BTF-Specification

Version History

Version	Author	Datum	Description
V1.0	[Timing-Architects]	2011-07-18	Initial specification approved with thanks by Continental Automotive GmbH, extended by source-entity-instance column
V2.0	[Timing-Architects]	2012-04-17	Added new data types
V2.0.*1*	[Timing-Architects]	2013-03-29	Added state charts and description of all entities
V2.0.*2*	[Timing-Architects]	2013-04-22	First public release
V2.1.*0*	[Timing-Architects, Robert Bosch GmbH]	2013-06-18	- Changed Process State Chart for compliance to OSEK 2.2.3 Extended Task Model - Some improvements of description
V2.1.*1*	[Timing-Architects]	2013-10-30	Clarified description and examples regarding difference preempt/suspend for processes/runnables.
V2.1.*3*	[Timing-Architects]	2014-04-10	-Process state chart: changed layout according to OSEK state order
V2.1.*4*	[Timing-Architects]	2015-03-24	- Added Scheduler, OS-Events, Semaphore-Events and Simulation-Events - Update examples - Correct diction of mtalimitexceeded - Changed allowed source type for activating a process
V2.1.*5*	[Timing-Architects]	2016-01-29	- Semaphore state chart: add missing state transition - Update description of Interrupt Service Routine: short version is *I* not *ISR* - added System-Events - update description of OS Events - improved description of SourceInstance in 2.3

Publisher: Timing-Architects Embedded Systems GmbH
This specification was extended in the context of the AMALTHEA ITEA2 project (ID 01IS11020E) and is released on the Eclipse Auto IWG

			- corrected description of #typeTable - improved description of Source and Target in 2.3

Note: In version key V x.x.y, x represents change in BTF, y is only specification update.

License Disclaimer

- *BTF is accessible to everyone free of charge.*
- *BTF and Timing-Architects Embedded Systems GmbH do not favor one implementer over another for any reason other than the technical standards compliance of a vendor's implementation.*
- *BTF is published under royalty-free terms*
- *BTF remains accessible and free of charge*
- *BTF is accessible free of charge and documented in all its details (i.e. all aspects of the standard are transparent and documented, and both access to and use of the documentation is free)*
- *BTF is free for all to implement, with no royalty or fee. Certification of compliance by Timing-Architects Embedded Systems GmbH may involve a fee.*
- *BTF implementations may be extended, or offered in subset form. However, certification organizations may decline to certify subset implementations, and may place requirements upon extensions*
- *BTF extensions have to be integrated in BTF and published under this open format license.*

CONTENT

1. List of Figures

2. List of Tables

1. INTRODUCTION

This document specifies a tracing format for timing evaluation of event based systems. The BTF (Best Trace Format, originating from Better Trace Format (BTF V1.0)) is a CSV-based format for representation of event-traces in ASCII. BTF is a format definition for full scale timing traces of simulator and profiling tools.

The Best Trace Format (BTF) is based on the Better Trace Format, initially defined by Continental Automotive GmbH. It allows analyzing the behavior of a system in a chronologically correct manner in order to apply timing, performance, or reliability evaluations. In general, it assumes a signal processing system, where one component of the system notifies another component of the system. These notifications are realized by events, stored in the BTF file. In comparison with compact trace formats from debugger traces, a BTF log of an event includes the entire information, namely: which component interacts with which component by an event.
Advanced scheduling concepts may be used in multicore processor systems where one traced component may have multiple instances at the same time, i.e. global scheduling or task migration concepts. This requires instance identification in order to derive which instance of a component is addressed in the event log. This means for example that each task execution can be exactly identified for the complete lifetime from activation till termination by its component name and instance counter.

The following figure shows the interaction between two component instances, where component Name1 (Instance #21452) sends an event X to component Name2 (Instance #124) at t=1200025. A component instance is generated from its parent component and duplicates its behavior (e.g. execution time according to a certain sequence). Nevertheless, it may be possible that a component instance exists over the complete traced time interval.

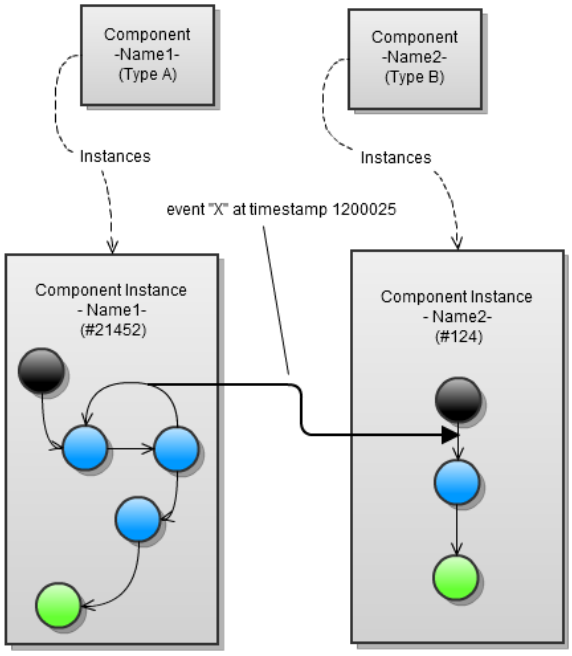

FIGURE 1: SCHEMATIC VISUALIZATION OF INTERACTION BETWEEN TWO ENTITY INSTANCES.

2. Structure of BTF-File

A BTF-file consists of two parts:

1. A header-section, containing meta-information on objects of the trace and optional comments.

   All lines start with a ‚#'. The meta-information is described by pragma-statements (see Section 2.1.2). Comments contain directly after the '#'-symbol a space, which allows the differentiation from pragma-statements.

2. A data-section, containing the trace-data of the simulation or measurement.

   The data-section consists of lines in CSV format with optional comment-lines. Each line represents one event of the traced system. The columns of the event-line describe the time, entities, and event. Comments are defined as in the header-section.

For the representation of the data-sections two ways exist. The symbolic-mode describes entities and event by names. The numeric-mode describes entities and event by a numerical identifier. In this case, the header-section includes the mapping between numerical identifier and a string of the name.

2.1. Header

The header includes parameters, used for the interpretation of the trace or information of the trace generator, and comments. Parameters and comments are indicated by a '#'-symbol.
The typical header of a BTF-file includes at least the version, creator, creation date and the time scale. Further information is optional.

2.1.1. Comments

Each row, starting with a '#'-symbol which is followed with a whitespace is a comment. Comments can be part of the header or can be entered at any position of the data section.

2.1.2. Parameters

Each row, starting with a '#'-symbol and one of the following parameter definitions is a parameter.
The parameter definition may not start with a whitespace. When the symbol '-' follows the '#' symbol, the row is an entry of the last defined table (e.g. typeTable). Following parameter definitions are predefined:

TABLE 1: PARAMETERS FOR BTF HEADER SECTION

Parameter	Description	Type	Example
#version	Version of BTF format definition	String	#version 1.0
#creator	Name and version of the program or device, which generated the trace	String	#creator TA-Simulator (12.10.2.47)

#creationDate	Timestamp of the start of simulation or measurement. The format has to comply with "ISO 8601 extended specification for representations of dates and times" YYYY-MM-DDTHH:MM:SS. The time should be in UTC time (indicated by a “Z” at the end)	String (ISO 8601)	#creationdate 2012-09-02T16:40:30Z
#inputFile	Filename of the model which was used for the simulation	String (URI)	#inputFile D:\Workspace\Project\DualCore.rte
#timescale	Defines the resolution of the timestamps in the trace. Default unit is nano-seconds (ns).	String (Enumeration [ps,ns,us,ms,s])	#timescale ns
#typeTable	Indicates the beginning of a mapping from all entities to a numerical Type-Id. See Table 3 for existing types. Type-Ids start with 0. Missing Ids are allowed	-<n> String	#typeTable #-0 T #-1 R #-2 SIG
#entityTable <n>	Indicates the beginning of a mapping from all entities to a numerical Entity-Id. An entity can be a task, runnable, etc. Type-Ids start with 0 and some Ids can be missing.	-<n> String	#entityTable #-0 Task_1ms #-1 GetSignal #-2 Main #-3 Temperature
#entityTypeTable <n>	Indicates the beginning of a mapping from all entities to types. Both, entity and type have to be defined before in	-<n> String	#entityTypeTable #-T Task_1ms #-R GetSignal #-R Main

	the entityTable and typeTable.		#-SIG Temperature

Example:

A typical header is shown in the following listing:

```
#version 1.0
#creator TA-Toolsuite 12.06.1
# Simulation of dualcore processor 120MHz, 16Kbyte RAM
#creationDate 2012-08-31T15:53:00
#inputFile c:\TAsc\doc\examples\ems.tap
#timeScale ns
#typeTable
#-0 T
#-1 R
#entityTable
#-0 Task_1ms
#-1 Task_2ms
#-2 Runnable_1ms_Init
#-3 Runnable_2ms_Store
#-4 Runnable_2ms_Read
#entityTypeTable
#-T Task_1ms
#-T Task_2ms
#-R Runnable_1ms_Init
#-R Runnable_2ms_Store
#-R Runnable_2ms_Read
```

2.2. DATA SECTION

The trace information is represented in CSV format. Each line describes one event. The interpretation of one line depends on the event type, shown in the next section.

For separating the content of one line the symbol ‘,’ (comma) is used. For the case using floating numbers, a ‘.’ (point) has to be used as decimal separator.

At any point of the trace section, a comment with pragma ‘#’ can be written.

2.3. ENTITIES AND EVENTS

The data section consists of line by line interpreted data. Each line has eight columns, whereas the last column is optional. A line contains the following elements:

<Time>,<Source>,<SourceInstance >,<TargetType>,<Target>,<TargetInstance>,<Event>,<Note>

The interpretation of the different columns depends on the <TargetType>-column.

TABLE 2: DESCRIPTION OF BTF COLUMNS

Column	Name	Description	Relevant for entity type
1	Time	Integer timestamp for one action. The timescale is given in the configuration section by the parameter #timescale.	all
2	Source	Arbitrary but unique name for the source which triggers the event. (e.g. core at start of a task or stimulus at the activation of a task)	all
3	SourceInstance	Instance counter for the source. Simulation entities have each time the instance '-1', all other non-instanceable entities (e.g. core) have instance '0' except at initialization events (see 2.3.4). Instanceable entities like stimuli starting with 0 and increment at each instantiation the counter. If there is no information for the instance this field can be empty.	all
4	Type	Type of the event target.	all
5	Target	Arbitrary but unique name for the target, which triggers the event (e.g. a task, runnable, signal-access).	all
6	TargetInstance	Instance counter for the target.	all
7	Event	Name of the event	all
8	Note	Optional field for further information (e.g. signal value at signal read or write access)	all

The fourth column (<TargetType>) includes the type of the event. Following types are defined in the BTF:

TABLE 3: ENTITY TYPES.

Category	Type-ID	Name	Description
Environment			
	STI	Stimulus	Trigger point for a Task or Interrupt-Service-Routine
Software			
	T	Task (Specialization of Process)	Object handled by OS Scheduler, and calling all "Top-Level" Runnables. A Task is the specialization of a Process type.
	I	Interrupt-Service-Routine (Specialization of Process)	Object handled by Interrupt-Management Unit and calling all "Top-Level" Runnables. An Interrupt-Service-Routine is the specialization of a Process type.
	R	Runnable	Object, called by a Process or another Runnable.
	IB	Instruction block	Sub-fraction of a Runnable
Hardware			
	ECU	Electronic Control Unit	Hardware device which has at least one processor.
	Processor	Processor	Hardware device which has at least one core
	C	Core	Hardware device which is part of a processor and executes software.
	M	Memory Module	Hardware device which is part of a processor.
Operating System			
	SCHED	Scheduler	Part of operating system which assigns processes to cores.
	SIG	Signal	Shared data object (e.g. variable) in a software.
	SEM	Semaphore	Operating system object, for restricting access to resources.
	EVENT	Event	Operating system object, for synchronization.
Information			
	SIM	Simulation	Used for notification events from simulation environment, e.g. simulation started or simulation stopped.

Example:

```
0,      Stimulus_Task_A,      0,      T,      Task_A,         0,      activate
100,    Core_1,               0,      T,      Task_A,         0,      start
100,    Task_A,               0,      R,      Runnable_A_1,   0,      start
7100,   Task_A,               0,      R,      Runnable_A_1,   0,      terminate
7100,   Task_A,               0,      R,      Runnable_A_2,   0,      start
10000,  Stimulus_Task_B,      0,      T,      Task_B,         0,      activate
10100,  Task_A,               0,      R,      Runnable_A_2,   0,      suspend
10100,  Core_1,               0,      T,      Task_A,         0,      preempt
10100,  Core_1,               0,      T,      Task_B,         0,      start
10100,  Task_B,               0,      R,      Runnable_B_1,   0,      start
17100,  Task_B,               0,      R,      Runnable_B_1,   0,      terminate
17100,  Core_1,               0,      T,      Task_B,         0,      terminate
17200,  Core_1,               0,      T,      Task_A,         0,      resume
17200,  Task_A,               0,      R,      Runnable_A_2,   0,      resume
21200,  Task_A,               0,      R,      Runnable_A_2,   0,      terminate
21200,  Core_1,               0,      T,      Task_A,         0,      terminate
```

Gantt-Chart of Example

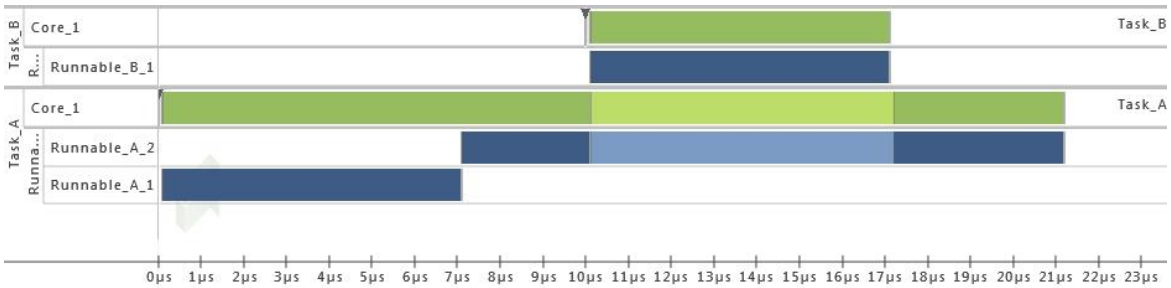

FIGURE 2: GANTT CHART OF EXAMPLE. DARK GREEN/BLUE AREAS SHOW EXECUTION OF TASK/RUNNABLE. LIGHT GREEN/BLUE AREAS SHOW TASKS/RUNNABLES IN READY/SUSPENDED STATE

2.3.1. STIMULUS-EVENTS

A stimulus is used to model external inputs or internal behavior, which is not modeled by other software or hardware parts. A stimulus is able to activate a task/interrupt-service-routine or set a signal value.

TABLE 4: COLUMNS FOR STIMULUS ENTITY.

Column	Entries
<Source>	Simulation (SIM), Task (T) or interrupt-service-routine (I)
<Event>	trigger

Example:
In the following example Task_A is activated by Stimulus_Task_A, which is a single stimulus triggered by the simulation system. After that Task_A activates Task_B by inter-process activation. Therefor Task_A triggers Stimulus_Task_B and this stimulus activates Task_B.

```
0,      SIM,                    -1,     STI,    Stimulus_Task_A,        0,      trigger
0,      Stimulus_Task_A,        0,      T,      Task_A,                 0,      activate
100,    Task_A,                 0,      STI,    IR_Scheduler_1,         0,      trigger
100,    Core_1,                 0,      T,      Task_A,                 0,      start
7100,   Task_A,                 0,      STI,    Stimulus_Task_B,        0,      trigger
7100,   Stimulus_Task_B,        0,      T,      Task_B,                 0,      activate
7200,   Task_B,                 0,      STI,    IR_Scheduler_1,         1,      trigger
7200,   Core_1,                 0,      T,      Task_A,                 0,      preempt
7200,   Core_1,                 0,      T,      Task_B,                 0,      start
```

2.3.2. PROCESS-EVENTS (TASK- AND ISR-EVENTS)

A process can be either a task or an interrupt service routine. A process is activated by a stimulus, as described in section 2.3.1. After activation, a scheduler assigns the process to a core where the process is executed. A running process can be preempted by another process and change to READY. Alternatively, a cooperative process can change itself to READY, e.g. at a schedule point, or explicitly migrate to another core. When a running process requests a resource (e.g. semaphore or event) which is not available, the process waits actively (e.g. “while(ResourceNotAvailible){...}”). This is indicated by the state POLLING. The scheduler could decide to remove a waiting process from the core and the process changes in state PARKING (passive waiting). When the requested resource becomes available but the process is in state parking, the process changes again to state READY.

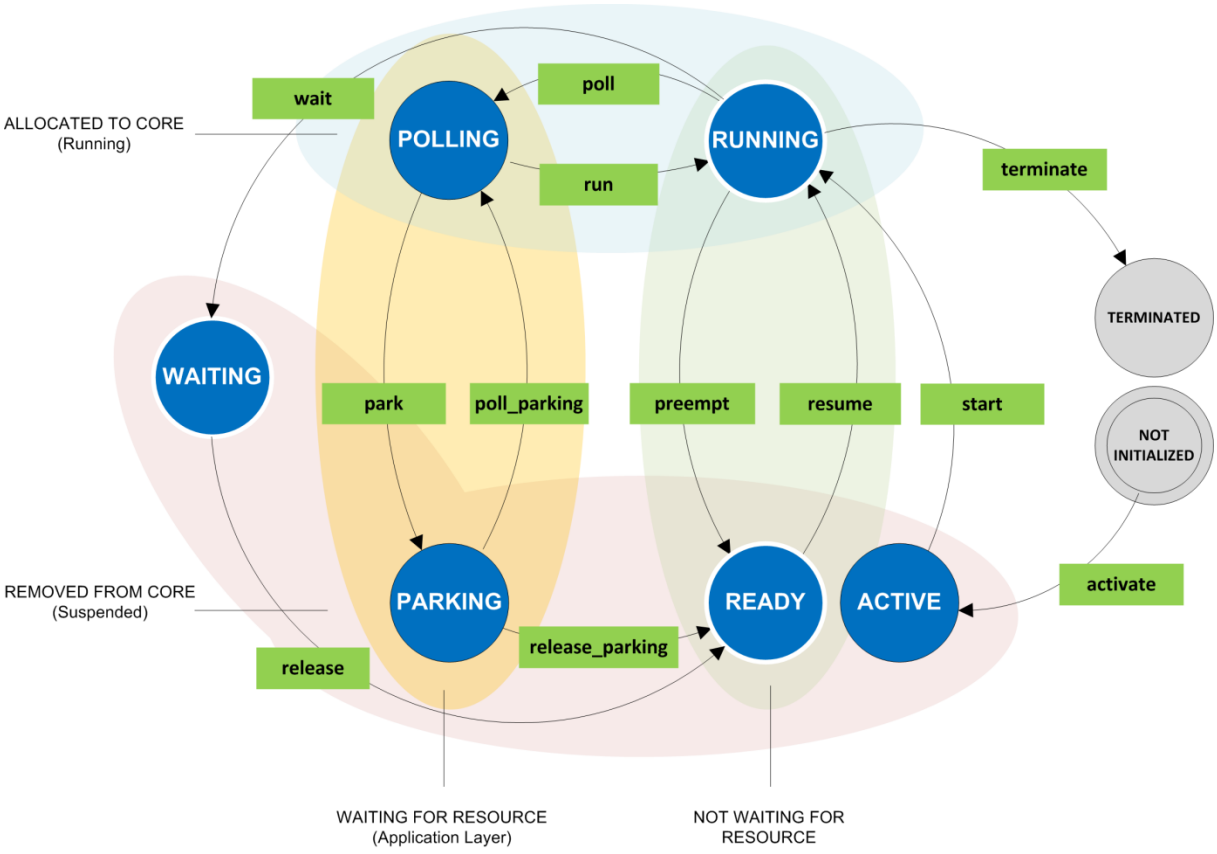

FIGURE 3: PROCESS STATE CHART.

***Note**: Whenever process (P) is used in the following description, this can either be a task (T) or interrupt-service-routine (I)*

TABLE 5: COLUMNS FOR PROCESS ENTITY.

Column	Entries
<Source>	Stimulus (STI), Core (C)
<Event>	activate, start, preempt, resume, terminate, poll, run, park, poll_parking, release_parking, wait, release, mtalimitexceeded, boundedmigration, phasemigration, fullmigration, enforcedmigration

TABLE 6: STATES FOR PROCESS ENTITY.

State	Description
ACTIVE	When the instance is ready for execution.
RUNNING	When the instance executes on a core.
READY	When the instance was preempted.
WAITING	When the instance has requested an OS Event which is not available and waits passively.
POLLING	When the instance has requested a resource which is not available and waits actively.
PARKING	When the instance waits for a not available resource and is preempted.
TERMINATED	When the instance has finished its execution.

TABLE 7: EVENTS FOR PROCESS ENTITY.

Internal Event	Description	Source
ACTIVATE	The process instance is activated by a stimulus.	STI
START	The process instance is allocated to the core and starts execution for the first time.	C
PREEMPT	The executing process instance is stopped by the scheduler, e.g. because of a higher priority process which is activated.	C
RESUME	The preempted process instance continues execution on the same or other core.	C
TERMINATE	The process instance has finished execution.	C
POLL	The process instance has requested a resource by polling (active waiting) which is not available.	C

RUN	The process instance resumes execution after polling (i.e. active waiting) for a resource.	C
PARK	The active waiting process instance is preempted by another process.	C
POLL_PARKING	The parking process instance is allocated to the core and again polls (i.e. actively waits) for a resource.	C
RELEASE_PARKING	The resource which is requested by a parking process instance becomes available, but the parking process stays preempted and changes to READY state.	C (last Core)
WAIT	The process has requested a non-set OS EVENT (see 3a: OSEK 2.2.3 Extended Task Model, WAIT_Event()).	C (last Core)
RELEASE	The OS EVENT which was requested by a process is set (see see 3a: OSEK 2.2.3 Extended Task Model, SET_Event()) and the process is ready to proceed execution.	C (last Core)

TABLE 8: INFO EVENTS FOR PROCESS ENTITY.

Notification-Event	Description
MTALIMITEXCEEDED	When there are more process instances of this process as the MTA-LIMIT value (MTA = Multiple Task Activation).
BOUNDEDMIGRATION	When the last executing core of the previous instance is not equal to first executing core of this instance.
PHASEMIGRATION	When the executing core before a preemption is not equal to the new executing core and there is no schedule point right before this execution.
FULLMIGRATION	When the executing core before a preemption is not equal to the new executing core and there is a schedule point right before this execution.
ENFORCEDMIGRATION	When a process migrates at a predefined position to execute on another scheduler.

Example:

The example shows the activation of TASK_InputProcessing, triggered by a timer. TASK_InputProcessing starts execution and is preempted by task TASK_1MS, also triggered by a timer. After TASK_1MS has finished execution, TASK_InputProcessing resumes execution.

```
6150000,        TIMER-A_2ms,    3,      T,      TASK_InputProcessing,  3,      activate
6150100,        Core_1,         0,      T,      TASK_InputProcessing,  3,      start
6250000,        TIMER-1MS,      6,      T,      TASK_1MS,              6,      activate
6250100,        TASK_1MS,       6,      STI,    IR_SCHED_Tasks_C1,     24,     trigger
6250100,        Core_1,         0,      T,      TASK_InputProcessing,  3,      preempt
6250100,        Core_1,         0,      T,      TASK_1MS,              6,      start
6721825,        Core_1,         0,      T,      TASK_1MS,              6,      terminate
6721925,        Core_1,         0,      T,      TASK_InputProcessing,  3,      resume
7110175,        Core_1,         0,      T,      TASK_InputProcessing,  3,      terminate
```

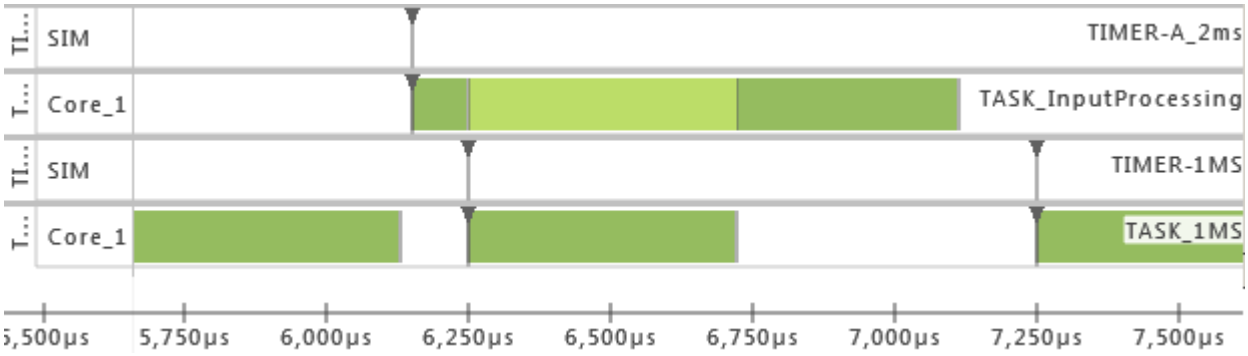

FIGURE 4: GANTT OF EXAMPLE. DARK GREEN AREAS SHOW EXECUTION OF TASK. LIGHT GREEN AREAS SHOW TASKS IN READY STATE.

2.3.3. RUNNABLE EVENTS

A runnable is called within a process instance or in the context of another runnable. When a runnable is called, it starts and changes to RUNNING. When the process instance which includes the runnable is suspended, the runnable itself is also suspended and changes to state SUSPENDED. When the process instance is resumed, the runnable also changes to RUNNING. After complete execution, the runnable changes to TERMINATED.

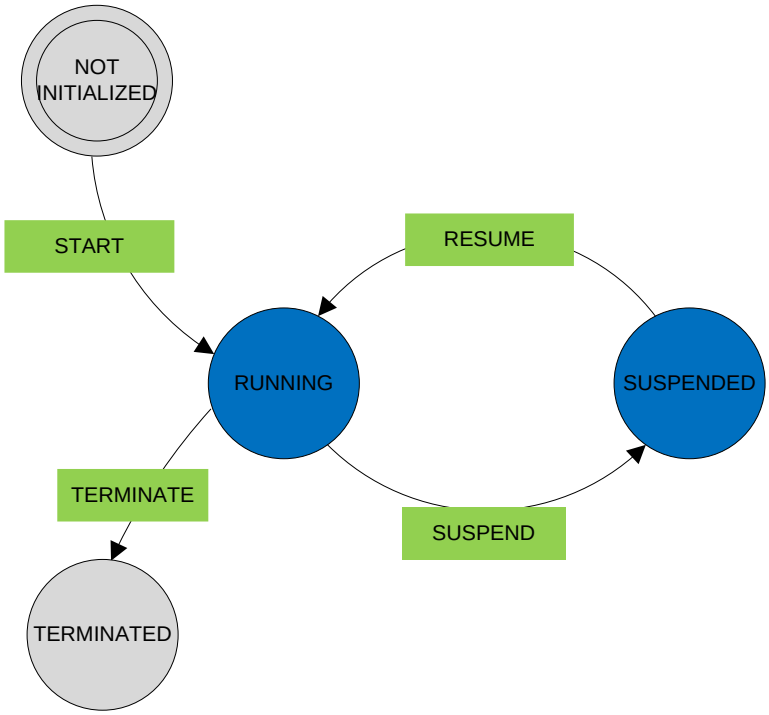

FIGURE 5: RUNNABLE STATE CHART.

TABLE 9: COLUMNS FOR RUNNABLE ENTITY.

Column	Entries
<Source>	Process (P)
<Event>	start, suspend, resume, terminate

TABLE 10: STATES FOR RUNNABLE ENTITY.

State	Description
RUNNING	Runnable instance executes on a core
SUSPENDED	Runnable instances has stopped execution on a core

TABLE 11: EVENTS FOR RUNNABLE ENTITY.

Events	Description	Source
START	The runnable instance is allocated to the core and starts execution for the first time.	P
SUSPEND	The executing runnable instance is stopped, because the calling process is suspended.	P
RESUME	The suspended runnable instance continues execution on the same or another core.	P
TERMINATE	Runnable instance has finished execution.	P

The runnable Runnable_A_2 is started and then suspended by runnable Runnable_B_1. After termination of Runnable_B_1, Runnable_A_2 resumes execution.

```
7100,   Task_A,         0,      R,      Runnable_A_2,  0,      start
10100,  Task_A,         0,      R,      Runnable_A_2,  0,      suspend
10100,  Task_B,         0,      R,      Runnable_B_1,  0,      start
17100,  Task_B,         0,      R,      Runnable_B_1,  0,      terminate
17200,  Task_A,         0,      R,      Runnable_A_2,  0,      resume
21200,  Task_A,         0,      R,      Runnable_A_2,  0,      terminate
```

2.3.4. SCHEDULER

The scheduler is part of the operating system and manages one or multiple cores. It is responsible for the execution order of all mapped processes on those cores.

TABLE 12: COLUMNS FOR SCHEDULER ENTITY.

Column	Entries
<Source>	SCHEDULER (SCHED), PROCESS (P)
<Event>	FINALIZE, SCHEDULE, PROCESSACTIVATE, SCHEDULEPOINT, PROCESSPOLLING, PROCESSTERMINATE

TABLE 13: EVENTS FOR SCHEDULER ENTITY.

Internal Event	Description	Source
SCHEDULE	The scheduling algorithm is executed.	SCHED
PROCESSACTIVATE	A process has been activated.	P
SCHEDULEPOINT	A process has hit a cooperative schedule point.	P
PROCESSPOLLING	A process has entered state polling.	P
PROCESSTERMINATE	A process has terminated.	P

Example:
TASK_A is preempted by TASK_B with a higher priority. After termination of TASK_B, a schedule decision is required and therefor Scheduler_1 is called. No other task is active so TASK_A can be resumed.

```
10100,        Core_1,        0,     T,      Task_A,        0,     preempt
10100,        Core_1,        0,     T,      Task_B,        0,     start
17100,        Core_1,        0,     T,      Task_B,        0,     terminate
17100,        Task_B,        0,     SCHED,  Scheduler_1,   -1,    processterminate
17200,        Scheduler_1,   -1,    SCHED,  Scheduler_1,   -1,    schedule
17200,        Core 1,        0,     T,      Task A,        0,     resume
```

TASK_B is preempted, because it reaches a schedule point. A schedule decision is required to call Scheduler_1. As no task with higher priority is active, TASK_B can be resumed.

```
10100,  Core_1,        0,     T,            Task_B,        0,     start
17100,  Task_B,        0,     SCHED,        Scheduler_1,   -1,    schedulepoint
17100,  Core_1,        0,     T,            Task_B,        0,     preempt
17200,  Scheduler_1,   -1,    SCHED,        Scheduler_1,   -1,    schedule
17200,  Core_1,        0,     T,            Task_B,        0,     resume
24200,  Core_1,        0,     T,            Task_B,        0,     terminate
```

TASK_A starts on Core_1. As it has to wait for Event_1, it enters state POLLING. This state transition triggers the execution of the scheduler. TASK_B starts on Core_2 and sets the required Event_1, so that TASK_A is able to switch to state run again on Core_1.

```
100,    Core_1,              0,     T,      Task_A,        0,     start
7108,   Task_A,              0,     EVENT,  Event_1,       0,     wait_event
7108,   Core_1,              0,     T,      Task_A,        0,     poll
7108,   Task_A,              0,     SCHED,  Scheduler_1,   -1,    processpolling
10000,  Stimulus_Task_B,     0,     T,      Task_B,        0,     activate
10000,  Task_B,              0,     SCHED,  Scheduler_2,   -1,    processactivate
10100,  Scheduler_2,         -1,    SCHED,  Scheduler_2,   -1,    schedule
10100,  Core_2,              0,     T,      Task_B,        0,     start
17100,  Task_B,              0,     EVENT,  Event_1,       0,     set_event,Task_A
17100,  Core_1,              0,     T,      Task_A,        0,     run
24100,  Core_1,              0,     T,      Task_A,        0,     terminate
```

2.3.1. OS-EVENTS

OS-Events are objects provided by the operating system. Operating OS-Events are executed in the context of a process. They offer a possibility to synchronize different processes. If a process instance requires information provided by another process at a predefined position, it waits for an OS-Event (WAIT_EVENT). So the process starts polling/waiting until the required event is set (SET_EVENT). In case the event should be reset it has to be cleared (CLEAR_EVENT). (see 3a: OSEK 2.2.3; 7. Event Mechanism)

TABLE 14: COLUMNS FOR SCHEDULER ENTITY.

Column	Entries
<Source>	Process (P)
<Event>	WAIT_EVENT, CLEAR_EVENT, SET_EVENT
<Note> (SET_EVENT)	Process (P) (target for which event should be set)

TABLE 15: EVENTS FOR SCHEDULER ENTITY.

Internal Event	Description	Source
WAIT_EVENT	A process has to wait for an OS-Event.	P
CLEAR_EVENT	A potentially set OS-Event for the calling task gets cleared.	P
SET_EVENT	A process sets an OS-Event for another process. The note column is the triggered process.	P

Example:
Task_A waits for OS-Event ExampleOsEvent and therefor has to poll. Task_B sets this event for Task_A, so that Task_A can resume. Afterwards the event gets cleared.

```
0,      Stimulus_Task_A,       0,      T,      Task_A,         0,      activate
100,    Core_1,                0,      T,      Task_A,         0,      start
1000,   Stimulus_Task_B,       0,      T,      Task_B,         0,      activate
1100,   Core_2,                0,      T,      Task_B,         0,      start
10108,  Task_A,                0,      EVENT,  ExampleOsEvent,0,      wait_event
10108,  Core_1,                0,      T,      Task_A,         0,      poll
11100,  Task_B,                0,      EVENT,  ExampleOsEvent,0,      set_event,Task_A
11100,  Core_1,                0,      T,      Task_A,         0,      run
11100,  Task_A,                0,      EVENT,  ExampleOsEvent,0,      clear_event
21100,  Core_1,                0,      T,      Task_A,         0,      terminate
21100,  Core_2,                0,      T,      Task_B,         0,      terminate
```

2.3.2. SIGNAL-EVENTS

A signal is basically an address in the memory of a micro-controller. This memory location contains a certain value, which can be accessed by a process instance if required. So generally a signal fulfills the function of naming the memory space, like a label. According to the stored value the process accessing it might change its behavior (READ).
Besides a process also a stimulus is able to write to the storage location and change the value of this label (WRITE).

TABLE 16: COLUMNS FOR SIGNAL ENTITY.

Column	Entries
<Source>	Process (P), Stimulus (STI)
<Event>	read, write
<Note>	Signal value (unused for READ-Event)

TABLE 17: EVENTS FOR SIGNAL ENTITY.

Events	Description	Source
READ	The signal is read by a process or a dynamic signal.	P, SIG
WRITE	The signal is written by a process or a stimulus. The value which should be written is stored in the <Note>-column.	P, STI

Example:

```
1222481,        STI_MODE_SWITCH,        0,      SIG,    HighPowerMode,          0,      write,  1
1222481,        TASK_200MS,             0,      SIG,    HighPowerMode,          0,      read,   1
4482566,        TASK_WritingActuator,   2,      SIG,    S16_C1_1,               0,      write,  0
5590428,        TASK_10MS,              0,      SIG,    S16_C1_1,               0,      read,   0
```

2.3.3. SEMAPHORE-EVENTS

If more than one process is able to access a common resource, it might be necessary to restrict the maximum amount of accesses in order to protect this resource from race conditions. Therefor the operating system provides the possibility to use a semaphore. It is possible to assign processes to the protected resource as long as the maximum value is not reached (semaphore is unlocked). If a process is assigned and the maximum amount of semaphore users is reached, the resource gets locked. In this state every new accessing entity has to wait until one of the previous accessing ones releases the resource.

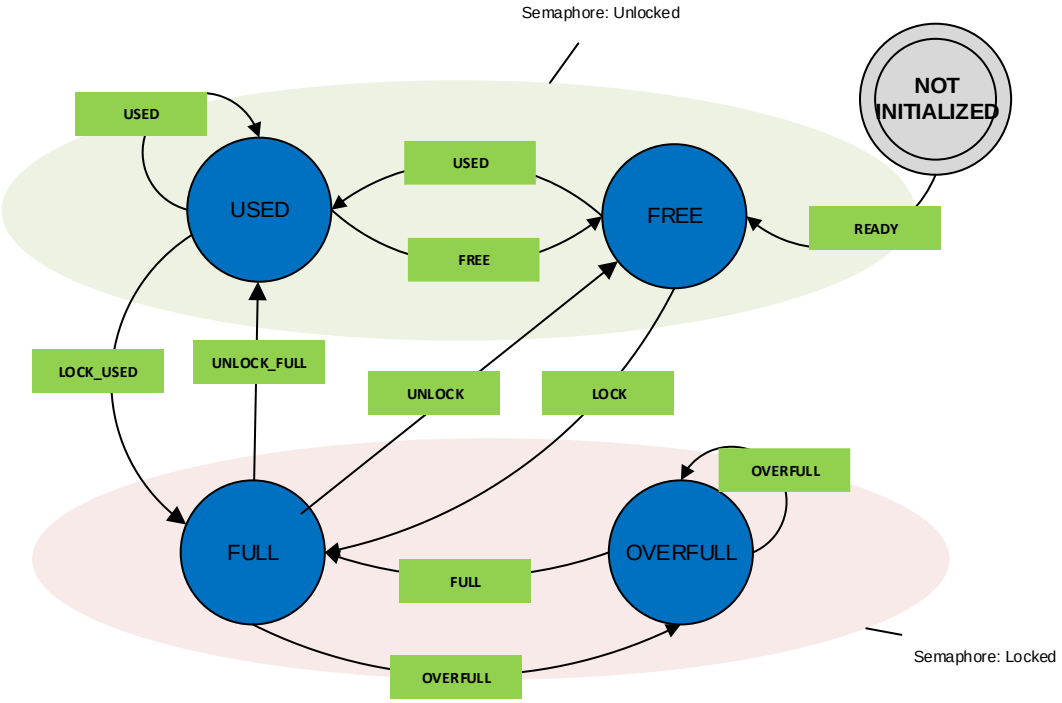

FIGURE 6: SEMAPHORE STATE CHART.

TABLE 18: COLUMNS FOR SEMAPHORE ENTITY.

Column	Entries
<Source>	Semaphore (SEM)
<Event>	Ready, lock, unlock, finalize, requestsemaphore, exclusivesemaphore, releasesemaphore, trigger, increment, decrement, queued, assigned, waiting, released, free, used, full, overfull, unlock_full, lock_used
<Note>	Count of requesting users.

TABLE 19: STATES FOR SEMAPHORE ENTITY.

State	Description
NOTINITALIZED	Semaphore is in this state before counter is initialized.
FREE	Semaphore has no assigned users.
USED	Semaphore has assigned requests and is still able to handle at least one request.
FULL	Semaphore has assigned requests and has reached its maximum value.
OVERFULL	Semaphore is locked and at least one request is waiting for the semaphore.

TABLE 20: EVENTS FOR SEMAPHORE ENTITY

Events	Description	Source
READY	The semaphore gets initialized.	SEM
LOCK	Semaphore reaches its maximum value of simultaneous accesses and had no users assigned before.	SEM
UNLOCK	Semaphore reaches 0 assigned users and was full before.	SEM
REQUESTSEMAPHORE	Semaphore is requested by a task.	P
EXCLUSIVESEMAPHORE	There is an exclusive semaphore request by a task.	P
QUEUED	Semaphore request is queued.	SEM
ASSIGNED	Semaphore request gets assigned to resource.	P
WAITING	Semaphore is requested, but locked, therefor semaphore request has to wait.	P
RELEASED	Assigned semaphore request releases semaphore.	P
FREE	Semaphore reaches 0 assigned users and was not full before.	SEM
USED	Semaphore is requested and has users assigned, but does not reach maximum value of simultaneous accesses.	SEM
UNLOCK_FULL	Semaphore is released by a user, has still other users assigned and was full before.	SEM
FULL	Semaphore is released by a user and reaches its maximum value of simultaneous accesses. Therefor no requesting user has to wait.	SEM
LOCK_USED	Semaphore is requested by a user and reaches its maximum value of simultaneous accesses.	SEM
OVERFULL	Semaphore is requested by a user and has more simultaneous accesses as allowed by its maximum value. Therefor at least one requesting user has to wait.	SEM
INCREMENT	Semaphore is requested by a task and therefor its counter gets incremented.	P
DECREMENT	Semaphore is released by a task and therefor its counter gets decremented.	P

Example:
The following example shows the behavior of a semaphore SEM_MemProtection, which has 1 as its maximum count of simultaneous accesses. It is requested by TASK_1ms_C1, which gets assigned to the resource. Therefor SEM_MemProtection gets locked and the second requesting task TASK_1ms_C2 has to wait until TASK_1ms_C1 releases the resource.

```
0,      SEM_MemProtection,    0,      SEM,    SEM_MemProtection,    0,      ready,          0
0,      SEM_MemProtection,    0,      SEM,    SEM_MemProtection,    0,      free,           0
3225,   TASK_1ms_C1,          0,      SEM,    SEM_MemProtection,    0,      increment
3225,   TASK_1ms_C1,          0,      SEM,    SEM_MemProtection,    0,      requestsemaphore,0
3225,   SEM_MemProtection,    0,      SEM,    SEM_MemProtection,    0,      queued,         0
3225,   SEM_MemProtection,    0,      SEM,    SEM_MemProtection,    0,      lock,           1
3225,   TASK_1ms_C1,          0,      SEM,    SEM_MemProtection,    0,      assigned,       1
4225,   TASK_1ms_C2,          0,      SEM,    SEM_MemProtection,    0,      increment
4225,   TASK_1ms_C2,          0,      SEM,    SEM_MemProtection,    0,      requestsemaphore,1
4225,   SEM_MemProtection,    0,      SEM,    SEM_MemProtection,    0,      queued,         1
4225,   SEM_MemProtection,    0,      SEM,    SEM_MemProtection,    0,      overfull,       2
4225,   TASK_1ms_C2,          0,      SEM,    SEM_MemProtection,    0,      waiting,        2
460031, TASK_1ms_C1,          0,      SEM,    SEM_MemProtection,    0,      released,       2
460031, TASK_1ms_C1,          0,      SEM,    SEM_MemProtection,    0,      decrement
460031, TASK_1ms_C2,          0,      SEM,    SEM_MemProtection,    0,      assigned,       1
460031, SEM_MemProtection,    0,      SEM,    SEM_MemProtection,    0,      full,           1
687875, TASK_1ms_C2,          0,      SEM,    SEM_MemProtection,    0,      released,       1
687875, TASK_1ms_C2,          0,      SEM,    SEM_MemProtection,    0,      decrement
687875, SEM_MemProtection,    0,      SEM,    SEM_MemProtection,    0,      unlock,         0
```

2.3.4. SIMULATION-EVENTS

Events of the simulation environment are events that provide additional information, such as simulation states and model information.

TABLE 21: COLUMNS FOR SIMULATION ENTITY.

Column	Entries
<Source>	SIM
<Event>	finalize, error, tag, description

TABLE 22: EVENTS FOR SIMULATION ENTITY.

Events	Description	Source
FINALIZE	Initialization of Simulation Environment complete.	SIM
ERROR	Error occurred during simulation	SIM
TAG	Transmit meta information (tag) for the corresponding source entity. The information is stored in the Note column.	*
DESCRIPTION	Transmit meta information (description) for the corresponding source entity. The information is stored in the Note column.	*

Example:

```
0,      Process_1,      0,      SIM,    Simulation,             0,      tag,    <tagName>
```

TABLE 23: DEFINED TAG EVENTS

Note	Description	Source
SIG_INIT_VALUE,<value>	Event is written at the beginning of the trace. Every signal is listed there with its initial value in <value>.	SIG
ECU_INIT	Event is written at the beginning of the trace. Used for mapping ECU<->Processor <->Core.	ECU
PROCESSOR_INIT	Event is written at the beginning of the trace. Used for mapping ECU<->Processor <->Core.	Processor
CORE_INIT	Event is written at the beginning of the trace. Used for mapping ECU<->Processor <->Core.	C
OSOVERHEAD_RUNNABLE	Event is written at the beginning of the trace. Used to identify runnables responsible for OS Overhead.	R
OSOVERHEAD_PROCESS	Event is written at the beginning of the trace. Used to identify processes responsible for OS Overhead.	P

Example:
The following example shows the events used for hardware mapping. Core_3 is mapped to Processor_2, Core_1 and Core_2 are mapped to Processor_1. Processor_1 and Processor_2 are mapped to Ecu_1.

```
0,      Ecu_1,                  -1,     SIM,    SIM,    -1,     tag,    ECU_INIT
0,      Processor_1,            -1,     SIM,    SIM,    -1,     tag,    PROCESSOR_INIT
0,      Core_1,                 -1,     SIM,    SIM,    -1,     tag,    CORE_INIT
0,      Core_2,                 -1,     SIM,    SIM,    -1,     tag,    CORE_INIT
0,      Processor_2,            -1,     SIM,    SIM,    -1,     tag,    PROCESSOR_INIT
0,      Core_3,                 -1,     SIM,    SIM,    -1,     tag,    CORE_INIT
```

Example:
The following example shows an event used for signal initialization. The signal SIG_Temperature has the initial value 0.

```
0,      SIG_Temperature,        -1,     SIM,    SIM,    -1,     tag,    SIG_INIT_VALUE,0
```

Example:
The following example shows events, which identify a runnable/process as operating system specific. So apiSetEvent and Sched_ISR will be handled for OS Overhead calculation.

```
0,      apiSetEvent,    -1,     SIM,    SIM,    -1,     tag,    OSOverhead_Runnable
0,      Sched_ISR,      -1,     SIM,    SIM,    -1,     tag,    OSOverhead_Process
```

2.3.1. SYSTEM-EVENTS

The System can be every device which is responsible for creating the BTF trace (e.g. TA Simulator, hardware).

TABLE 24: COLUMNS FOR SYSTEM ENTITY.

Column	Entries
<Source>	SIM
<Event>	start, stop

TABLE 25: EVENTS FOR SYSTEM ENTITY.

Events	Description	Source
START	System starts tracing; first event in trace except there are initialization events (see 2.3.4)	SIM
STOP	System stops tracing; last event in trace	SIM

Example:

```
0,      Ecu_1,          -1,     SIM,    SIM,            -1,     tag,    ECU_INIT
0,      Processor_1,    -1,     SIM,    SIM,            -1,     tag,    PROCESSOR_INIT
0,      Core_1,         -1,     SIM,    SIM,            -1,     tag,    CORE_INIT
0,      SIM,            -1,     SYS,    SYSTEM,          0,     start
0,      Sti_Task_1,      0,     T,      Task_1,          0,     activate
100,    Core_1,          0,     T,      Task_1,          0,     start
170,    Core_1,          0,     T,      Task_1,          0,     terminate
1000,   SIM,            -1,     SYS,    SYSTEM,          0,     stop
```

3. REFERENCES

a. OSEK Specification 2.2.3. (2005). Retrieved from http://portal.osek-vdx.org/files/pdf/specs/os223.pdf