasible to determine the source entity for a *mtalimitexceeded* event. Consequently, Hypothesis 2 does not hold.

However, it should be noted that ORTI allows it to specify OS vendor

specific attributes. This means in case a mapping is basically possible as claimed by Hypothesis 3 then it would be possible to include the required information in the ORTI file.

Nevertheless, to the best of my knowledge this thesis is the first work to show that BTF *trigger* actions and all process actions except *mtalimitexceeded* can be created based on the ORTI sections specified by OSEK/VDX.

Software to System Mapping

No complete mapping from software to system entities is feasible by relying solely on the information in the ORTI file. Additional information is required to achieve a complete mapping. On the one hand a detailed understanding of the OS internals is required, on the other hand meta information must be provided to the transformation algorithm.

The concept of runnables and signals is not specified by OSEK/VDX. Basically, runnables are functions and signals are variables. It is possible to create runnable and signal events via function and data tracing. A list of all entities is required to distinguish regular functions from runnables and regular variables from signals.

To create BTF events for the event entity type it is necessary to understand the respective code of the OS. By parsing the statically created C header files the event IDs can be retrieved and the correct events can be created.

Semaphore events are the most complex entity types to reconstruct via hardware tracing. BTF supports all possible types of semaphore like synchronization mechanisms. Hence, a variety of different actions are specified. A possible mapping for OSEK/VDX resource entities is nevertheless provided in this thesis.

To the best of my knowledge this is the first work to show that all BTF signal, runnable, event, and semaphore actions can be recreated from an OSEK/VDX compliant OS. Therefore, Hypothesis 3 is true.

7 Future Work

Improve Trace Interface Standard

It has been shown that a complete software to system mapping is possible for an OSEK OS and should accordingly also be possible for an AUTOSAR OS. However, detailed knowledge of the OS is required to understand and implement this mapping. OSEK/VDX tries to minimize this effort via the ORTI trace interface. Unfortunately, this interface is only regulated for a subset of all OS entity types.

Some entities like spinlocks, semaphores, and inter-process communication techniques like AUTOSAR sender-receiver-communication are not covered at all. In theory OSEK/VDX allows it to add additional attributes to the ORTI file, but this option is currently not comprehensively used by the OS vendors. To solve this problem further efforts to reach a common trace interface standard for all AUTOSAR system entities should be made.

Evaluate Different Hardware Platforms

In this thesis the feasibility of recording cycle accurate hardware traces was validated for the Infineon Aurix TriCore processor family using the Infineon Multi-Core Debug System. As described in section 3.2 there exists different trace standards for other processor families.

In order to achieve a better understanding of the trace capabilities of various hardware platforms different other processor families should be tested in the future. It has been shown that cycle accurate recording of data events on the Infineon TC298TF processor is only feasible for certain clock settings. It would be interesting to know if similar constraints also exist for other platforms.

Evaluate Different Operating Systems

ERIKA Enterprise Operating System is used as a representative for an OSEK/VDX compliant OS in this thesis. It is a sufficient choice because of the available source code and the permissive license. For EE, it could be shown that a mapping from software to system entities is feasible.

However, OSEK/VDX has been taken over from AUTOSAR. Since AU-

TOSAR is a superset of OSEK/VDX the reasoning for most system entities is legitimate for both OS standards. Nevertheless, AUTOSAR introduces new synchronization patterns (of which some have been adopted by EE) and it would be interesting to know if a mapping is possible for those new techniques as well.

Additionally, a complete mapping could only be created because the source code of EE is freely available. It would be interesting to know if the same approach is feasible for a commercial OS that does not make its source code available. This is an important question to answer since the automotive industry relies predominantly on commercial OSs.

Validate Mapping With Real World Applications

Finally, the feasibility of the software to system mapping has been shown and validated for several test applications. One part of those applications was created manually to cover specific test cases, the other part was created randomly. However, all test applications have in common that they do not execute real functionality. Instead, dummy instructions are used to simulate runtime that would emerge on real hardware due the computation of algorithms and feedback loops.

It may be possible that the trace capability of the tested hardware is limited for real applications. If this is the case the mapping introduced in this thesis may not be completely applied in the real world for example because the bandwidth for recording OS data events is limited. To investigate this question industrial case studies should be conducted based on the approaches discussed in this thesis.

Trace a Multi-ECU Setup

In many environments microcontrollers operate in big networks. For example, in modern cars up to 70 ECUs are installed and connected via at least five different field bus systems [20]. In such systems correct system performance is not only dependent on the behavior of a single controller, but also on the interaction of the system as a whole. The ability to trace multiple ECUs in parallel would provided enormous benefits in the analysis and validation of multi-ECU systems.

In order to get meaningful results from the analysis of a multi-ECU trace it is mandatory that the timestamps from all ECUs are synchronous. Otherwise, the delay between different processor would result in wrong evaluation metrics and no valid conclusions could be drawn. Therefore, the feasibility of a multi-ECU trace environment is an interesting and important topic for future work.

A Appendix

List of Figures

List of Tables

List of Listings

Keywords

AGBT Aurora Gigabit Interface. 36–38, 63

AUTOSAR AUTomotive Open System ARchitecture. 5, 8, 12, 20, 85, 86

AUTOSAR OS AUTOSAR Operating System. 8, 85

BOB Bus Observation Block. 31, 34

BTF Best Trace Format. 2, 5–7, 18–27, 41–49, 51, 52, 54–59, 62, 69, 70, 72–74, 79, 80, 83, 84

CC OSEK Conformance Class. 7, 11, 12, 43

DAQ Data AcQuisition. 37

DMA Direct Memory Access. 35

EE ERIKA Enterprise Operating System. 51, 53–55, 62, 63, 65, 68, 85, 86

ELF Executable and Linkable Format. 39, 41

ETM Embedded Trace Macrocell. 34, 36

I/O Input/Output. 8, 33, 46

ID Identifier. 13, 15, 19, 21, 46, 48, 50, 51, 55, 84

IMDS Infineon Multicore Debug Solution. 34, 36, 63

IPA Inter-Process Activation. 44, 70, 73, 75, 79

ISR Interrupt Service Routine. 9–16, 19–22, 25, 45–48, 52, 72, 73, 75, 80

JTAG Joint Test Action Group. 37

KOIL Kernel Object Interface Language. 15

MCAL Microcontroller Abstraction Layer. 8

MTA Multiple Task Activation. 11–13, 19, 43, 46, 51, 52, 79

OEM Original equipment manufacturer. 8

OIL OSEK Implementation Language. 14, 15, 61–63

ORTI OSEK Run Time Interface. 4, 5, 15–17, 41–44, 46–54, 56, 57, 62, 83–85

OS Operating System. 3–9, 11–16, 21, 22, 24, 25, 44–46, 48, 51–56, 58–60, 62, 66, 67, 75, 79, 80, 83–86

OSEK OS OSEK Operating System. 6–13, 15, 24, 45, 51, 85

OSEK/VDX Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen. 5–8, 11–16, 23–25, 43, 46, 47, 50, 51, 53, 61, 62, 83–86

POB Processor Observation Block. 31, 34

RPC Remote Procedure Call. 52, 75

RTE Real Time Evaluation. 41, 59, 60

TA Timing-Architects Embedded Systems GmbH. 1, 59, 60, 67, 72

VFB Virtual Function Bus. 12

Bibliography

[1] Infineon Technologies AG. Tc29xa user manual. Confidental.

[2] Infineon Technologies AG. Tricore processor family. `http://www.infineon.com/export/sites/default/media/Applications/Automotive/TriCore_Family-br-2012.pdf`. Accessed: 2015-07-22.

[3] Martin Alfranseder, Matthias Mucha, Stefan Schmidhuber, Alfons Sailer, Michael Niemetz, and Jurgen Mottok. A modified synchronization model for dead-lock free concurrent execution of strongly interacting task sets in embedded systems. In *Applied Electronics (AE), 2013 International Conference on*, pages 1–6. IEEE, 2013.

[4] AUTOSAR. General Specification of Basic Software Modules. `http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/general/standard/AUTOSAR_SWS_BSWGeneral.pdf`, 2014.

[5] Autosar Consortium. AUTomotive Open System ARchitecture. `http://www.autosar.org/`, 2003. Accessed: 2015-05-10.

[6] Osman Balci. Principles and techniques of simulation validation, verification, and testing. In *Simulation Conference Proceedings, 1995. Winter*, pages 147–154. IEEE, 1995.

[7] J Banks, JS Carson, and BL Nelson. *DM Nicol, Discrete-Event System Simulation*. Prentice hall Englewood Cliffs, NJ, USA, 2000.

[8] Christos G Cassandras et al. *Introduction to discrete event systems*. Springer Science & Business Media, 2008.

[9] Michael Deubzer. *Robust Scheduling of Real-Time Applications on Efficient Embedded Multicore Systems*. PhD thesis, Technische Universität München, 2011.

[10] Dixon, Brad and O'Keeffe, Hugh. The Advantages of Real-Time Trace Debug in Complex Embedded Systems. `http://www.mentor.com/`, 2013.

[11] Elektrobit. Eb tresos autocore. https://www.elektrobit.com/products/ecu/eb-tresos/autocore/. Accessed: 2015-02-11.

[12] Domenico Ferrari. *Computer systems performance evaluation*, volume 21. Prentice-Hall Englewood Cliffs, 1978.

[13] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[14] Barbara Geddes. How the cases you choose affect the answers you get: Selection bias in comparative politics. *Political analysis*, pages 131–150, 1990.

[15] HighTec EDV-Systeme GmbH. Development platform - hightec edv-systeme gmbh. https://www.hightec-rt.com/en/products/development-platform.html. Accessed: 2015-08-26.

[16] Lauterbach GmbH. Practice script language user's guide. http://www2.lauterbach.com/pdf/practice_user.pdf. Accessed: 2015-08-28.

[17] Lauterbach GmbH. Trace32 in-circuit debugger. http://www.lauterbach.com/tutorial.pdf. Accessed: 2015-08-26.

[18] Lauterbach GmbH. Trace Export for Third-Party Timing Tools. http://www2.lauterbach.com/pdf/app_timing_tools.pdf, 2015. Accessed: 2015-09-17.

[19] Christian Helm. Event Trace Based Detection and Analysis of Memory Bottlenecks in Multicore Real-Time Systems. Master thesis, Ostbayerische Technische Hochschule Regensburg, 2014.

[20] Maximilian Hempe. Modelling the Dynamic Behaviour of an AUTOSAR Operating System. Master thesis, Hochschule München, 2015.

[21] Andrew BT Hopkins and Klaus D McDonald-Maier. Debug support for complex systems on-chip: A review. *IEE Proceedings-Computers and Digital Techniques*, 153(4):197–207, 2006.

[22] Standard for a Global Embedded Processor Debug Interface, 2012.

[23] iSYSTEM AG. Ocd tricore. http://www.isystem.com/downloads/winIDEA/help/index.html?OCDTriCore.html. Accessed: 2015-08-15.

[24] iSYSTEM AG für Informatiksysteme. Target access hardware ic6000. http://www.isystem.com/index.php/products/hardware/ic6000-on-chip-analyzer. Accessed: 2015-07-22.

[25] iSYSTEM AG für Informatiksysteme. winidea integrated development environment. `http://www.isystem.com/products/software/winidea`. Accessed: 2015-07-23.

[26] Daniel Kästner, Marek Jersak, Christian Ferdinand, Peter Gliwa, and Reinhold Heckmann. An integrated timing analysis methodology for real-time systems. Technical report, SAE Technical Paper, 2011.

[27] Sascha Konrad and Betty HC Cheng. Real-time specification patterns. In *Proceedings of the 27th international conference on Software engineering*, pages 372–381. ACM, 2005.

[28] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[29] Johan Kraft, Anders Wall, and Holger Kienle. Trace recording for embedded systems: Lessons learned from five industrial projects. In *Runtime Verification*, pages 315–329. Springer Berlin Heidelberg, 2010.

[30] Robyn R Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 126–133. IEEE, 1993.

[31] Felix Martin. Hardware Based Tracing of Embedded Multi-Core Systems. Project thesis 1, Ostbayerische Technische Hochschule Regensburg, 2014.

[32] Felix Martin. Automation of a Closed Loop Model-Based Development Process. Project thesis 2, Ostbayerische Technische Hochschule Regensburg, 2015.

[33] Felix Martin, Andreas Sailer, Michael Deubzer, and Jürgen Mottok. Automation of a Closed Loop Model-Based Development Process. In *Applied Research Conference 2015 Conference Book*, 2015.

[34] Felix Martin, Armin Stingl, Michael Deubzer, Stefan Krämer, Martin Hobelsberger, and Jürgen Mottok. Hardware-Based Tracing of Embedded Multi-Core Real-Time Systems. In *Applied Research Conference 2014 Conference Book*, 2014.

[35] A Mayer, H Siebert, A Kolof, and S el Baradie. Debug support for complex system-on-chips. In *CMP media LLC, Embedded Systems Conference*, 2003.

[36] Albrecht Mayer, Alfred Kless, and Stefan Weisse. Automotive Tool Interfaces. *Techonline*, 2013.

[37] Alan Mink, Robert J. Carpenter, George Nacht, and John W. Roberts. Multiprocessor performance-measurement instrumentation. *Computer*, 23(9):63–75, 1990.

[38] Alan Mink and George Nacht. Performance measurement of a shared-memory multiprocessor using hardware instrumentation. In *System Sciences, 1989. Vol. I: Architecture Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, volume 1, pages 267–276. IEEE, 1989.

[39] George Nacht and Alan Mink. A hardware instrumentation approach for performance measurement of a shared-memory multiprocessor. In *Modeling Techniques and Tools for Computer Performance Evaluation*, pages 249–264. Springer, 1989.

[40] Nico Naumann. Autosar runtime environment and virtual function bus. *Hasso-Plattner-Institut, Tech. Rep*, page 38, 2009.

[41] OSEK/VDX. OSEK Implementation Language. `http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf`, 2004.

[42] OSEK/VDX. Operating System. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, 2005.

[43] OSEK/VDX. OSEK Run Time Interface. `http://portal.osek-vdx.org/files/pdf/specs/orti-a-22.pdf`, 2005.

[44] OSEK/VDX. OSEK Run Time Interface. `http://portal.osek-vdx.org/files/pdf/specs/orti-b-22.pdf`, 2005.

[45] OSEK/VDX. OSEK/VDX Standard. `http://portal.osek-vdx.org/`, 2005. Accessed: 2015-05-05.

[46] Kai Richter, Marek Jersak, Christian Ferdinand, and Peter Gliwa. Eine ganzheitliche methodik für den automatisierten echtzeit-nachweis zur absicherung hoch integrierter, sicherheitskritischer software-systeme. *Proceedings Automotive Safety and Security*, 2010.

[47] Andreas Sailer. Towards an Automated Modelling Approach of Real-Time Systems from Trace Recordings. `subs.emis.de/LNI/Proceedings/Proceedings232/2233.pdf`, 2014.

[48] Stefan Schmidhuber. Genetic Optimization of Embedded Multicore Real-Time Systems. Master thesis, Hochschule Regensburg, 2012.

[49] Stan Schneider and Lori Fraleigh. The ten secrets of embedded debugging. *Embedded Systems Programming*, 17:21–32, 2004.

[50] Joseph Sifakis, Stavros Tripakis, and Sergio Yovine. Building models of real-time systems from application software. *Proceedings of the IEEE*, 91(1):100–111, 2003.

[51] Neal Stollon. Infineon multicore debug solution. In *On-Chip Instrumentation*, pages 219–230. Springer, 2011.

[52] Evidence Embedding Technology. Ee alarm tick implementation. `http://svn.tuxfamily.org/viewvc.cgi/erika_erikae/repos/ee/trunk/ee/pkg/kernel/oo/src/ee_altick.c`. Accessed: 2015-08-26.

[53] Evidence Embedding Technology. Erika enterprise real-time operating system. `http://erika.tuxfamily.org/drupal/`. Accessed: 2015-02-11.

[54] Evidence Embedding Technology. Rt-druid reference manual. `http://download.tuxfamily.org/erika/webdownload/manuals_pdf/rtdruid_refman_1_5.0.pdf`. Accessed: 2015-08-28.

[55] Timing Architects Embedded Systems GmbH. Timing architects tool suite. `http://www.timing-architects.com/`. Accessed: 2015-03-10.

[56] Timing Architects Embedded Systems GmbH. Timing architects tool suite simulator. `http://www.timing-architects.com/ta-tool-suite/simulator/`. Accessed: 2015-08-25.

[57] Timing Architects Embedded Systems GmbH. BTF-Specification. `https://wiki.eclipse.org/images/e/e6/TA_BTF_Specification_2.1.3_Eclipse_Auto_IWG.pdf`, 2014.

[58] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-time mach: Towards a predictable real-time system. In *USENIX Mach Symposium*, pages 73–82. Citeseer, 1990.

[59] Lauterbach Development Tools. Target access hardware powertrace-ii. `http://www.lauterbach.com/powertrace2.html`. Accessed: 2015-07-22.

[60] Jonas Trümper, Stefan Voigt, and Jürgen Döllner. Maintenance of embedded systems: Supporting program comprehension using dynamic analysis. In *Software Engineering for Embedded Systems*, pages 58–64. IEEE, 2012.

[61] Jim Turley. Nexus standard brings order to microprocessor debugging. *A White Paper www.nexus5001.org*, 2004.

[62] Joseph Yiu. *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*. Newnes, 2013.

[63] Haibo Zeng and Marco Di Natale. Mechanisms for guaranteeing data consistency and flow preservation in autosar software on multi-core platforms. In *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*, pages 140–149. IEEE, 2011.

Declaration of original authorship

Mir ist bekannt, dass dieses Exemplar der Masterarbeit als Prüfungsleistung in das Eigentum des Freistaates Bayern übergeht.

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und außer den angeführten keine weiteren Hilfsmittel benützt habe.

Soweit aus den im Literaturverzeichnis angegebenen Werken und Internetquellen einzelne Stellen dem Wortlaut oder dem Sinn nach entnommen sind, sind sie in jedem Fall unter der Angabe der Entlehnung kenntlich gemacht.

Die Versicherung der selbständigen Arbeit bezieht sich auch auf die in der Arbeit enthaltenen Zeichen-, Kartenskizzen und bildlichen Darstellungen.

Ich versichere, dass meine Masterarbeit bis jetzt bei keiner anderen Stelle veröffentlicht wurde. Mir ist bewusst, dass eine Veröffentlichung vor der abgeschlossenen Bewertung nicht erfolgen darf.

Ich bin mir darüber im Klaren, dass ein Verstoß hiergegen zum Ausschluss von der Prüfung führt oder die Prüfung ungültig macht.

Regensburg, den 28.10.2015