CODE TIME TECHNOLOGIES

Abassi RTOS

Porting Document

AVR32A – IAR

Disclaimer

Table of Contents

1 INTRODUCTION 6

1.1 DISTRIBUTION CONTENTS 6
1.2 LIMITATIONS 6

2 TARGET SET-UP 7

2.1 INTC CONTROLLER SET-UP 8
2.2 INTERRUPT STACK SET-UP 9
2.3 FAST INTERRUPTS SET-UP 10
2.4 SATURATION BIT SET-UP 11
2.5 MULTITHREADING 12
2.5.1 Reentrance Protection 13
2.5.2 Full Multithreading Protection 13
2.5.3 Partial Multithreading Protection 14
2.6 RETE ERRATA 14
2.7 INTERRUPT MASKING ERRATA 15
2.8 SPURIOUS INTERRUPT ERRATA 17

3 INTERRUPTS 19

3.1 INTERRUPT HANDLING 19
3.1.1 Interrupt Installer 19
3.2 INTERRUPT PRIORITY AND ENABLING 20
3.3 FAST INTERRUPTS 20
3.4 NESTED INTERRUPTS 20

4 STACK USAGE 22

5 SEARCH SET-UP 23

6 CHIP SUPPORT 26

7 MEASUREMENTS 27

7.1 MEMORY 27
7.2 LATENCY 30

8 APPENDIX A: BUILD OPTIONS FOR CODE SIZE 37

8.1 CASE 0: MINIMUM BUILD 37
8.2 CASE 1: + RUNTIME SERVICE CREATION / STATIC MEMORY 38
8.3 CASE 2: + MULTIPLE TASKS AT SAME PRIORITY 39
8.4 CASE 3: + PRIORITY CHANGE / PRIORITY INHERITANCE / FCFS / TASK SUSPEND 40
8.5 CASE 4: + TIMER & TIMEOUT / TIMER CALL BACK / ROUND ROBIN 41
8.6 CASE 5: + EVENTS / MAILBOXES 42
8.7 CASE 6: FULL FEATURE BUILD (NO NAMES) 43
8.8 CASE 7: FULL FEATURE BUILD (NO NAMES / NO RUNTIME CREATION) 44
8.9 CASE 8: FULL BUILD ADDING THE OPTIONAL TIMER SERVICES 45

9 APPENDIX B: AVR32A INTERRUPT COMPONENTS 47

9.1 OSISRINSTALL 47
9.2 OSAVR32AISRMAP 48
9.3 OSAVR32AISRPRIO 49

10 REVISION HISTORY 50

List of Figures

FIGURE 2-1 PROJECT FILE LIST 7
FIGURE 2-2 GUI SET OF `OS_AVR32A_INT_LINE (C)` 8
FIGURE 2-4 GUI SET OF `OS_ISR_STACK` 10
FIGURE 2-5 GUI SET OF `OS_FAST_INTS` 11
FIGURE 2-6 GUI SET OF `OS_HANDLE_SR_Q` 12
FIGURE 2-7 MULTITHREAD-SAFE PROJECT FILE LIST 13
FIGURE 2-8 FULL MULTITHREAD PROTECTION GUI CONFIGURATION 14
FIGURE 2-9 GUI SET OF `OS_FIX_RETE_L` 15
FIGURE 2-10 GUI SET OF `OS_FIX_SR_GIM` 16
FIGURE 2-11 GUI SET OF `OS_SPURIOUS_ISR` 18
FIGURE 7-1 MEMORY MEASUREMENT CODE OPTIMIZATION SETTINGS 27
FIGURE 7-2 LATENCY MEASUREMENT CODE OPTIMIZATION SETTINGS 30

List of Tables

TABLE 1-1 DISTRIBUTION 6
TABLE 2-1 ASSEMBLER REQUIRED OPTIONS 7
TABLE 2-2 INTC CONFIGURATION FOR "C" (COMMAND LINE) 8
TABLE 2-5 INTERRUPT STACK ENABLED 9
TABLE 2-6 INTERRUPT STACK DISABLED 9
TABLE 2-7 COMMAND LINE SET OF OS_ISR_STACK 9
TABLE 2-8 FAST INTERRUPT CONFIGURATION 10
TABLE 2-9 COMMAND LINE SET OF OS_FAST_INTS 10
TABLE 2-10 FAST INTERRUPTS VS. PRIORITIES 11
TABLE 2-11 SATURATION BIT CONFIGURATION 12
TABLE 2-12 COMMAND LINE SET OF OS_HANDLE_SR_Q 12
TABLE 2-13 FULL MULTITHREAD PROTECTION COMMAND LINE CONFIGURATION 13
TABLE 2-14 SETTING A TASK TO BE MULTITHREAD SAFE 14
TABLE 2-15 RETE INSTRUCTION FIX CONFIGURATION 14
TABLE 2-16 COMMAND LINE SET OF OS_FIX_RETE_L 15
TABLE 2-17 INTERRUPT MASKING FIX CONFIGURATION 16
TABLE 2-18 COMMAND LINE SET OF OS_FIX_SR_GIM 16
TABLE 2-19 SPURIOUS INTERRUPT FIX CONFIGURATION 17
TABLE 2-20 COMMAND LINE SET OF OS_SPURIOUS_ISR 17
TABLE 3-1 ATTACHING A FUNCTION TO AN INTERRUPT 19
TABLE 3-2 INVALIDATING AN ISR HANDLER 20
TABLE 3-3 REMOVING INTERRUPT NESTING 21
TABLE 3-4 PROPAGATING INTERRUPT NESTING 21
TABLE 4-1 CONTEXT SAVE STACK REQUIREMENTS 22
TABLE 5-1 SEARCH ALGORITHM CYCLE COUNT 24
TABLE 7-1 "C" CODE MEMORY USAGE (MEDIUM CODE) 28
TABLE 7-2 "C" CODE MEMORY USAGE (LARGE CODE) 29
TABLE 7-3 ASSEMBLY CODE MEMORY USAGE 29
TABLE 7-4 MEASUREMENT WITHOUT TASK SWITCH 31
TABLE 7-5 MEASUREMENT WITHOUT BLOCKING 31
TABLE 7-6 MEASUREMENT WITH TASK SWITCH 31
TABLE 7-7 MEASUREMENT WITH TASK UNBLOCKING 32
TABLE 7-8 LATENCY MEASUREMENTS (MEDIUM CODE / SMALL DATA) 33
TABLE 7-9 LATENCY MEASUREMENTS (MEDIUM CODE / LARGE DATA) 34
TABLE 7-10 LATENCY MEASUREMENTS (LARGE CODE / SMALL DATA) 35
TABLE 7-11 LATENCY MEASUREMENTS (LARGE CODE / LARGE DATA) 36
TABLE 8-1: CASE 0 BUILD OPTIONS 37
TABLE 8-2: CASE 1 BUILD OPTIONS 38
TABLE 8-3: CASE 2 BUILD OPTIONS 39
TABLE 8-4: CASE 3 BUILD OPTIONS 40
TABLE 8-5: CASE 4 BUILD OPTIONS 41
TABLE 8-6: CASE 5 BUILD OPTIONS 42
TABLE 8-7: CASE 6 BUILD OPTIONS 43
TABLE 8-8: CASE 7 BUILD OPTIONS 44
TABLE 8-9: CASE 8 BUILD OPTIONS 45

1 Introduction

This document details the port of the Abassi RTOS to the ATMEL AVR32A processor. The software suite used for this specific port is the IAR Embedded Workbench for the AVR32, more commonly known as EWAVR; the version used for the port and all tests is Version 4.10.

1.1 Distribution Contents

The set of files supplied with this distribution are listed in Table 1-1 below:

Table 1-1 Distribution

File Name	Description
`Abassi.h`	Include file for the RTOS
`Abassi.c`	RTOS "C" source file
`Abassi_AVR32A_IAR.s82`	RTOS assembly file for the AVR32A to use with the IAR Embedded Workbench
`Abassi_IAR_MTX_IF.c`	Abassi interface functions for multithread-safe operation of the IAR DLIB.
`Demo_0_EVK1101_IAR.c`	Demo code for the EVK1101 evaluation board using the LEDs & serial port
`Demo_3_EVK1101_IAR.c`	Demo code for the EVK1101 evaluation board using the serial port
`Demo_5_EVK1101_IAR.c`	Demo for on the EVK1101 evaluation board using the serial port
`AbassiDemo.h`	Build option settings for the demo code

1.2 Limitations

To optimize the reaction time of the Abassi RTOS components, it was decided to require the processor to always operate in the supervisor mode (which is the start-up default mode for AVR32A microcontrollers) and not the application mode. The IAR Embedded Workbench regular start-up code fulfills these constraints and one must be careful to not change the mode of operation.

Applications using the RTOS built with the small code model have not been verified. The reason for this is due to the family of devices used for the port and testing. The flash memory of all device of the AT32UC3B family is located at address 0x80000000, which does not fulfill the requirement of the small code model. The devices in the AT32UC3A and AT32UC3A3 families also have the same flash mapping.

2 Target Set-up

Very little is needed to configure the Atmel Studio development environment to use the Abassi RTOS in an application. All there is to do is to add the files `Abassi.c` and `Abassi_AVR32A_IAR.s` in the source files of the application project, and make sure the seven configuration settings in the file `Abassi_AVR32_IAR.s` (`OS_ISR_STACK` described in Section 2.2, `OS_FAST_INTS` described in Section 2.3, `OS_HANDLE_PSR_Q` described in Section 2.4, `OS_FIX_RETE_L` described in Section 2.6, `OS_FIX_SR_GIM` described in Section 2.7, and `OS_SPURIOUS_ISR` described in Section 2.8) are set according to the needs of the application. As well, update the include file path in the C/C++ compiler preprocessor options with the location of `Abassi.h`. There is no need to include a file for the interrupt table, as the `Abassi_AVR32A_IAR.s` file contains all the interrupt table and default exception handlers.

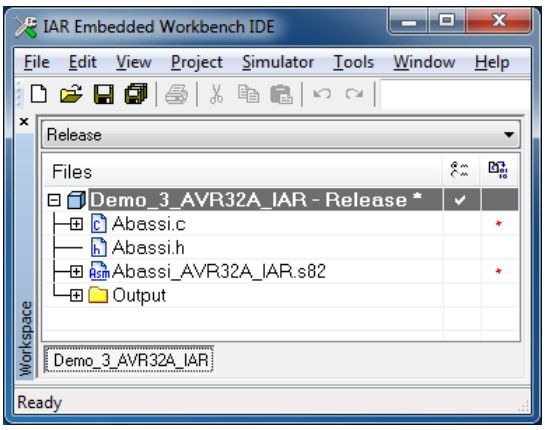

Figure 2-1 Project File List

The file `Abassi_AVR32A_IAR.s82` contains everything related to the AVR32A events and interrupts, and it overloads the regular IAR Embedded Workbench interrupt handlers. The RTOS supplies its own API for the interrupts, as the RTOS needs to be interrupt-aware, therefore any non-Abassi interrupt API cannot be used. Finally, there are a few configuration settings in the files `Abassi_AVR32A_IAR.s82` and `Abassi.h` that need to be set according to the target device and the needs of the application. These settings are described in the following subsections.

If the application is not built though the Embedded Workbench GUI, then it is important to specify the same options on the command line for both the compiler and the assembler, as the file `Abassi_AVR32A_IAR.s82` relies on command line options to generate the correct code. The options the file `Abassi_AVR32A_IAR.s82` requires are listed in the following table:

Table 2-1 Assembler required options

Option	Description
`--data_model=XXX`	Data model, where `XXX` is either `small` (`s`) or `large` (`l`)

2.1 INTC controller Set-up

The AVR32A interrupt controller (INTC) supports up to 64 groups of interrupts, where each individual group can handle up to 32 lines of interrupts. This means the interrupt controller is capable at handling up to 2048 individual sources of interrupts. If the group/line mapping is not reduced in one form or another, it would be necessary to reserve 8192 bytes (2048 * 4 bytes, as function pointers are 4 bytes) for the interrupt table internally used by the interrupt dispatcher. The way the Abassi RTOS reduces the size of the interrupt table is to consider the maximum number of groups and maximum number of lines (in any group) supported on the target device. For example, on the `AT32UC3B` device family, the highest group number (not number of groups) is 18, meaning the device spans 19 groups of interrupts. And the group that has the highest line number (not number of lines) is group number 1, with the largest line number being 9. This means it is possible to fully support all interrupt triggers on this device family by using an array of 19 groups and 10 lines.

The number of groups and number of lines must be specified either directly in the `Abassi.h` file where all the build option definitions are located, or through the GUI, or on the command line.

These values can be set through the command as shown in Table 2-2 below:

Table 2-2 INTC Configuration for "C" (Command line)

```
iccavr32 … -DOS_AVR32A_INT_GRP=19 -DOS_AVR32A_INT_LINE=10 …
```

The number of interrupt groups and lines can also be set for the "C" code through the GUI, in the "*C/C++ Compiler / Preprocessor*" menu, as shown in the following figure:

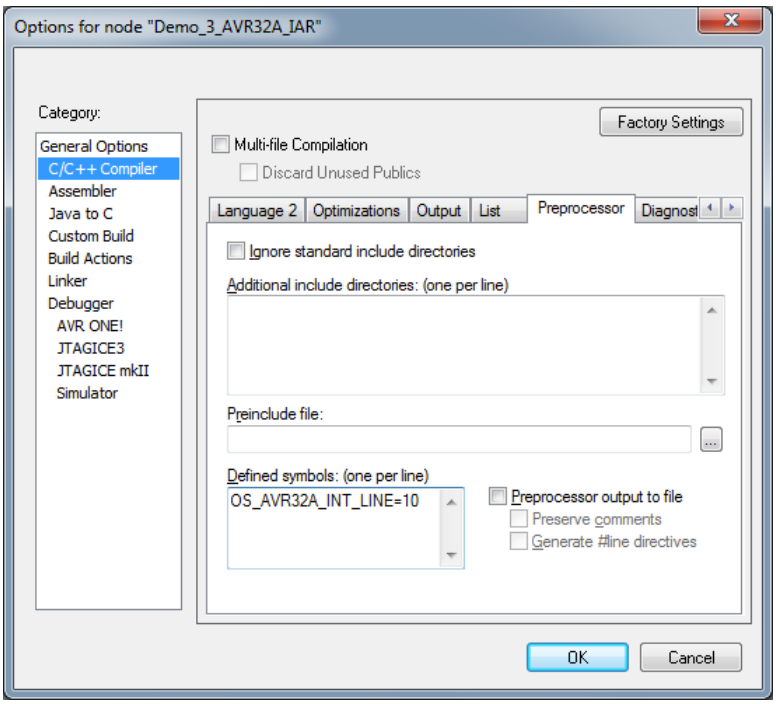

Figure 2-2 GUI set of `OS_AVR32A_INT_LINE (C)`

The values shown in the table above are the values set in the distribution, as the port was performed on an EVK1101 Evaluation board, which is populated with an AT32UC3B0256 device.

2.2 Interrupt Stack Set-up

It is possible, and is highly recommended, to use a hybrid stack when nested interrupts occur in an application. Using this hybrid stack, specially dedicated to the interrupts, removes the need to allocate extra room to the stack of every task in the application to handle the interrupt nesting. This feature is controlled by the value set by the definition OS_ISR_STACK, located around line 30 in the file Abassi_AVR32A_IAR.s82. To disable this feature, set the definition of OS_ISR_STACK to a value of zero. To enable it, and specify the interrupt stack size, set the definition of OS_ISR_STACK to the desired size in bytes (see Section 4 for information on stack sizing). As supplied in the distribution, the hybrid stack feature is enabled, and a stack size of 1024 bytes is allocated; this is shown in the following table:

Table 2-3 Interrupt Stack enabled

```
#ifndef OS_ISR_STACK
OS_ISR_STACK       EQU 1024       /* If using a dedicated stack for the nested ISRs  */
#endif                            /* 0 if not used, otherwise size of stack in bytes */
```

Table 2-4 Interrupt Stack disabled

```
#ifndef OS_ISR_STACK
OS_ISR_STACK       EQU 0          /* If using a dedicated stack for the nested ISRs  */
#endif                            /* 0 if not used, otherwise size of stack in bytes */
```

Alternatively, it is possible to overload the OS_ISR_STACK value set in Abassi_AVR32A_IAR.s82 by using the assembler command line option –D and specifying the desired hybrid stack size. In the following example, the stack size is set to 512:

Table 2-5 Command line set of OS_ISR_STACK

```
aavr32 … -DOS_ISR_STACK=512 …
```

The number of interrupt line can also be set for the assembly code through the GUI, in the "*Assembler / Preprocessor*" menu, as shown in the following figure:

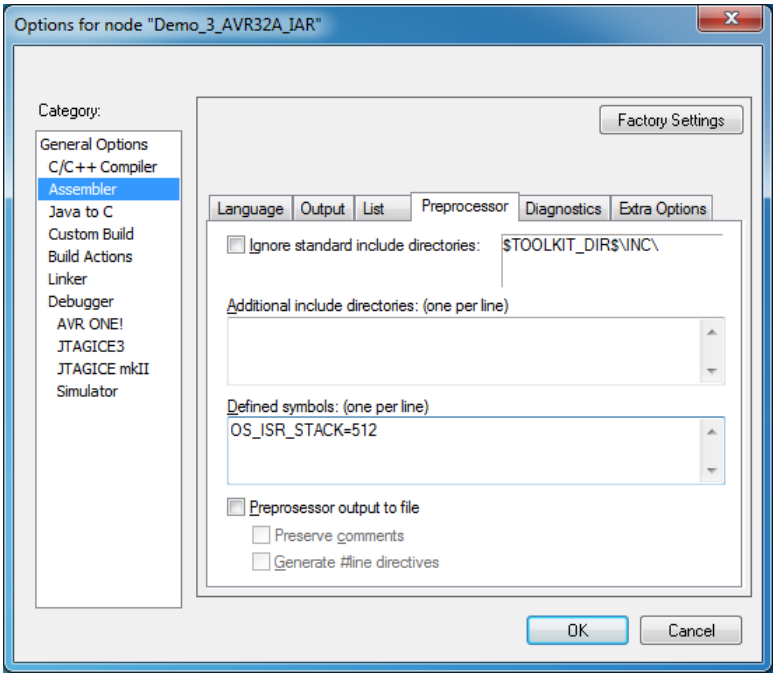

Figure 2-3 GUI set of `OS_ISR_STACK`

2.3 Fast Interrupts Set-up

Fast interrupts are supported on this port. A fast interrupt is an interrupt that never uses any component from Abassi, and as the name says, is desired to operate as fast as possible. To configure the fast interrupts, all there is to do is to set the value of the token `OS_FAST_INTS`, located around line 30 in the file `Absssi_AVR32A_IAR.s82`, to the priority threshold at which the interrupts are mapped to fast interrupts:

Table 2-6 Fast Interrupt Configuration

```
#ifndef OS_FAST_INTS
OS_FAST_INTS     EQU 0          /* Fast interrupts enable? and if so, level threshold */
#endif                          /* e.g if fast ISRs for prio 2 & 3, then set to 2     */
```

Alternatively, it is possible to overload the `OS_FAST_INTS` value set in `Abassi_AVR32A_IAR.s82` by using the assembler command line option `–D` and specifying the desired fast interrupt threshold. In the following example, the threshold is set to 2:

Table 2-7 Command line set of `OS_FAST_INTS`

```
aavr32 … -DOS_FAST_INTS=2 …
```

The fast interrupt priority threshold value can also be set for the assembly code through the GUI, in the "*Assembler / Preprocessor*" menu, as shown in the following figure:

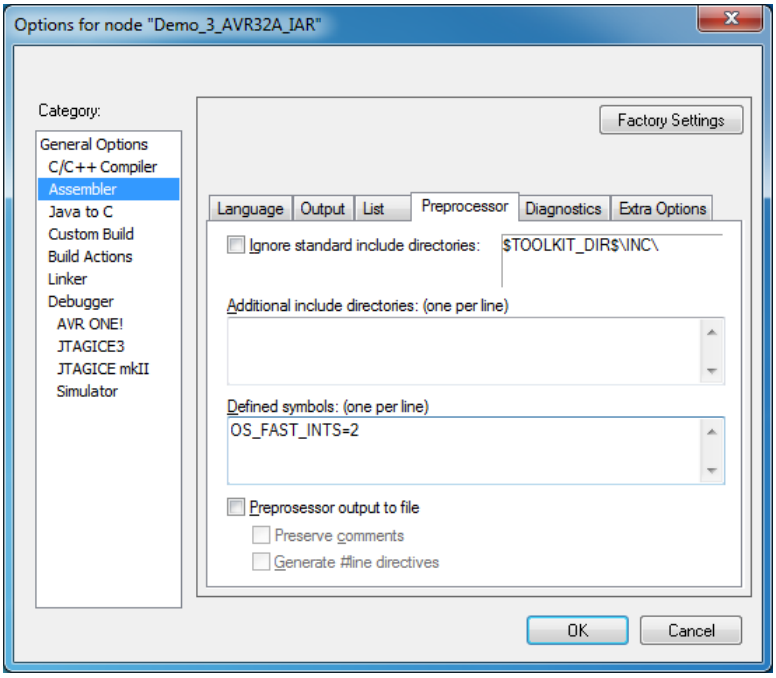

Figure 2-4 GUI set of `OS_FAST_INTS`

The following table indicates the priorities of the interrupts that are re-mapped to operate as fast interrupts, according to the setting of `OS_FAST_INTS`:

Table 2-8 Fast Interrupts vs. Priorities

`OS_FAST_INTS` setting	Priority Level
`OS_FAST_INTS EQU 0`	`Disable`
`OS_FAST_INTS EQU 1`	`1, 2, 3`
`OS_FAST_INTS EQU 2`	`2, 3`
`OS_FAST_INTS EQU 3 (or larger than 3)`	`3`

If an interrupt priority level is configured to the fast interrupt operations, all interrupts at that level are treated as fast interrupts; it is not possible to distribute some interrupt handlers to a regular interrupt and some others to a fast interrupt within the same priority level.

Even if the hybrid interrupt stack feature is enabled (see Section 2.2), fast interrupts will not use that stack. This translates into the need to reserve room on all task stacks for the possible nesting of fast interrupts.

2.4 Saturation Bit set-up

In the AVR32A status register, there is a sticky bit to indicate if an arithmetic saturation has occurred; this is the Q flag in the status register (bit 3). By default, this bit is not kept localized at the task level as it needs extra processing to do so; instead, it is propagated across all tasks. This choice was made because most applications do not care about the value of this bit.

If this bit is relevant for an application, even in a single task, then it must be kept locally in each task. To keep the meaning of the saturation bit localized, the token `OS_HANDLE_SR_Q` must be set to a non-zero value; to disable it, it must be set to a zero value. This is located at around line 40 in the file `Abassi_AVR32A_IAR.s82`. The distribution code disables the localization of the Q bit, setting the token `OS_HANDLE_SR_Q` to zero, as shown in the following table:

Table 2-9 Saturation Bit configuration

```
#ifndef OS_HANDLE_SR_Q
OS_HANDLE_SR_Q       EQU 0          /* If we keep the Q bit (saturation) on per tasks */
#endif
```

Alternatively, it is possible to overload the `OS_HANDLE_SR_Q` value set in `Abassi_AVR32A_IAR.s82` by using the assembler command line option `–D` and specifying the desired handling of the saturation bit. In the following example, it is configured to be localized at the task level:

Table 2-10 Command line set of `OS_HANDLE_SR_Q`

```
aavr32 … -DOS_HANDLE_SR_Q=1 …
```

The handling of the saturation bit can also be set for the assembly code through the GUI, in the "*Assembler / Preprocessor*" menu, as shown in the following figure:

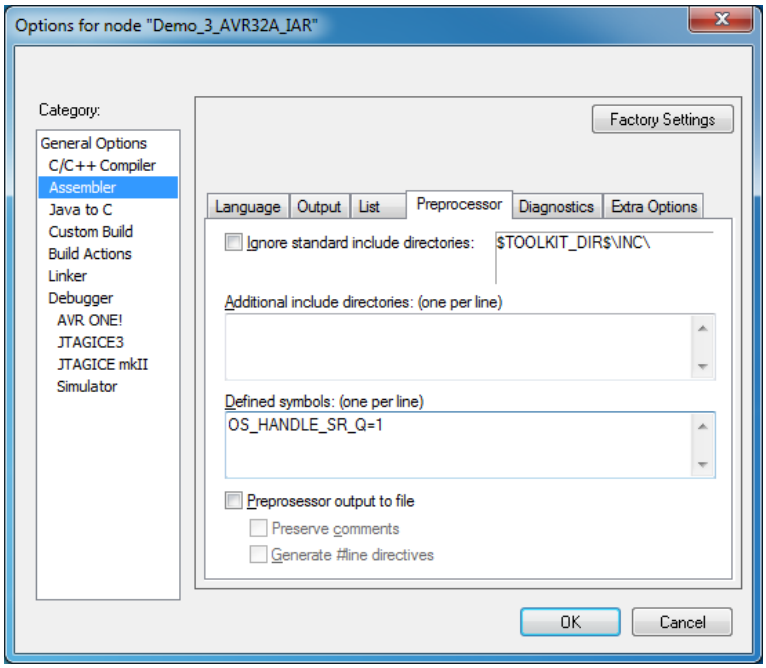

Figure 2-5 GUI set of `OS_HANDLE_SR_Q`

2.5 Multithreading

By default, the IAR DLIB runtime library is not multithread safe. There are two aspects to take into account when protecting the library for multithreading. The first one involves reentrance; some library functions are not reentrant, therefore two tasks accessing the same non-reentrant function at the same time can create major issues. The classic example of non-reentrant functions are the family of functions for

dynamic memory allocation: e.g. `malloc()` and `free()`. As they internally use a static buffer, a few pointers, and some linked lists, if two tasks use functions that access the internals of the dynamic memory allocation at the same time, corruption could occur. Protecting the non-reentrant functions is straightforward: all there is to do is to make sure there is only a single task that can access the non-reentrant functions at any time. This is done with a mutex, as it is the perfect mechanism to guarantee exclusive access to a resource.

The second type of functions and variables that are not multithread safe are due to internal data used by the library, data that is truly a global resource. Examples of these are: the `errno` variable or the `locale` information; these are called TLS (Thread Local Storage) by IAR. The only efficient way to protect these functions and variables against multithreading is to have the library configured to use a unique sets of variables for each task. There are multiple ways to implement the data access or swapping, but fundamentally, if the library does not provided such a dedicated mechanism, it becomes cumbersome to solve the issue, as it would require a manual swap of the each individual internal static variable of the library at every task switch.

More detailed information on what functions require re-entrance protection and which global variables require multi-threading protection can be found in the IAR EWAVR32 Compiler Reference Guide, in the section titled "*Multithread Support in the DLIB Library*".

The IAR DLIB library fully support both mechanisms to make the library multithread safe. The following sub-sections describe how to make each of the two libraries multithread safe.

NOTE: The "out of the box" AVR32A DLIB is not compiled nor archived with the multi-threading protection hooks. As specified in the IAR EWAVR32 Compiler Reference Guide the DLIB must be re-compiled and archived. The procedure on how to do so is in described the sections *Building and using a customized library* and *Multithread Support in the DLIB Library*.

2.5.1 Reentrance Protection

Reentrance protection is achieved by giving access to mutexes to the library. The DLIB reentrance protection requires a specific API and these custom API modules are provided in the file `Abassi_IAR_MTX_IF.c`, which is part of the distribution. All there is to do to protect the DLIB against reentrance is to add the file `Abassi_IAR_MTX_IF.c` in the project.

Figure 2-6 Multithread-safe Project File List

2.5.2 Full Multithreading Protection

For full multithreading of the library, all there is to do is to define for the compiler the build option `OS_IAR_MTHREAD` with a positive value. Setting `OS_IAR_MTHREAD` to a positive value does two things. The first change is to insert a custom function that provides the address of the global variables associated to the running task. Then, any time a TLS variable is accessed, either directly in the task, or internally by the library, it is the task's TLS being accessed. The second change occurs during task creation, where there is an allocation of memory through the component `OSalloc()` in order to hold one set of TLS for every task.

NOTE: The Adam&Eve task (the one associated with the function `main()`) uses the default TLS.

Table 2-11 Full Multithread Protection Command Line Configuration

```
iccarm … -DOS_IAR_MTHREAD=1 …
```

The library multithreading protection used by `Abassi_CORTEXM3_IAR.s` can also be set through the GUI, in the "*Assembler / Preprocessor*" menu, as shown in the following figure:

Figure 2-7 Full Multithread Protection GUI Configuration

2.5.3 Partial Multithreading Protection

It may not be necessary to make the library multithread safe for all tasks in an application; e.g. tasks that don't access or use the TLS, or call library functions using TLS, do not require the library to be protected. It may also be desirable to share the TLS amongst a set of tasks. Setting the build option `OS_IAR_MTHREAD` to a negative value allows the selection of the tasks where multithreading protection is required. The build option `OS_IAR_MTHREAD` is set the same way as described in the previous section.

A task is set to use the library in a multithread safe manner with the following:

Table 2-12 Setting a task to be multithread safe

```
#include "Abassi.h"

TSK_t *TskReent;
void _DLIB_TLS_MEMORY *Mthread;

…
                                                /* First the task must be created    */
                                                /* in the suspended state            */
TskReent = TSKcreate("TaskName", TskPrio, StackSize, TaskFct, 0);
                                                /* Get memory for the TLS            */
Mthread = OSalloc(__IAR_DLIB_PERTHREAD_SIZE);
                                                /* Initialize the TLS                */
__iar_dlib_perthread_initialize((void *) Mthread);

TskReent->XtraData[0] = (intptr_t)Mthread;      /* Attach the TLS to the task        */

TSKreseum(TskReent);                            /* The task may now be resumed       */
```

If the same TLS is desired to be shared amongst multiple tasks, simply set the field `XtraData[0]` of the tasks descriptors to the same TLS memory block, initialized once only.

2.6 RETE Errata

There is a known and well-documented hardware problem on some revisions of the AVR32A CPU core, which is related to not clearing of the L bit (lock bit) in the status register when a RETE instruction executes. As the instruction RETE is used in the file `Abassi_AVR32A_IAR.s82`, it is possible to include special code that fixes the problem. This is controlled with the token `OS_FIX_RETE_L`, which must be set to a non-zero value to activate the fix; to disable it, it must be set to a zero value. The token is defined at around line 45 in the file `Abassi_AVR32A_IAR.s82`. The distribution code enables the fix on the RETE instruction as a safety measure; this is shown in the following table:

Table 2-13 RETE Instruction Fix Configuration

```
#ifndef OS_FIX_RETE_L
OS_FIX_RETE_L       EQU 1      /* If patching errata on rete not clearing L bit      */
#endif
```

NOTE: The operation performed by this fix has not been verified as correct due to the lack of access to a device with the problem. The fix was tested to verify it does not impact in any way the operation of the RTOS.

Alternatively, it is possible to overload the `OS_FIX_RETE_L` value set in `Abassi_AVR32A_IAR.s82` by using the assembler command line option `–D` to enable/disable the fix. In the following example, it is enabled:

Table 2-14 Command line set of `OS_FIX_RETE_L`

```
aavr32 … -DOS_FIX_RETE_L=1 …
```

The enabling / disabling of the fix can also be set for the assembly code through the GUI, in the "*Assembler / Preprocessor*" menu, as shown in the following figure:

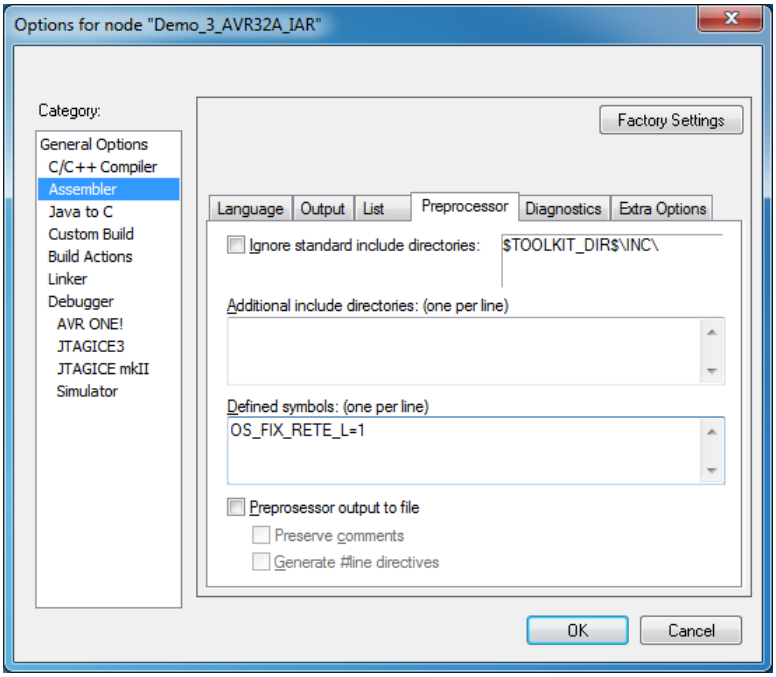

Figure 2-8 GUI set of `OS_FIX_RETE_L`

2.7 Interrupt Masking Errata

There is a known and well-documented hardware problem on some revisions of the AVR32A CPU core, which is related to masking the interrupt by setting the global interrupt mask bit in the status register: the two next instructions after the flag setting may not execute properly. As interrupts are masked in the `Abassi_AVR32A_IAR.s82`, it is possible to include special code that fixes the problem. This is controlled with the token `OS_FIX_SR_GIM`, which must be set to a non-zero value to activate the fix; to disable it, it must be set to a zero value. The token is defined at around line 50 in the file `Abassi_AVR32A_IAR.s82`. The distribution code enables this fix as a safety measure; this is shown in the following table:

Table 2-15 Interrupt Masking Fix Configuration

```
#ifndef OS_FIX_SR_GIM
OS_FIX_SR_GIM       EQU 1    /* If patching errata on 2 nop after interrupt masking  */
#endif
```

NOTE: The operation performed by this fix has not been verified as correct due to the lack of access to a device with the problem. The fix was tested to verify it does not impact in any way the operation of the RTOS.

Alternatively, it is possible to overload the `OS_FIX_SR_GIM` value set in `Abassi_AVR32A_IAR.s82` by using the assembler command line option `–D` to enable/disable the fix. In the following example, it is enabled:

Table 2-16 Command line set of `OS_FIX_SR_GIM`

```
aavr32 … -DOS_FIX_SR_GIM=1 …
```

The enabling / disabling of the fix can also be set for the assembly code through the GUI, in the "*Assembler / Preprocessor*" menu, as shown in the following figure:

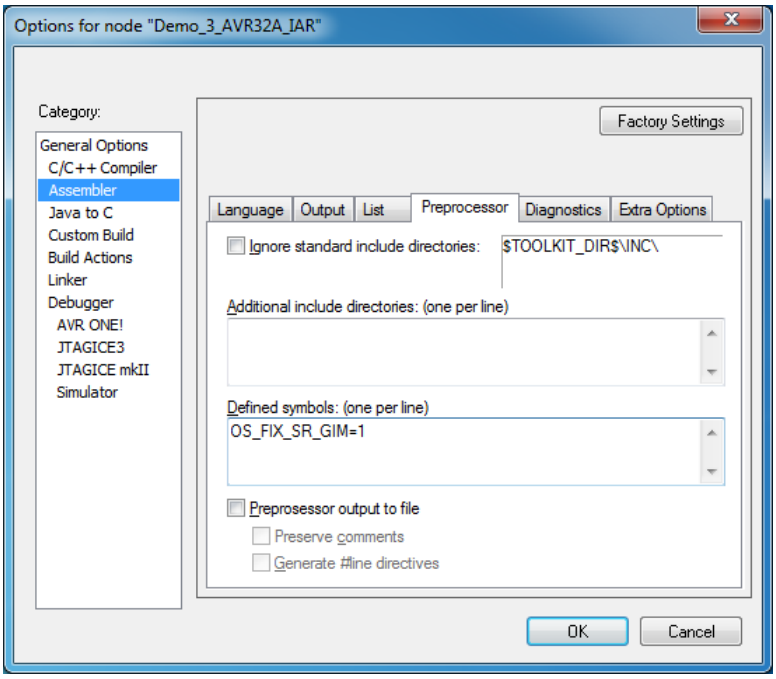

Figure 2-9 GUI set of `OS_FIX_SR_GIM`

2.8 Spurious Interrupt Errata

There is a known and well-documented hardware race condition on some revisions of the AVR32A CPU core when the interrupt request is cleared on the peripheral that has raised the interrupt line. As interrupts are handled in the `Abassi_AVR32A_IAR.s82`, it is possible to include special code that detects and rejects spurious interrupts. This is controlled with the token `OS_SPURIOUS_ISR`, which must be set to a non-zero value to activate the fix; to disable it, it must be set to a zero value. The token is defined at around line 55 in the file `Abassi_AVR32A_IAR.s82`. The distribution code does not enable this fix as spurious interrupts should not happen in a well-designed application:

Table 2-17 Spurious Interrupt Fix Configuration

```
#ifndef OS_SPURIOUS_ISR
OS_SPURIOUS_ISR       EQU 0          /* If code reject spurious interrupts is added  */
#endif
```

NOTE: The operation performed by this fix has not been verified as correct because spurious interrupts where never detected during testing; even after test code was written that broke the rules specified by Atmel to eliminate spurious interrupts. This fix was tested to verify it does not impact in any way the operation of the RTOS.

Alternatively, it is possible to overload the `OS_SPURIOUS_ISR` value set in `Abassi_AVR32A_IAR.s82` by using the assembler command line option `–D` to enable/disable the fix. In the following example, it is enabled:

Table 2-18 Command line set of `OS_SPURIOUS_ISR`

```
aavr32 … -DOS_SPURIOUS_ISR=1 …
```

The enabling / disabling of the fix can also be set for the assembly code through the GUI, in the "*Assembler / Preprocessor*" menu, as shown in the following figure:

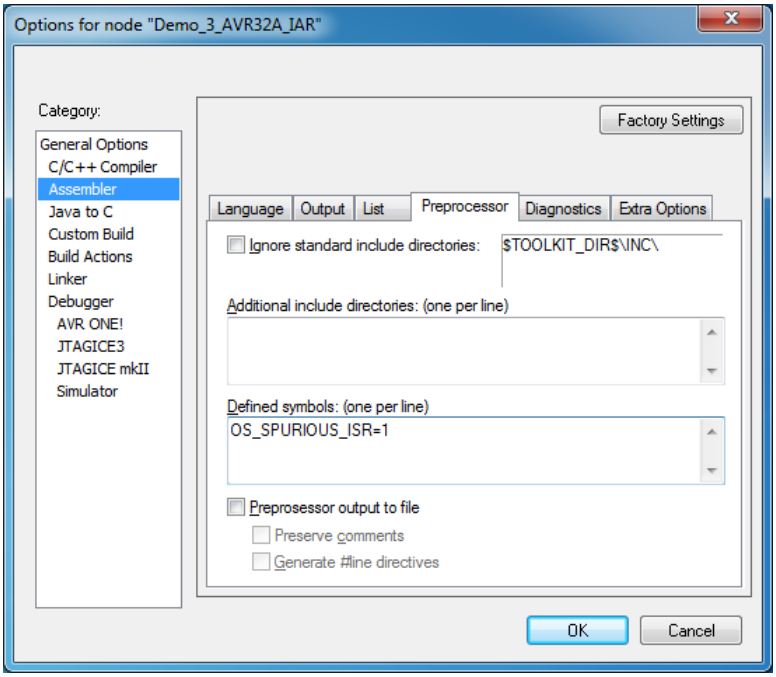

Figure 2-10 GUI set of `OS_SPURIOUS_ISR`

3 Interrupts

The Abassi RTOS needs to be aware when kernel requests are performed inside or outside an interrupt context. For all interrupt sources the Abassi RTOS provides an interrupt dispatcher, which allows it to be interrupt-aware. This dispatcher achieves two goals. First, the kernel uses it to know if a request occurs within an interrupt context or not. Second, using this dispatcher reduces the code size, as all interrupts share the same code for the decision making of entering the kernel or not at the end of the interrupt. When Fast Interrupts are used, the same dispatching operation is performed to determine the interrupt function handler (see Section 2.3).

The number of sources of interrupts is specified by the build options `OS_AVR32A_INT_GRP` and `OS_AVR32_INT_LINE` defined in the file `Abassi.h`[1] (see Section 2).

3.1 Interrupt Handling

3.1.1 Interrupt Installer

Attaching a function to a regular interrupt is quite straightforward. All there is to do is use the RTOS components `OSisrInstall()` and `OSavr32aISRmap()` to specify the function to be attached to that interrupt number. The `OSavr32aISRmap()` component must be used to properly re-map the interrupt group number and interrupt line number for the interrupt dispatcher. Then, the component `OSavr32aISRprio()` must be used to set the priority level of an interrupt group. These components are described further detailed in Section 9.

For example,

Table 3-1 shows the code required to attach the Real Time Clock (RTC) interrupt and set the priority of the interrupt to level 3. On the `AT32UC3B` device family, the RTC interrupt line is attached to group number 1 and line number 8. The example attaches the function `RTChandler` to the RTC interrupt:

Table 3-1 Attaching a Function to an Interrupt

```
#include "Abassi.h"

   …
   OSstart();
   …
   OSisrInstall(OSavr32aISRmap(1,8), &RTChandler);
   OSavr32aISRprio(1, 3);

   … /* More ISR setup */

   OSeint(1);                              /* Global enable of all interrupts        */
```

NOTE: `OSisrInstall()` must be used with the `OSavr32aISRmap()` component.

[1] These build options must be set according to the target device.

At start-up, once `OSstart()` has been called, all interrupt handler functions are set to a "do nothing" function, named `OSinvalidISR()` and the priority level of all interrupt groups are set to 0. If an interrupt function is attached to an interrupt number using the `OSisrInstall()` component before calling `OSstart()`, this attachment will be removed by `OSstart()`; the same will happen to the priority levels. This implies that `OSisrInstall()` should never be used before `OSstart()` has ran. When an interrupt handler is removed, it is very important and necessary to first disable the interrupt source, then the handling function can be set back to `OSinvalidISR()`. This is shown in the Table 3-2:

Table 3-2 Invalidating an ISR handler

```
#include "Abassi.h"

   …
   /* Disable the interrupt source */
   OSisrInstall(OSavr32aISRmap(x,y), &OSinvalidISR);
   …
```

When an application needs to disable/enable the interrupts, the RTOS supplied functions `OSdint()` and `OSeint()` should be used.

The Interrupt Controller (INTC) on the AVR32A does not clear the interrupt generated by a peripheral; neither does the RTOS. This means the peripheral generating the interrupt must be informed to remove the interrupt request. This operation must be performed in the interrupt handler otherwise the interrupt will be re-entered over and over.

As clearly explained in the Atmel documentation, to eliminate the risk of encountering spurious interrupts, a very specific sequence of operations must be performed. Since the RTOS does not generate the interrupt acknowledge to the peripheral, the onus is on the designer to make sure the code sequence is correct. It is also possible to enable a fix aimed at trapping spurious interrupts (see Section 2.8), but, as described in the Atmel documentation, this fix does not eliminate *all* spurious interrupts.

3.2 Interrupt Priority and Enabling

To properly configure interrupts, the interrupt handler must be set with the component `OSisrInstall()` and the interrupt priority level of the interrupt groups must be set with the component `OSavr32aISRprio()`. A key step is to also configure the peripheral to generate interrupts. There is no software provided to configure peripherals, and the IAR Embedded Workbench does not provide any either. However, the Atmel AVR32 Studio provides everything, in source code form, that is required for programming the processor peripherals. Also, most chip manufacturers provide code to configure the specifics on their devices.

3.3 Fast Interrupts

Fast interrupts are supported on this port. A fast interrupt is an interrupt that never uses any component from Abassi and as the name says, is desired to operate as fast as possible. To set-up a fast interrupt, refer to Section 2.3.

NOTE: If an Abassi component is used inside a fast interrupt, the application will misbehave.

3.4 Nested Interrupts

The interrupt controller allows nesting of interrupts; this means an interrupt of higher priority will interrupt the processing of an interrupt of lower priority. Individual interrupt sources can be set to one of 4 levels, where level 0 is the lowest and 3 is the highest.

This implies that the RTOS build option `OS_NESTED_INTS` must be set to a non-zero value. The exception to this is if all enabled interrupts in an application are all set, without exception, to the same priority; then interrupt nesting will not occur. In that case, and only that case, can the build option `OS_NESTED_INTS` be set to zero. As this latter case is quite unlikely, the build option `OS_NESTED_INTS` is always overloaded when compiling the RTOS for the AVR32A. If the latter condition is guaranteed, the overloading located after the pre-processor directive can be modified. The code affected in `Abassi.h` is shown in Table 3-3 below and the line to modify is the one with `#define OX_NESTED_INTS 1`:

Table 3-3 Removing interrupt nesting

```
#elif defined(__ICCARV32__)
  #if (__CORE__ == __AVR32A__)
    …
  #define OX_NESTED_INTS 0     /* The AVR32 has 4 nested interrupt levels */
```

Or if the build option `OS_NESTED_INTS` is desired to be propagated:

Table 3-4 Propagating interrupt nesting

```
#elif defined(__ICCARV32__)
  #if (__CORE__ == __AVR32A__)
    …
  #define OX_NESTED_INTS OS_NESTED_INTS
```

The Abassi RTOS kernel never disables interrupts, but there are a few very small regions within the interrupt dispatcher where interrupts are temporarily disabled due to the nesting (a total of between 10 to 20 instructions).

The kernel is never entered as long as interrupt nesting exists. In all interrupt functions, when a RTOS component that needs to access some kernel functionality is used, the request(s) is/are put in a queue. Only once the interrupt nesting is over (i.e. when only a single interrupt context remains) is the kernel entered at the end of the interrupt, when the queue contains one or more requests, and when the kernel is not already active. This means that only the interrupt handler function operates in an interrupt context, and only the time the interrupt function is using the CPU are other interrupts of equal or lower level blocked by the interrupt controller.

4 Stack Usage

The RTOS uses the tasks' stack for two purposes. When a task is blocked or ready to run but not running, the stack holds the register context that was preserved when the task got blocked or preempted. Also, when an interrupt occurs, the register context of the running task must be preserved in order for the operations performed during the interrupt to not corrupt the contents of the registers used by the task when it got interrupted. For the AVR32A, the context save contents of a blocked or pre-empted task is different from the one used in an interrupt. The following table lists the number of bytes required by each type of context save operation:

Table 4-1 Context Save Stack Requirements

Description	Context save
Blocked/Preempted task context save	`36 bytes`
Blocked/Preempted task context save (Saturation bit kept)	`40 bytes`
Interrupt dispatcher context save (no hybrid stack)	`32 bytes`
Interrupt dispatcher context save (with hybrid stack)	`36 bytes`

The numbers for the interrupt dispatcher context save include the 32 bytes the processor pushes on the stack when it enters the interrupt servicing.

When sizing the stack to allocate to a task, there are three factors to take in account. The first factor is simply that every task in the application needs at least the area to preserve the task context when it is preempted or blocked. Second, one must take into account how many levels of nested interrupts exist in the application. As a worst case, all levels of interrupts may occur and becoming fully nested. So, if N levels of interrupts are used in the application, provision should be made to hold N times the size of an ISR context save on each task stack, plus any added stack used by the interrupt handler functions. Finally, add to all this the stack required by the code implementing the task operation.

NOTE: The AVR32A processor needs alignment on 4 bytes for the instructions accessing word or double word memory. When stack memory is allocated, Abassi guarantees 8 bytes alignment. This said, when sizing `OS_STATIC_STACK` or `OS_ALLOC_SIZE`, make sure to take in account that all allocation performed through these memory pools are by block size multiple of 8 bytes.

If the hybrid interrupt stack (see Section 2.2) is enabled, then the above description changes: it is only necessary to reserve room on task stacks for a single interrupt context save and not the worst-case nesting. With the hybrid stack enabled, the second, third, and so on interrupts use the stack dedicated to the interrupts. The hybrid stack is enabled when the `OS_ISR_STACK` token in file `Abassi_AVR32A_IAR.s82` is set to a non-zero value (see Section 2.2).

5 Search Set-up

The Abassi RTOS build option `OS_SEARCH_FAST` offers three different algorithms to quickly determine the next running task upon task blocking. The following table shows the measurements obtained for the number of CPU cycles required when a task at priority 0 is blocked, and the next running task is at the specified priority. The number of cycles includes everything, not just the search cycle count. The number of cycles was measured using the cycle count register, which increments the counter once every CPU cycle. The second column is when `OS_SEARCH_FAST` is set to zero, meaning a simple array traversing. The third column, labeled Look-up, is when `OS_SEARCH_FAST` is set to 1, which uses an 8 bit look-up table. Finally, the last column is when `OS_SEARCH_FAST` is set to 5 (IAR/AVR32A `int` are 32 bits, so 2^5), meaning a 32 bit look-up table, further searched through successive approximation. The compiler optimization for this measurement was set to *Level High / Speed* optimization for the medium code and small data model. The RTOS build options were set to the minimum feature set, except for option `OS_PRIO_CHANGE` set to non-zero. The presence of this extra feature provokes a small mismatch between the result for a difference of priority of 1, with `OS_SEARCH_FAST` set to zero, and the latency results in Section 7.2.

When the build option `OS_SEARCH_ALGO` is set to a negative value, indicating to use a 2-dimensional linked list search technique instead of the search array, the number of CPU is constant at 201 cycles.

Table 5-1 Search Algorithm Cycle Count

Priority	Linear search	Look-up	Approximation
1	203	224	271
2	205	231	271
3	212	238	271
4	219	245	271
5	226	252	271
6	233	259	271
7	240	266	271
8	247	232	271
9	254	233	271
10	261	240	271
11	268	247	271
12	275	254	271
13	282	261	271
14	289	268	271
15	296	275	271
16	303	241	271
17	310	242	271
18	317	249	271
19	324	256	271
20	331	263	271
21	338	270	271
22	345	277	271
23	352	284	271
24	359	250	271

When `OS_SEARCH_FAST` is set to 0, each extra priority level to traverse requires exactly 7 CPU cycles. When `OS_SEARCH_FAST` is set to 1, each extra priority level to traverse requires exactly 7 CPU cycles, except when the priority level is an exact multiple of 8; then there is a sharp reduction of CPU usage. Overall, setting `OS_SEARCH_FAST` to 1 adds 34 cycles of CPU for the search compared to setting `OS_SEARCH_FAST` to zero. But when the next ready to run priority is less than 8, 16, 24, … then there is around an extra 9 cycles needed but without the 8 times 7 cycle accumulation. Finally, the third option, when `OS_SEARCH_FAST` is set to 5, delivers an almost perfectly constant CPU usage as the algorithm utilizes a successive approximation search technique (when the delta is 32 or more, the CPU cycle count is 280, for 64 or more it is 290).

When the priority span is less than 13, the build option `OS_SEARCH_FAST` should be set to 0, as it is the most efficient algorithm. For a priority span greater than 12, then the build option should be set to 1 as it delivers overall a better performance than when set to 5.

Setting the build option `OS_SEARCH_ALGO` to a non-negative value minimizes the time needed to change the state of a task from blocked to ready to run, and not the time needed to find the next running task upon blocking/suspending of the running task. If the application needs are such that the critical real-time requirement is to get the next running task up and running as fast as possible, then set the build option `OS_SEARCH_ALGO` to a negative value.

6 Chip Support

No custom chip support is provided with the distribution code. Also, most device manufacturers provide code to configure the peripherals on their devices. The distribution code contains some of the manufacturer's open source libraries, e.g. Atmel Software Framework.

7 Measurements

This section gives an overview of the memory requirements and the CPU latency encountered when the RTOS is used on the AVR32A and compiled with IAR. The CPU cycles are exactly the CPU clock cycles, as the processor executes one instruction at every clock transition.

7.1 Memory

The memory numbers are supplied for the two limit cases of build options (and some in-between): the smallest footprint is the RTOS built with only the minimal feature set, and the other with almost all the features. For both cases, names are not part of the build. This feature was removed from the metrics because it is highly probable that shipping products utilizing this RTOS will not include the naming of descriptors, as its usefulness is mainly limited to debugging and making the opening/creation of components run-time safe.

The code size numbers are expressed with “less than” as they have been rounded up to multiples of 25 for the “C” code. These numbers were obtained using the release version 1.122.205 of the RTOS and may change in other versions. One should interpret these numbers as the “very likely” numbers for other released versions of the RTOS.

The code memory required by the RTOS includes the “C” code and assembly language code used by the RTOS. The code optimization settings of the compiler that were used for the memory measurements are:

1. Optimization level: High
2. Optimize for: Size
3. All transformations are enabled

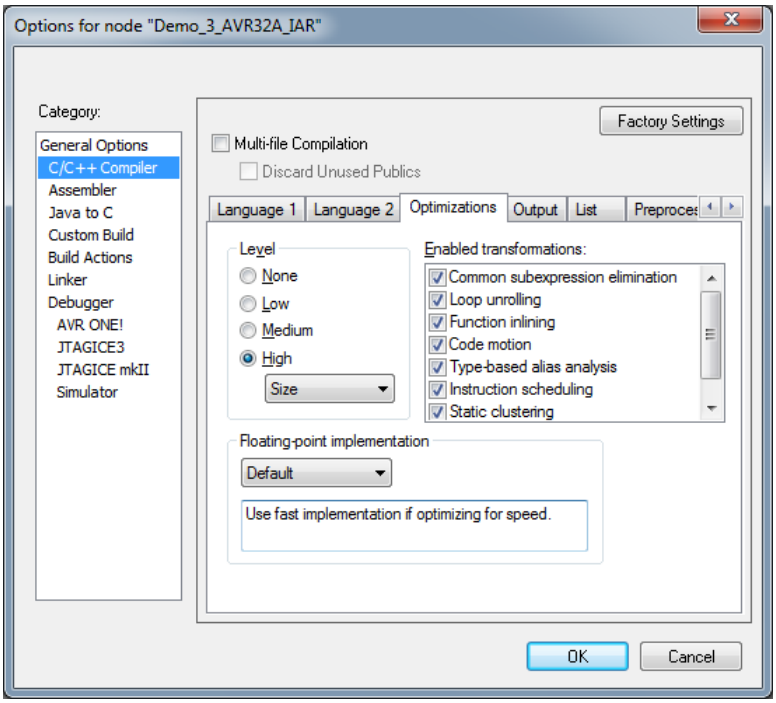

Figure 7-1 Memory Measurement Code Optimization Settings

Table 7-1 "C" Code Memory Usage (Medium Code)

Description	(Small Data)Size	(Large Data) Size
Minimal Build	< 625 bytes	< 650 bytes
+ Runtime service creation / static memory	< 850 bytes	< 875 bytes
+ Multiple tasks at same priority	< 950 bytes	< 1000 bytes
+ Runtime priority change + Mutex priority inheritance + FCFS + Task suspension	< 1375 bytes	< 1475 bytes
+ Timer & timeout + Timer call back + Round robin	< 1825 bytes	< 2000 bytes
+ Events + Mailbox	< 2475 bytes	< 2675 bytes
Full Feature Build (no names)	< 3000 bytes	< 3225 bytes
Full Feature Build (no names / no runtime creation)	< 2675 bytes	< 2875 bytes
Full Feature Build (no names / no runtime creation) + Timer services module	< 3025 bytes	< 3250 bytes

Table 7-2 "C" Code Memory Usage (Large Code)

Description	(Small Data)Size	(Large Data) Size
Minimal Build	< 650 bytes	< 675 bytes
+ Runtime service creation / static memory	< 850 bytes	< 875 bytes
+ Multiple tasks at same priority	< 975 bytes	< 1025 bytes
+ Runtime priority change + Mutex priority inheritance + FCFS + Task suspension	< 1400 bytes	< 1500 bytes
+ Timer & timeout + Timer call back + Round robin	< 1850 bytes	< 2025 bytes
+ Events + Mailbox	< 2500 bytes	< 2700 bytes
Full Feature Build (no names)	< 3025 bytes	< 3225 bytes
Full Feature Build (no names / no runtime creation)	< 2700 bytes	< 2900 bytes
Full Feature Build (no names / no runtime creation) + Timer services module	< 3050 bytes	< 3275 bytes

Table 7-3 Assembly Code Memory Usage

Description	ISR stack
Assembly code size	270 bytes
Interrupt vector table	258 bytes
ISR stack	+ 22 bytes
Fast Interrupts (independent of `OS_FAST_INTS` value)	+ 40 bytes
Handle Saturation bit	+ 30 bytes
Fix for RETE	+ 10 bytes
Fix for interrupt masking	+ 8 bytes
Fix for spurious interrupts	+ 54 bytes

The assembly code sizes are the values for each feature individually enabled; all other features been disabled. There are interdependencies; for example, the fast interrupt code has optional code to deal with the spurious interrupts. The measurements shown for the spurious interrupt does not take in account the extra code added in the fast interrupt code when the spurious interrupt feature is enabled.

There are two aspects when describing the data memory usage by the RTOS. First, the RTOS needs its own data memory to operate, and second, most of the services offered by the RTOS require data memory for each instance of the service. As the build options affect either the kernel memory needs or the service descriptors (or both), an interactive calculator has been made available on the Code Time Technologies website.

7.2 Latency

Latency of operations has been measured on an Atmel EVK1101 evaluation board. This evaluation board is populated with a 66 MHz `AT32UC3B0256` device, but for the measurements, the CPU was clocked at the same frequency as the external 12 MHz crystal. All measurements have been performed on the real platform, using the cycle count register as the measuring element. Other models will have slightly different code and data requirements. The code optimization settings that were used for the latency measurements are:

1. Optimization level: High
2. Optimize for: Speed
3. All transformations are enabled

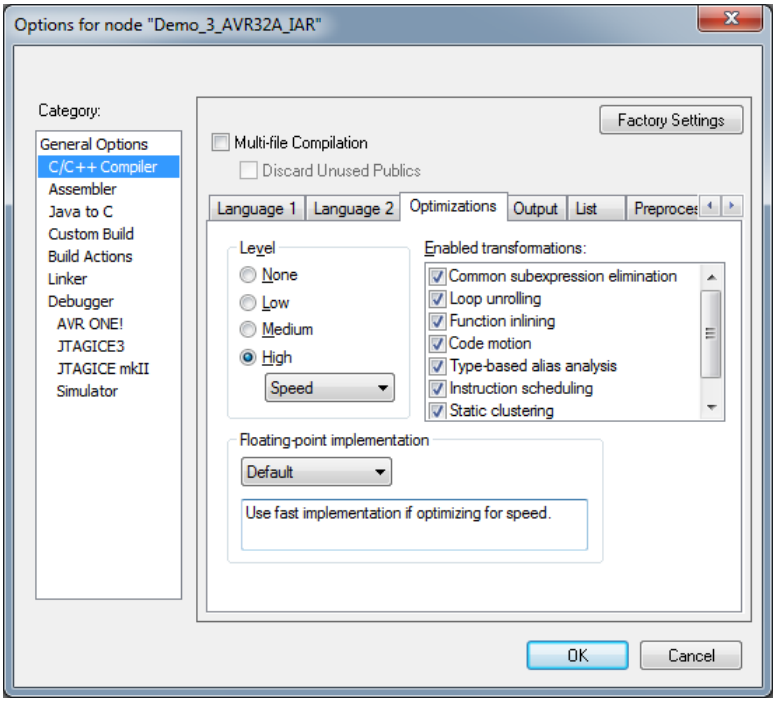

Figure 7-2 Latency Measurement Code Optimization Settings

There are 5 types of latencies that are measured, and these 5 measurements are expected to give a very good overview of the real-time performance of the Abassi RTOS for this port. For all measurements, three tasks were involved:

1. Adam & Eve set to a priority value of 0;
2. A low priority task set to a priority value of 1;
3. The Idle task set to a priority value of 20.

The sets of 5 measurements are performed on a semaphore, on the event flags of a task and finally on a mailbox. The first 2 latency measurements use the component in a manner where there is no task switching. The third measurements involve a high priority task getting blocked by the component. The fourth measurements are about the opposite: a low priority task getting pre-empted because the component unblocks a high priority task. Finally, the reaction to unblocking a task, which becomes the running task, through an interrupt is provided.

The first set of measurements counts the number of CPU cycles elapsed starting right before the component is used until it is back from the component. For these measurement there is no task switching. This means:

Table 7-4 Measurement without Task Switch

```
Start CPU cycle count
SEMpost(…);  or  EVTset(…);  or  MBXput();
Stop CPU cycle count
```

The second set of measurements, as for the first set, counts the number of CPU cycles elapsed starting right before the component is used until it is back from the component. For these measurement there is no task switching. This means:

Table 7-5 Measurement without Blocking

```
Start CPU cycle count
SEMwait(…, -1);  or  EVTwait(…, -1);  or  MBXget(…, -1);
Stop CPU cycle count
```

The third set of measurements counts the number of CPU cycles elapsed starting right before the component triggers the unblocking of a higher priority task until the latter is back from the component used that blocked the task. This means:

Table 7-6 Measurement with Task Switch

```
main()
{
    …
    SEMwait(…, -1);  or  EVTwait(…, -1);  or  MBXget(…, -1);
    Stop CPU cycle count
    …
}

TaskPrio1()
{
    …
    Start CPU cycle count
    SEMpost(…);      or  EVTset(…);       or  MBXput(…);
    …
}
```

The forth set of measurements counts the number of CPU cycles elapsed starting right before the component blocks of a high priority task until the next ready to run task is back from the component it was blocked on; the blocking was provoked by the unblocking of a higher priority task. This means:

Table 7-7 Measurement with Task unblocking

```
main()
{
    …
    Start CPU cycle count
    SEMwait(…, -1);  or  EVTwait(…, -1);  or  MBXget(…, -1);
    …
}

TaskPrio1()
{
    …
    SEMpost(…);      or  EVTset(…);       or  MBXput(…);
    Stop CPU cycle count
    …
}
```

The fifth set of measurements counts the number of CPU cycles elapsed from the beginning of an interrupt using the component, until the task that was blocked becomes the running task and is back from the component used that blocked the task. It is the same as the third set of measurement except the count register paired with the compare register are used as the source of interrupt. The interrupt latency measurement includes everything involved in the interrupt operation, even the cycles the processor needs to push the interrupt context before entering the interrupt code. The interrupt function, attached with `OSisrInstall()`, is simply a three line function that, first clears the interrupt request by writing to the compare register, which is then followed by the use of the appropriate RTOS component, and then finally a return.

The following table lists the results obtained, where the cycle count is measured using the count register on the AVR32A. This counter increments its count by 1 at every CPU cycle. As was the case for the memory measurements, these numbers were obtained with a beta release of the RTOS. It is possible the released version of the RTOS may have slightly different numbers.

The interrupt latency is the number of cycles elapsed when the interrupt trigger occurred and the ISR function handler is entered. This includes the number of cycles used by the processor to set-up the interrupt stack and to branch to the address specified in the interrupt vector table.

The interrupt overhead without entering the kernel is the measurement of the number of CPU cycles used between the entry point in the interrupt vector and the return from interrupt, with a “do nothing” function in the `OSisrInstall()`. The interrupt trigger was the cycle counter itself. The interrupt overhead when entering the kernel is calculated using the results from the third and fifth tests. Finally, the OS context switch is the measurement of the number of CPU cycles it takes to perform a task switch, without involving the wrap-around code of the synchronization component.

None of the features that can be enabled in the file `Abassi_AVR32A_IAR.s82` are enabled. This means:

- No hybrid stack
- Fast interrupt are disabled
- The saturation bit is not propagated
- The RETE errata fix in not enabled
- The interrupt masking fix is not enabled

In the following table, the latency numbers between parentheses are the measurements when the build option `OS_SEARCH_ALGO` is set to a negative value. The regular number is the latency measurements when the build option `OS_SEARCH_ALGO` is set to 0.

Table 7-8 Latency Measurements (Medium Code / Small Data)

Description	Minimal Features	Full Features
Semaphore posting no task switch	111 (109)	159 (161)
Semaphore waiting no blocking	116 (112)	170 (172)
Semaphore posting with task switch	171 (194)	271 (301)
Semaphore waiting with blocking	187 (181)	303 (308)
Semaphore posting in ISR with task switch	392 (410)	492 (520)
Event setting no task switch	n/a	155 (157)
Event getting no blocking	n/a	176 (178)
Event setting with task switch	n/a	286 (316)
Event getting with blocking	n/a	315 (320)
Event setting in ISR with task switch	n/a	510 (535)
Mailbox writing no task switch	n/a	200 (202)
Mailbox reading no blocking	n/a	227 (229)
Mailbox writing with task switch	n/a	329 (359)
Mailbox reading with blocking	n/a	359 (364)
Mailbox writing in ISR with task switch	n/a	560 (585)
Interrupt Latency	52	52
Interrupt overhead entering the kernel	221 (216)	221 (219)
Interrupt overhead NOT entering the kernel	73	73
Context switch	35	36

Table 7-9 Latency Measurements (Medium Code / Large Data)

Description	Minimal Features	Full Features
Semaphore posting no task switch	112 (110)	159 (161)
Semaphore waiting no blocking	117 (113)	172 (174)
Semaphore posting with task switch	172 (195)	273 (303)
Semaphore waiting with blocking	188 (182)	303 (308)
Semaphore posting in ISR with task switch	396 (414)	496 (521)
Event setting no task switch	n/a	157 (159)
Event getting no blocking	n/a	177 (179)
Event setting with task switch	n/a	288 (318)
Event getting with blocking	n/a	316 (321)
Event setting in ISR with task switch	n/a	513 (541)
Mailbox writing no task switch	n/a	202 (204)
Mailbox reading no blocking	n/a	228 (230)
Mailbox writing with task switch	n/a	330 (360)
Mailbox reading with blocking	n/a	361 (366)
Mailbox writing in ISR with task switch	n/a	564 (589)
Interrupt Latency	52	52
Interrupt overhead entering the kernel	224 (219)	223 (218)
Interrupt overhead NOT entering the kernel	73	73
Context switch	35	35

Table 7-10 Latency Measurements (Large Code / Small Data)

Description	Minimal Features	Full Features
Semaphore posting no task switch	111 (109)	159 (161)
Semaphore waiting no blocking	116 (112)	170 (172)
Semaphore posting with task switch	171 (194)	271 (301)
Semaphore waiting with blocking	187 (181)	303 (308)
Semaphore posting in ISR with task switch	392 (410)	492 (520)
Event setting no task switch	n/a	155 (157)
Event getting no blocking	n/a	176 (178)
Event setting with task switch	n/a	286 (316)
Event getting with blocking	n/a	315 (320)
Event setting in ISR with task switch	n/a	510 (535)
Mailbox writing no task switch	n/a	200 (202)
Mailbox reading no blocking	n/a	227 (229)
Mailbox writing with task switch	n/a	329 (359)
Mailbox reading with blocking	n/a	359 (364)
Mailbox writing in ISR with task switch	n/a	560 (585)
Interrupt Latency	52	52
Interrupt overhead entering the kernel	221 (216)	221 (219)
Interrupt overhead NOT entering the kernel	73	73
Context switch	35	36

Table 7-11 Latency Measurements (Large Code / Large Data)

Description	Minimal Features	Full Features
Semaphore posting no task switch	112 (110)	159 (161)
Semaphore waiting no blocking	117 (113)	172 (174)
Semaphore posting with task switch	172 (195)	273 (303)
Semaphore waiting with blocking	188 (182)	303 (308)
Semaphore posting in ISR with task switch	396 (414)	496 (521)
Event setting no task switch	n/a	157 (159)
Event getting no blocking	n/a	177 (179)
Event setting with task switch	n/a	288 (318)
Event getting with blocking	n/a	316 (321)
Event setting in ISR with task switch	n/a	513 (541)
Mailbox writing no task switch	n/a	202 (204)
Mailbox reading no blocking	n/a	228 (230)
Mailbox writing with task switch	n/a	330 (360)
Mailbox reading with blocking	n/a	361 (366)
Mailbox writing in ISR with task switch	n/a	564 (589)
Interrupt Latency	52	52
Interrupt overhead entering the kernel	224 (219)	223 (218)
Interrupt overhead NOT entering the kernel	73	73
Context switch	35	35

8 Appendix A: Build Options for Code Size

8.1 Case 0: Minimum build

Table 8-1: Case 0 build options

```
#define OS_ALLOC_SIZE       0    /* When !=0, RTOS supplied OSalloc                  */
#define OS_COOPERATIVE      0    /* When !=0, the  kernel is in cooperative mode     */
#define OS_EVENTS           0    /* If event flags are supported                     */
#define OS_FCFS             0    /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK       0    /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE     0    /* Type of logging to use                           */
#define OS_MAILBOX          0    /* If mailboxes are used                            */
#define OS_MAX_PEND_RQST    2U   /* Maximum number of requests posted in ISRs        */
#define OS_MIN_STACK_USE    0    /* If the kernel minimizes its stack usage          */
#define OS_MTX_DEADLOCK     0    /* To enable the mutex deadlock detection           */
#define OS_MTX_INVERSION    0    /* To enable & type of protection against prio inv  */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes   */
#ifndef OS_NESTED_INTS
  #define OS_NESTED_INTS    0    /* If operating with nested interrupts              */
#endif
#define OS_PRIO_CHANGE      0    /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN         2    /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME        0    /* Does not Support multiple same priority tasks    */
#define OS_ROUND_ROBIN      0    /* Use round-robin, value specifies period in uS    */
#define OS_RUNTIME          0    /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO      0    /* If using a fast search                           */
#define OS_STARVE_PRIO      0    /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX   0    /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX  0    /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX   0    /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX       0    /* If !=0 how many mailboxes                        */
#define OS_STATIC_NAME      0    /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM       0    /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK     0    /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK      0    /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_STATIC_TIM_SRV   0    /* If !=0 how many timer services                   */
#define OS_TASK_SUSPEND     0    /* If a task can suspend another one                */
#define OS_TIMEOUT          0    /* !=0 enables blocking timeout                     */
#define OS_TIMER_CB         0    /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV        0    /* !=0 includes the timer services                  */
#define OS_TIMER_US         0    /* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG     0    /* If tasks have arguments                          */
```

8.2 Case 1: + Runtime service creation / static memory

Table 8-2: Case 1 build options

```
#define OS_ALLOC_SIZE       0    /* When !=0, RTOS supplied OSalloc                  */
#define OS_COOPERATIVE      0    /* When !=0, the  kernel is in cooperative mode     */
#define OS_EVENTS           0    /* If event flags are supported                     */
#define OS_FCFS             0    /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK       0    /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE     0    /* Type of logging to use                           */
#define OS_MAILBOX          0    /* If mailboxes are used                            */
#define OS_MAX_PEND_RQST    2U   /* Maximum number of requests posted in ISRs        */
#define OS_MIN_STACK_USE    0    /* If the kernel minimizes its stack usage          */
#define OS_MTX_DEADLOCK     0    /* To enable the mutex deadlock detection           */
#define OS_MTX_INVERSION    0    /* To enable protection against priority inversion  */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes   */
#ifndef OS_NESTED_INTS
  #define OS_NESTED_INTS    0    /* If operating with nested interrupts              */
#endif
#define OS_PRIO_CHANGE      0    /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN         2    /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME        0    /* Support multiple tasks with the same priority    */
#define OS_ROUND_ROBIN      0    /* Use round-robin, value specifies period in uS    */
#define OS_RUNTIME          1    /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO      0    /* If using a fast search                           */
#define OS_STARVE_PRIO      0    /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX   0    /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX  0    /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX   0    /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX       0    /* If !=0 how many mailboxes                        */
#define OS_STATIC_NAME      0    /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM       5    /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK     128  /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK      5    /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_STATIC_TIM_SRV   0    /* If !=0 how many timer services                   */
#define OS_TASK_SUSPEND     0    /* If a task can suspend another one                */
#define OS_TIMEOUT          0    /* !=0 enables blocking timeout                     */
#define OS_TIMER_CB         0    /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV        0    /* !=0 includes the timer services                  */
#define OS_TIMER_US         0    /* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG     0    /* If tasks have arguments                          */
```

8.3 Case 2: + Multiple tasks at same priority

Table 8-3: Case 2 build options

```
#define OS_ALLOC_SIZE       0    /* When !=0, RTOS supplied OSalloc                  */
#define OS_COOPERATIVE      0    /* When !=0, the  kernel is in cooperative mode     */
#define OS_EVENTS           0    /* If event flags are supported                     */
#define OS_FCFS             0    /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK       0    /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE     0    /* Type of logging to use                           */
#define OS_MAILBOX          0    /* If mailboxes are used                            */
#define OS_MAX_PEND_RQST    32U  /* Maximum number of requests posted in ISRs        */
#define OS_MIN_STACK_USE    0    /* If the kernel minimizes its stack usage          */
#define OS_MTX_DEADLOCK     0    /* To enable the mutex deadlock detection           */
#define OS_MTX_INVERSION    0    /* To enable protection against priority inversion  */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes   */
#ifndef OS_NESTED_INTS
  #define OS_NESTED_INTS    0    /* If operating with nested interrupts              */
#endif
#define OS_PRIO_CHANGE      0    /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN         20   /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME        1    /* Support multiple tasks with the same priority    */
#define OS_ROUND_ROBIN      0    /* Use round-robin, value specifies period in uS    */
#define OS_RUNTIME          1    /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO      0    /* If using a fast search                           */
#define OS_STARVE_PRIO      0    /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX   0    /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX  0    /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX   0    /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX       0    /* If !=0 how many mailboxes                        */
#define OS_STATIC_NAME      0    /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM       5    /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK     128  /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK      5    /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_STATIC_TIM_SRV   0    /* If !=0 how many timer services                   */
#define OS_TASK_SUSPEND     0    /* If a task can suspend another one                */
#define OS_TIMEOUT          0    /* !=0 enables blocking timeout                     */
#define OS_TIMER_CB         0    /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV        0    /* !=0 includes the timer services                  */
#define OS_TIMER_US         0    /* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG     0    /* If tasks have arguments                          */
```

8.4 Case 3: + Priority change / Priority inheritance / FCFS / Task suspend

Table 8-4: Case 3 build options

```
#define OS_ALLOC_SIZE       0    /* When !=0, RTOS supplied OSalloc                  */
#define OS_COOPERATIVE      0    /* When !=0, the  kernel is in cooperative mode     */
#define OS_EVENTS           0    /* If event flags are supported                     */
#define OS_FCFS             1    /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK       0    /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE     0    /* Type of logging to use                           */
#define OS_MAILBOX          0    /* If mailboxes are used                            */
#define OS_MAX_PEND_RQST    32U  /* Maximum number of requests posted in ISRs        */
#define OS_MIN_STACK_USE    0    /* If the kernel minimizes its stack usage          */
#define OS_MTX_DEADLOCK     0    /* To enable the mutex deadlock detection           */
#define OS_MTX_INVERSION    1    /* To enable protection against priority inversion  */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes   */
#ifndef OS_NESTED_INTS
  #define OS_NESTED_INTS    0    /* If operating with nested interrupts              */
#endif
#define OS_PRIO_CHANGE      1    /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN         20   /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME        1    /* Support multiple tasks with the same priority    */
#define OS_ROUND_ROBIN      0    /* Use round-robin, value specifies period in uS    */
#define OS_RUNTIME          1    /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO      0    /* If using a fast search                           */
#define OS_STARVE_PRIO      0    /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX   0    /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX  0    /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX   0    /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX       0    /* If !=0 how many mailboxes                        */
#define OS_STATIC_NAME      0    /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM       5    /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK     128  /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK      5    /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_STATIC_TIM_SRV   0    /* If !=0 how many timer services                   */
#define OS_TASK_SUSPEND     1    /* If a task can suspend another one                */
#define OS_TIMEOUT          0    /* !=0 enables blocking timeout                     */
#define OS_TIMER_CB         0    /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV        0    /* !=0 includes the timer services                  */
#define OS_TIMER_US         0    /* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG     0    /* If tasks have arguments                          */
```

8.5 Case 4: + Timer & timeout / Timer call back / Round robin

Table 8-5: Case 4 build options

```
#define OS_ALLOC_SIZE       0    /* When !=0, RTOS supplied OSalloc                  */
#define OS_COOPERATIVE      0    /* When !=0, the  kernel is in cooperative mode     */
#define OS_EVENTS           0    /* If event flags are supported                     */
#define OS_FCFS             1    /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK       0    /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE     0    /* Type of logging to use                           */
#define OS_MAILBOX          0    /* If mailboxes are used                            */
#define OS_MAX_PEND_RQST    32U  /* Maximum number of requests posted in ISRs        */
#define OS_MIN_STACK_USE    0    /* If the kernel minimizes its stack usage          */
#define OS_MTX_DEADLOCK     0    /* To enable the mutex deadlock detection           */
#define OS_MTX_INVERSION    1    /* To enable protection against priority inversion  */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes   */
#ifndef OS_NESTED_INTS
  #define OS_NESTED_INTS    0    /* If operating with nested interrupts              *
#endif
#define OS_PRIO_CHANGE      1    /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN         20   /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME        1    /* Support multiple tasks with the same priority    */
#define OS_ROUND_ROBIN      100000  /* Use round-robin, value specifies period in uS */
#define OS_RUNTIME          1    /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO      0    /* If using a fast search                           */
#define OS_STARVE_PRIO      0    /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX   0    /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX  0    /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX   0    /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX       0    /* If !=0 how many mailboxes                        */
#define OS_STATIC_NAME      0    /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM       5    /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK     128  /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK      5    /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_STATIC_TIM_SRV   0    /* If !=0 how many timer services                   */
#define OS_TASK_SUSPEND     1    /* If a task can suspend another one                */
#define OS_TIMEOUT          1    /* !=0 enables blocking timeout                     */
#define OS_TIMER_CB         10   /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV        0    /* !=0 includes the timer services                  */
#define OS_TIMER_US         50000/* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG     0    /* If tasks have arguments                          */
```

8.6 Case 5: + Events / Mailboxes

Table 8-6: Case 5 build options

```
#define OS_ALLOC_SIZE       0    /* When !=0, RTOS supplied OSalloc                  */
#define OS_COOPERATIVE      0    /* When !=0, the  kernel is in cooperative mode     */
#define OS_EVENTS           1    /* If event flags are supported                     */
#define OS_FCFS             1    /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK       0    /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE     0    /* Type of logging to use                           */
#define OS_MAILBOX          1    /* If mailboxes are used                            */
#define OS_MAX_PEND_RQST    32U  /* Maximum number of requests posted in ISRs        */
#define OS_MIN_STACK_USE    0    /* If the kernel minimizes its stack usage          */
#define OS_MTX_DEADLOCK     0    /* To enable the mutex deadlock detection           */
#define OS_MTX_INVERSION    1    /* To enable protection against priority inversion  */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes   */
#ifndef OS_NESTED_INTS
  #define OS_NESTED_INTS    0    /* If operating with nested interrupts              */
#endif
#define OS_PRIO_CHANGE      1    /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN         20   /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME        1    /* Support multiple tasks with the same priority    */
#define OS_ROUND_ROBIN      100000  /* Use round-robin, value specifies period in uS */
#define OS_RUNTIME          1    /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO      0    /* If using a fast search                           */
#define OS_STARVE_PRIO      0    /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX   0    /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX  0    /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX   10   /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX       2    /* If !=0 how many mailboxes                        */
#define OS_STATIC_NAME      0    /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM       5    /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK     128  /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK      5    /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_STATIC_TIM_SRV   0    /* If !=0 how many timer services                   */
#define OS_TASK_SUSPEND     1    /* If a task can suspend another one                */
#define OS_TIMEOUT          1    /* !=0 enables blocking timeout                     */
#define OS_TIMER_CB         10   /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV        0    /* !=0 includes the timer services                  */
#define OS_TIMER_US         50000/* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG     0    /* If tasks have arguments                          */
```

8.7 Case 6: Full feature Build (no names)

Table 8-7: Case 6 build options

```
#define OS_ALLOC_SIZE       0    /* When !=0, RTOS supplied OSalloc                  */
#define OS_COOPERATIVE      0    /* When !=0, the  kernel is in cooperative mode     */
#define OS_EVENTS           1    /* If event flags are supported                     */
#define OS_FCFS             1    /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK       0    /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE     0    /* Type of logging to use                           */
#define OS_MAILBOX          1    /* If mailboxes are used                            */
#define OS_MAX_PEND_RQST    32U  /* Maximum number of requests posted in ISRs        */
#define OS_MIN_STACK_USE    0    /* If the kernel minimizes its stack usage          */
#define OS_MTX_DEADLOCK     0    /* To enable the mutex deadlock detection           */
#define OS_MTX_INVERSION    1    /* To enable protection against priority inversion  */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes   */
#ifndef OS_NESTED_INTS
  #define OS_NESTED_INTS    0    /* If operating with nested interrupts              */
#endif
#define OS_PRIO_CHANGE      1    /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN         20   /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME        1    /* Support multiple tasks with the same priority    */
#define OS_ROUND_ROBIN      -100000 /* Use round-robin, value specifies period in uS */
#define OS_RUNTIME          1    /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO      0    /* If using a fast search                           */
#define OS_STARVE_PRIO      -3   /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX   -10  /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX  -100 /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX   10   /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX       2    /* If !=0 how many mailboxes                        */
#define OS_STATIC_NAME      0    /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM       5    /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK     128  /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK      5    /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_STATIC_TIM_SRV   0    /* If !=0 how many timer services                   */
#define OS_TASK_SUSPEND     1    /* If a task can suspend another one                */
#define OS_TIMEOUT          1    /* !=0 enables blocking timeout                     */
#define OS_TIMER_CB         10   /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV        0    /* !=0 includes the timer services                  */
#define OS_TIMER_US         50000/* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG     1    /* If tasks have arguments                          */
```

8.8 Case 7: Full feature Build (no names / no runtime creation)

Table 8-8: Case 7 build options

```
#define OS_ALLOC_SIZE       0    /* When !=0, RTOS supplied OSalloc                  */
#define OS_COOPERATIVE      0    /* When !=0, the  kernel is in cooperative mode     */
#define OS_EVENTS           1    /* If event flags are supported                     */
#define OS_FCFS             1    /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK       0    /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE     0    /* Type of logging to use                           */
#define OS_MAILBOX          1    /* If mailboxes are used                            */
#define OS_MAX_PEND_RQST    32U  /* Maximum number of requests posted in ISRs        */
#define OS_MIN_STACK_USE    0    /* If the kernel minimizes its stack usage          */
#define OS_MTX_DEADLOCK     0    /* To enable the mutex deadlock detection           */
#define OS_MTX_INVERSION    1    /* To enable protection against priority inversion  */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes   */
#ifndef OS_NESTED_INTS
  #define OS_NESTED_INTS    0    /* If operating with nested interrupts              */
#endif
#define OS_PRIO_CHANGE      1    /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN         20   /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME        1    /* Support multiple tasks with the same priority    */
#define OS_ROUND_ROBIN      -100000 /* Use round-robin, value specifies period in uS */
#define OS_RUNTIME          0    /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO      0    /* If using a fast search                           */
#define OS_STARVE_PRIO      -3   /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX   -10  /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX  -100 /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX   0    /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX       0    /* If !=0 how many mailboxes                        */
#define OS_STATIC_NAME      0    /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM       0    /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK     0    /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK      0    /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_STATIC_TIM_SRV   0    /* If !=0 how many timer services                   */
#define OS_TASK_SUSPEND     1    /* If a task can suspend another one                */
#define OS_TIMEOUT          1    /* !=0 enables blocking timeout                     */
#define OS_TIMER_CB         10   /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV        0    /* !=0 includes the timer services                  */
#define OS_TIMER_US         50000/* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG     1    /* If tasks have arguments                          */
```

8.9 Case 8: Full build adding the optional timer services

Table 8-9: Case 8 build options

```
#define OS_ALLOC_SIZE       0    /* When !=0, RTOS supplied OSalloc                  */
#define OS_COOPERATIVE      0    /* When !=0, the  kernel is in cooperative mode     */
#define OS_EVENTS           1    /* If event flags are supported                     */
#define OS_FCFS             1    /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK       0    /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE     0    /* Type of logging to use                           */
#define OS_MAILBOX          1    /* If mailboxes are used                            */
#define OS_MAX_PEND_RQST    32U  /* Maximum number of requests posted in ISRs        */
#define OS_MIN_STACK_USE    0    /* If the kernel minimizes its stack usage          */
#define OS_MTX_DEADLOCK     0    /* To enable the mutex deadlock detection           */
#define OS_MTX_INVERSION    1    /* To enable protection against priority inversion  */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes   */
#ifndef OS_NESTED_INTS
  #define OS_NESTED_INTS    0    /* If operating with nested interrupts              */
#endif
#define OS_PRIO_CHANGE      1    /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN         20   /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME        1    /* Support multiple tasks with the same priority    */
#define OS_ROUND_ROBIN      -100000 /* Use round-robin, value specifies period in uS */
#define OS_RUNTIME          0    /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO      0    /* If using a fast search                           */
#define OS_STARVE_PRIO      -3   /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX   -10  /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX  -100 /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX   0    /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX       0    /* If !=0 how many mailboxes                        */
#define OS_STATIC_NAME      0    /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM       0    /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK     0    /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK      0    /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_STATIC_TIM_SRV   0    /* If !=0 how many timer services                   */
#define OS_TASK_SUSPEND     1    /* If a task can suspend another one                */
#define OS_TIMEOUT          1    /* !=0 enables blocking timeout                     */
#define OS_TIMER_CB         10   /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV        1    /* !=0 includes the timer services                  */
#define OS_TIMER_US         50000/* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG     1    /* If tasks have arguments                          */
```

9 Appendix B: AVR32A Interrupt Components

9.1 OSisrInstall

Synopsis

```
#include "Abassi.h"

void OSisrInstall(int IntrptID, void (* Fct)(void));
```

Description

The component `OSisrInstall()` attaches the interrupt function handler, specified by the argument `Fct`, to an interrupt source. The interrupt source, which is defined by the group it is part of and which line it is attached to, is specified by the argument `IntrptID`; this argument must always be provided through the component `OSavr32aISRmap`.

Availability

AVR32A port only.

Arguments

`IntrptID`	Interrupt identifier, as computed by the component `OSavr32aISRmap()`, that specifies the group and line of the interrupt to attach the function `Fct` to.
`Fct`	Function to attach to the interrupt source indicated by the argument `IntrpID`.

Returns

`void`

Component type

Macro (safe)

Options

None.

Notes

See also

`OSavr32aISRmap` (Section 9.2)
`OSavr32ISRprio` (Section 9.3)

9.2 OSavr32aISRmap

Synopsis

```
#include "Abassi.h"

int OSavr32aISRmap(int Group, int Line);
```

Description

The component `OSavr32aISRmap()` computes a unique identifier for an interrupt source the RTOS interrupt dispatcher uses.

Availability

AVR32A port only.

Arguments

`Group`	Group number the interrupt source belongs to
`Line`	Line number the interrupt source is attached to in the group.

Returns

`int`	Unique ID

Component type

Macro (safe)

Options

None.

Notes

See also

`OSisrIinstall` (Section 9.1)

9.3 OSavr32aISRprio

Synopsis

```
#include "Abassi.h"

void OSavr32aISRprio(int Group, int Prio);
```

Description

The component `OSavr32aISRprio()` sets the interrupt priority level of an interrupt group.

Availability

AVR32A port only.

Arguments

`Group`	Group number to set the interrupt priority
`Prio`	Priority level to set (0 lowest, 3 highest)

Returns

`void`

Component type

Macro (safe)

Options

None.

Notes

See also

`OSisrInstall` (Section 9.1)