RTA-OSEK

Binding Manual: V850E/IAR

Contact Details

ETAS Group **www.etasgroup.com**	**ETAS GmbH** 70469 Stuttgart, Germany Tel.:+49 711 89661-0 Fax:+49 711 89661-300 **sales.de@etas.com**
ETAS Inc. Ann Arbor, MI 48103, USA Tel.: +1 888 ETAS INC Fax: +1 734 997 9449 **sales.us@etas.com**	**ETAS S.A.S.** 94588 Rungis Cedex, France Tel.: +33 (1) 56 70 00 50 Fax: +33 (1) 56 70 00 51 **sales.fr@etas.com**
ETAS K.K. Yokohama 220-6217, Japan Tel.: +81 45 222-0900 Fax: +81 45 222-0956 **sales.jp@etas.com**	**ETAS Ltd.** Derby DE21 4SU, UK Tel.: +44 1332 253770 Fax: +44 1332 253779 **sales.uk@etas.com**
ETAS Korea Co. Ltd. Seoul 137-889, Korea Tel.: +82 2 5747-016 Fax: +82 2 5747-120 **sales.kr@etas.com**	**ETAS (Shanghai) Co., Ltd.** Shanghai 200120, P.R. China Tel.: +86 21 5037 2220 Fax: +86 21 5037 2221 **sales.cn@etas.com**
ETAS in Italy 10135 TORINO, Italy Tel.: +39 011 3285 988 Fax: +39 (011) 3285 256 **sales.it@etas.com**	**ETAS Automotive India Pvt. Ltd.** Bangalore 560 068, India Tel.: +91 80 4191 2585 Fax: +91 80 4191 2586 **sales.in@etas.com**
ETAS in Brazil CEP-05802-140 São Paulo, Brazil Tel.: +55 11 2162-0252 **sales.br@etas.com**	**ETAS in Russia** Moscow, 129515, Russia Tel.: +7 495 937 0400 998 **sales.ru@etas.com**

Copyright Notice

Disclaimer

Trademarks

Contents

1 About this Guide

This guide provides target-specific information for the V850E/IAR port of ETAS' RTA-OSEK. It supplements the more general information in the *RTA-OSEK User Guide*.

A port is defined as a specific target microcontroller/target toolchain pairing. This guide tells you about integration issues with your target toolchain and issues that you need to be aware of when using RTA-OSEK on your target hardware. Port specific parameters of implementation are also provided, giving the RAM and ROM requirements for each object in the RTA-OSEK Component and execution times for each API call to the RTA-OSEK Component.

1.1 Who Should Read this Guide?

The reader should have an understanding of real time embedded programming in an OSEK context. You should read this guide if you want to know low-level technical information to integrate the RTA-OSEK Component into your application.

1.2 Conventions

Important: Notes that appear like this contain important information that you need to be aware of. Make sure that you read them carefully and that you follow any instructions that you are given.

Portability: Notes that appear like this describe things that you will need to know if you want to write code that will work on any processor running the RTA-OSEK Component.

Program code, file names, C types and symbols, and RTA-OSEK API call names all appear in the `courier` typeface. When the name of an object is made available to the programmer the name also appears in the `courier` typeface, so, for example, a task named Task1 appears as a task handle called `Task1`.

2 Toolchain Issues

This chapter contains important details about RTA-OSEK and your toolchain. A port of the RTA-OSEK Component is specific to both the target hardware and a specific version of the compiler toolchain. You must make sure that you build your application with the supported toolchain.

If you are interested in using a different version of the same toolchain, please contact ETAS to confirm whether or not this is possible.

2.1 Compiler

The RTA-OSEK Component was built using the following compiler:

Vendor	IAR
Compiler	IAR Embedded Workbench
Version	v3.71.1.0

The compulsory compiler options for application code are shown in the following table:

Option	**Description**
`V1`	Specifies the V850E and V850ES cores

The C file that RTA-OSEK generates from your OIL configuration file is called `osekdefs.c`. This file defines configuration parameters for the RTA-OSEK Component when running your application.

2.2 Assembler

The RTA-OSEK Component was built using the following assembler:

Vendor	IAR
Assembler	IAR Universal Library Builder
Version	V4.61R

The compulsory assembler options for application code are shown in the following table:

Option	**Description**
V1	Specifies the V850E and V850ES cores

The assembly file that RTA-OSEK generates from your OIL configuration file is called `osgen.s85`. This file defines configuration parameters for the RTA-OSEK Component when running your application.

2.3 Linker/Locator

In addition to the sections used by application code, the following RTA-OSEK sections must be located:

Sections	Rom/Ram	Description
`os_pid`	ROM	RTA-OSEK read-only data
`os_pird`	ROM	RTA-OSEK initialization data
`os_intvec`	ROM	Vector table if generated by RTA-OSEK GUI
`os_pir`	RAM	RTA-OSEK initialized data
`os_pur`	RAM	RTA-OSEK uninitialized data
`os_wrappers`	ROM	RTA-OSEK interrupt wrappers.
`os_text`	ROM	RTA-OSEK code section.
`os_jumptable`	ROM	Jump table for indirect vector table
`os_pirf`	RAM	RTA-OSEK initialized data. Must be initialized during C-startup
`os_trace_ram`	RAM	RTA-TRACE uninitialized data. Must be zeroed during C-startu

In order to avoid the cost of the C-startup unnecessarily clearing `os_pir`, this section should be given the `NOCLEAR` attribute in the linker directive (`*.ld`) file. The supplied example application's linker directive file shows how this can be performed.

2.4 Debugger

Information about ORTI for RTA-OSEK can be found in the *RTA-OSEK ORTI Guide*

At the time of writing, we were not aware of any debuggers for the NEC V850E Series with support for ORTI.

If you are using an ORTI version 2.0 aware debugger on this platform you can use the "Unknown ORTI debugger" option in the RTA-OSEK GUI to generate an ORTI output file. The ORTI generated will not have been tested on the debugger and, therefore, is not guaranteed to work.

Please contact LiveDevices if you have any questions about ORTI support in RTA-OSEK.

3 Target Hardware Issues

3.1 Interrupts

This section explains the implementation of RTA-OSEK's interrupt model for V850E/IAR. You can find out more about configuring interrupts for RTA-OSEK in the *RTA-OSEK User Guide*.

3.1.1 Interrupt Levels

In RTA-OSEK interrupts are allocated an Interrupt Priority Level (IPL). This is a processor independent abstraction of the interrupt priorities that are available on the target hardware. You can find out more about IPLs in the *RTA-OSEK User Guide*. The hardware interrupt controller is explained in the *appropriate NEC manuals*.

The following table shows how RTA-OSEK IPLs relate to interrupt priorities on the target hardware:

IPL Value	Hardware Priority	ISPR Values	Description
0	N/A	00000000	User level
1	7	1E+07	Category 1 and 2 interrupts
2	6	x1000000	Category 1 and 2 interrupts
3	5	xx100000	Category 1 and 2 interrupts
4	4	xxx10000	Category 1 and 2 interrupts
5	3	xxxx1000	Category 1 and 2 interrupts
6	2	xxxxx100	Category 1 and 2 interrupts
7	1	xxxxxx10	Category 1 and 2 interrupts
8	0	xxxxxxx1	Category 1 and 2 interrupts

3.1.2 Interrupt Vectors

For the allocation of Category 1 and Category 2 interrupt handlers to interrupt vectors on your target hardware, the following restrictions apply:

Vector	Legality
0x0010 to 0x0050 and 0x00700	Category 1
0x0080 to maximum vector for CPU variant	Category 1 or 2

The valid base addresses for the vector table are:

Base Address	Notes
`0x0010`	The vector table must start just above the reset vector.

3.1.3 Category 1 Handlers

Category 1 interrupt service routines (ISRs) must correctly handle the interrupt context themselves, without support from the operating system. The IAR C compiler can generate appropriate interrupt handling code for a C function decorated with the `__interrupt` function qualifier. You can find out more in your compiler documentation.

3.1.4 Category 2 Handlers

Category 2 ISRs are provided with a C function context by the RTA-OSEK Component, since the RTA-OSEK Component handles the interrupt context itself. The handlers are written using the OSEK OS standard `ISR()` macro, shown in Code Example 3:1.

```
#include "MyISR.h"
ISR(MyISR) {
  /* Handler routine */
}
```

Code Example 3:1 - Category 2 ISR Interrupt Handler

You must not insert a return from interrupt instruction in such a function. The return is handled automatically by the RTA-OSEK Component.

3.1.5 Vector Table Issues

The number of vectors available depends upon the specific V850 chip variant selected in RTA-OSEK. The variants directly supported are CAG4-M, DJ3, FE2, FE3, FF2, FF3, FG2, FG3, FJ2, FJ3, FK3, PH2, PH3, PHO3, RS1, SG2, SG3, SJ2 and SJ3. These are in addition to the 'Generic V850ES' variant. Further variants can be supported by contacting ETAS.

When RTA-OSEK generates an interrupt vector table for the V850, it only emits data for addresses 0x10 up to the *highest declared interrupt*. This allows RTA-OSEK to cope efficiently with chip variants with differently sized vector tables.

Reserved vectors

The RTA-OSEK component requires eight interrupt vectors to be reserved for its use. By default these will be generated in the vector table in the first eight unbound vectors starting from vector 0x80. This can be overridden by creating a dummy ISR in the RTA-OSEK GUI with the name '`reserved_os_vector`'. The first reserved vector will then be generated in its place and the remaining reserved vectors will be generated in the next seven unbound vectors.

The RTA-OSEK component also needs to know the addresses of the eight interrupt control registers (PICn) corresponding with the reserved vectors. These should be mapped at link-time to the `_os_reserved_icr`*n* labels, where *n* ranges from 1 to 8. The linker directive file provided with the example application shows how this can be achieved.

3.1.6 Interrupt-handling assembler files

RTA-OSEK generates three different interrupt-handling assembler source files, each with a different approach to supporting ISRs. They are mutually exclusive: only one of the three should be compiled and linked into an application. `osvec1.s85` is the only one to include a vector table; `osvec2.s85` and `osvec3.s85` both require a vector table to be supplied by the user. As such, it is recommended that `osvec1.s85` is used unless the added flexibility of `osvec2.s85` or `osvec3.s85` is required. An explanation of the contents of each of the files follows below.

Note that when you configure your application with the RTA-OSEK GUI you can choose whether or not a vector table is generated. This option dictates whether or not the file `osvec1.s85` is generated; `osvec2.s85` and `osvec3.s85` are always generated, regardless of this option.

In the following discussion, an 'outer wrapper' is a small function, specific to an ISR, which sets up sufficient context for that ISR's entry function pointer to be passed along to the 'mid-wrapper.' The mid-wrapper is common to all Category 2 ISRs and saves and restores the register context around the call to the ISR's entry function.

osvec1. s85

The file `osvec1.s85` contains the interrupt vector table (containing the outer wrappers) and the mid-wrapper. The vector table is placed in the `os_intvec` section, which should be linked starting at address 0x10, and the mid-wrapper is placed in the `os_wrappers` section.

osvec2. s85

The file `osvec2.s85` does not contain a traditional vector table, but does contain a 'jump table' with the label `_os_vec2_table`, which is placed in the `os_jumptable` section. The table contains four-byte entries, one for each vector from `0x10` up to the highest bound vector. The content of each table entry depends on its corresponding vector as follows:

1. If a Category 2 ISR is bound to the vector, the entry is a jump (`jr`) to the outer interrupt wrapper for that ISR.
2. If a Category 1 ISR is bound to the vector, the entry is a jump to the ISR's entry function.
3. If the vector is unbound, greater than or equal to vector 0x80 (or the vector for the dummy ISR `reserved_os_vector`, if it exists) and fewer than eight entries have been reserved for the OS, then the entry is a jump to `os_reserved_vector` (see 'Reserved vectors' above for an explanation).
4. If the vector is unbound and there exists a default interrupt, the entry is a jump to the default interrupt's entry function.
5. If the vector is unbound and there is no default interrupt, the entry is a `nop` instruction.

The file also contains each outer wrapper referenced in the jump table, the mid-wrapper and the code for `os_reserved_vector`. These are all placed in the `os_wrappers` section.

The file can be assembled with only the wrappers (i.e. without the jump table) by defining the symbol `OS_NO_JUMP_TABLE` on the command line.

osvec3. s85

The file `osvec3.s85` contains a jump table similar to that in `osvec2.s85`, with the difference that the entries contain addresses rather than jump instructions. The content of each four-byte table entry depends on its corresponding vector as follows:

6. If a Category 2 ISR is bound to the vector, the entry is the address of the outer interrupt wrapper for that ISR.
7. If a Category 1 ISR is bound to the vector, the entry is the address of the ISR's entry function.
8. If the vector is unbound and there exists a default interrupt, the entry is the address of the default interrupt's entry function.
9. If the vector is unbound and there is no default interrupt, the entry is zero.

The file also contains the outer wrappers referenced in the jump table, which differ from the wrappers in `osvec2.s85` by omitting the code to preserve `r6` on the stack.

`osvec3.s85` also contains a special form of mid-wrapper. Unlike the 'regular' mid-wrapper used in `osvec1.s85` and `osvec2.s85`, this mid-wrapper does not restore `r6` from the stack after the ISR has run, and does not return from interrupt (`reti`). The final instruction of the mid-wrapper is a jump to `os_end_wrapper`. A default implementation of `os_end_wrapper`, which restores `r6` from the stack and returns from interrupt, is provided. The default implementation can be removed by the preprocessor by defining the symbol `OS_NO_END_WRAPPER` on the command line.

Important: When using `osvec3.s85`, it is the responsibility of the user to preserve `r6` in interrupt-handling code before execution reaches the outer wrapper. The default implementation of `os_end_wrapper` can be used if only `r6` is preserved on the stack. If any other registers are used before execution reaches the outer wrappers, then they must be preserved on the stack, and `os_end_wrapper` overridden to restore them (and `r6`) from the stack. Any such additional stack usage must be accounted for in the idle task's stack usage.

Important: When using osvec3.`s85`, the eight reserved vectors, each consisting of the following code, must be manually placed on the vector table. They may be placed anywhere, with the condition that the locations must match the `_os_reserved_icr`*`n`* addresses provided at link-time.

```
-- OS reserved vector
stsr  EIPC, r2
jmp   [r2]
.align     16
```

The following table shows the syntax for labels attached to RTA-OSEK Category 2 interrupt handlers (VVVV represents the 4 hex digit, upper-case, zero-padded value of the vector location).

Vector Location	Label
`0xVVVV`	`os_wrapper_vvvv`
eg :0x03A0	`os_wrapper_03a0`

3.1.7 Traps

If a trap instruction is used, a Category 1 interrupt handler can be set up to service the exception. However, RTA-OSEK does not preserve the EP (exception in progress) bit in the PSW if an API call is made that manipulates the IPL. If such calls are used, the EP bit must be set back to 1 prior to leaving the interrupt handler as shown in the sample code below.

```
asm void set_PSW(ByteType m) {
%reg m ;
  ldsr m, PSW ;
%error
  Macro has not expanded
}

asm ByteType get_PSW(void) {
  stsr PSW,r10;
}

__interrupt void sync_isr(void)
{
  register ByteType psw_val = get_PSW();

  DisableAllInterrupts();
  ...
  EnableAllInterrupts();

  set_PSW(psw_val);
}
```

Note that in reality an exception handler never needs to make such calls, because it is already executing at the highest IPL and it is illegal for it to lower the interrupt priority. In this case, no special processing will be needed.

3.1.8 Default Interrupt

The 'default interrupt' is intended to be used to catch all unexpected interrupts. All unused interrupts have their interrupt vectors directed to the named routine that you specify. This routine must correctly handle the interrupt context, in the same way as a Category 1 ISR. The Green Hills C compiler can generate appropriate interrupt handling code using the `__interrupt` function qualifier.

Because RTA-OSEK only emits interrupt vectors for addresses 0x0010 up to the highest declared interrupt, it will only fill unused vectors with the default interrupt *up to the highest declared interrupt*. To fill the entire vector table for your chip variant, create a dummy Category 1 interrupt and place it on the highest vector used by the chip. The default interrupt will then be used to fill all unused vectors below this.

3.1.9 Flash Security ID

To protect the contents of internal ROM some V850 variants such as the FE2 and FF2 support a 10 byte security number located at memory address 0x70. This address falls within the interrupt vector table range. If the user wishes to enter a 10 byte security number at this address there are three available methods:

1. The `OS_SECURITY_ID` macro in `osvec1.s85` can be defined on the command line. For example, assembling `osvec1.s85` with the command line option `-DOS_SECURITY_ID=0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA` will insert the security number 0x112233445566778899AA at address 0x70.
2. `OS_SECURITY_ID` can alternatively be defined in a header file, and `osvec1.s85` can be made to include that file by defining `OS_SECURITY_ID_HEADER` to be the file's name. For example, assembling `osvec1.s85` with the option `-DOS_SECURITY_ID_HEADER =\"security_id.h\"` will have the effect of including `security_id.h` at the start of `osvec1.s85`.
3. The security ID can also be specified in an assembler source file, and `osvec1.s85` can be made to include that file at vector 0x70. This is achieved using the preprocessor symbol `OS_SECURITY_ID_ASM`. For example, creating the file `security_id.s85` containing "`.byte 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA`" and assembling `osvec1.s85` with the option `-DOS_SECURITY_ID_ASM=\"security_id.s85 \"` will insert the security ID 0x112233445566778899AA at address 0x70.

Some V850 variants may contain other registers in flash after the security ID. For example, the CAG4-M variant contains the FMOP0 and FMOP1 registers at addresses 0x7A and 0x7B, respectively. These can be set simply by adding two (or more) additional bytes to the `OS_SECURITY_ID` macro. The provided example application demonstrates this by building osvec1.s85 to write 0xDF to FMOP0, 0xFF to FMOP1, and to fill the remaining bytes from 0x7C to 0x7F with 0xFF.

3.1.10 Helper Macros

Category 1 ISR Macros

The macros `CAT1ISR_BEGIN()` and `CAT1ISR_END()` should be placed at the start and end of Category 1 ISR functions, to allow nested execution of interrupts by enabling/disabling interrupts and to perform some housekeeping required for the RTA-OSEK component to function correctly. A Category 1 ISR function should therefore look like the following code:

```
__interrupt void some_isr(void)
{
  CAT1ISR_BEGIN()
  /* user code goes here */
  CAT1ISR_END()
}
```

Enabling and Disabling Interrupt Sources

The macros `OS_ENABLE_INTERRUPT(PICn)` and `OS_DISABLE_INTERRUPT(PICn)` will enable/disable a specific interrupt source by writing to the provided interrupt control register. The arguments provided to these macros should be the register macros defined in the device header files provided by NEC. For example, the macro `PIC10` is defined in the file ‘df3461.h’ provided with the example application. The following code would disable, and then enable, the INTP10 interrupt.

```
/* disable INTP10 */
OS_DISABLE_INTERRUPT(PIC10);
/* enable INTP10 */
OS_ENABLE_INTERRUPT(PIC10);
```

3.2 Register Settings

The RTA-OSEK Component does not require the initialization of registers before calling `StartOS()`.

The RTA-OSEK Component does not reserve the use of any hardware registers.

3.3 Stack Usage

3.3.1 Number of Stacks

A single stack is used. The first argument to `StackFaultHook` is always 0.

`osStackOffsetType` is a scalar, representing the number of bytes on the stack, with C type `unsigned long`.

3.3.2 Stack Usage within API Calls

The maximum stack usage within RTA-OSEK API calls, excluding calls to hooks and callbacks, is as follows:

Standard

API max usage (bytes): 64

Timing

API max usage (bytes): 64

Extended

API max usage (bytes): 88

To determine the correct stack usage for tasks that use other library code, you may need to contact the library vendor to find out more about call stack usage.

4 Parameters of Implementation

This chapter provides detailed information on the functionality, performance and memory demands of the RTA-OSEK Component.

NB: This is a placeholder for the tables of sizes and times collected by the Binding Manual Performance Measurement application. At the time of generation of this manual, this application is not yet available for the V850E/IAR port of RTA-OSEK.

Support

For product support, please contact your local ETAS representative.

Office locations and contact details can be found at the front of this manual and on the ETAS Group website www.etasgroup.com.