

# iSYSTEM EB tresos Safety OS 2.x Thread: Profiling Application Note



This document and all documents accompanying it are copyrighted by iSYSTEM and all rights are reserved. Duplication of these documents is allowed for personal use. For every other case, written consent from iSYSTEM is required.

Copyright © iSYSTEM, AG.  
All rights reserved.  
All trademarks are property of their respective owners.

iSYSTEM is an ISO 9001 certified company

## Table of Contents

1	Introduction .....	2
1.1	OS Threads .....	2
2	Timing Analysis Concepts .....	7
2.1	Overview .....	7
2.2	OS Running Task Profiling without Code Instrumentation .....	9
2.3	OS Thread-State Profiling by means of Code Instrumentation.....	10
2.4	OS Thread-State Profiling without Code Instrumentation.....	11
3	Running Thread/Task Profiling .....	12
3.1	Operating System Configuration.....	12
3.2	iSYSTEM Profiler XML.....	12
3.3	Analyzer Configuration .....	13
3.4	Profiler Display .....	14
3.5	Extraction of current Task Object in a running Thread.....	15
4	Thread-State Profiling by means of Code Instrumentation.....	17
4.1	Overview .....	17
4.2	Required Code Instrumentation .....	17
4.3	Operating System Configuration.....	19
4.4	iSYSTEM Profiler XML.....	20
4.5	Analyzer Configuration .....	21
4.6	Profiler Display .....	22
5	Thread-State Profiling without Code Instrumentation.....	23
5.1	Overview .....	23
5.2	Thread Control/Status Structures .....	23
5.3	Operating System Configuration.....	25
5.4	iSYSTEM Profiler XML.....	26
5.5	Analyzer Configuration .....	27
5.6	Profiler Display .....	28
5.7	Hardware Trace Configuration Options for various Processor Architectures .....	29
6	Inspectors .....	31
6.1	Task Metric Analysis.....	31
7	BTF Export.....	34
8	Technical Support .....	36
8.1	Online Resources .....	36
8.2	Contact.....	36

# 1 Introduction

This application note describes three approaches for OS scheduling analysis on EB tresos Safety OS. EB tresos Safety OS is a micro-kernel based OS and thus does not follow the approaches known from classic AUTOSAR OS (i.e. OSEK) implementations.

Tasks and ISR2 objects are all managed by EB tresos Safety OS as threads. The OS kernel operates in Supervisor mode, whereas the OS threads may run in (lower privilege) User mode or also Supervisor mode of the CPU.

The memory protection is based on OS objects and OS applications. Each OS object or OS application has its own memory region, protected by means of the hardware memory protection unit (MPU) of the processor. In addition, the kernel itself has its own protected memory region. For more information about EB tresos Safety OS, please refer to the product documentation provided by Elektrobit.

## 1.1 OS Threads

A thread comprises an execution context for various OS objects. The OS manages several types of threads such as kernel threads, task threads or ISR threads. A thread also represents a schedulable entity managed by the OS scheduler. Task threads represent user-defined task, i.e. resemble the tasks of OSEK-compliant AUTOSAR OSes.

Multiple OS objects may execute within the same thread context, e.g. multiple user tasks may share the same task thread. However, typically a one-to-one mapping of user tasks to task thread is applied.

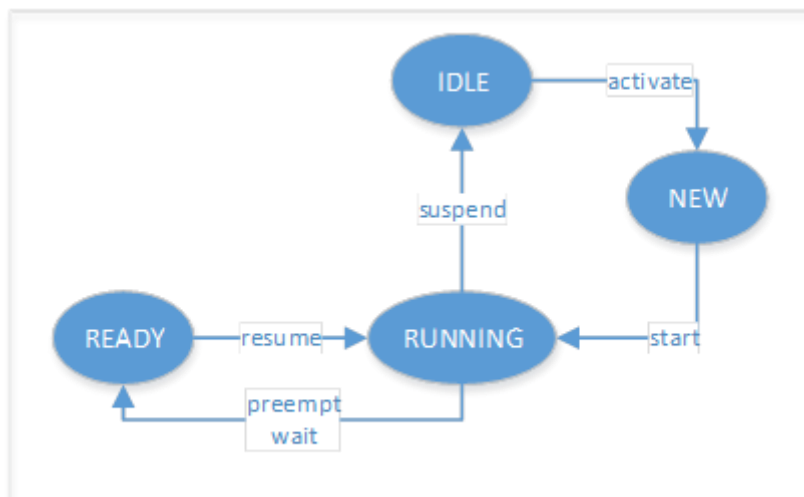


Figure 1: EB tresos Safety OS Thread-State Model

EB tresos Safety OS defines the following thread state enumeration type:

```

\include\private\Mk_thread.h:

enum mk_threadstate_e
{
    MK_THS_IDLE      = 0,
    MK_THS_READY     = 1,
    MK_THS_RUNNING   = 2,
    MK_THS_NEW       = 3,
};
  
```

Listing 1: Thread-State Enumeration Type

EB tresos Safety OS manages the following objects which can execute within an OS thread:

```
\include\public\Mk_public_types.h:

enum mk_objecttype_e
{
    MK_OBJTYPE_KERNEL,          /* Objects belonging to the microkernel */
    MK_OBJTYPE_TASK,            /* Task objects belonging to the user */
    MK_OBJTYPE_ISR,             /* ISR objects belonging to the user */
    MK_OBJTYPE_QMOS,            /* Objects belonging to the QM-OS */
    MK_OBJTYPE_QMOSISR,         /* ISR objects belonging to the QM-OS */
    MK_OBJTYPE_SHUTDOWNHOOK,    /* A shutdown-hook */
    MK_OBJTYPE_ERRORHOOK,       /* An error-hook */
    MK_OBJTYPE_PROTECTIONHOOK,  /* The protection-hook */
    MK_OBJTYPE_TRUSTEDFUNCTION, /* A trusted function */
    MK_OBJTYPE_UNKNOWN          /* Must be last */
};
```

Listing 2: Thread Type Enumeration Type

In the sample shown below, the OS maintains an array of `mk_thread_t` objects. Each element contains status information for each task thread, i.e. threads that are used to run user tasks. The sample below implements six task threads on core 0 (**Error! Reference source not found.**) and one of core 1 (**Error! Reference source not found.**).

Name	Value	Type
▣ MK_c0_taskThreads	((Ptr(0x70001A84),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294))	mk_thread_t [6]
▣ [0]	(Ptr(0x70001A84),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294))	mk_thread_t
▣ [1]	(Ptr(0x70001AB4),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294))	mk_thread_t
▣ [2]	(Ptr(0x70001A74),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294))	mk_thread_t
▣ [3]	(Ptr(0x70001A94),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294))	mk_thread_t
▣ regs	Ptr(0x70001A94)	mk_threadregisters_t *
▣ name	Ptr(0x8000C294)	char *
▣ *(name),s	"Task_St1"	char [256]
▣ next	Ptr(0x00000000) = NULL	mk_culprit_t
▣ parentThread	Ptr(0x00000000) = NULL	mk_culprit_t
▣ parentCookie	0x00000000	unsigned long
▣ xcoreReply	(0x00000000,0x00000000)	mk_statusandvalue_t
▣ accounting	(Ptr(0x8000C458),0xFFFFFFFF)	mk_accounting_t
▣ state	MK_THS_IDLE	mk_threadstate_t
▣ queueingPriority	0x00000003	long
▣ runningPriority	0x00000003	long
▣ currentPriority	0x00000000	long
▣ lastLockTaken	Ptr(0x00000000) = NULL	mk_lock_t *
▣ jobQueue	Ptr(0x00000000) = NULL	mk_jobqueue_t *
▣ eventStatus	Ptr(0x00000000) = NULL	mk_eventstatus_t *
▣ memoryPartition	0x00000005	long
▣ currentObject	0x00000005	long
▣ objectType	MK_OBJTYPE_TASK	mk_objecttype_t
▣ applicationId	0x00000001	long
▣ parentCore	0xFFFFFFFF	long
▣ [4]	(Ptr(0x70001AA4),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294))	mk_thread_t
▣ [5]	(Ptr(0x70001A64) = Ptr(MK_c0_taskThreads[0]),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294),Ptr(0x8000C294))	mk_thread_t

Figure 2: Core 0 Task Thread Array

Name	Value	Type
▣ MK_c1_taskThreads	((Ptr(0x6000161C) = Ptr(MK_c1_...))	mk_thread_t [1]
▣ [0]	(Ptr(0x6000161C) = Ptr(MK_c1_...))	mk_thread_t
▣ regs	Ptr(0x6000161C) = Ptr(MK_c1_...)	mk_threadregisters_t *
▣ name	Ptr(0x8000C26C)	char *
* (name), s	"Cyclic2C2"	char [256]
▣ next	Ptr(0x00000000) = NULL	mk_culprit_t
▣ parentThread	Ptr(0x00000000) = NULL	mk_culprit_t
▣ parentCookie	0x00000000	unsigned long
▣ xcoreReply	(0x00000000, 0x00000000)	mk_statusandvalue_t
▣ accounting	(Ptr(0x8000C318), 0xFFFFFFFF)	mk_accounting_t
▣ state	MK_THS_IDLE	mk_threadstate_t
▣ queueingPriority	0x00000001	long
▣ runningPriority	0x00000001	long
▣ currentPriority	0x00000000	long
▣ lastLockTaken	Ptr(0x00000000) = NULL	mk_lock_t *
▣ jobQueue	Ptr(0x00000000) = NULL	mk_jobqueue_t *
▣ eventStatus	Ptr(0x00000000) = NULL	mk_eventstatus_t *
▣ memoryPartition	0x00000002	long
▣ currentObject	0x00000001	long
▣ objectType	MK_OBJTYPE_TASK	mk_objecttype_t
▣ applicationId	0x00000002	long
▣ parentCore	0xFFFFFFFF	long

Figure 3: Core 1 Task Thread Array

The profiler timelines in **Error! Reference source not found.** show a trace of the currently running thread on two cores. Both timelines are based on the same trace recording, but display different time spans. The upper timeline is zoomed in at the location of the blue and yellow markers of the lower timeline.

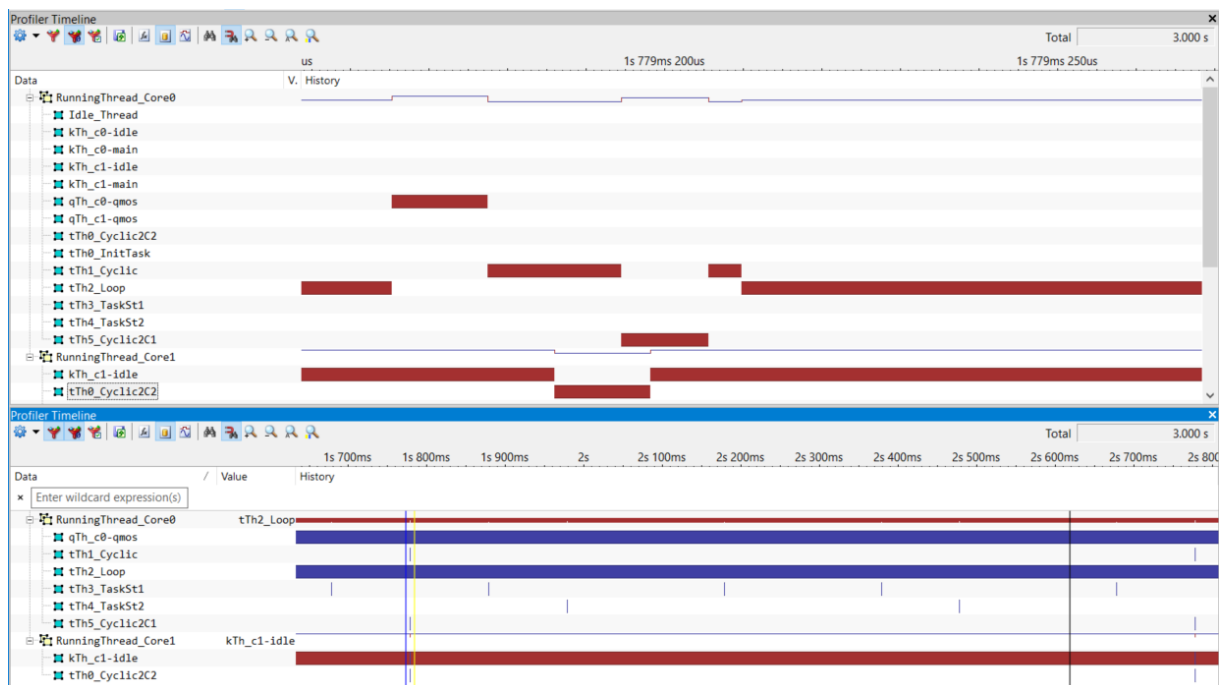


Figure 4: Sample Profiler Timeline (Dual-Core Running Thread)

A complete list of all user tasks, covering all cores, can be found in the array `MK_taskCfgTable[]` (see **Error! Reference source not found.**). The element `MK_taskCfgTable[n].thread` contains a pointer to the corresponding task thread `MK_cX_taskThreads[m]` element.

For instance, the element `MK_taskCfgTable[4].thread` contains a pointer to the corresponding task thread `MK_c0_taskThreads[2]` element (see **Error! Reference source not found.**).

Name	Value	Type
▣ MK_taskCfgTable	((Ptr(0x70001A64) = Ptr(MK	mk_taskcfg_t [7]
▣ [0]	((Ptr(0x70001A64) = Ptr(MK	mk_taskcfg_t
▣ [1]	((Ptr(0x6000161C) = Ptr(MK	mk_taskcfg_t
▣ [2]	((Ptr(0x70001A84), Ptr(0x800	mk_taskcfg_t
▣ [3]	((Ptr(0x70001AB4), Ptr(0x800	mk_taskcfg_t
▣ [4]	((Ptr(0x70001A74), Ptr(0x800	mk_taskcfg_t
▣ threadCfg	(Ptr(0x70001A74), Ptr(0x8000	mk_threadcfg_t
▣ dynamic	Ptr(0x70001A58)	mk_task_t *
▣ thread	Ptr(0x70001B6C)	mk_culprit_t
▣ *(thread)	(Ptr(0x70001A74), Ptr(0x8000	mk_thread_t
▣ regs	Ptr(0x70001A74)	mk_threadregisters_t *
▣ name	Ptr(0x8000C28C)	char *
▣ *(name), s	"Loop"	char [256]
▣ next	Ptr(0x700016F8) = Ptr(MK_c0	mk_culprit_t
▣ parentThread	Ptr(0x00000000) = NULL	mk_culprit_t
▣ parentCookie	0x00000000	unsigned long
▣ xcoreReply	(0x00000000, 0x00000000)	mk_statusandvalue_t
▣ accounting	(Ptr(0x8000C408), 0xFFFFFFFF	mk_accounting_t
▣ state	MK_THS_RUNNING	mk_threadstate_t
▣ queueingPriority	0x00000001	long
▣ runningPriority	0x00000001	long
▣ currentPriority	0x00000001	long
▣ lastLockTaken	Ptr(0x00000000) = NULL	mk_lock_t *
▣ jobQueue	Ptr(0x00000000) = NULL	mk_jobqueue_t *
▣ eventStatus	Ptr(0x00000000) = NULL	mk_eventstatus_t *
▣ memoryPartition	0x00000004	long
▣ currentObject	0x00000004	long
▣ objectType	MK_OBJECTTYPE_TASK	mk_objecttype_t
▣ applicationId	0x00000000	long
▣ parentCore	0xFFFFFFFF	long
▣ stack	Ptr(0x70000C08)	unsigned long *
▣ maxActivations	0x00000001	long
▣ eventStatus	Ptr(0x00000000) = NULL	mk_eventstatus_t *
▣ [5]	((Ptr(0x70001A94), Ptr(0x800	mk_taskcfg_t
▣ [6]	((Ptr(0x70001AA4), Ptr(0x800	mk_taskcfg_t

Figure 5: MK\_taskCfgTable Array

Name	Value	Type	Address
▣ MK_c0_taskThreads	((Ptr(0x70001A84), Ptr(0x8000C	mk_thread_t [6]	(Virtual) 70001AC4
▣ [0]	(Ptr(0x70001A84), Ptr(0x8000C	mk_thread_t	(Virtual) 70001AC4
▣ [1]	(Ptr(0x70001AB4), Ptr(0x8000C	mk_thread_t	(Virtual) 70001B18
▣ [2]	(Ptr(0x70001A74), Ptr(0x8000C	mk_thread_t	(Virtual) 70001B6C
▣ regs	Ptr(0x70001A74)	mk_threadregisters_t *	(Virtual) 70001B6C
▣ name	Ptr(0x8000C28C)	char *	(Virtual) 70001B70
▣ *(name), s	"Loop"	char [256]	(Virtual) 8000C28C
▣ next	Ptr(0x700016F8) = Ptr(MK_c0	mk_culprit_t	(Virtual) 70001B74

Figure 6: MK\_c0\_taskThreads Array



However, as mentioned earlier, it is possible that multiple user tasks share the execution context of the same task thread, i.e. multiple `MK_taskCfgTable[]` elements refer to the same `MK_cX_taskThreads[]` element.

The profiler timeline below shows a sample configuration where the user tasks *SchMComTask\_5ms* and *SchMComTask\_10ms* execute within the task thread 4.

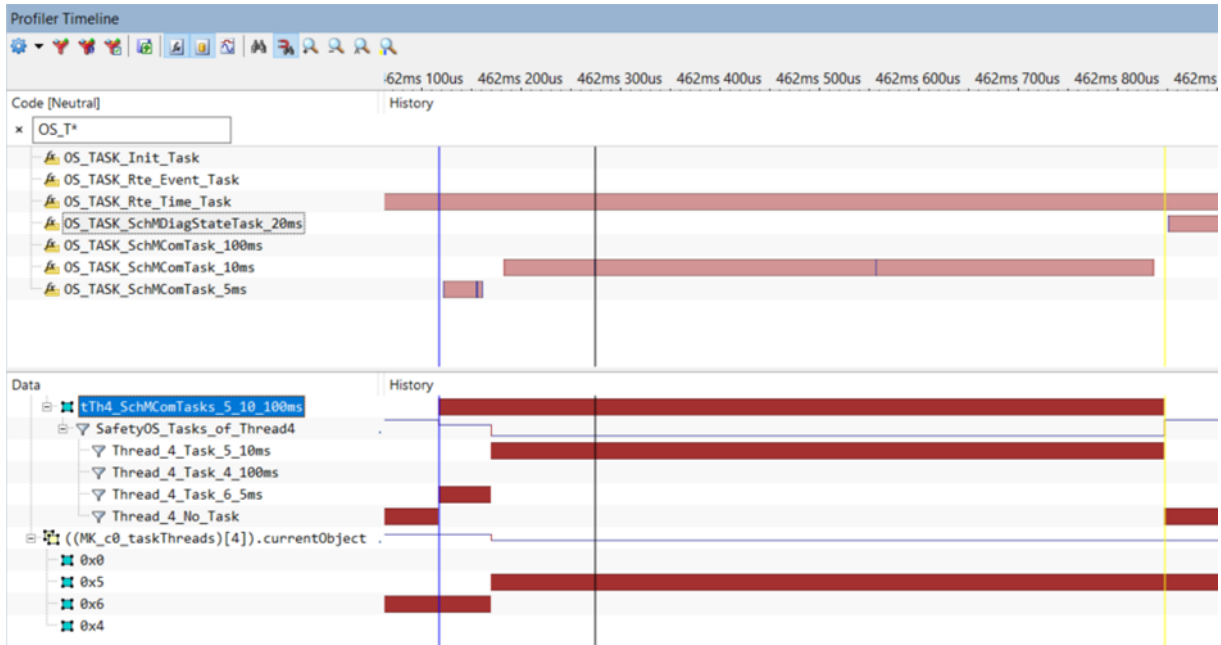


Figure 7: Multiple User Tasks executing within the same Task Thread context

## 2 Timing Analysis Concepts

### 2.1 Overview

Timing analysis of EB tresos Safety OS cannot be accomplished purely by standard methods based on an ORTI file generated by EB tresos Studio.

The EB tresos Studio does in fact generate an ORTI file also for EB tresos Safety OS. A concept for running task tracing (RUNNINGTASK) is not provided, instead only a vendor-specific extension is supported for running thread tracing (vs\_RUNNINGTHREAD).

The enumeration vs\_RUNNINGTHREAD lists all objects running within threads, including all user tasks, isrs and kernel threads. However, the OS object to be used for identifying the currently running thread is not suitable for hardware-based tracing as it uses multiple memory locations referenced by a pointer.

```
os.orti file:

OS
{
  ENUM [
    "NO_THREAD" = 0x0,
    "Cyclic2C1" = "MK_taskCfgTable[0].threadCfg.name",
    "Cyclic2C2" = "MK_taskCfgTable[1].threadCfg.name",
    "InitTask" = "MK_taskCfgTable[2].threadCfg.name",
    "Cyclic" = "MK_taskCfgTable[3].threadCfg.name",
    "Loop" = "MK_taskCfgTable[4].threadCfg.name",
    "Task_St1" = "MK_taskCfgTable[5].threadCfg.name",
    "Task_St2" = "MK_taskCfgTable[6].threadCfg.name",
    "Os_Counter_STM0_T0" = "MK_isrCfgTable[0].threadCfg.name",
    "mk_boot_thread_core0" = "MK_bootThreadConfig[0]->name",
    "mk_init_thread_core0" = "MK_initThreadConfig[0]->name",
    "mk_idle_thread_core0" = "MK_idleThreadConfig[0]->name",
    "mk_shutdown_thread_core0" = "MK_shutdownThreadConfig[0]->name",
    "mk_aux1_thread_core0" = "MK_aux1Thread[0]->name",
    "mk_aux2_thread_core0" = "MK_aux2Thread[0]->name",
    "mk_qmos_thread_core0" = "MK_qmosThreadConfig[0]->name",
    "mk_protection_hook_thread_core0" = "MK_protectionHookThreadConfig[0]->name",
    "mk_error_hook_thread_core0" = "MK_errorHookThreadConfig[0]->name",
    "mk_boot_thread_core1" = "MK_bootThreadConfig[1]->name",
    "mk_init_thread_core1" = "MK_initThreadConfig[1]->name",
    "mk_idle_thread_core1" = "MK_idleThreadConfig[1]->name",
    "mk_shutdown_thread_core1" = "MK_shutdownThreadConfig[1]->name",
    "mk_aux1_thread_core1" = "MK_aux1Thread[1]->name",
    "mk_aux2_thread_core1" = "MK_aux2Thread[1]->name",
    "mk_qmos_thread_core1" = "MK_qmosThreadConfig[1]->name",
    "mk_protection_hook_thread_core1" = "MK_protectionHookThreadConfig[1]->name",
    "mk_error_hook_thread_core1" = "MK_errorHookThreadConfig[1]->name"
  ] vs_RUNNINGTHREAD[], "Running thread identification";
};

OS XYZ
{
  vs_RUNNINGTHREAD = "MK_c0_coreVars.currentThread->name";
};
```

Listing 3: Sample ORTI file generated by EB tresos Studio

Each element of the vs\_RUNNINGTHREAD enumeration maps a thread name (e.g. *Task\_St1*) with an address of the MK\_taskCfgTable array element, containing a string of the thread object name (e.g. MK\_taskCfgTable[5].threadCfg.name).

In other words, the pointer MK\_c0\_coreVars.currentThread->name points to a name string which is a sub-element of a MK\_taskCfgTable[n].threadCfg element associated with a user task n. This relation is depicted in **Error! Reference source not found.**



Name	Value	Type	Address
▢ MK_c0_coreVars	(Ptr(0x70001BC0), Ptr(0x70001BC0))	mk_kernelcontrol	(Virtual)700015F0
▢ currentThread	Ptr(0x70001BC0)	mk_culprit_t	(Virtual)700015F0
▢ *(currentThread)	(Ptr(0x70001A94), Ptr(0x70001A94))	mk_thread_t	(Virtual)70001BC0
▢ regs	Ptr(0x70001A94)	mk_hwthreadregis	(Virtual)70001BC0
▢ name	Ptr(0x8000C294)	char *	(Virtual)70001BC4
▢ *(name), s	"Task_St1"	char [256]	(Virtual)8000C294
▢ next	Ptr(0x70001B6C)	mk_culprit_t	(Virtual)70001BC8

Name	Value	Type	Address
▢ MK_taskCfgTable	((Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64))	mk_taskcfg_t [7]	(Virtual)8000C2B4
▢ [0]	((Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64))	mk_taskcfg_t	(Virtual)8000C2B4
▢ [1]	((Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64))	mk_taskcfg_t	(Virtual)8000C304
▢ [2]	((Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64))	mk_taskcfg_t	(Virtual)8000C354
▢ [3]	((Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64))	mk_taskcfg_t	(Virtual)8000C3A4
▢ [4]	((Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64))	mk_taskcfg_t	(Virtual)8000C3F4
▢ [5]	((Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64), Ptr(0x70001A64) = Ptr(0x70001A64))	mk_taskcfg_t	(Virtual)8000C444
▢ threadCfg	(Ptr(0x70001A94), Ptr(0x70001A94))	mk_threadcfg_t	(Virtual)8000C444
▢ regs	Ptr(0x70001A94)	mk_threadregiste	(Virtual)8000C444
▢ name	Ptr(0x8000C294)	char *	(Virtual)8000C448
▢ *(name), s	"Task_St1"	char [256]	(Virtual)8000C294
▢ coreIndex	0x00000000	long	(Virtual)8000C44C

Name	Value	Type	Address
▢ MK_c0_taskThreads	((Ptr(0x70001A84), Ptr(0x70001A84), Ptr(0x70001A84), Ptr(0x70001A84), Ptr(0x70001A84), Ptr(0x70001A84))	mk_thread_t [6]	(Virtual)70001AC4
▢ [0]	(Ptr(0x70001A84), Ptr(0x70001A84))	mk_thread_t	(Virtual)70001AC4
▢ [1]	(Ptr(0x70001A84), Ptr(0x70001A84))	mk_thread_t	(Virtual)70001B18
▢ [2]	(Ptr(0x70001A84), Ptr(0x70001A84))	mk_thread_t	(Virtual)70001B6C
▢ [3]	(Ptr(0x70001A84), Ptr(0x70001A84))	mk_thread_t	(Virtual)70001BC0
▢ regs	Ptr(0x70001A94)	mk_threadregiste	(Virtual)70001BC0
▢ name	Ptr(0x8000C294)	char *	(Virtual)70001BC4
▢ *(name), s	"Task_St1"	char [256]	(Virtual)8000C294
▢ next	Ptr(0x70001B6C)	mk_culprit_t	(Virtual)70001BC8

Figure 8: Sample MK\_c0\_coreVars.currentThread-&gt;name, pointing to MK\_taskCfgTable[5]-&gt;name (Task\_St1)

## 2.2 OS Running Task Profiling without Code Instrumentation

OS running task profiling is based on tracing a global OS data object which contains status information about the currently running thread. As mentioned earlier these threads may either be kernel threads, user threads or ISR threads.

The core-specific global variable, e.g. `MK_c0_coreVars.currentThread`, contains a pointer to an element of a `mk_thread_t` type array.

Data Object pointed to by <code>MK_cX_coreVars.currentThread</code>	Description
<code>MK_cX_taskThreads[]</code>	user task thread element of core X
<code>MK_cX_isrThreads[]</code>	user ISR2 thread element of core X
<code>MK_cX_auxN_Threads[]</code>	kernel thread elements (aux1/aux2) of core X
<code>MK_cX_idleThread</code>	Idle thread of core X

In case multiple user tasks are mapped into a task thread, winIDEA Analyzer Inspectors can be used to derive the currently running user task. However, this requires that, in addition to the currently running thread, also the currently active object of the corresponding thread is traced (`MK_cX_taskThreads[n].currentObject`).

Name	Value	Type
<code>MK_c0_taskThreads</code>	<code>((Ptr(0x70001A84), Ptr(0x8000C...))</code>	<code>mk_thread_t [6]</code>
[0]	<code>(Ptr(0x70001A84), Ptr(0x8000C...))</code>	<code>mk_thread_t</code>
[1]	<code>(Ptr(0x70001AB4), Ptr(0x8000C...))</code>	<code>mk_thread_t</code>
[2]	<code>(Ptr(0x70001A74), Ptr(0x8000C...))</code>	<code>mk_thread_t</code>
[3]	<code>(Ptr(0x70001A94), Ptr(0x8000C...))</code>	<code>mk_thread_t</code>
regs	<code>Ptr(0x70001A94)</code>	<code>mk_threadregisters_t *</code>
name	<code>Ptr(0x8000C294)</code>	<code>char *</code>
*(name), s	<code>"Task_St1"</code>	<code>char [256]</code>
next	<code>Ptr(0x00000000) = NULL</code>	<code>mk_culprit_t</code>
parentThread	<code>Ptr(0x00000000) = NULL</code>	<code>mk_culprit_t</code>
parentCookie	<code>0x00000000</code>	<code>unsigned long</code>
xcoreReply	<code>(0x00000000, 0x00000000)</code>	<code>mk_statusandvalue_t</code>
accounting	<code>(Ptr(0x8000C458), 0xFFFFFFFF)</code>	<code>mk_accounting_t</code>
state	<code>MK_THS_IDLE</code>	<code>mk_threadstate_t</code>
queueingPriority	<code>0x00000003</code>	<code>long</code>
runningPriority	<code>0x00000003</code>	<code>long</code>
currentPriority	<code>0x00000000</code>	<code>long</code>
lastLockTaken	<code>Ptr(0x00000000) = NULL</code>	<code>mk_lock_t *</code>
jobQueue	<code>Ptr(0x00000000) = NULL</code>	<code>mk_jobqueue_t *</code>
eventStatus	<code>Ptr(0x00000000) = NULL</code>	<code>mk_eventstatus_t *</code>
memoryPartition	<code>0x00000005</code>	<code>long</code>
currentObject	<code>0x00000005</code>	<code>long</code>
objectType	<code>MK_OBJTYPE_TASK</code>	<code>mk_objecttype_t</code>
applicationId	<code>0x00000001</code>	<code>long</code>
parentCore	<code>0xFFFFFFFF</code>	<code>long</code>
[4]	<code>(Ptr(0x70001AA4), Ptr(0x8000C...))</code>	<code>mk_thread_t</code>
[5]	<code>(Ptr(0x70001A64) = Ptr(MK_c0...</code>	<code>mk_thread_t</code>

Figure 9: currentObject Element of the Core 0 Task Thread 3 (`MK_c0_taskThread[3]`)

## 2.3 OS Thread-State Profiling by means of Code Instrumentation

The OS supports thread-state tracing by means of the macro `MK_TRACE_STATE_THREAD`. This macro has already been placed into the source code at all the relevant code sections, but is disabled (i.e. defined as empty) per default in the source code files (lib\_src) of the MicroOS plugin provided by Elektrobit.

The concept is to overwrite the default (empty) macro with iSYSTEM tool specific instrumentation code that collects all relevant information and copies this data into a global data object which is monitored by means to hardware trace.

The concepts for inserting the trace instrumentation code into the code generation and build process to EB tresos Studio and the configuration of the iSYSTEM trace analyzer are described in section 4 Thread-State Profiling.

**Error! Reference source not found.** shows a sample iSYSTEM profiler timeline of threads distributed over two cores, including the detailed state (IDLE/NEW/READY/RUNNING) of each thread.

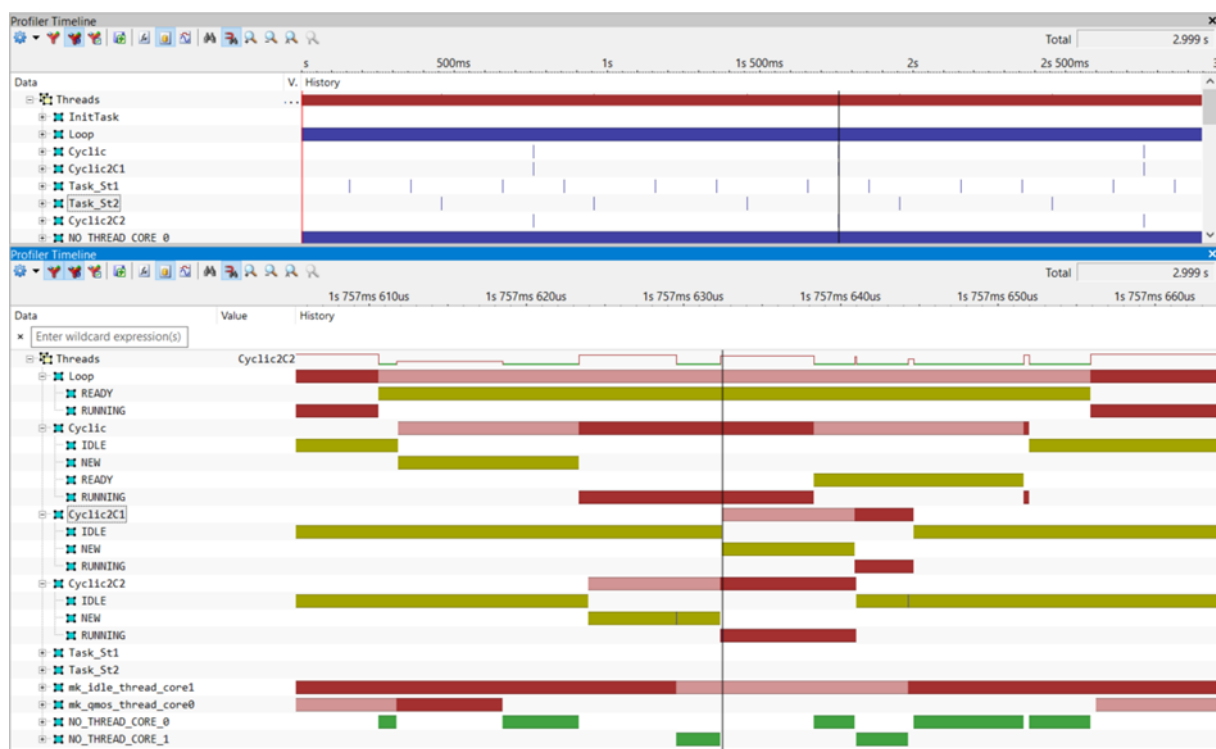


Figure 10: Thread-State Profiling Timeline by means of `MK_TRACE THREAD_STATE` Instrumentation

## 2.4 OS Thread-State Profiling without Code Instrumentation

In case code instrumentation is not an option, the current state of each thread can still be traced. Alternately, the current state of each thread can be monitored by tracing a dedicated variable within a thread control and status structure. These structures are grouped into arrays. EB tresos Safety OS maintains one array per core and per thread type. Typically, EB tresos Safety OS uses 6 thread types, i.e. there are 6 arrays per core. The most appropriate trace configuration for tracing the thread state variables depends on the capabilities of the on-chip trace logic of the processor, i.e. available trace interface bandwidth and data trace filtering (qualifier) features.

Figure 11 shows a sample iSYSTEM profiler timeline of threads derived from a data trace of the various thread state variables.

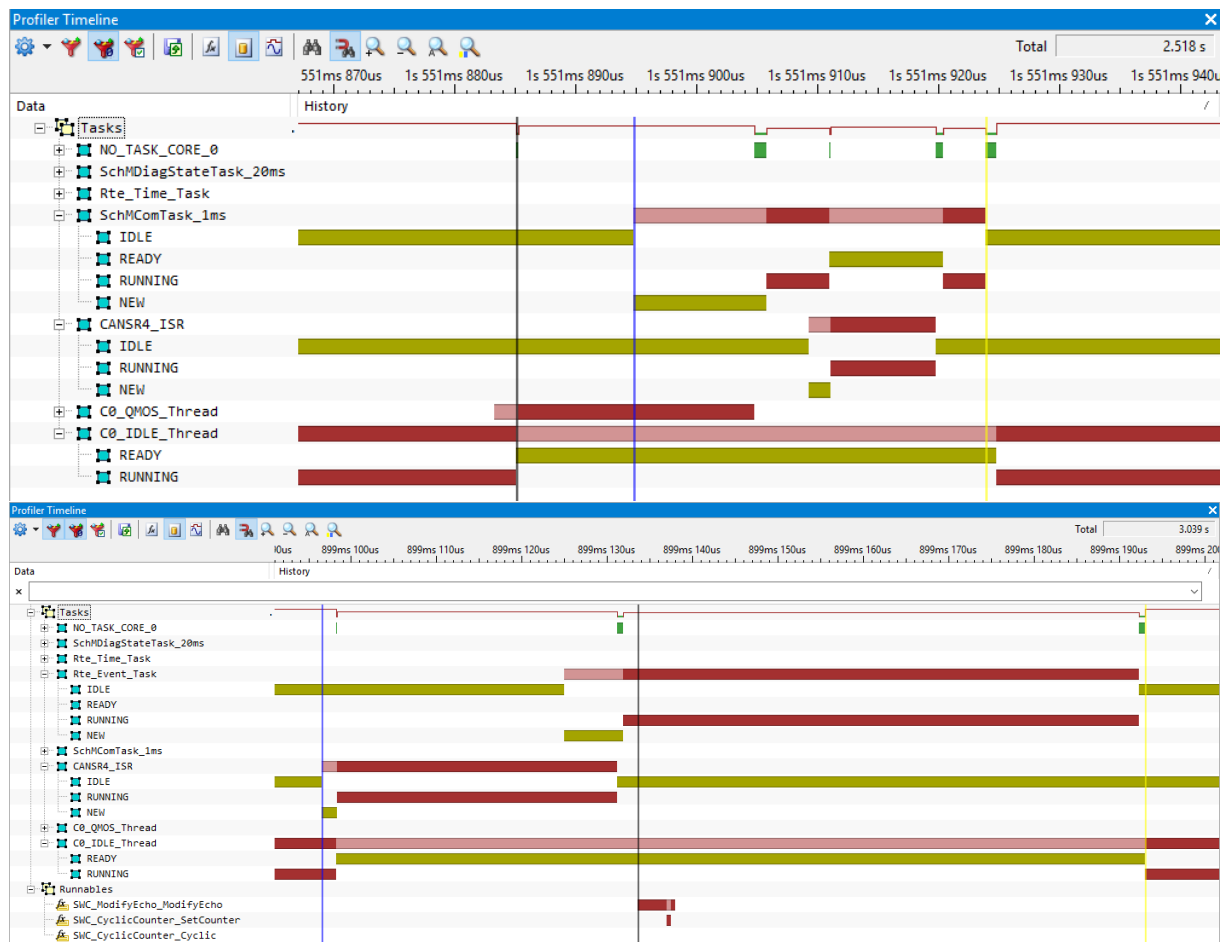


Figure 11: Thread-State Profiling Timeline by means of Data Trace of Thread State Variable

### 3 Running Thread/Task Profiling

#### 3.1 Operating System Configuration

Making winIDEA aware of the target operating system (OS) can be accomplished by reading in an OS description file. In case of an OSEK-compliant AUTOSAR OS, this OS description file is the so-called ORTI file, generated by the OS generator of the AUTOSAR tool (e.g. EB tresos Studio). However, as mentioned earlier, for running thread tracing of EB tresos Safety OS, this ORTI file based approach is not applicable. Instead, an iSYSTEM-proprietary XML needs to be used to describe the target OS.

An OS description file can be imported into winIDEA via the menu: “Debug – Operating System...” as shown in Figure 12: Selection of the iSYSTEM Profiler XML File

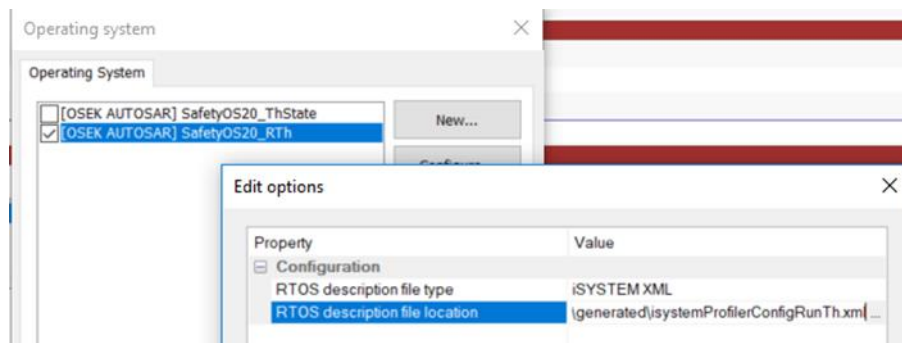


Figure 12: Selection of the iSYSTEM Profiler XML File

#### 3.2 iSYSTEM Profiler XML

The figure below shows a sample profiler XML file.

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2  <OperatingSystem>
3    <Name>SafetyOsDemo</Name>
4    <NumCores>2</NumCores>
5    <Types>
6      <TypeEnum><Name>TypeRunningThreadSymbol</Name>
7        <Enum><Name>Idle_Thread</Name> <Value>0x00</Value></Enum>
8        <Enum><Name>tTh0_InitTask</Name> <Value>&MK_c0_taskThreads[0]</Value></Enum>
9        <Enum><Name>tTh1_Cyclic</Name> <Value>&MK_c0_taskThreads[1]</Value></Enum>
10       <Enum><Name>tTh2_Loop</Name> <Value>&MK_c0_taskThreads[2]</Value></Enum>
11       <Enum><Name>tTh3_TaskSt1</Name> <Value>&MK_c0_taskThreads[3]</Value></Enum>
12       <Enum><Name>tTh4_TaskSt2</Name> <Value>&MK_c0_taskThreads[4]</Value></Enum>
13       <Enum><Name>tTh5_Cyclic2C1</Name> <Value>&MK_c0_taskThreads[5]</Value></Enum>
14       <Enum><Name>qTh_c0-qmos</Name> <Value>&MK_c0_aux1Thread</Value></Enum>
15       <Enum><Name>kTh_c0-main</Name> <Value>&MK_c0_aux2Thread</Value></Enum>
16       <Enum><Name>kTh_c0-idle</Name> <Value>&MK_c0_idleThread</Value></Enum>
17       <Enum><Name>tTh0_Cyclic2C2</Name> <Value>&MK_c1_taskThreads[0]</Value></Enum>
18       <Enum><Name>qTh_c1-qmos</Name> <Value>&MK_c1_aux1Thread</Value></Enum>
19       <Enum><Name>kTh_c1-main</Name> <Value>&MK_c1_aux2Thread</Value></Enum>
20       <Enum><Name>kTh_c1-idle</Name> <Value>&MK_c1_idleThread</Value></Enum>
21     </TypeEnum>
22   </Types>
23   <Profiler>
24     <Object>
25       <Definition>RunningThread_Core0</Definition>
26       <Expression>(MK_c0_coreVars).currentThread</Expression>
27       <Type>TypeRunningThreadSymbol</Type>
28       <DefaultValue>Idle_Thread</DefaultValue>
29     </Object>
30     <Object>
31       <Definition>RunningThread_Core1</Definition>
32       <Expression>(MK_c1_coreVars).currentThread</Expression>
33       <Type>TypeRunningThreadSymbol</Type>
34       <DefaultValue>Idle_Thread</DefaultValue>
35     </Object>
36   </Profiler>
37 </OperatingSystem>

```

Figure 13: Sample iSYSTEM Profiler XML File

The XML file consists of two major sections. The `Types` section contains an enumeration type `TypeRunningThreadSymbol` that maps the thread names (displayed within the profiler) to data values (addresses of data objects in the ELF file). The data values represent the data obtained by the profiler by tracing the data objects as described in the »Profiler« section of the XML file.

The `Profiler` section describes that the global data objects, used by the OS for signaling the currently running thread, have the symbol name `MK_c0_coreVars.currentThread` for core 0, or »`MK_c1_coreVars.currentThread`« for core 1, respectively. The content of the XML can be derived from the OS objects `MK_cX_taskThreads`, `MK_cX_isrThreads` and also `MK_cX_auxThread` and `MK_cX_idleThread`).

### 3.3 Analyzer Configuration

The OS profiler of the winIDEA analyzer can be enabled by selecting »OS objects« in the »Profiler« tab of the analyzer configuration dialog. The »RTOS Profiler Options« dialog (opened via »OS Setup...«) allows enabling/disabling of individual OS objects in the analysis.

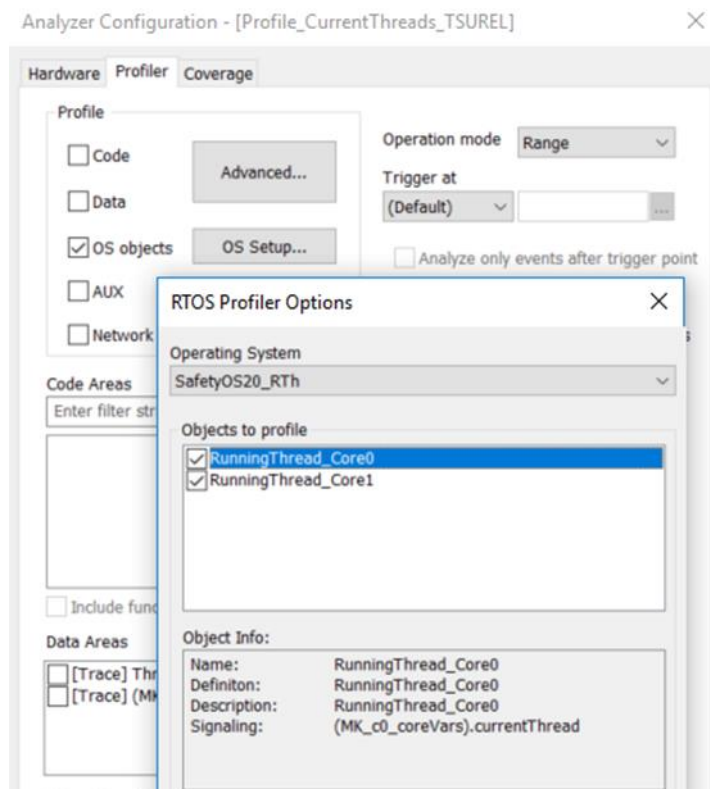


Figure 14: OS Running Thread Configuration in the iSYSTEM Analyzer



### 3.4 Profiler Display

The profiler timelines in Figure 15 show a trace of the currently running thread on two cores. Both timelines are based on the same trace recording, but display different time spans. The upper timeline is zoomed in at the location of the blue and yellow markers of the lower timeline.

A dark-red profiler state bar indicates that the corresponding core is currently executing this thread. A dark-blue bar indicates that there are multiple state transitions and the user must zoom in to see further details of each state transition.

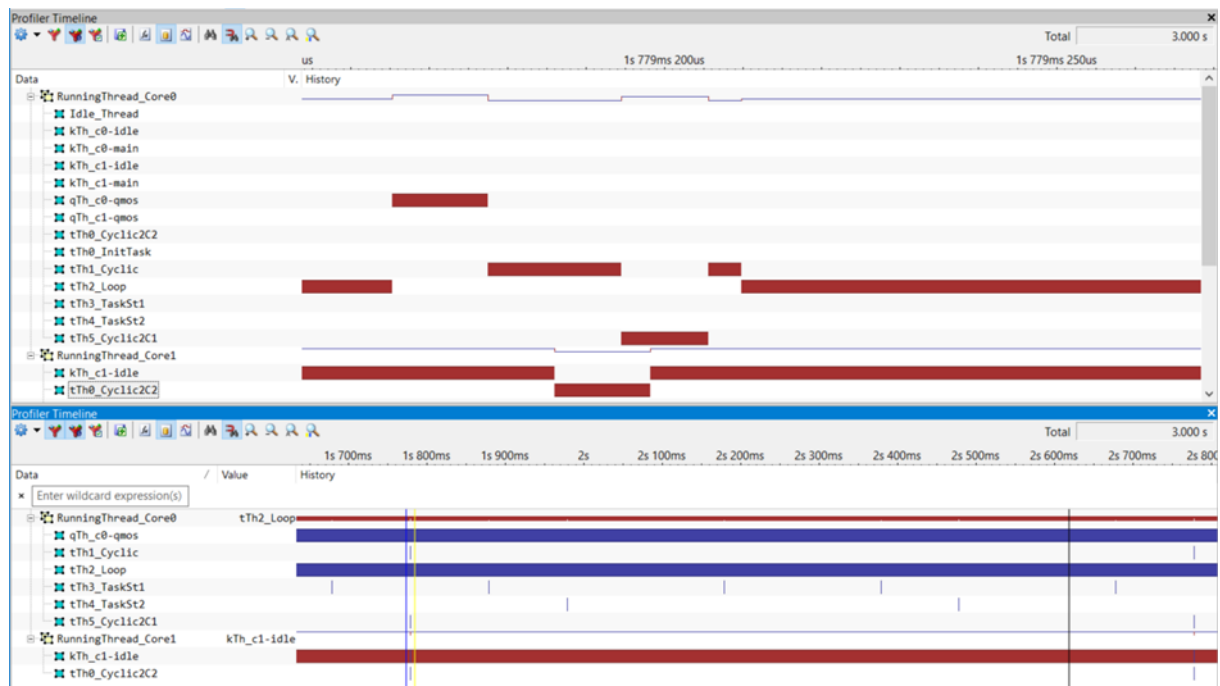


Figure 15: Sample Running Thread Profiler Timeline

### 3.5 Extraction of current Task Object in a running Thread

However, as mentioned earlier, it is possible that multiple user tasks share the execution context of one task thread, i.e. multiple `MK_taskCfgTable[]` elements refer to the same `MK_cX_taskThreads[]` element.

Figure 16 and Figure 17 **Error! Reference source not found.** show a sample thread configuration. Threads 0 to 3 are used to run only one dedicated user task, whereas thread 4 is used to execute the tasks `SchMComTask_100ms`, `SchMComTask_10ms` or `SchMComTask_5ms`, i.e. the element `MK_c0_taskThread[4].currentObject` can either be 4, 5 or 6.

The value `x` indicated by `MK_c0_taskThread[n].currentObject` relates to an element `x` in the `MK_taskCfgTable[]`.

Name	Value
<code>((MK_taskCfgTable)[4]).threadCfg.name</code>	<code>Ptr(0x40129E82)</code>
<code>*(((MK_taskCfgTable)[4]).threadCfg.name)</code>	<code>"SchMComTask_100ms"</code>
<code>((MK_taskCfgTable)[5]).threadCfg.name</code>	<code>Ptr(0x40129E94)</code>
<code>*(((MK_taskCfgTable)[5]).threadCfg.name)</code>	<code>"SchMComTask_10ms"</code>
<code>((MK_taskCfgTable)[6]).threadCfg.name</code>	<code>Ptr(0x40129EBC)</code>
<code>*(((MK_taskCfgTable)[6]).threadCfg.name)</code>	<code>"SchMComTask_5ms"</code>

Figure 16: Sample User Task Configuration

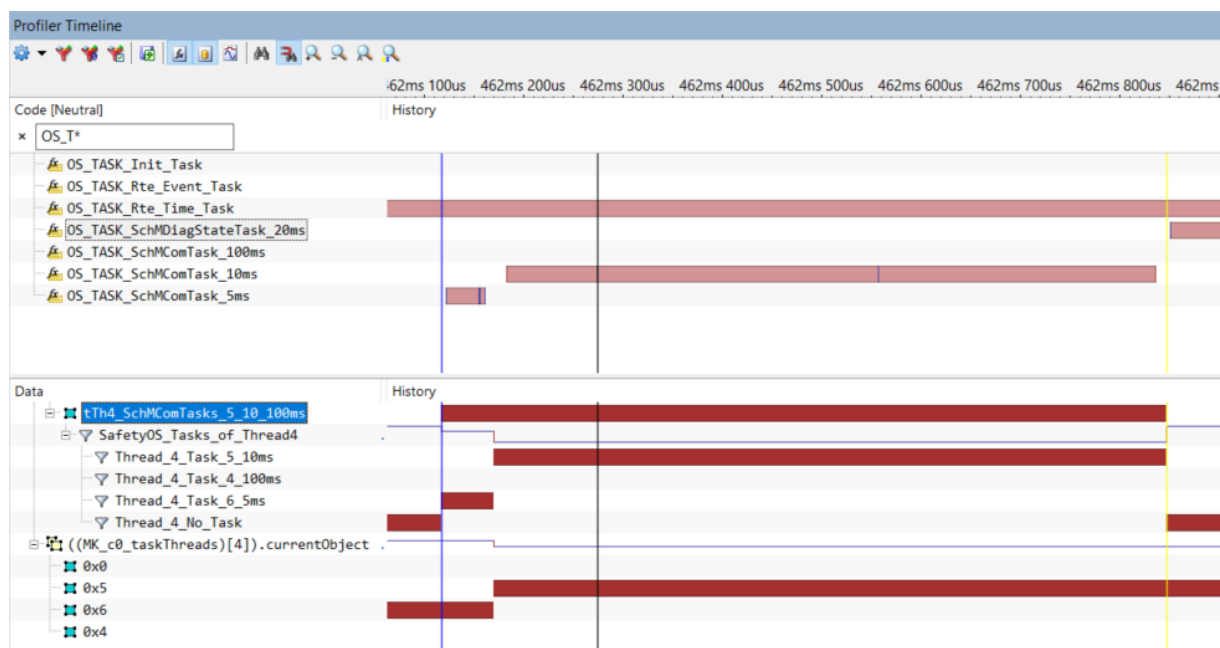


Figure 17: Deriving user tasks from a Task Thread by means of Inspectors

The iSYSTEM profiler features so-called profiler “Inspectors”. These inspectors basically allow a user-defined post-analysis of the profiler timeline and can be used for sophisticated, used-specific event-chain analysis of trace recordings, presented by the profiler.

An inspector allows to create a new profiler object, derived from the analysis of already existing profiler objects.

In the given example, the inspector creates 4 new profiler objects, derived from the existing profiler objects “RunningThread” and the current object (i.e. currently running user task) of thread 4. This inspector basically implements (describes) a state machine as depicted in **Error! Reference source not found..**

Each state of the state-machine represents an Inspector Object on the Profiler Timeline e.g. the state “Task 5” represents the Inspector Object `Thread_4_Task_5_10ms` as shown in Figure 18.

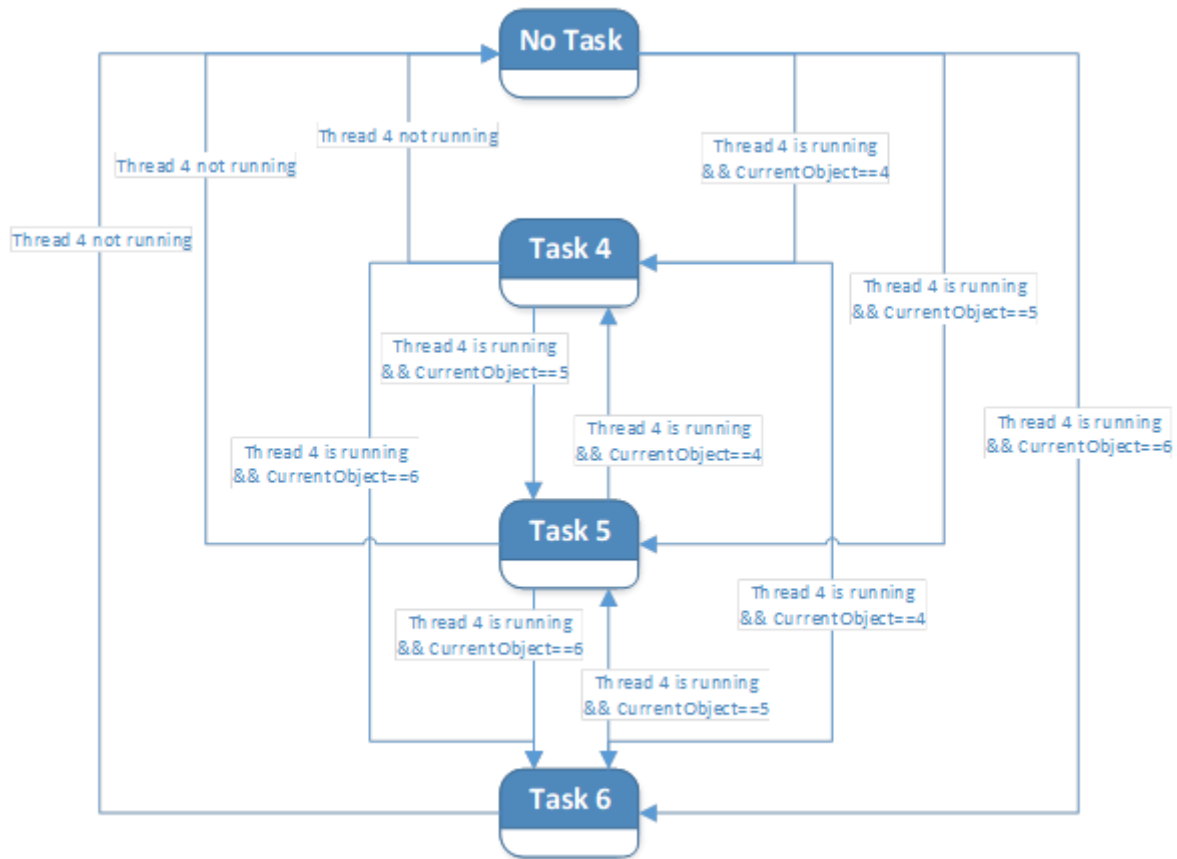


Figure 18: Inspector state-machine

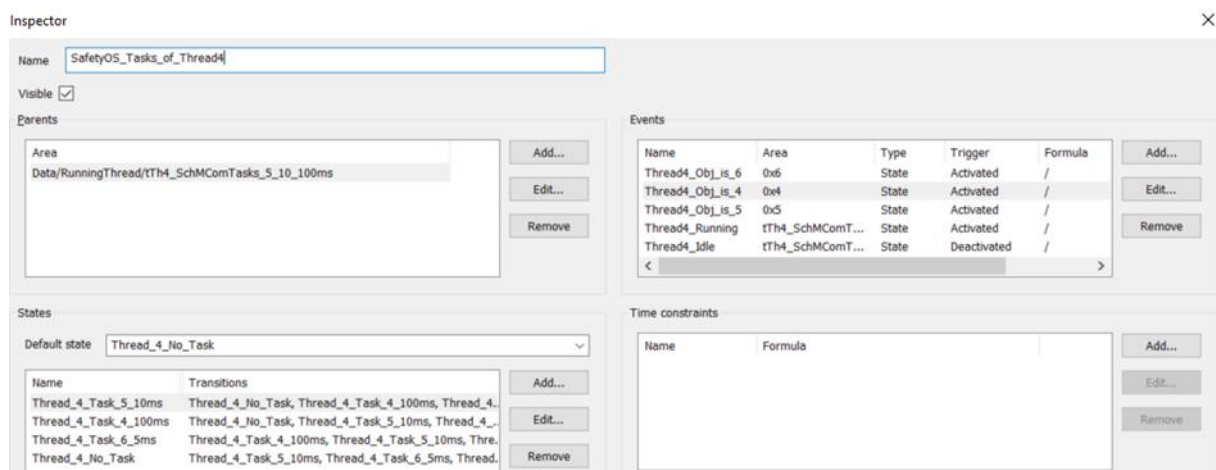


Figure 19: Sample Inspector Configuration in the Inspector GUI of the winIDEA Analyzer

## 4 Thread-State Profiling by means of Code Instrumentation

### 4.1 Overview

The OS supports thread state tracing by means of the macro `MK_TRACE_STATE_THREAD`. This macro has already been placed into the source code at all the relevant code sections, but is disabled (i.e. defined as empty) per default in the source code files (`lib_src`) of the MicroOS plugin provided by Elektrobit.

### 4.2 Required Code Instrumentation

#### 4.2.1 Definition of the Thread State Tracing Hook

The default definition of the macro can be found in the `MK_kconfig.h` header file included by all relevant kernel source files. Per default the macro is defined as empty, thus it must be replaced by an iSYSTEM trace specific definition.

In the `MK_kconfig.h` below, the default definition has been replaced by an include of another header file `isystemOsTrace.h`.

```
\include\MK_kconfig.h:

/* Use external trace tool if selected.
 *
 * !LINKSTO Microkernel.Function.MK_TRACE_STATE_THREAD,1
 * !doctype src
 */
#if MK_USE_TRACE
/* Include the header file that defines the trace tool's implementation of
MK_TRACE_STATE_THREAD.
*/
#include <isystemOsTrace.h>

/*
#define MK_TRACE_STATE_THREAD(typ,id,name,old,new) \
    MK_QmDumpThreadStateChange(typ, id, name, old, new)
*/

#else

#include <isystemOsTrace.h>
/*
#define MK_TRACE_STATE_THREAD(typ, id, name, old, new) do { } while(0)
*/
#endif
```

Listing 4: Enabling the iSYSTEM Instrumentation Code in `MK_kconfig.h`

The iSYSTEM trace specific implementation of the `MK_TRACE_STATE_THREAD` macro is listed below. The `isystemOsTrace.h` header file must be in the folder `\include\isystemOsTrace.h`.

```
/* =====
 * File: isystemOsTrace.h
 * iSYSTEM EB tresos SafetyOS Thread State Trace Instrumentation
 * MK_TRACE_STATE_THREAD macro definition
 * ===== */
#ifndef __isystemOsTrace_H
#define __isystemOsTrace_H

extern unsigned long isystem_os_trace[1];

#ifndef MK_TRACE_STATE_THREAD

#define MK_TRACE_STATE_THREAD(typ, id, name, old, new) \
do { \
    isystem_os_trace[0] = (mfc(0xFE1C)<<30) | (new<<28) | (0xFFFFF&(int)name); \
} while (0)

#endif

#endif /* if !defined( __isystemOsTrace_H ) */
```

Listing 5: `MK_TRACE_STATE_THREAD` Macro Defintion of iSYSTEM Trace

Parameter Name	Description
Typ	Thread object type (not used for thread state trace)
Id	Integer ID of the thread (not used for thread state trace)
Name	Pointer to the name string of the thread (e.g. <code>MK_taskCfgTable[5].threadCfg.name</code> ) Only the lower 24 bits of the pointer value are communicated to the trace tool. The upper bits are derived from the ELF file.
Old	Previous state of the thread, before the current state-transition. (not used for thread state trace)
New	New state of the thread, after the current state-transition.

#### 4.2.2 Definition of the global Variable `isystem_os_trace`

The global data object `isystem_os_trace` must be linked into a data memory region that is assigned to the kernel (.BSS section of the kernel). This can be achieved by added an iSYSTEM trace specific source file, containing the global variable definition, to the build process of the kernel library. As all OS kernel source files use the prefix “`MK_k_`” also the iSYSTEM source file needs to be named accordingly, e.g. `MK_k_isystem.c`. The code listing below shows the definition of the global variable `isystem_os_trace` in the `Mk_k_isystem.c` source file.

```
/* =====
 * file: Mk_k_isystem.c
 * iSYSTEM EB tresos Safety OS Thread State Trace Instrumentation
 * Global variable used for OS Thread State tracing.
 * Needs to be accessible by kernel.
 * ===== */
unsigned long isystem_os_trace[1];
```

Listing 6: Definition of `isystem_os_trace` Variable

### 4.2.3 Adding the Instrumentation Code into the Build System

The *Mk\_k\_isystem.c* source file can be added to the kernel library file list by extending the `MK_LIBSRCKERN_KLIB_BASELIST` macro in the *MicroOs\_filelist.mak* file.

```
\make\MicroOS_filelist.mak:

=====
# Lists of base filenames for the kernel library
=====
# MK_LIBSRCKERN_KLIB_BASELIST is the list of all files in the plugin/lib_src/kernel
directory that
# go into the kernel library.
# The files are listed without prefix or extension.
# DON'T PUT SYSTEM-CALL KERNEL-SIDE FUNCTIONS HERE! - Put them in
MK_SYSTEMCALL_BASELIST!!!
MK_LIBSRCKERN_KLIB_BASELIST += isystem
```

Listing 7: Makefile modifications for adding MK\_k\_isystem.c to the Build Process

## 4.3 Operating System Configuration

Making winIDEA aware of the target operating system (OS) can be accomplished by reading in an OS description file. In case of an OSEK-compliant AUTOSAR OS, this OS description file is the so-called ORTI file, generated by the OS generator of the AUTOSAR tool (e.g. EB tresos Studio).

For thread-state tracing of EB tresos Safety OS, this ORTI file based approach must be extended by means of an iSYSTEM-proprietary XML file. An OS description file can be imported into winIDEA via the menu: “Debug – Operating System...” as shown in Figure 20.

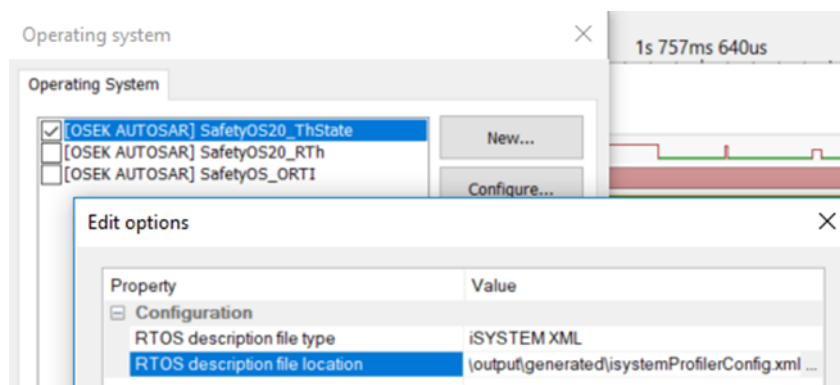


Figure 20: Selection of the iSYSTEM Profiler XML File



## 4.4 iSYSTEM Profiler XML

Figure 21 shows a sample profiler XML file used for thread-state profiling via hook instrumentation.

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2  <OperatingSystem>
3    <Name>SafetyOsDemo</Name>
4    <NumCores>2</NumCores>
5    <ORTI>orti\os.orti</ORTI>
6  </OperatingSystem>
7  <Types>
8    <TypeEnum><Name>TypeThreadStateSymbol</Name>
9      <Enum><Name>UNDEF</Name>    <Value>0xFF</Value></Enum>
10     <Enum><Name>IDLE</Name>     <Value>0x00</Value></Enum>
11     <Enum><Name>READY</Name>    <Value>0x01</Value></Enum>
12     <Enum><Name>RUNNING</Name>  <Value>0x02</Value></Enum>
13     <Enum><Name>NEW</Name>      <Value>0x03</Value></Enum>
14   </TypeEnum>
15 </Types>
16 <Profiler>
17   <Object>
18     <Definition>TASKSTATE</Definition>
19     <Description>Threads</Description>
20     <Type>OS:vs_RUNNINGTHREAD</Type>
21     <DefaultValue>NO_THREAD</DefaultValue>
22     <Name>TASKSTATE</Name>
23     <Level>Task</Level>
24     <Expression>isystem_os_trace[0]</Expression>
25     <TaskState>
26       <MaskID>0xFFFF</MaskID>
27       <MaskState>0x30000000</MaskState>
28       <MaskCore>0xC0000000</MaskCore>
29       <Type>TypeThreadStateSymbol</Type>
30       <StateInfo><Name>IDLE</Name><Property>Terminate</Property></StateInfo>
31       <StateInfo><Name>RUNNING</Name><Property>Run</Property></StateInfo>
32     </TaskState>
33   </Object>
34 </Profiler>
35 </OperatingSystem>

```

Figure 21: Sample iSYSTEM Profiler XML file for Thread-State Profiling via Hook Instrumentation

The XML file can be separated in three major sections:

1. The “ORTI” section includes the standard ORTI file generated by EB tresos Studio. From the ORTI file the profiler obtains the mapping between the `MK_taskCfgTable[].threadCfg.name` values obtained by instrumentation and tracing the `isystem_os_trace` variable.
2. The “Types” section contains an enumeration type `TypeThreadStateSymbol` that maps thread-state names (displayed within the profiler) to data values obtained by instrumentation and tracing the `isystem_os_trace` variable as described in the “Profiler” section of the XML file.
3. The “Profiler” section describes how to decode the data values captured by tracing the `isystem_os_trace` variable.

## 4.5 Analyzer Configuration

The OS profiler of the winIDEA analyzer can be enabled by selecting “OS objects” in the “Profiler” tab of the analyzer configuration dialog.

The “RTOS Profiler Options” dialog (opened via »OS Setup...«) allows enabling/disabling of individual OS objects in the analysis.

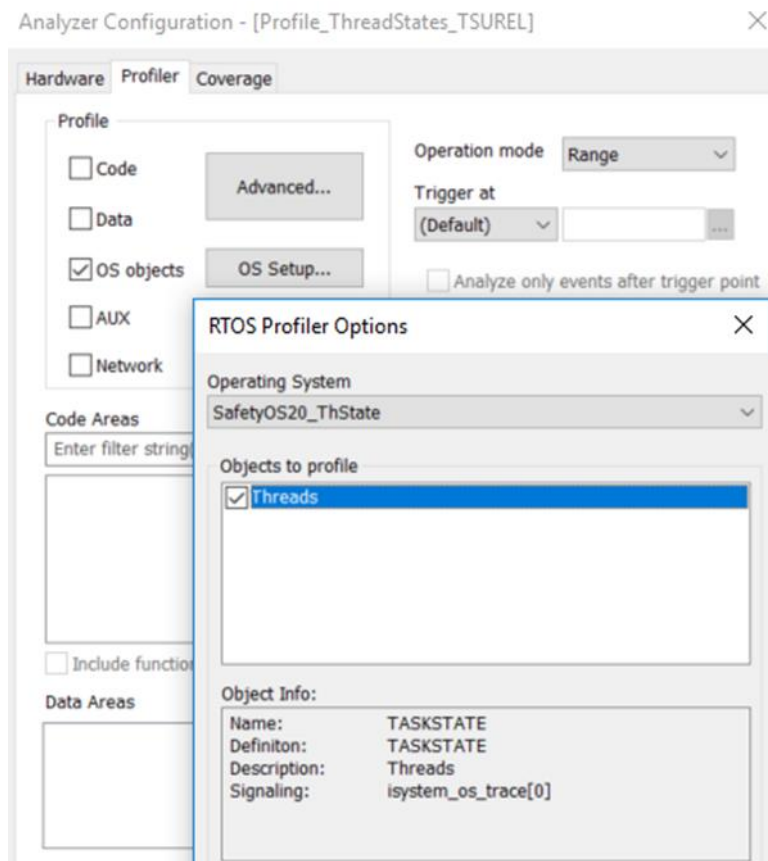


Figure 22: OS Thread-State Configuration in the iSYSTEM Analyzer

## 4.6 Profiler Display

The screen shot below shows a sample dual-core thread-state profile. The threads “Loop”, “Cyclic”, “Cyclic2C1”, “Task\_St1” and “Task\_St2” run on the primary core, the thread “Cyclic2C2” executes on the secondary core.

The profiler timelines in Figure 23 show a state trace of the threads running on two cores. Both timelines are based on the same trace recording, but display different time spans. The upper timeline is zoomed in at the location of the blue and yellow markers of the lower timeline.

A dark-red profiler state bar indicates that the corresponding core is currently executing this thread. A dark-blue bar indicates that there are multiple state transitions and the user must zoom in to see further details of each state transition.

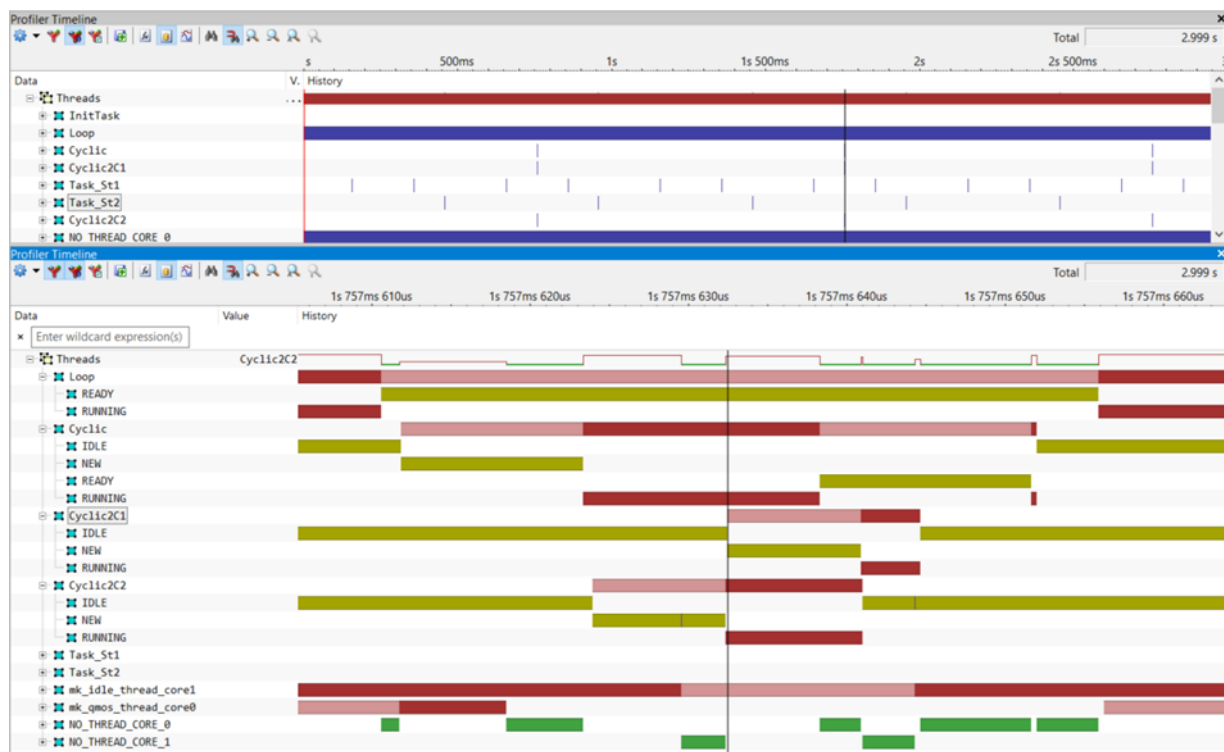


Figure 23: Sample Thread-State Profiler Timeline

## 5 Thread-State Profiling without Code Instrumentation

### 5.1 Overview

The current state of each thread can be monitored by tracing a dedicated variable within a thread control and status structure. These structures are grouped into arrays. EB tresos Safety OS maintains one array per core and per thread type. Typically, EB tresos Safety OS uses 6 thread types, i.e. there are 6 arrays per core. The most appropriate trace configuration for tracing the thread state variables depends on the capabilities of the on-chip trace logic of the processor, i.e. available trace interface bandwidth and data trace filtering (qualifier) features.

### 5.2 Thread Control/Status Structures

As mentioned above, EB tresos Safety OS typically uses 6 thread types:

Thread Type	Description
Aux1 Thread	Kernel thread, typically used for the “QMOS” task.
Aux2 Thread	Kernel thread, typically used for the “MAIN” task.
Error Hook Thread	Kernel thread, used of the error hook task.
Idle Thread	Kernel thread, used for the idle task.
ISR Threads	User threads, used for ISRs of category 2.
Task Threads	User threads, used for user tasks.

The figure below shows all thread control/status arrays of core 0 listed in a winIDEA Watch window. The task thread array has been expanded, as well as the structure of task thread 2, showing its individual structure elements, such as the “state” object (of type “mk\_threadstate\_t”).

Name	Value	Type	Address
⊞ MK_c0_aux1Thread	(Ptr(0x7000'4084) =	mk_thread_t	(Virtual) 7000'4094
⊞ MK_c0_aux2Thread	(Ptr(0x7000'40E8) =	mk_thread_t	(Virtual) 7000'40F8
⊞ MK_c0_errorHookThread	(Ptr(0x7000'41A8) =	mk_thread_t	(Virtual) 7000'41B8
⊞ MK_c0_idleThread	(Ptr(0x7000'4244) =	mk_thread_t	(Virtual) 7000'4254
⊞ MK_c0_isrThreads	((Ptr(0x7000'42CC),	mk_thread_t [2]	(Virtual) 7000'42DC
⊞ [0]	(Ptr(0x7000'42CC),	mk_thread_t	(Virtual) 7000'42DC
⊞ [1]	(Ptr(0x0000'0000) =	mk_thread_t	(Virtual) 7000'4330
⊞ MK_c0_taskThreads	((Ptr(0x7000'4644),	mk_thread_t [5]	(Virtual) 7000'4664
⊞ [0]	(Ptr(0x7000'4644),	mk_thread_t	(Virtual) 7000'4664
⊞ [1]	(Ptr(0x7000'4624),	mk_thread_t	(Virtual) 7000'46B8
⊞ [2]	(Ptr(0x7000'4634),	mk_thread_t	(Virtual) 7000'470C
⊞ regs	Ptr(0x7000'4634)	mk_hwthreadregisters_t *	(Virtual) 7000'470C
⊞ name	Ptr(0x8003'DF54)	char *	(Virtual) 7000'4710
* (name), s	"Rte_Time_Task"	char [256]	(Virtual) 8003'DF54
⊞ next	Ptr(0x0000'0000) =	mk_culprit_t	(Virtual) 7000'4714
⊞ parentThread	Ptr(0x0000'0000) =	mk_culprit_t	(Virtual) 7000'4718
parentCookie	0	unsigned long	(Virtual) 7000'471C
⊞ xcoreReply	(0,0)	mk_statusandvalue_t	(Virtual) 7000'4720
⊞ accounting	(Ptr(0x8003'E090), 42	mk_accounting_t	(Virtual) 7000'4728
state	MK_THS_IDLE	mk_threadstate_t	(Virtual) 7000'4730
queueingPriority	3	long	(Virtual) 7000'4734
runningPriority	3	long	(Virtual) 7000'4738
currentPriority	0	long	(Virtual) 7000'473C
⊞ lastLockTaken	Ptr(0x0000'0000) =	mk_lock_t *	(Virtual) 7000'4740
⊞ jobQueue	Ptr(0x0000'0000) =	mk_jobqueue_t *	(Virtual) 7000'4744
⊞ eventStatus	Ptr(0x0000'0000) =	mk_eventstatus_t *	(Virtual) 7000'4748
memoryPartition	-1	long	(Virtual) 7000'474C
currentObject	3	long	(Virtual) 7000'4750
objectType	MK_OBJTYPE_TASK	mk_objecttype_t	(Virtual) 7000'4754
applicationId	-1	long	(Virtual) 7000'4758
parentCore	-1	long	(Virtual) 7000'475C
⊞ [3]	(Ptr(0x7000'4614) =	mk_thread_t	(Virtual) 7000'4760
⊞ [4]	(Ptr(0x7000'4654),	mk_thread_t	(Virtual) 7000'47B4

Figure 24: Thread Control/Status Structures of all six Thread Types

### 5.3 Operating System Configuration

Making winIDEA aware of the target operating system (OS) can be accomplished by reading in an OS description file. In case of an OSEK-compliant AUTOSAR OS, this OS description file is the so-called ORTI file, generated by the OS generator of the AUTOSAR tool (e.g. EB tresos Studio).

For thread-state tracing of EB tresos Safety OS, this ORTI file based approach must be extended by means of an iSYSTEM-proprietary XML file. An OS description file can be imported into winIDEA via the menu: “Debug – Operating System...” as shown in Figure 25.

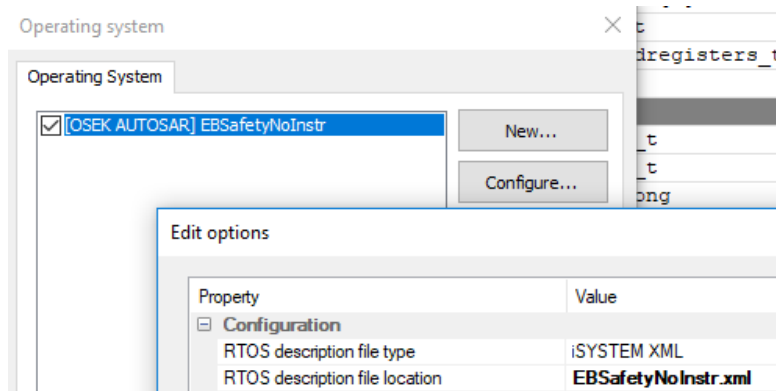


Figure 25: Selection of the iSYSTEM Profiler XML File



## 5.4 iSYSTEM Profiler XML

The figure below shows a sample profiler XML file used for thread-state profiling without instrumentation.

```
<TypeEnum>
  <Name>Type_ThreadState_MAPPING</Name>
  <Enum>
    <Name>NO_TASK</Name>
    <Value>0xFF</Value></Enum>
    <Name>Init_Task</Name>
    <Value>0</Value><Property><Name>Expression</Name><Value>MK_c0_taskThreads[0].state</Value></Property></Enum>
    <Name>SchMDiagStateTask_20ms</Name>
    <Value>1</Value><Property><Name>Expression</Name><Value>MK_c0_taskThreads[1].state</Value></Property></Enum>
    <Name>Rte_Time_Task</Name>
    <Value>2</Value><Property><Name>Expression</Name><Value>MK_c0_taskThreads[2].state</Value></Property></Enum>
    <Name>Rte_Event_Task</Name>
    <Value>3</Value><Property><Name>Expression</Name><Value>MK_c0_taskThreads[3].state</Value></Property></Enum>
    <Name>SchMComTask_1ms</Name>
    <Value>4</Value><Property><Name>Expression</Name><Value>MK_c0_taskThreads[4].state</Value></Property></Enum>
    <Name>CANSR4_ISR</Name>
    <Value>5</Value><Property><Name>Expression</Name><Value>MK_c0_isrThreads[0].state</Value></Property></Enum>
    <Name>UNUSED_ISR</Name>
    <Value>6</Value><Property><Name>Expression</Name><Value>MK_c0_isrThreads[1].state</Value></Property></Enum>
    <Name>C0_QMOS_Thread</Name>
    <Value>7</Value><Property><Name>Expression</Name><Value>MK_c0_aux1Thread.state</Value></Property></Enum>
    <Name>C0_MAIN_Thread</Name>
    <Value>8</Value><Property><Name>Expression</Name><Value>MK_c0_aux2Thread.state</Value></Property></Enum>
    <Name>C0_IDLE_Thread</Name>
    <Value>9</Value><Property><Name>Expression</Name><Value>MK_c0_idleThread.state</Value></Property></Enum>
    <Name>C0_ErrorHook_Thread</Name>
    <Value>10</Value><Property><Name>Expression</Name><Value>MK_c0_errorHookThread.state</Value></Property></Enum>
  </Enum>
</TypeEnum>

</Types>

<Profiler>
  <Object>
    <Definition>TASKSTATE</Definition>
    <Description>Tasks</Description>
    <Type>Type_ThreadState_MAPPING</Type>
    <Expression>$(EnumType)</Expression>
    <DefaultValue>NO_TASK</DefaultValue>
    <Name>TASKSTATE</Name>
    <Level>Task</Level>
    <TaskState>
      <MaskID>0x0</MaskID>
      <MaskState>0xFF</MaskState>
      <MaskCore>0x0</MaskCore>
      <Type>Type_ThreadState</Type>
      <BTFMappingType>Type_BTSTATE_MAPPING</BTFMappingType>
      <StateInfo><Name>IDLE</Name><Property>Terminate</Property></StateInfo>
      <StateInfo><Name>WAITING</Name><Property>Terminate</Property></StateInfo>
      <StateInfo><Name>RUNNING</Name><Property>Run</Property></StateInfo>
    </TaskState>
  </Object>
</Profiler>
```

Figure 26: Sample iSYSTEM Profiler XML file for Thread-State Profiling

In the upper section (“Types”) an enumeration type is defined (“Type\_ThreadState\_MAPPING”), which maps a thread name, displayed in the winIDEA Profiler to its corresponding state variable in the OS thread status/control structure/array.

In the lower section (“Profiler”), a new profiler object is created. It is defined as a “TASKSTATE” object, telling the profiler that this object is used for OS task state (or thread) reconstruction. The “Type” and “Expression” tags tell the profiler to use the “Type\_ThreadState\_MAPPING” for the thread state analysis.

## 5.5 Analyzer Configuration

The OS profiler of the winIDEA analyzer can be enabled by selecting »OS objects« in the »Profiler« tab of the analyzer configuration dialog.

The “RTOS Profiler Options” dialog (opened via “OS Setup...”) allows enabling/disabling of individual OS objects in the analysis.

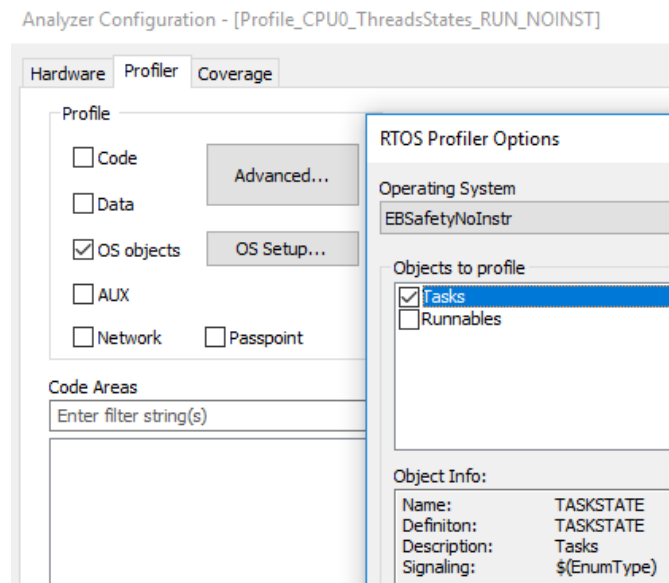


Figure 27: OS Thread-State Configuration in the iSYSTEM Analyzer

## 5.6 Profiler Display

The profiler timelines in Figure 28 show a thread state trace of a sample EB tresos Safety OS application. Both timelines are based on the same trace recording, but display different time spans. The upper timeline is zoomed in at the location of the blue and yellow markers of the lower timeline. A dark-red profiler state bar indicates that the corresponding core is currently executing this thread. A light-red bar indicates that a thread is active, but currently not running, i.e. it is each in NEW (activated but not running yet) or in READY (pre-empted by other thread) state.

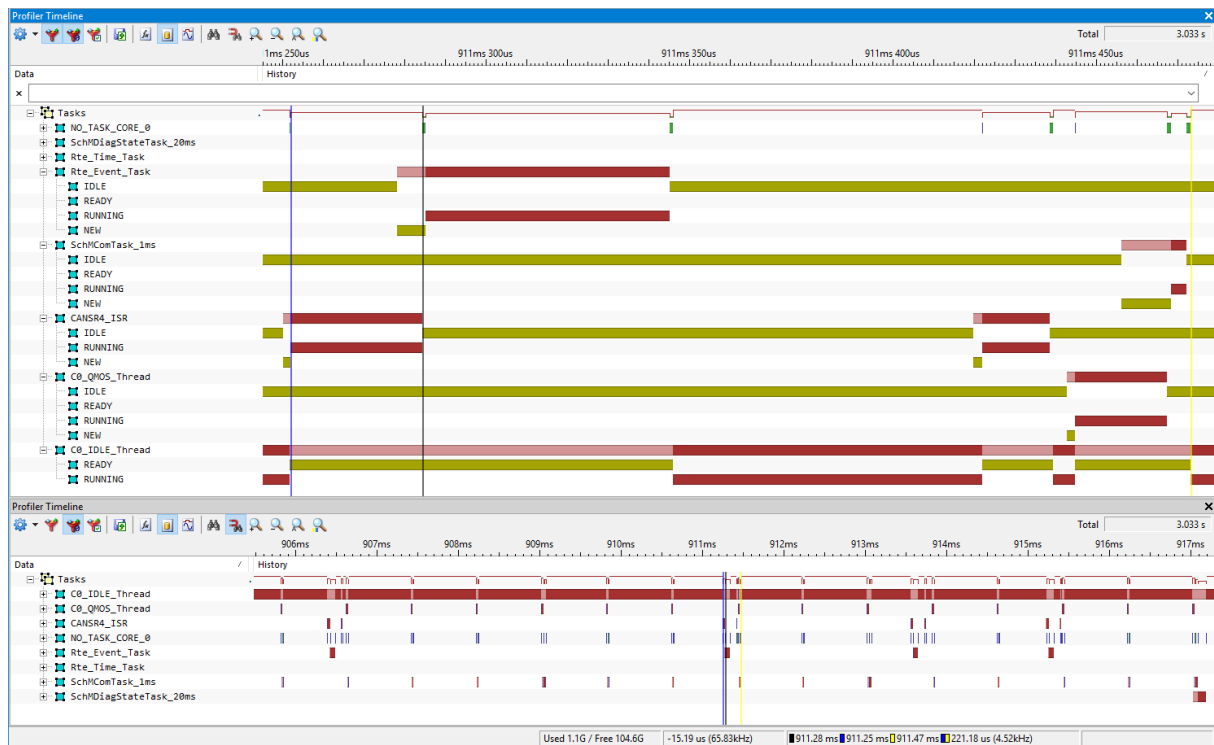


Figure 28: Sample Thread-State Profiler Timeline

## 5.7 Hardware Trace Configuration Options for various Processor Architectures

The on-chip trace logic (of each core) must be configured for monitoring all data write accesses to the thread state variable of each thread. As mentioned earlier this typically involves six thread status/control arrays.

### 5.7.1 ARM ETM (e.g. on ARM Cortex R7)

The ARM ETM can be configured to observe the write access to the thread status/control arrays. However, the number of available address comparator depends on the actual ETM configuration on the given processor.

The figure below shows a sample ETM configuration on an R7 ETM implementation a Renesas RCAR M3 SoC.

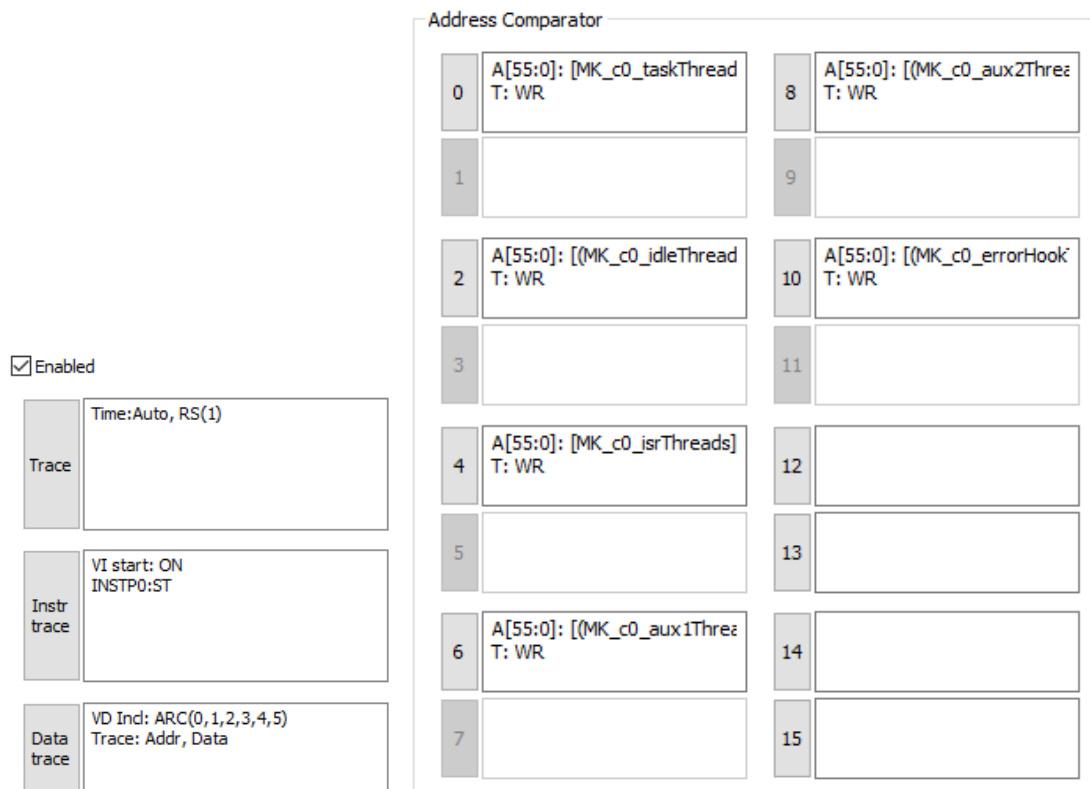


Figure 29: R7 ETM configuration to trace write accesses to the Thread Status/Control Arrays

### 5.7.2 Infineon AURIX MCDS (e.g. on TC277ED)

The AURIX MCDS can either be configured to observe the entire arrays or the (many) individual thread state objects.

The entire arrays can be observed via the DTU Magnitude comparators (dtu\_ea\_trig\_[7:0]). This approach only required 6 on-chip data trace channels (i.e. address comparators), but the disadvantage is that also write access to array/structure elements are traced, which are not required for thread state tracing. This means, unnecessary trace messages may lead to trace interface bandwidth issues (i.e. trace overflow to reduced trace duration).

```

dtu_ea_trig_0    [MK_c0_taskThreads]
dtu_ea_trig_1    [MK_c0_aux1Thread]
dtu_ea_trig_2    [MK_c0_aux2Thread]
dtu_ea_trig_3    [MK_c0_isrThreads]
dtu_ea_trig_4    [MK_c0_idleThread]
dtu_ea_trig_5    [MK_c0_errorHookThread]

```

Figure 30: AURIX MCDS DTU trigger configuration to trace the Thread Status/Control Arrays

Alternatively, the individual thread state can be observed by means of the MCDS Fine Grain Comparator.

The figure below shows a sample Fine Grain Comparator configuration for a system with 5 task threads.

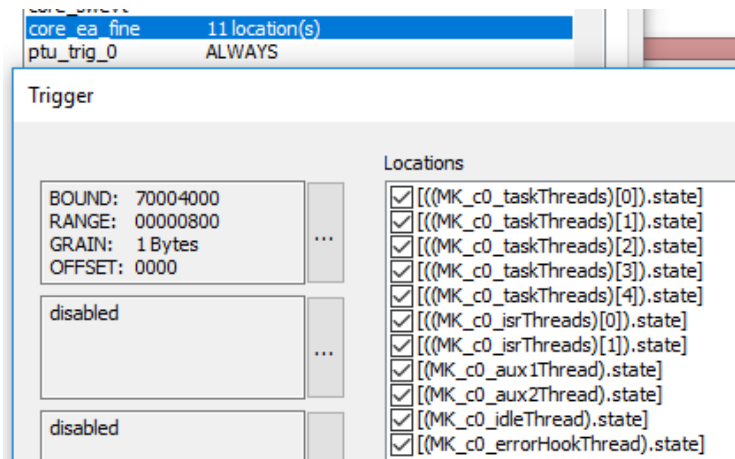


Figure 31: Sample Fine Grain Comparator configuration for a system with 5 task threads

## 6 Inspectors

Inspectors are a winIDEA feature to analyze user-defined metrics in the winIDEA profiler timeline. It allows the creation of new Profiler objects, so called Inspectors, which can change their state depending on different events, such as state changes of other objects and timing parameters. This section demonstrates how inspectors can be used to cover certain advanced timing-analysis use-cases for the EB tresos AutoCore operating system.

### 6.1 Task Metric Analysis

Inspectors can be used to calculate the metrics defined in the AUTOSAR Timing Extensions Specification. Predefined Inspectors exist for a certain subset of those metrics. The Inspectors are defined in a generic way meaning the metrics are calculated for all threads in the trace. There is no need to add a separate Inspector for each task and metric.

If you are interested in using those Inspectors, ask your iSYSTEM contact for the respective Inspectors JSON file which can be imported into the winIDEA Profiler to make the metrics available.

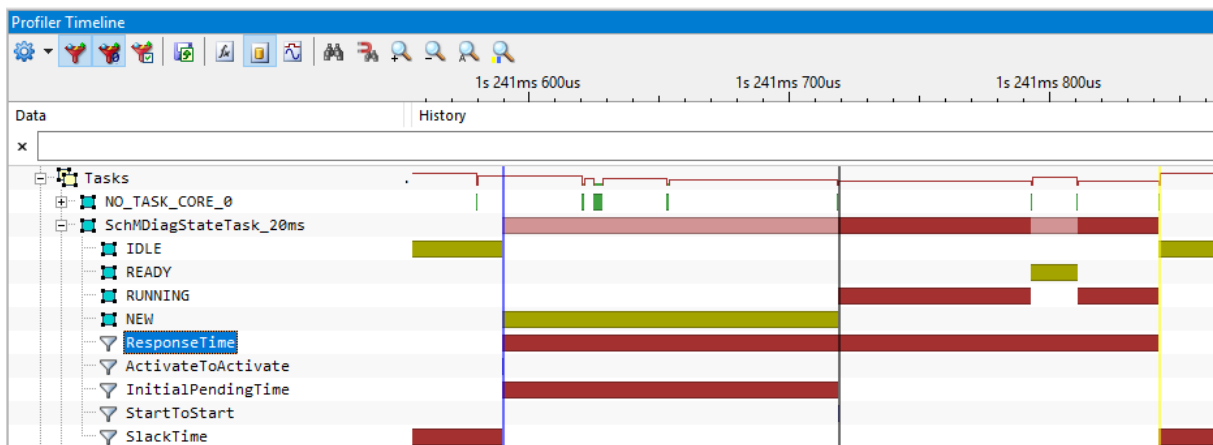


Figure 32: Inspectors to calculate Task Metrics for the Thread SchMDiagStateTask\_20ms

For a further analysis of the Inspector objects, you can utilize the Properties view of the winIDEA Analyzer. To open the Properties view, select the desired object and press “Alt + Enter”.

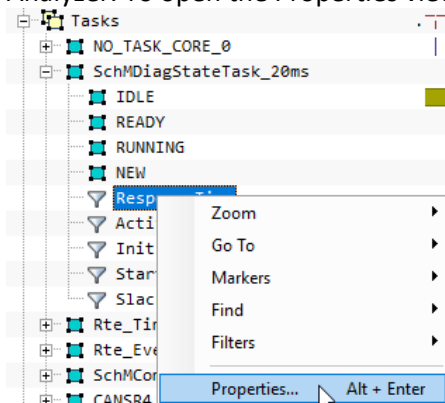


Figure 33: Opening the “Properties” View for a Profiler Inspector Object

For the task metric “StartToStart”, the relevant object statistic is “Period”. It measures the time difference between the NEW-to-RUNNING state transitions of two consecutive instances of the same thread.



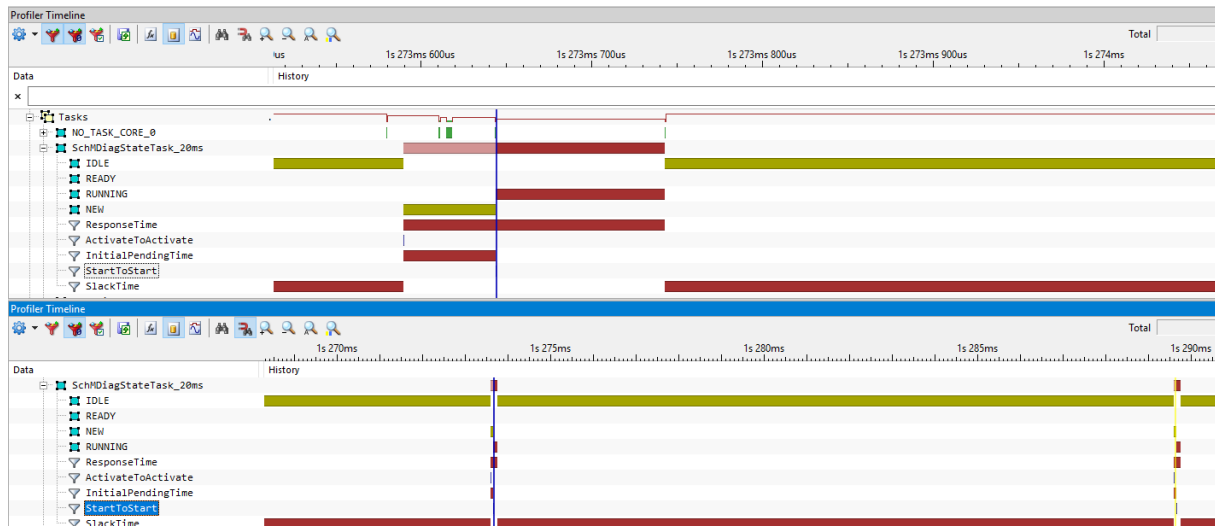


Figure 34: Inspector Object “StartToStart” for the Thread SchMDiagStateTask\_20ms

Note: The Figure shows two profiler timelines of the same trace recording only using different zoom factors.

The Properties view provides the measurements for average, maximum and minimum period (i.e. “StartToStart” time) along with the time (and link “->”) to its occurrence.

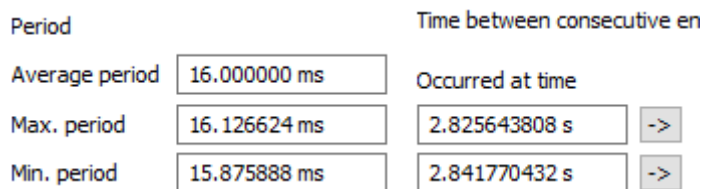


Figure 35: Period Properties for the “ActivateToActivate” Inspector Object

For the task metric “ResponseTime”, the relevant object statistic is “Net Time”. It measures the time between the thread activation (IDLE-to-NEW transition) and the start of the thread (NEW-to-RUNNING transition).

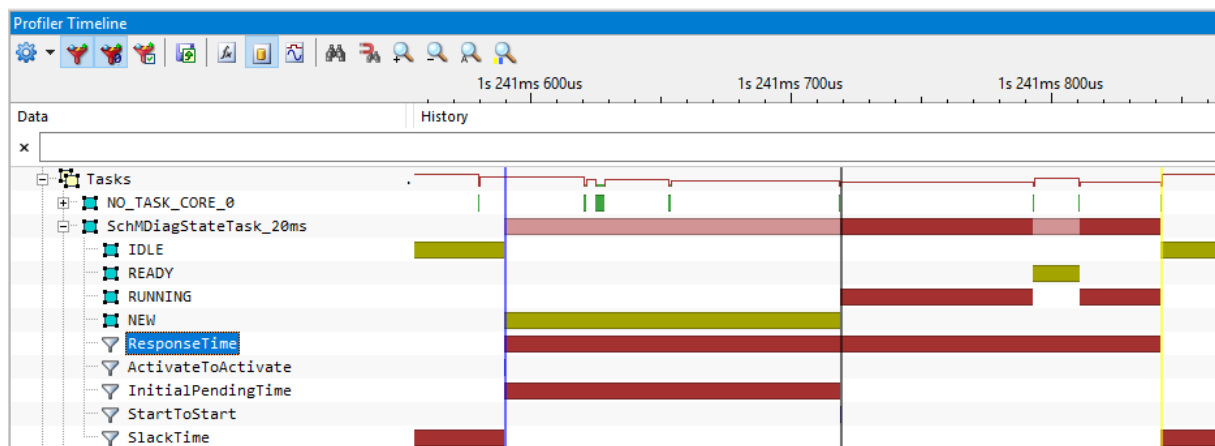


Figure 36: Inspector Object “ResponseTime” for the Thread SchMDiagStateTask\_20ms

The Properties view provides the measurements for average, maximum and minimum Net Time (i.e. “ResponseTime”) along with the time (and link “->”) to its occurrence.

Net Time	32.235672 ms	
Average	170.559 us	Occurred at time
Max	300.512 us	2.041589608 s ->
Min	151.576 us	537.589616 ms ->

Figure 37: Net Time Properties for the “ResponseTime” Inspector Object


## 7 BTF Export

The winIDEA Profiler supports the export of traces into the BTF format. BTF is a CSV based trace format that is supported by different timing tool vendors. Before the BTF export is usable the iSYSTEM profiler XML file must be updated. The Profiler supports the export of tasks, ISR2s, Runnables and signals. For tasks and threads the following BTF mapping reference must be added to the `TASKSTATE` object.

```
<BTFMappingType>Type_BTFSIZE_MAPPING</BTFMappingType>
```

The *btf\_mapping* itself must then be added to the TypeEnum section of the XML file. For EB tresos Safety OS the mapping in **Error! Reference source not found.** can be used. The mapping is required to tell winIDEA which thread state maps to which BTF task state transition.

The following steps must be executed to export a BTF trace file.

1. Load symbols  to make sure that the updated iSYSTEM Profiler XML is in use.
2. Record a trace with the necessary configuration to record threads and Runnables.
3. Select the export button in the Profiler timeline, choose BTF export, and export.

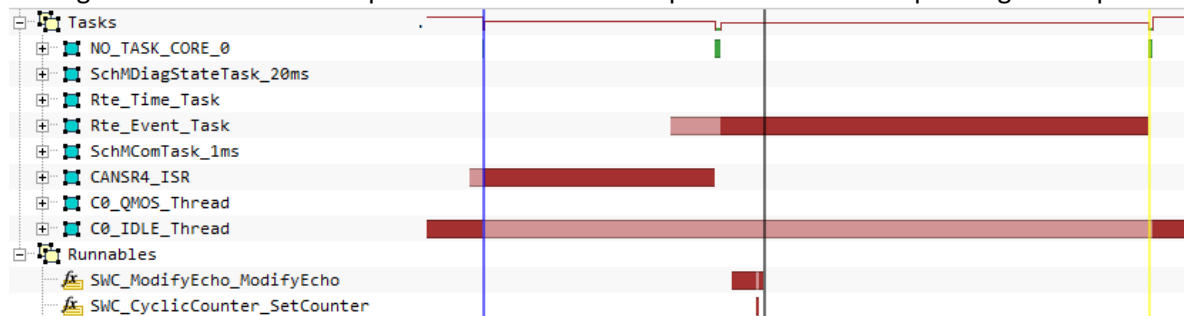


This generates a BTF trace file which matches the profiler timeline as shown in Figure 38.

```
<TypeEnum>
  <Name>Type_BTFSIZE_MAPPING</Name>
  <Enum><Name>NEW</Name><Value>Active</Value></Enum>
  <Enum><Name>READY</Name><Value>Ready</Value></Enum>
  <Enum><Name>RUNNING</Name><Value>Running</Value></Enum>
  <Enum><Name>IDLE</Name><Value>Terminated</Value></Enum>
</TypeEnum>
```

Listing 8: Mapping from EB tresos Safety OS thread states to BTF task state transitions.

The figure below shows a sample thread and Runnable profile and its corresponding BTF export.



```
128844384,CORE_0,97,T,CANSR4_ISR,97,start
128870360,STI_Rte_Event_Task,47,T,Rte_Event_Task,47,activate
128876392,CORE_0,97,T,CANSR4_ISR,97,terminate
128877080,CORE_0,47,T,Rte_Event_Task,47,start
128878816,Rte_Event_Task,47,R,SWC_ModifyEcho_ModifyEcho,47,start
128882144,Rte_Event_Task,47,R,SWC_ModifyEcho_ModifyEcho,47,suspend
128882144,Rte_Event_Task,47,R,SWC_CyclicCounter_SetCounter,47,start
128882556,Rte_Event_Task,47,R,SWC_CyclicCounter_SetCounter,47,terminate
128882556,Rte_Event_Task,47,R,SWC_ModifyEcho_ModifyEcho,47,resume
128883060,Rte_Event_Task,47,R,SWC_ModifyEcho_ModifyEcho,47,terminate
128936896,CORE_0,47,T,Rte_Event_Task,47,terminate
128937584,CORE_0,0,T,C0_IDLE_Thread,0,resume
```

Figure 38: The winIDEA Profiler allows a trace export to the BTF format. BTF is supported by various timing tool vendors.

Note: The winIDEA Profiler also allows an export of Runnables. However, Runnable trace and profiling is beyond the scope of this Application Note. Please refer to the dedicated Application Note about Runnable trace with EB tresos AutoCore and EB tresos Safety.

## 8 Technical Support

### 8.1 Online Resources

<a href="#">Online Help</a> ▶ winIDEA and testIDEA online help	<a href="#">Knowledge Base</a> ▶ Tips & tricks categorized by issue type and architecture	<a href="#">Tutorials</a> ▶ From beginner to expert
<a href="#">Technical Notes</a> ▶ How-tos for winIDEA functionalities with scripts	<a href="#">Application Notes</a> ▶ How-to notes on advanced use-cases	<a href="#">Webinars</a> ▶ Technical webinars about ISYSTEM tools with use cases

### 8.2 Contact

Please visit <https://www.isystem.com/contact.html> for contact details.

iSYSTEM has made every effort to ensure the accuracy and reliability of the information provided in this document at the time of publishing. Whilst iSYSTEM reserves the right to make changes to its products and/or the specifications detailed herein, it does not make any representations or commitments to update this document.

© iSYSTEM. All rights reserved.