MICROSAR OS RH850

Technical Reference

Authors	Senol Cendere, Yohan Humbert
Version	1.11
Status	Released

Document Information

History

Author	Date	Version	Remarks
S. Cendere	2014-01-21	1.00	Creation
S. Cendere	2014-02-03	1.01	Release for RH850 SafeContext
S. Cendere	2014-04-24	1.02	Release for RH850 P1M
S. Cendere	2014-09-30	1.03	Added error numbers for interrupt consistency checks
Y. Humbert	2014-10-14	1.04	Update
Y. Humbert	2015-01-29	1.05	Added ASID support
Y. Humbert	2015-02-26	1.06	Added D1M, E1L, E1M and F1M
Y. Humbert	2015-06-10	1.07	Added Multicore chapter
S. Cendere	2015-06-29	1.08	Added Timing Protection chapter
S. Cendere	2015-08-20	1.09	Removed core exception attributes
Y. Humbert	2015-11-10	1.10	Support Multicore SC3
S. Cendere	2016-01-22	1.11	Added description for osCheckAndRefreshTimer

Reference Documents

Ref.	Source	Title	Version
[1]	AUTOSAR	AUTOSAR Operating System Specification (downloadable from www.Autosar.org)	3.0.x 4.0.x 4.1.x
[2]	OSEK	OSEK/VDX Operating System Specification (downloadable from www.osek-vdx.org)	2.2.3
[3]	Vector Informatik GmbH	Technical Reference MICROSAR OS	9.01

Scope of the Document

This technical reference describes the specific use of the MICROSAR OS for Renesas RH850. It supplements the general technical reference for MICROSAR OS [3].

Caution
We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector´s release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Overview

This document describes the implementation specific part of the AUTOSAR operating system for the Renesas RH850 microcontroller family. In this document a processor of the family may be referred as RH850.

The common part of all MICROSAR OS implementations is described in document [3].

The implementation is based on the OSEK-OS-specification 2.2.3 and on AUTOSAR OS specifications 3.0.x/4.0.x/4.1.x. This document assumes that the reader is familiar with the OSEK and AUTOSAR OS specifications.

OSEK/VDX is a registered trademark of Continental Automotive GmbH (until 2007: Siemens AG).

Contents

1 Overview of MICROSAR OS

1.1 Overview of Properties

Property Class	Version / Range / Support
AUTOSAR OS specification	3.0.x, 4.0.x, 4.1.x
Scalability Classes supported	SC1 SC3 (SafeContext) SC4 (SafeContext)
Conformance Classes supported	SC1: BCC1, BCC2, ECC1, ECC2 SC3 and SC4: ECC2
Scheduling policy	SC1: full-, non- and mixed-preemptive SC3 and SC4: full- and mixed-preemptive
Scheduling points	depending on scheduling policy
Maximum number of tasks	65535
Maximum number of events per task	32
Maximum number of activations per task	255
Maximum number of priorities	8192
Maximum number of counters	256
Maximum number of alarms	32767
Maximum number of resources	8192
Maximum number of resources locked simultaneously	255
Maximum number of schedule tables	65535
Maximum number of expiry points	65535 (cyclical expiry points counted by their multiplicity)
Maximum number of ISRs	depending on derivative
Status Levels	SC1: STANDARD and EXTENDED SC3: EXTENDED SC4: EXTENDED
Nested Interrupts	supported
Interrupt level resource handling	SC1: supported SC3: not supported SC4: not supported
ORTI support	SC1: 2.1 Standard and 2.1 Additional 2.2 Standard and 2.2 Additional SC3: 2.2 Additional SC4: 2.2 Additional
Library Version	not supported
FPU support	supported (if provided by derivative)

Table 1 OS Properties

2 Installation

The general installation is described in the common document [3].

The RH850 specific files are described below.

2.1 OIL-Configurator

The OIL-configurator is a general tool for different AUTOSAR implementations. The implementation specific parts are the code generator and the OIL-implementation files for the code generator.

- OIL-Configurator root\OILTOOL
- OIL-Implementation files root\OILTOOL\GEN
- Code Generator root\OILTOOL\GEN

2.1.1 OIL-Implementation Files

The implementation specific files will be copied onto the local hard disk. The OIL-tool has knowledge about these files through the file OILGEN.INI (the correct path is set by the installation program).

CPU	Implementation file	Standard object file	Description
D1L	RH850_D1L.i41	RH850_D1L.s41	Source code version
D1M	RH850_D1M.i41	RH850_D1M.s41	Source code version
E1L	RH850_E1L.i41	RH850_E1L.s41	Source code version
E1M	RH850_E1M.i41	RH850_E1M.s41	Source code version
E1x-FCC2	RH850_E1x_FCC2.i41	RH850_E1x_FCC2.s41	Source code version
F1H	RH850_F1H.i41	RH850_F1H.s41	Source code version
F1L	RH850_F1L.i41	RH850_F1L.s41	Source code version
F1M	RH850_F1M.i41	RH850_F1M.s41	Source code version
P1M	RH850_P1M.i41	RH850_P1M.s41	Source code version

Table 2 OIL implementation Files

3 Configuration

Before an application can be compiled all static MICROSAR OS objects have to be defined in a configuration file. The OS generator generates code in accordance to this configuration file.

The configuration can be done either in XML language (AUTOSAR ECU configuration format) or in OIL (OSEK implementation language).

Chapter 4 shows the program flow of a configuration / generation / compilation process in detail.

There are configuration attributes which are standard for all MICROSAR OS implementations. These are described in [3].

Platform specific attributes (which only apply to this implementation of MICROSAR OS) are described hereafter.

3.1 XML Configuration

An XML configuration of the OS must conform to the AUTOSAR XML schema. To edit such a configuration the DaVinci configurator of Vector Informatik GmbH can be used.

3.2 OIL Configuration

MICROSAR OS systems for RH850 can also be described using OIL. The OIL configurator tool is capable of reading and writing OIL files. The finished OIL file is passed to the code generator which generates the configuration files of the OS.

3.3 OS Attributes

The OS object controls general aspects of the operating system. The following attributes are provided for scalability class SC1, SC3 and SC4:

Attribute Name		Values Default value is written in bold	Description
OIL	XML		
Compiler	OsOSCompiler	**GHS**	Selects the supported compiler.
SystemStackSize	OsOSSystemStackSize	0 … 0xFFFC	Size of the system stack in bytes.
EnumeratedUnhandledISRs	OsOSEnumeratedUnhandledISRs	TRUE **FALSE**	This attribute determines handling of unused interrupt sources. If set TRUE then variable ossUnhandledExceptionDetail is set to the exception number before calling osUnhandledException.
ORTIDebugSupport	OsOSORTIDebugSupport	TRUE **FALSE**	The RH850 implementation supports ORTI debug information if this attribute is selected.
ORTIDebugLevel	OsOSORTIDebugLevel	**ORTI_22_Additional**	Support ORTI 2.2 with additional features which require some additional runtime and memory.
UserConfigurationVersion	OsOSUserConfigurationVersion	0 ... 0xFFFF	This value specifies the current user specific version of the OS configuration. It can be read back by the user for validation.
SupportFPU	OsOSSupportFPU	TRUE **FALSE**	Switches on the support for FPU. (if provided by derivative)
TimingProtectionTimerClock	OsOSTimingProtectionTimerClock	0 … 999999 **No default**	Only SC4: peripheral clock of timer unit TAUJ0 specified in [kHz]

Table 1: RH850 specific attributes of OS

For scalability class SC3 and SC4 the following additional attributes are provided:

Attribute Name		Values Default value is written in bold	Description
OIL	XML		
CheckIntAPIStatus	OsOSCheckIntAPIStatus	**TRUE** FALSE	If set to `FALSE` then the OS API functions `CallTrustedFunction` and `CallNonTrustedFunction` do not check the interrupt status.
PeripheralRegion	OsOSPeripheralRegion	**No default**	List of peripheral regions. A peripheral region defines an address range where access is allowed for selected applications (trusted or non-trusted). The access is granted by means of the API functions.
MemoryProtection	OsOSMemoryProtection	TRUE **FALSE**	Enables memory protection via MPU. Mandatory for SC3 and SC4.
MpuRegion	OsOSMpuRegion	**Enable**	Configures static MPU regions

3.3.1 MpuRegion Sub-Attributes (SC3 and SC4)

If a static MPU region is enabled then the memory area is specified by the following sub-attributes:

Sub-Attribute Name		Values (default value is written in bold)	Description
OIL	XML		
StartAddr	OsOSStartAddr	string **No default**	Start address of static MPU region hexadecimal value: StartAddr = 0x… or memory area specific linker symbol: StartAddr = <linker_start_symbol>
EndAddr	OsOSEndAddr	string **No default**	End address of static MPU region hexadecimal value: EndAddr = 0x… or memory area specific linker symbol: EndAddr = <linker_end_symbol>
AccessRights	OsOSAccessRights	uint32 **No default**	MPU region access configuration
ASID	OsOSASID	TRUE **FALSE**	Specifies, whether ASID matching is enabled for this MPU region.
Identifier	OsOSIdentifier	0 ... 0x3FE **0x3FF**	ASID value to be used as area match condition. The maximum value 0x3FF is used as default value.
CORE	OsOSCORE	uint32 **No default**	Only Multicore OS: Core assignment for corresponding MPU region

Value of StartAddr must always point to the first valid Byte in the specified memory area.

Value of EndAddr must always point to the last valid Byte in the specified memory area.

The number of configurable static MPU regions depends on the derivative.

```
MpuRegion = Enable
{
    StartAddr  = 0x... ;
    EndAddr   = 0x… ;
    AccessRights = 0x…;
    ASID = TRUE;
    Identifier = 0x01;
    CORE = 0;                /* Multicore OS */
};
```

3.3.2 PeripheralRegion Sub-Attributes (SC3 and SC4)

Sub-Attribute Name		Values (default value is written in bold)	Description
OIL	XML		
StartAddress	OsOSStartAddress	uint32 **No default**	Numeric value. Specifies the start address of the peripheral region which shall be configured. Any 32 bit value can be used.
EndAddress	OsOSEndAddress	uint32 **No default**	Numeric value. Specifies the end address of the peripheral region which shall be configured. Any 32 bit value can be used.
Identifier	OsOSIdentifier	string **No default**	C-String Must be a unique C Identifier which can be used in an application or BSW module to access the peripheral region.
ACCESSING_APPLICATION	OsOSAccessingApplication	ApplicationType	Application reference. Defines access rights of an application for this PeripheralRegion. This attribute can be defined multiple times, so that different applications might have access right to the same PeripheralRegion.

Caution

The application is allowed to access memory addresses in the interval of StartAddress <= memory to be accessed <= EndAddress

The “EndAddress” value is included! All bytes of a peripheral access must fit into the peripheral region.

3.4 Counter Attributes

Platform specific attributes are located within the container “DRIVER”.

Sub-Attribute Name		Values (default value is written in bold)	Description
OIL	XML		
Timer	OsCounterTimer	OSTM OSTM_HIRES **No default**	Selects the timer hardware which drives the hardware counter. OSTM: OSTM timer is used and generates cyclic interrupts. OSTM_HIRES: OSTM timer is used and runs in high resolution timer mode. Which interrupt channel is used for a specific derivative can be found in chapter 8.1.

3.4.1 OSTM Sub-Attributes

Sub-Attribute Name		Values (default value is written in bold)	Description
OIL	XML		
EnableNesting	OsCounterEnableNesting	TRUE FALSE **No default**	Specifies whether the timer interrupt which drives the hardware counter can be interrupted by higher priority interrupts.
InterruptPriority	OsCounterInterruptPriority	0 ... **15** 0 ... **7** (F1L)	The priority of the timer ISR which drives the hardware counter.
StackSize	OsCounterStackSize	0 … 0xFFFC	Stack size of the timer ISR which drives the hardware counter.

3.4.2 OSTM_HIRES Sub-Attributes

Sub-Attribute Name		Values (default value is written in bold)	Description
OIL	XML		
EnableNesting	OsCounterEnableNesting	TRUE FALSE **No default**	Specifies whether the timer interrupt which drives the hardware counter can be interrupted by higher priority interrupts.
InterruptPriority	OsCounterInterruptPriority	0 ... **15** 0 ... **7** (F1L)	The priority of the timer ISR which drives the hardware counter.
StackSize	OsCounterStackSize	0 … 0xFFFC	Stack size of the timer ISR which drives the hardware counter.
MinTimeBetweenTimerIrqs	OsCounterMinTimeBetweenTimerIrqs	uint32 **0**	Defines the number of timer ticks which at least must be between two timer interrupts (shortest possible time between two timer interrupts).

Detailed description how to configure software and hardware counter can be found in [3].

3.5 ISR Attributes

Attribute Name		Values Default value is written in bold	Description
OIL	XML		
ExceptionType	OsOSExceptionType	GENERAL_EXCEPTION **EIINT**	Select EIINT for EI level interrupts. Select GENERAL_EXCEPTION for core exceptions.
EnableNesting	OsOSEnableNesting	TRUE FALSE	Must be set FALSE if the configured CAT2 ISR shall not be interrupted by CAT1 or CAT2 ISRs with higher priority level. This attribute is ignored for CAT1 ISRs.
UseSpecialFunctionName	OsOSUseSpecialFunctionName	TRUE **FALSE**	This is a feature for mapping different interrupt / exception sources to one interrupt handler. This feature is supported as described in [3].

Table 2: RH850 specific ISR attributes

3.5.1 ExceptionType Sub-Attributes

ExceptionType = GENERAL_EXCEPTION

Attribute Name		Values	Description
OIL	XML		
ExceptionAddress	OsOSExceptionAddress	**No default**	Interrupt vector address offset.

Table 3: Sub-attributes of ExceptionType=GENERAL_EXCEPTION

ExceptionType = EIINT

Attribute Name		Values	Description
OIL	XML		
IntChannel	OsOSIntChannel	0 … 511 **No default**	Channel index of EI level interrupt. Max channel number depends on used CPU derivative.
InterruptPriority	OsOSInterruptPriority	SC1 and SC3: 0…15 SC4: 1…15 **No default**	Defines the interrupt priority level of the ISR. Lower value means higher priority.
InterruptStackSize	OsOSInterruptStackSize	0 … 0xFFFC **No default**	Size of the ISR stack

Table 4: Sub-attributes of ExceptionType=EIINT

3.6 Application Attributes

The following specific attributes are provided for SC3 and SC4:

Attribute Name		Values Default value is written in bold	Description
OIL	XML		
ASID	OsApplicationASID	0 … 0x3FE **0x3FF**	Value to be written to ASID register on application switch. The maximum value 0x3FF is used as default value.
MpuRegion	OsApplicationMpuRegion	Enable	Configures application specific dynamic MPU region

Table 5: RH850: Application specific attributes

3.6.1 Attribute MpuRegion

Attribute MpuRegion must be configured for non-trusted applications which need write access to application specific memory areas. If dynamic MPU region is enabled then the memory area is specified by following sub-attributes:

Sub-Attribute Name		**Values** (default value is written in bold)	**Description**
OIL	XML		
StartAddr	OsApplication StartAddr	string **No default**	Start address of application specific MPU region hexadecimal value: StartAddr = 0x… or application specific linker symbol: StartAddr = <appl_linker_start_symbol>
EndAddr	OsApplication EndAddr	string **No default**	End address of application specific MPU region hexadecimal value: EndAddr = 0x… or application specific linker symbol: EndAddr = <appl_linker_end_symbol>

Value of StartAddr must always point to the first valid Byte in the specified memory area.
Value of EndAddr must always point to the last valid Byte in the specified memory area.

```
MpuRegion = Enable
{
    StartAddr  = 0x... ;
    EndAddr   = 0x... ;
};
```

The total number of used dynamic MPU regions depends on the application which has the most number of dynamic regions.
Example: if an application has 3 dynamic regions and other application has 5 dynamic regions then the total number of used dynamic MPU regions is 5.

3.7 Event Attributes

Events in the MICROSAR OS operating system are always implemented as bits in bit fields. The user could use bit masks like ‘0x00000001’ but to achieve portability between different MICROSAR OS implementation he should use event names which are mapped by the code generator to the defined bits. The MICROSAR OS RH850 implementation allows up to 32 events per task. The required size of the event masks is calculated automatically by the code generator. Possible event mask sizes are 8, 16 and 32 Bits.

3.8 Linker Include Files (SC3 and SC4)

The generated linker include files osdata.dld, osrom.dld, ossdata.dld, osstacks.dld and ostdata.dld are example files which can be used for mapping the OS and application data.

osdata.dld

Include file osdata.dld contains mapping for application and OS data. It should be included immediately after the default .data section.

osrom.dld

Include osrom.dld contains the mapping of initialized data which is copied by the start-up code from ROM to RAM area. It should be included at end of ROM section.

ossdata.dld

Include file ossdata.dld contains the mapping of application and OS data in SDA section. Due to limited number of MPU protection areas the SDA section of non-trusted application contain SDA and non-SDA data. This affects also the global shared data sections. This file must be included after the .sdata section.

osstacks.dld

Include file osstacks.dld contains the mapping of system stack, all task and all ISR stacks. This file should be included before application specific data sections.

ostdata.dld

Include file ostdata.dld contains mapping for application data in TDA section. This file should be included after the .zdata section.

4 System Generation

The system generation process is described in the document [3] which is common to all implementation. The following section describes the RH850 specific parts of the generating process.

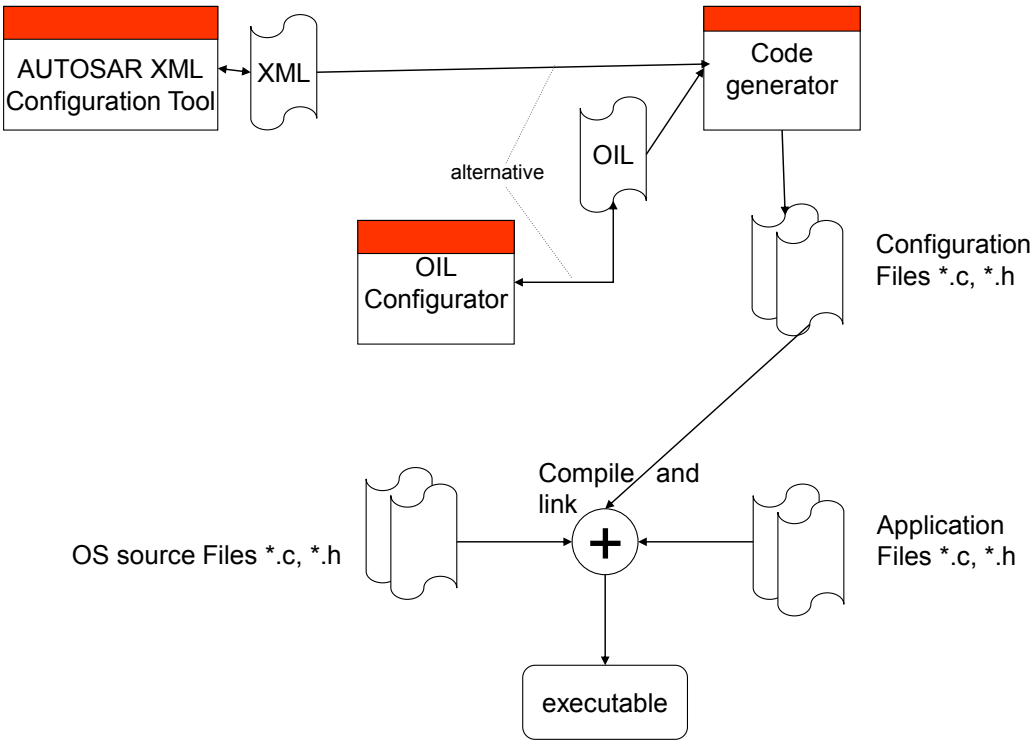

Figure 4-1 System Generation

4.1 Code Generator

The following files are generated by the code generator genRH850.exe

- config.xml configuration information in XML-format
- intvect_c0.c interrupt vector tables
- tcb.c applications and OS configuration
- tcb.h
- tcbpost.h
- trustfct.h
- trustfct.c trusted function stubs
- osConfigBlock.c contains the configuration block
- osStacks.h
- osStacks.c stack definitions
- Os_MemMap.h contains definitions for MemMap usage
- OILFileName.ort
- OILFileName.htm

For SC3 and SC4 the following additional files are generated:

- osdata.dld example linker include file for application data
- osrom.dld example linker include file for application data initialization
- ossdata.dld example linker include file for application data in SDA
- osstacks.dld example linker include file for stacks
- ostdata.dld example linker include file for application data in TDA
- nontrustfct.h

For multicore systems the following additional files are generated:

- ioc.h Header for IOC related functions
- ioc.c IOC related functions
- ccb.h Header for multicore related attributes
- ccb.c Multicore related attributes
- intvect_c1.c Interrupt vector tables for second core

The files tcb.c, tcb.h, trustfct.c, trustfct.h and nontrustfct.h are described in document [3].

The module intvect_c<X>.c contains the interrupt vector tables for the Green Hills compiler, where X is the corresponding logical core ID.

The module *OILFileNam*e.ort is only generated if the configuration attribute O*RTIDebugSupport* is selected.

*OILFileNam*e.htm is containing information about the configuration settings.

5 Stack Handling

5.1 Task Stacks

Each task runs on an own separate task stack. Tasks might share a stack as described in [3]. The PreTaskHook function always uses the system stack. The PostTaskHook function uses the system stack or in case of a task termination, it uses the task stack of the task that is terminated. Note that all task stacks must provide sufficient size for the maximum nesting level of ISR category 1. Therefore it is recommended to use ISR category 2 if possible.

The size of each task stack is determined by the configuration attribute StackSize of the configuration object TASK.

5.2 ISR Stacks

An interrupt stack is defined for each interrupt priority level which is assigned to a CAT2 ISR. Each CAT2 ISR runs on the interrupt stack which is assigned to its priority level.

5.3 System Stack

The system stack is used by the dispatcher and by the hook wrappers.

5.4 Startup Stack

In function osStartOSasm the stack pointer is initialized to point to the system stack.

The system stack can be used as start-up stack.

The following linker symbol is provided for each core X to use the system stack as start-up stack:

_osSystemStack_EndAddr_c<X>

5.5 Stack Usage Size

The OS offers the possibility to determine the maximum stack usage of the OS stacks.

Please be aware, that the function stack usage determination for system stacks depends heavily on the positions in the code, where an ISR is interrupted by another ISR.

In the single stack model, the measured value for the system stack usage also depends heavily on the position in the code, where a basic task is preempted by another task. For this reason, it is extremely difficult, to find a conclusion for the worst case system stack usage from the measured stack usage.

NOTE: The API functions for determination of the stack usage are only available with the following configuration settings:

STACKMONITORING = TRUE
StackUsageMeasurement = TRUE

5.5.1 Task Stack Usage

API function *osGetStackUsage* is described in [3].

5.5.2 System Stack Usage

The usage of the system stack since the start of the OS can be determined by using the function *osGetSystemStackUsage*.

Prototype:

```
typedef osuint16 osStackUsageType;

osStackUsageType osGetSystemStackUsage(void);
```

Argument: none

Return value: Maximum stack usage (bytes) of the system stack since StartOS().

5.5.3 ISR Stack Usage

The usage of the ISR stacks since the start of the OS can be determined by using the function *osGetISRStackUsage*.

Prototype:

```
typedef osuint16 osStackUsageType;

osStackUsageType osGetISRStackUsage(ISRType IsrIndex);
```

Argument: Index of the ISR

Return value: Maximum stack usage (bytes) of the ISR stack since StartOS().

6 Interrupt Handling

Note
This implementation supports interrupt handling according to the RH850 “expanded specifications” with a separate vector table for EIINT interrupts.

6.1 Interrupt Vectors

The code generator generates the interrupt vector tables. Therefore, all interrupt-service-routines have to be defined in the configuration. The interrupt vector tables are generated into the file intvect_c<X>.c for the Green Hills compiler, where X is the corresponding logical core ID.

The interrupt vector tables are generated into three sections:

- `". osExceptionVectorTable_c<X>"`
  contains the core exception vector table
- `". osEIINTVectorTable_c<X>"`
  contains the EIINT exceptions vector table
- `".os_text"`
  Contains the ISR prologue for CAT2 ISR wrappers

The length of the mentioned vector tables depends on the concrete derivative.

6.1.1 Reset Vector

The user can specify an own reset vector. For this the user must configure a category 1 ISR which has the vector address 0x0, i.e. attribute ExceptionAddress=0x0.

If a category 1 ISR is configured for vector 0x0, this vector is generated as a jump to the specified address.

If this vector is not configured in the configuration, the vector 0x0 is generated as a jump to the startup code, delivered with the compiler.

Please note that all examples use the start-up module which is delivered with the compiler. For the Green Hills compiler this module is crt0.o

6.1.2 Level Initialization

The user has to configure an interrupt priority level for all ISRs. MICROSAR OS initializes the appropriate interrupt control register with the level, configured by the user. The application is not allowed to change the interrupt priority level after StartOS is called.

6.2 Interrupt Level and Category

The RH850 controllers support interrupt levels. ISRs with lower level have priority over those with higher level. ISRs of lower level might therefore be nested into ISRs of higher level. Interrupt levels cannot be chosen independently from the category.

Because ISRs of category 2 can activate tasks, the exit code of a non-nested category 2 ISR has to check, if a task has been activated by the ISR itself or by any nested ISRs. If so, a task switch has to be set off. As category 1 ISRs have no such exit code, they must not be interrupted by category 2 ISRs. For this reason, MICROSAR OS checks, that all category 1 ISRs have lower or equal level value than the category 2 ISR with lowest level.

Note: Scalability class SC4 does not allow use of category 2 ISRs with priority level 0

6.3 Interrupt Category 1

For interrupts of category 1 no MICROSAR OS API functions can be used. Note that the category 1 ISR with highest priority value must have lower or equal priority value than the category 2 ISR with lowest priority value.

Example: if category 2 ISRs have priority levels 8, 10, 11 and 12 then category 1 cannot use priority levels 8 ... 15. Category 1 ISRs must then use priority levels 0 ... 7

Note: Scalability class SC4 does not allow use of category 1 ISRs for EIINT exceptions

6.3.1 Interrupt Processing in C

Using the Green Hills compiler, category 1 interrupt functions written in C must be implemented as described in the compiler manual:

For EIINT, you can use either of the following methods to declare a function as an interrupt routine:

- *Place #pragma ghs interrupt immediately before the function.*
- *Prepend the __interrupt keyword to the function definition.*

Independently from the compiler, the user should not provide the interrupt vector number to the compiler, as the compiler would generate an interrupt vector in this case. Interrupt vector generation is always done by the operating system.

6.3.2 Unhandled Exception Determination

If an unexpected interrupt occurs which is caused by an unused interrupts source then the ErrorHook and ShutdownHook are called and the system shutdown is requested.

The error type is reported to the application via error code E_OS_SYS_ABORT.

If an unhandled core exception has occurred then the error reason is reported to the application via error number osdErrUEUnhandledCoreException. The global variable ossUnhandledCoreExceptionDetail contains the content of register FEIC and the global variable ossUnhandledExceptionDetail contains the offset of the core exception.

If an unhandled EIINT exception has occurred then the error reason is reported to the application via error number osdErrUEUnhandledEIINTException. The global variable ossUnhandledEIINTDetail contains the content of register EIIC and the global variable ossUnhandledExceptionDetail contains the index of the EIINT exception.

6.4 Interrupt Category 2

6.4.1 Interrupt Entry

The entry code of category 2 ISRs is automatically generated by MICROSAR OS. This code stores the GPR registers onto stack and calls the CAT2 ISR wrapper.

SC1 and SC3: The CAT2 ISR wrapper clears the global interrupt disable flag if the attribute *EnableNesting* is set for this ISR before calling the corresponding ISR function. If this attribute is not set then the CAT2 ISR wrapper does not clear the global interrupt disable flag.

SC4: The CAT2 ISR wrapper sets register PMR to system level and clears the global interrupt disable flag if the attribute *EnableNesting* is set for this ISR before calling the corresponding ISR function. If this attribute is not set then the CAT2 ISR wrapper sets register PMR to task level and then clears the global interrupt disable.

6.4.2 Interrupt Exit

The necessary return from exception is implemented in the CAT2 ISR wrapper exit code. The application is not allowed to issue a return from exception instruction.

6.4.3 CAT2 ISR Function

For category 2 ISRs, the user has to use the ISR-macro provided by MICROSAR OS. The name which is given to this macro must be identical to the name in the configuration.

```
ISR( myISR )
{
   … /* ISR function body */
}
```

6.5 Disabling Interrupts

Caution

It is not allowed to disable interrupts longer than the tick time `SECONDSPERTICK` of the hardware counter (timer). If interrupts are disabled longer than the tick time the alarm management could be handled wrong. This error is *not* detected by the operating system.

Only when compiling the operating system with the extended status additional error checking is performed.

7 MPU Handling (SC3 and SC4)

7.1 MPU Region Usage

MPU region 0 is used for stack area protection and all other MPU regions can be configured by the user to be static or dynamic (reprogrammed).

The total number of available MPU regions depends on the derivative group:

Derivative group	Number of MPU regions (per Core)
D1L and D1M	12
E1L and E1M	12
E1x-FCC2	16
F1H and F1M	16
F1L	4
P1M	12

7.1.1 MPU Region 0

MPU region 0 is used for stack area protection and cannot be configured by the user. It is always reprogrammed by the OS when the context is switched. Therefore trusted and non-trusted applications can only write to the task and ISR stacks which belong to the application.

7.1.2 Static MPU Regions

If a MPU region shall be static, i.e. it is only initialized in StartOS and not reprogrammed when context is switched then it has to be configured in the OS specific section. The region number is not required. The OS generator assigns a region number for each configured static MPU region.

The user must specify start address, end address and region attributes of the memory area. Furthermore, for Multicore OS the core assignment has to be specified.

If a MPU region is configured to be static then it is initialized in StartOS with settings from the configuration block.

7.1.3 Dynamic MPU Regions

If a MPU region shall be dynamic, i.e. it is always reprogrammed when context is switched then it has to be configured in each corresponding non-trusted application section. The region number is not required. The OS generator assigns a region number for each configured dynamic MPU region. Trusted applications cannot be configured with MPU regions.

The user must specify start and end address of the memory area. The region attributes are configured by the OS.

If a MPU region is configured to be dynamic then it is initialized in StartOS to be unused. After StartOS it is always reprogrammed when a non-trusted application is started or terminated.

Non-trusted applications can have different number of MPU regions. The total number of reprogrammed MPU regions depends on the application which has the most dynamic regions on the corresponding core. Example: If a system has 2 non-trusted applications on the same core, one application configured with 2 MPU regions and the other application configured with 3 MPU regions, then the total number of dynamic MPU regions is 3. During runtime when context is switched then always 3 MPU regions are reprogrammed.

Non-trusted application Appl1:
- MPU region 1 used
- MPU region 2 used

Non-trusted application Appl2:
- MPU region 1 used
- MPU region 2 used
- MPU region 3 used

When application Appl1 is started then MPU regions 1 and 2 are reprogrammed with application specific settings and MPU region 3 is set to unused.
When application Appl2 is started then MPU regions 1, 2 and 3 are reprogrammed with application specific settings.

8 RH850 Peripherals

8.1 Supported System Timer

MICROSAR OS RH850 uses the timer unit OSTM as driver for a configured hardware counter (see 3.4). The corresponding interrupt channel depends on the derivative group:

Derivative group	Used Interrupt Channel	Used Hardware Unit
D1L / D1M	125	OSTM0
E1L / E1M	25	OSTM0
E1x-FCC2	25 26 for second core (Multicore OS)	OSTM0 OSTM1 for second core (Multicore OS)
F1M	84	OSTM0
F1H	84 314 for second core (Multicore OS)	OSTM0 OSTM5 for second core (Multicore OS)
F1L	76	OSTM0
P1M	74	OSTM0

Table 3 RH850 driver for hardware counter

8.2 Supported Time Monitoring Timer

MICROSAR OS RH850 uses the timer unit TAUJ0 for SC4 timing protection:

Derivative group	Used Interrupt Channels	Used Hardware Unit
D1L / D1M	121, 122, 123	TAUJ0
E1L / E1M	Timing Protection not supported	-
F1H / F1M	80, 81, 82	TAUJ0
F1L	72, 73, 74	TAUJ0
P1M	133, 134, 135	TAUJ0

Table 4 RH850 timer units for timing protection

8.3 Initialization

The OS initializes the interrupt controller INTC before the StartupHook is called.

Register SCBASE and SCCFG are initialized to use the OS syscall table (SC3 and SC4).

The OS initializes the timer OSTM after the StartupHook is called.

In case of SC3 and SC4, the OS initializes the memory protection unit MPU before the StartupHook is called.

In case of SC4, the OS initializes the timer unit TAUJ0 before the StartupHook is called.

9 Implementation Specifics

9.1 API Functions

9.1.1 DisableAllInterrupts

SC1 and SC3: The function *DisableAllInterrupts* disables all interrupts. This is achieved by setting the ID-Bit in register PSW. The old value of the ID-Bit is stored for later restore.

SC4: The function *DisableAllInterrupts* disables interrupts up to priority level 1. Interrupts with priority level 0 stay enabled. This is performed by setting bits 1…15 in register PMR. The old value of the register is stored for later restore. Interrupts of priority level 0 stay enabled so that timing protection exceptions can still occur.

Remark: Nested calls are **not** possible.

9.1.2 EnableAllInterrupts

SC1 and SC3: The function *EnableAllInterrupts* restores the content of the ID-Bit which was stored by *DisableAllInterrupts*.

SC4: The function *EnableAllInterrupts* restores the content of register PMR which was stored by *DisableAllInterrupts*.

Remark: Nested calls are **not** possible.

9.1.3 SuspendAllInterrupts

SC1 and SC3: The function *SuspendAllInterrupts* disables all interrupts. This is achieved by setting the ID-Bit in register PSW. The old value of the ID-Bit is stored for later restore.

SC4: The function *SuspendAllInterrupts* disables interrupts up to priority level 1. Interrupts with priority level 0 stay enabled. This is performed by setting bits 1…15 in register PMR. The old value of the register is stored for later restore. Interrupts of priority level 0 stay enabled so that timing protection exceptions can still occur.

Remark: Nested calls are possible.

9.1.4 ResumeAllInterrupts

SC1 and SC3: The function *ResumeAllInterrupts* restores the content of the ID-Bit which was stored by *SuspendAllInterrupts*.

SC4: The function *ResumeAllInterrupts* restores the content of register PMR which was stored by *SuspendAllInterrupts*.

Remark: Nested calls are possible.

9.1.5 SuspendOSInterrupts

The function *SuspendOSInterrupts* disables all category 2 interrupts. This is achieved by setting the PMR register to system level (highest interrupt priority of all category 2 interrupts). The old value of the PMR register is stored for later restore.

Remark: Nested calls are possible.

9.1.6 ResumeOSInterrupts

The function *ResumeOSInterrupts* restores the content of register PMR which was stored by *SuspendOSInterrupts*.

Remark: Nested calls are possible.

Note

If OS API functions are used before StartOS then osInitialize must be called before any OS API function is called. If SC3 or SC4 is used, then also osInitINTC must be called before StartOS.

9.1.7 GetResource

MICROSAR OS RH850 SC3/SC4 does not support the extension of the resource concept for interrupt levels.

9.1.8 ReleaseResource

MICROSAR OS RH850 SC3/SC4 does not support the extension of the resource concept for interrupt levels.

9.1.9 GetAlarmBase

MICROSAR OS RH850 SC3/SC4 does not support API function GetAlarmBase.

9.1.10 osInitialize

Function osInitialize is used for initializing OS global variables which are used by osDispatcher and interrupt handling API. It is called by the OS in StartOS.

Prototype: void osInitialize(void)

9.1.11 osInitINTC

Function osInitINTC initializes the interrupt controller INTC and some core registers for the corresponding core X:

- EBASE = &osExceptionVectorTable_c<X>
- INTBP = &osEIINTVectorTable_c<X>
- INTCFG = 0
- SCBP = &osSysCallTable_c<X> (SC3 and SC4)
- SCCFG = osdNumberOfSysCallFunctions (SC3 and SC4)
- set table mode and priority level for each configured EIINT interrupt source assigned to core X

osInitINTC is called by the OS in StartOS.

Prototype: void osInitINTC(void)

9.2 Peripheral Region API

In a safety application there is the need to access peripheral components from QM software (non-trusted).

For QM software, which runs in restricted mode (e.g. user mode) the peripheral access must be granted by the MPU. Sometimes there are peripheral registers which cannot be written at all in restricted mode.

Therefore the OS offers the concept of the peripheral region API.

The peripheral regions are defined in the configuration. Access rights are also configured on application level.

With an API any Software (also non trusted) is capable to write to peripheral registers even if the access is not granted by the MPU.

9.2.1.1 Read Functions

There are three reading functions.

Prototype	

`osuint8 osReadPeripheral8 ( osuint16 area, osuint32 address )`
`osuint16 osReadPeripheral16( osuint16 area, osuint32 address )`
`osuint32 osReadPeripheral32( osuint16 area, osuint32 address )`	

Parameter	
`area`	Identifier of peripheral regions to the read from
`address`	Address to be read from
Return code	
	The content of “address” interpreted as 8 bit, 16 bit or 32 bit value
Functional Description	
> reads either an 8 bit, or a 16 bit or a 32 bit value from “address” > The function performs accessing checks (whether the caller has accessing rights to the peripheral region and whether the address to be read from is within the configured range of the peripheral region) > The error hook is raised in case of an error > A shutdown is not issued in case of an error	
Particularities and Limitations	
> These functions may not be called from OS hooks	
Call context	
> These functions may be called from Task context > These functions may be called from category 2 ISR context > These functions can be called with interrupts enabled or with interrupts disabled	

9.2.1.2 Write Functions

There are three writing functions.

Prototype	

`void osWritePeripheral8 ( osuint16 area, osuint32 address, osuint8 value)`
`void osWritePeripheral16( osuint16 area, osuint32 address, osuint16 value)`
`void osWritePeripheral32( osuint16 area, osuint32 address, osuint32 value)`	

Parameter	
`area`	Identifier of peripheral regions to the read from
`address`	Address to write to
`value`	Value to be written
Return code	
`None`	
Functional Description	
> Writes to either an 8 bit, or a 16 bit or a 32 bit value > The function performs accessing checks (whether the caller has accessing rights to the peripheral region and whether the address to be read from is within the configured range of the peripheral region) > The error hook is raised in case of an error > A shutdown is not issued in case of an error	
Particularities and Limitations	
> These functions may not be called from OS hooks	
Call context	
> These functions may be called from Task context > These functions may be called from category 2 ISR context > These functions can be called with interrupts enabled or with interrupts disabled	

9.2.1.3 Modify Functions

There are three modifying functions.

Prototype	

`void osModifyPeripheral8 ( osuint16 area, osuint32 address, osuint8 clearmask, osuint8 setmask)`
`void osModifyPeripheral16( osuint16 area, osuint32 address, osuint16 clearmask, osuint16 setmask)`
`void osModifyPeripheral32( osuint16 area, osuint32 address, osuint32 clearmask, osuint32 setmask)`	

Parameter	
`area`	Identifier of peripheral regions to the read from
`address`	Address to be modified
`clearmask`	Bitmask which is bitwise “ANDed” to “address”
`setmask`	Bitmask which is bitwise “ORed” to “address”
Return code	
`None`	
Functional Description	
> The function performs accessing checks (whether the caller has accessing rights to the peripheral region and whether the address to be read from is within the configured range of the peripheral region) > The error hook is raised in case of an error > A shutdown is not issued in case of an error > After the access checks has passed first the “clearmask” is ANDed to “address” and afterwards the “setmask” is ORed to it.	
Particularities and Limitations	
> These functions may not be called from OS hooks	
Call context	
> These functions may be called from Task context > These functions may be called from category 2 ISR context > These functions can be called with interrupts enabled or with interrupts disabled	

9.3 Peripheral Interrupt API Functions (SC3 and SC4)

The OS provides functions which can be used to perform write access to the interrupt controller INTC control registers in user mode.

If read access to SFR registers is enabled in user mode then the following macros can be used to read from interrupt controller INTC registers.

```
#define osReadICR8(addr)    (*((osuint8*)(addr)))
#define osReadICR16(addr)   (*((osuint16*)(addr)))
```

This chapter describes API functions which can be used to access the interrupt control registers within the corresponding address range:

Derivative group	Address range
D1L/D1M	FFFE EA00 - FFFE EA3E (EIC0 to EIC31) FFFF B040 - FFFF B1FE (EIC32 to EIC255)
E1L/E1M	FFFE EA00 - FFFE EA3E (EIC0 to EIC31) FFFF B040 - FFFF B3FE (EIC32 to EIC511)
E1x-Fcc2	FFFE EA00 - FFFE EA3E (EIC0 to EIC31) FFFF B040 - FFFF B3FE (EIC32 to EIC511)
F1H	FFFE EA00 - FFFE EA3E (EIC0 to EIC31) FFFF B040 - FFFF B2BC (EIC32 to EIC350)
F1L	FFFF 9000 - FFFF 903E (EIC0- EIC31) FFFF A040 - FFFF A232 (EIC32- EIC281)
F1M	FFFE EA00 - FFFE EA3E (EIC0 to EIC31) FFFF B040 - FFFF B25C (EIC32 to EIC297)
P1M	FFFE EA00 - FFFE EA3E (EIC0 to EIC31) FFFF B040 - FFFF B2FE (EIC32 to EIC383)

9.3.1 Write to Interrupt Control Register

Writing 1 or 2 Byte to the interrupt controller INTC control register is achieved by osWriteICR8/osWriteICR16 and osWriteICRxLo/ osWriteICRxHi/ osWriteICRx16:

osWriteICR8

This function writes 1 Byte at specified destination address. Before writing the address parameter is checked to be in valid range.

Prototype	
`void osWriteICR8(uint32 addr, uint8 val);`	
Parameter	
`addr` `val`	destination address value to be written at destination address
Return Code	
`void`	-
Functional Description	
`*(uint8*)addr = (uint8)val;`	

osWriteICR16

This function writes 2 Bytes at specified destination address. Before writing the address parameter is checked to be in valid range.

Prototype	
`void osWriteICR16(uint32 addr, uint16 val);`	
Parameter	
`addr` `val`	destination address value to be written at destination address
Return Code	
`void`	-
Functional Description	
`*(uint16*)addr = (uint16)val;`	

osWriteICRxLo

This function writes the lower Byte of the control register of the specified interrupt number. Before writing the index is checked to be in valid range.

Prototype	
`void osWriteICRxLo(uint32 index, uint8 val);`	
Parameter	
`index` `val`	interrupt number value to be written to control register
Return Code	
`void`	-
Functional Description	
`*(uint8*)addr = (uint8)val;`	

osWriteICRxHi

This function writes the upper Byte of the control register of the specified interrupt number. Before writing the index is checked to be in valid range.

Prototype	
`void osWriteICRxHi(uint32 index, uint8 val);`	
Parameter	
`index` `val`	interrupt number value to be written to control register
Return Code	
`void`	-
Functional Description	
`*(uint8*)addr = (uint8)val;`	

osWriteICRx16

This function writes both Bytes of the control register of the specified interrupt number. Before writing the index is checked to be in valid range.

Prototype	
`void osWriteICRx16(uint32 index, uint16 val);`	
Parameter	
`index` `val`	interrupt number value to be written to control register
Return Code	
`void`	-
Functional Description	
`*(uint8*)addr = (uint8)val;`	

9.3.2 Set or Clear Mask Flag

The mask flag of the interrupt control register can be set or cleared by osSetICRMask and osClearICRMask.

osSetICRMask

This function sets the mask flag at specified destination address which must be even. Before writing the address parameter is checked to be in valid range. The address parameter is not checked to be even. The user must take care about it.

Prototype	
`void osSetICRMask(uint32 addr);`	
Parameter	
`addr`	destination address
Return Code	
`void`	-
Functional Description	
`*(uint8*)addr |= (uint8)0x80;`	

osClearICRMask

This function clears the mask flag at specified destination address which must be even. Before writing the address parameter is checked to be in valid range. The address parameter is not checked to be even. The user must take care about it.

Prototype	
`void osClearICRMask(uint32 addr);`	
Parameter	
`addr`	destination address
Return Code	
`void`	-
Functional Description	
`*(uint8*)addr &= (uint8)0x7F;`	

9.3.3 Set or Clear ICR Request Flag

The request flag of the interrupt control register can be set and cleared by calling the functions osSetICRReq and osClearICRReq:

osSetICRReq

This function sets the interrupt request flag at specified destination address. Before writing the address parameter is checked to be in valid range. The destination address is automatically made an odd address by the function.

Prototype	
`void osSetICRReq(uint32 addr);`	
Parameter	
`addr`	destination address
Return Code	
`void`	-
Functional Description	
`*(uint8*)(addr | 0x01) |= (uint8)0x10;`	

osClearICRReq

This function clears the interrupt request flag at specified destination address. Before writing the address parameter is checked to be in valid range. The destination address is automatically made an odd address by the function.

Prototype	
`void osClearICRReq(uint32 addr);`	
Parameter	
`addr`	destination address
Return Code	
`void`	-
Functional Description	
`*(uint8*)(addr | 0x01) &= (uint8)0xEF;`	

9.3.4 Read, Set or Clear Mask Bit in Registers IMRm

The mask bits in IMRm registers can be read, set and cleared by calling functions osGetIMRmEI, osSetIMRmEI and osClearIMRmEI.

osGetIMRmEI

This function returns the current value of the mask bit specified by index (interrupt number). Before read access the index parameter is checked to be in the valid range.

Prototype	
`void osGetIMRmEI(uint16 index);`	
Parameter	
`index`	mask register index
Return Code	
`uint8`	return mask flag IMRmEIMK<index>
Functional Description	
`return (uint8)(IMRmEIMK<index>);`	

osSetIMRmEI

This function sets the mask bit specified by index (interrupt numer). Before write access the index parameter is checked to be in the valid range.

Prototype	
`void osSetIMRmEI(uint16 index);`	
Parameter	
`index`	mask register index
Return Code	
`void`	-
Functional Description	
`IMRmEIMK<index> = 1;`	

osClearIMRmEI

This function clears the mask bit specified by index (interrupt number). Before write access the index parameter is checked to be in the valid range.

Prototype	
`void osClearIMRmEI(uint16 index);`	
Parameter	
`index`	mask register index
Return Code	
`void`	-
Functional Description	
`IMRmEIMK<index> = 0;`	

9.3.5 Write to Registers IMRm

The registers IMRm can be written with 1 Byte, 2 Byte and 4 Byte value by calling functions osWriteIMR8, osWriteIMR16 and osWriteIMR32.

osWriteIMR8

Function osWriteIMR8 writes 1 Byte to register IMRm specified by address. Before write access the address parameter is checked to be in the valid range.

Prototype	
`void osWriteIMR8(uint32 addr, uint8 val);`	
Parameter	
`addr` `val`	destination address value to be written at destination address
Return Code	
`void`	-
Functional Description	
`*(uint8*)addr = (uint8)val;`	

osWriteIMR16

This function writes 2 Bytes to register IMRm specified by address. Before write access the address parameter is checked to be in the valid range.

Prototype	
`void osWriteIMR16(uint32 addr, uint16 val);`	
Parameter	
`addr` `val`	destination address value to be written at destination address
Return Code	
`void`	-
Functional Description	
`*(uint16*)addr = (uint16)val;`	

osWriteIMR32

This function writes 4 Bytes to register IMRm specified by address. Before write access the address parameter is checked to be in the valid range.

Prototype	
`void osWriteIMR32(uint32 addr, uint32 val);`	
Parameter	
`addr` `val`	destination address value to be written at destination address
Return Code	
`void`	-
Functional Description	
`*(uint32*)addr = (uint32)val;`	

9.4 Hook Routines

The MICROSAR OS specification [3] allows implementation specific additional parameters in hook routines. The prototypes of the hook routines are described in [1].

The contexts where hook routines are called are implementation specific and are described below. All hook routines are called with disabled interrupts.

9.4.1 ErrorHook

The ErrorHook is called if an error is detected in an API-function and if a system error is detected. The ErrorHook runs on the task or ISR stack if an API error is reported and it runs on the system stack if an unrecoverable system error is reported. Interrupts are disabled and CPU runs in supervisor mode.

9.4.2 StartupHook

The StartupHook runs always on the system stack. Interrupts are disabled and CPU runs in supervisor mode.

9.4.3 ShutdownHook

SC1: The ShutdownHook runs on the stack of the task or ISR which has called ShutdownOS(). EI level interrupts are disabled.

SC3 and SC4: The ShutdownHook runs always on the system stack. EI level interrupts are disabled and the CPU runs in supervisor mode.

9.4.4 PreTaskHook

The PreTaskHook runs always on the system stack. Interrupts are disabled and CPU runs in supervisor mode.

9.4.5 PostTaskHook

The PostTaskHook runs on the task stack or on the system stack. Interrupts are disabled and CPU runs in supervisor mode.

9.4.6 PreAlarmHook (SC1 only)

When entering the ISR *“osTimerInterrupt”* there is a call of the PreAlarmHook when the corresponding configuration switch is set. The user has to take care for the implementation of this routine. Please implement this hook routine as follows:

```
void PreAlarmHook(void)
{
   /* user specific code */
}
```

The Hook Routine is called before incrementing the system counter and before handling all alarms! The Hook Routine runs on system stack.

9.4.7 ISRHooks (SC1 only)

The ISRHooks – UserPreISRHook and UserPostISRHook – run on the system stack. If EnableNesting is set to TRUE for the respective ISR, these hooks run with interrupts enabled; otherwise they run with interrupts disabled.. For a more detailed description of these hooks see [3].

9.4.8 Callbacks (SC1 only)

Callbacks run on the stack of the entity that led to their activation. E.g. if an alarm callback is activated through a call to IncrementCounter() by a task, it runs on this tasks stack. If IncrementCounter() was called by an interrupt (for example the system timer interrupt), the Callback runs onthe corresponding ISR stack.

9.4.9 ProtectionHook (SC3 and SC4)

The ProtectionHook is called if a memory protection or privileged instruction violation was detected. It runs always on the system stack. Interrupts are disabled and CPU runs in supervisor mode.

9.5 Functions for MPU functionality checks

The OS provides 2 functions which can be called to check the functionality of the MPU.

9.5.1 Function osCheckMPUAccess

Function osCheckMPUAccess can be called to check the current MPU protection settings.

Prototype	
`uint8 osCheckMPUAccess(const uint8* addr)`	
Parameter	
`addr`	The address to be checked for read and write access
Return Type	
`uint8`	0 = E_OK: read and write access to given address is possible 1 = E_OS_ACCESS: access to given address has caused memory protection violation
Functional Description	
• disable interrupts • call internal function osAsmCheckMPU to check read and write access to destination address • store the return value of internal function osAsmCheckMPU into local variable • restore previous interrupt state • return with given return value	
Particularities and Limitations	
The internal function osAsmCheckMPU first reads 1 byte from the destination address and then writes this value back to the destination address. If read and write access is possible the value on the given destination address is not changed. If read or write access is not possible then an access violation is detected by the MPU and a protection exception occurs. The protection exception handler returns to internal function osAsmCheckMPU with return value = 1. This return value is returned to the caller of function osCheckMPUAccess to signal the access violation.	
Call Context	
Not allowed before call of StartOS()	

Table 5: Function osCheckMPUAccess

9.5.2 Function osCheckAndRefreshMPU

Function osCheckAndRefreshMPU can be called to check and re-initialize all MPU registers which are not reprogrammed during runtime.

Prototype		
`StatusType osCheckAndRefreshMPU(void)`		
Parameter		
-	-	
Return Type		
`StatusType`	E_OK:	all MPU registers have expected content
	E_OS_SYS_API_ERROR:	invalid content was detected in MPU registers
Functional Description		
• checks all MPU registers which are not reprogrammed • re-initialize all MPU registers which are not reprogrammed • returns E_OK if all MPU registers have expected content • returns E_OS_SYS_API_ERROR if invalid content was detected in MPU registers		
Particularities and Limitations		
-		
Call Context		
▪ Call is only allowed after StartOS() ▪ Call is only allowed by trusted applications		

Table 6: Function osCheckAndRefreshMPU

9.6 Function for OSTM functionality checks

9.6.1 Function osCheckAndRefreshTimer

Function osCheckAndRefreshTimer can be called by trusted applications at any time after StartOS to check and re-initialize the register settings of system timer OSTM.

Prototype		
`StatusType osCheckAndRefreshTimer(void)`		
Parameter		
-	**-**	
Return Type		
`StatusType`	E_OK:	all registers of OSTM have expected content
	E_OS_SYS_API_ERROR:	invalid content was detected in OSTM register
Functional Description		
• check content of all used OSTM registers • re-initialize all OSTM registers which have wrong content • returns E_OK if all OSTM registers have expected content • returns E_OS_SYS_API_ERROR if invalid content was detected in OSTM registers		
Particularities and Limitations		
-		
Call Context		
▪ Call is only allowed after StartOS() ▪ Call is only allowed by trusted applications		

Table 7: Function osCheckAndRefreshTimer

10 Non-Trusted Functions (SC3 and SC4)

Non-trusted functions are a VECTOR extension to the AUTOSAR OS specification. This concept allows non-trusted applications to provide interface functions which might be called by trusted or non-trusted tasks and ISRs.

10.1 Functionality

Non-trusted functions can be used to provide service functions by non-trusted applications. Non-trusted functions are called with memory access rights and service protection rights of the owner application, independent from the access rights of the caller. These functions can access local data of the owner application without the possibility to overwrite data of other applications.

The caller might be a task of a trusted application with global memory access, developed according to high safety standards. The called non-trusted function might be developed according to lower standards but is not able to access any other memory than limited accessible memory of the non-trusted owner application. During the call, the non-trusted function is executed on a separate stack, isolated from the caller stack by the MPU.

10.2 API

Prototype	
`StatusType osCallNonTrustedFunction(NonTrustedFunctionIndexType FunctionIndex, NonTrustedFunctionParameterRefType FunctionParams);`	
Parameter	
FunctionIndex	Index of the function to be called.
FunctionParams	Pointer to the parameters for the function to be called. If no parameters are provided, a NULL pointer has to be passed.
Return code	
`E_OK`	No error
`E_OS_SERVICEID`	No function defined for this index
Functional Description	
Executes the non-trusted function referenced by FunctionIndex and passes argument FunctionParams. The non-trusted function must conform to the following C prototype: `void NONTRUSTED_<name of the non-trusted function(NonTrustedFunctionIndexType, NonTrustedFunctionParameterRefType);` The arguments are the same as the arguments of CallNonTrustedFunction.	
Particularities and Limitations	
> The non-trusted function is called in user mode with memory protection enabled > The function has memory access rights of the owner application > The function has the service protection rights of the owner application	
Call context	
> Task, CAT2 ISR, trusted function, non-trusted function	

Table 8 API CallNonTrustedFunction

Note
Vector MICROSAR OS implementations offer the possibility of stub function generation for trusted functions. This mechanism is **not** available for non-trusted functions.

10.3 Call Context

The following table shows the API functions callable from non-trusted functions.

API Functions	Call allowed	API Functions	Call allowed
ActivateTask	X	GetTaskID	X
TerminateTask	DC	GetTaskState	X
ChainTask	DC	StartOS	-
Schedule	DC	ShutdownOS	-
DisableAllInterrupts	P	GetApplicationID	X
EnableAllInterrupts	P	GetActiveApplicationMode	X
SuspendAllInterrupts	P	GetISRID	X
ResumeAllInterrupts	P	CallTrustedFunction	X
SuspendOSInterrupts	P	CallNonTrustedFunction	X
ResumeOSInterrupts	P	CheckObjectAccess	X
GetResource	X	CheckObjectOwnership	X
ReleaseResource	X	StartScheduleTableRel	X
SetEvent	X	StartScheduleTableAbs	X
ClearEvent	DC	StopScheduleTable	X
GetEvent	X	NextScheduleTable	X
WaitEvent	DC	StartScheduleTableSynchron	X
GetAlarm	X	SyncScheduleTable	X
SetRelAlarm	X	GetScheduleTableStatus	X
SetAbsAlarm	X	SetScheduleTableAsync	X
CancelAlarm	X	IncrementCounter	X

Abbreviations:

X: Allowed

DC: Dependent on caller. Allowed if called from task, not allowed from ISR

P: Pairwise, when interrupts are disabled within the (trusted/non-trusted) function, they need to be re-enabled within the same function

10.3.1 Example

Non-trusted functions have to be defined and called as followed:

Example for calling a non-trusted function (configured function name = MyNTF used as index number):

```
TASK(Task1)
{
   MyNTFParametersStruct callArg;
   callArg.a = 2;
   CallNonTrustedFunction( MyNTF, (NonTrustedFunctionParameterRefType)
(&callArg));
}
```

Definition and prototype of the non-trusted function must have the prefix **NONTRUSTED_** :

```
void NONTRUSTED_MyNTF (NonTrustedFunctionIndexType idx,
NonTrustedFunctionParameterRefType  param)
{
    if( ((MyNTFParametersStruct *) param)->a == 2)
    {
         /* do something */
    }
}
```

The non-trusted function parameters must be declared via typedef struct:

```
typedef struct
{
     unsigned char a;
} MyNTFParametersStruct;
```

11 Multicore

This OS implements the Autosar OS Multi Core feature according to Autosar specification 4.0.3.

11.1 Configuration

Each application has to be assigned to a core. This is done with the application attribute “CORE”.

Since each configuration object has to be assigned to an application the core assignment of the object is implicitly done with the application assignment.

11.1.1 Core IDs

The physical core ID which is provided in hardware register HTCFG0 differs from the logical core ID used by the OS internally, which is returned by `GetCoreID()`.

	CPU1 (PE1)	CPU2 (PE2)
Physical core ID	1	2
Logical core ID	0	1

Table 9: Mapping of physical and logical core ID

11.2 Multi-Core start-up

As immediately after reset both cores begin execution, the master-slave startup behavior is emulated in software. Therefore, a handshake synchronization is performed by calling `osInitMultiCoreOS()` on PE1 and calling `osInitSlaveCore()` on PE2. It is not allowed to use any other API function before this initial synchronization step is done.

By calling `osInitMultiCoreOS()` PE1 initializes the multi-core OS related variables and synchronizes with PE2. By calling `osInitSlaveCore()` PE2 synchronizes with PE1 and resides in a busy waiting state, until it is started by `StartCore()` or `StartNonAutosarCore()`. If only PE2 should be controlled by the OS, then PE2 has to call `osInitMultiCoreOS()` after `osInitSlaveCore()`.

Caution

The hardware register **`OSTM0_CMP`** is used for synchronization purpose.

Therefore, **`OSTM0_CMP`** must not be modified before initial synchronization.

The following chapters provide code examples for the multi-core startup.

11.2.1 Both PEs controlled by OS

```
void main()
{
    […]
    switch(GetCoreID())
    {
         case OS_CORE_ID_0:
              […]
              osInitMultiCoreOS();
              StartCore(OS_CORE_ID_1);
              StartOS(OSDEFAULTAPPMODE);
              break;
         case OS_CORE_ID_1:
              […]
              osInitSlaveCore();
              StartOS(OSDEFAULTAPPMODE);
              break;
         default:
              […]
              break;
    }
    […]
}
```

11.2.2 Only PE1 controlled by OS

```
void main()
{
    […]
    switch(GetCoreID())
    {
         case OS_CORE_ID_0:
              […]
              osInitMultiCoreOS();
              StartNonAutosarCore(OS_CORE_ID_1);    /* may be called later */
              StartOS(OSDEFAULTAPPMODE);
              break;
         case OS_CORE_ID_1:
              […]
              osInitSlaveCore();
              […]
              break;
         default:
              […]
              break;
    }
    […]
}
```

11.2.3 Only PE2 controlled by OS

```
void main()
{
    […]
    switch(GetCoreID())
    {
         case OS_CORE_ID_0:
              […]
              osInitMultiCoreOS();
              StartCore(OS_CORE_ID_1);
              […]
              break;
         case OS_CORE_ID_1:
              […]
              osInitSlaveCore();
              osInitMultiCoreOS();
              StartNonAutosarCore(OS_CORE_ID_0);    /* adjusts the state of PE1*/
              StartOS(OSDEFAULTAPPMODE);
              break;
         default:
              […]
              break;
    }
    […]
}
```

12 Timing Protection (SC4)

MICROSAR OS RH850 SC4 provides timing protection by using timer unit TAUJ0:
TAUJ0 timer channel 0 is used for inter arrival time monitoring.
TAUJ0 timer channel 1 is used for execution time monitoring.
TAUJ0 timer channel 2 is used for blocking time monitoring.
TAUJ0 timer channel 3 is not used and therefore disabled.
TAUJ0 timer channels 0, 1 and 2 are initialized with interrupt priority level 0.

12.1 Configuration Attributes

The common SC4 attributes are described in the general MICROSAR OS manual.

RH850 specific SC4 attributes

OS attribute *TimingProtectionTimerClock* must be specified in [kHz] by user:
Example: If the peripheral clock of TAUJ0 is 20 MHz then set
TimingProtectionTimerClock = 20000

12.2 Restrictions for SC4 Configurations

MICROSAR OS RH850 has the following restrictions for SC configurations:

- It is not allowed to use category 1 ISRs
- It is not allowed to configure category 2 ISRs with priority level 0
- It is not allowed to modify any register of unit TAUJ0 after calling StartOS

13 Error Handling

13.1 MICROSAR OS RH850 Error Numbers

In addition to the AUTOSAR OS error numbers all MICROSAR OS implementations provide unique error numbers for an exact error description. All error numbers are defined as a 16 bit value. The error numbers are defined in the header file osekerr.h and are defined according to the following syntax:

```
0xgfee
  ||+--- consecutive error number
  |+---- number of function in the function group
  +----- number of function group
```

Error numbers common for all MICROSAR OS implementations are described in [3].

13.1.1 RH850 specific Error Numbers

The RH850 implementation specific error numbers have a function group number starting at 0xA101 and are described in the following table:

Name	Error	Type	Reason
osdErrYOSystemStackOverflow	0xA101	SysCheck	system stack overflow is detected
osdErrYOTaskStackOverflow	0xA102	SysCheck	task stack overflow is detected
osdErrYOISRStackOverflow	0xA103	SysCheck	ISR stack overflow is detected
osdErrSCWrongSysCallParameter	0xA201	SysCheck	invalid syscall parameter is used
osdErrDPStartValidContext	0xA401	SysCheck	new task is started with valid context
osdErrDPResumeInvalidContext	0xA402	SysCheck	preempted task is resumed with invalid context
osdErrDPInvalidTaskIndex	0xA403	SysCheck	invalid active task index
osdErrDPInvalidApplicationID	0xA404	SysCheck	invalid active application ID
osdErrEXMemoryViolation	0xA501	SysCheck	memory protection violation
osdErrEXPrivilegedInstruction	0xA502	SysCheck	privileged instruction violation
osdErrSUInvalidTaskIndex	0xA601	SysCheck	invalid task index used in osGetStackUsage
osdErrSUInvalidIsrIndex	0xA602	SysCheck	invalid ISR index used in osGetISRStackUsage
osdErrSUInvalidIsrPrioLevel	0xA603	SysCheck	invalid ISR priority level used in osGetISRStackUsage
osdErrCIInvalidIsrIndex	0xA701	SysCheck	CAT2 ISR wrapper called with invalid ISR ID
osdErrCIInvalidIsrPrioLevel	0xA702	SysCheck	invalid ISR priority level used in CAT2 ISR wrapper
osdErrCIInvalidApplicationID	0xA703	SysCheck	invalid application ID used in CAT2 ISR wrapper

Name	Error	Type	Reason
osdErrCIMissingIntRequest	0xA704	SysCheck	Missing interrupt request
osdErrCIInterruptIsMasked	0xA705	SysCheck	Interrupt is masked
osdErrCIWrongIntPriority	0xA706	SysCheck	Wrong interrupt priority
osdErrPIGetIMRInvalidIndex	0xA801	SysCheck	invalid IMR index used in osGetIMR
osdErrPISetIMRInvalidIndex	0xA802	SysCheck	invalid IMR index used in osSetIMR
osdErrPIClearIMRInvalidIndex	0xA803	SysCheck	invalid IMR index used in osClearIMR
osdErrPIWriteIMR8InvalidAddr	0xA804	SysCheck	invalid IMR address used in osWriteIMR8
osdErrPIWriteIMR16InvalidAddr	0xA805	SysCheck	invalid IMR address used in osWriteIMR16
osdErrPIWriteIMR32InvalidAddr	0xA806	SysCheck	invalid IMR address used in osWriteIMR32
osdErrPISetICRMaskInvalidAddr	0xA807	SysCheck	invalid ICR address used in osSetICRMask
osdErrPIClearICRMaskInvalidAddr	0xA808	SysCheck	invalid ICR address used in osClearICRMask
osdErrPISetICRReqInvalidAddr	0xA809	SysCheck	invalid ICR address used in osSetICRReq
osdErrPIClearICRReqInvalidAddr	0xA80A	SysCheck	invalid ICR address used in osClearICRReq
osdErrPIWriteICR8InvalidAddr	0xA80B	SysCheck	invalid ICR address used in osWriteICR8
osdErrPIWriteICR16InvalidAddr	0xA80C	SysCheck	invalid ICR address used in osWriteICR16
osdErrPIWriteICRxLoInvalidIndex	0xA80D	SysCheck	invalid ICR index used in osWriteICRxLo
osdErrPIWriteICRxHiInvalidIndex	0xA80E	SysCheck	invalid ICR index used in osWriteICRxHi
osdErrPIWriteICRx16InvalidIndex	0xA80F	SysCheck	invalid ICR index used in osWriteICRx16
osdErrCRInvalidSettingOSTM	0xA901	SysCheck	invalid register content found in osCheckAndRefreshTimer
osdErrCRInvalidSettingMPU	0xA902	SysCheck	invalid register content found in osCheckAndRefreshMPU
osdErrUEUnhandledCoreException	0xAA01	SysCheck	unhandled core exception occurred
osdErrUEUnhandledDirectBranch	0xAA02	SysCheck	unhandled direct branch exception occurred

Table 10 RH850 specific Error Numbers

14 Modules

MICROSAR OS RH850 source and header files depend on the scalability class:

14.1 Source Files

Module Name	Description	SC1	SC3	SC4
atosappl.c	Memory protection related functions		●	●
atostime.c	Schedule table related functions	●	●	●
atosTProt.c	Timing protection related functions			●
osek.c	System initialisation, scheduler, interrupt control	●	●	●
osekalrm.c	Alarm related functions	●	●	●
osekasm.c	Compiler specific assembler functions and syscalls	●	●	●
osekerr.c	Error handling	●	●	●
osekevnt.c	Event handling	●	●	●
osekrsrc.c	Resource handling	●	●	●
oseksched.c	Schedule table related functions	●	●	●
osekstart.c	OS start-up related functions	●	●	●
osektask.c	Task management	●	●	●
osektime.c	Timer interrupt routine and alarm management	●	●	●
osSysCall.c	Compiler specific syscall table		●	●
osOstmHiRes.c	Timer specific functions	●	●	●
osMultiCore.c	Multicore related functions	**MC** [1]	**MC**	

Table 11 List of source files

[1] Only for multi core systems

14.2 Header Files

Module Name	Description	SC1	SC3	SC4
Os.h	This header has to be included by the application	●	●	●
Os_Cfg.h	Included by Os.h, includes the Autosar Header files if selected by the attribute TypeHeaderInclude	●	●	●
osek.h	Included by Os_cfg.h, basic OS header	●	●	●
osekasm.h	Compiler specific header for assembler code	●	●	●
osekasrt.h	Included by osek.h, macro-definitions for assertion-handling	●	●	●
osekcov.h	Coverage macros	●	●	●
osekerr.h	Included by osek.h, definitions of all error-numbers	●	●	●
osekext.h	Header file for OS internal functions	●	●	●
oseksched.h	Header for schedule table related functions	●	●	●
emptymac.h	Empty API hook macros	●	●	●
testmac1.h	User API hook macros (contains macros for ORTI debug support)	●	●	●
testmac3.h	User API hook macros (contains macros for ORTI debug support)	●		
testmac4.h	User API hook macros (contains macros for ORTI debug support)	●		
vrm.h	Included by all headers and system modules, Vector release management	●	●	●
osSysCallTable.dld	Linker specific symbols for the syscall table. This file must be included into the global project linker file.		●	●
osDerivatives.h	File for including the necessary derivative dependent header.	●	●	●
osRH850_<Derivative>.h	RH850 Derivative specific header file.	●	●	●
osINTC2.h	Contains specific code for the interrupt controller.	●	●	●
osMultiCore.h	Header for multicore related functions	**MC**	**MC**	

Table 12 List of header files

15 Contact

Visit our website for more information on

- News
- Products
- Demo software
- Support
- Training data
- Addresses

www.vector.com

In case of OSEK / MICROSAR OS related problems you may write an email to
osek-support@vector.com