CODE TIME TECHNOLOGIES

Abassi RTOS

Porting Document

AVR32A – GCC

Disclaimer

Table of Contents

List of Figures

List of Tables

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 Atmel Studio using the GCC tools suite; the version used for the port and all tests is Version 6.0.1996 Service Pack 2.

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_GCC.s`	RTOS assembly file for the AVR32A to use with the Atmel Studio
`Demo_0_EVK1101_GCC.c`	Demo code that runs on the Atmel EVK1101 evaluation board using the LEDs & serial port
`Demo_0_EVK1101_GCC.mak`	Makefile for Demo #1
`Demo_3_EVK1101_GCC.c`	Demo code that runs on the Atmel EVK1101 evaluation board using the serial port
`Demo_3_EVK1101_GCC.mak`	Makefile for Demo #4
`Demo_5_EVK1101_GCC.c`	Demo code that runs on the Atmel EVK1101 evaluation board using the serial port
`Demo_5_EVK1101_GCC.mak`	Makefile for Demo #5
`AbassiDemo.h`	Build option settings for the demo code

1.2 Limitations

To optimize 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), not the application mode, and to always use the main stack pointer (MSP). The Atmel Studio regular start-up code in the file `crt0.s` fulfills these constraints and one must be careful to not change the mode of operation.

The "C" optimization option "Generate Position-independent code" cannot be used when one or more services are created at compile time. This means if `TSKSTATIC()`, `SEM_STATIC()`, `MTX_STATIC()` or `MBX_STATIC()` are used, this optimization option cannot be used.

NOTE: Makefiles are supplied instead of Atmel Studio projects because the Atmel Studio environment does not support linked files (also known as shortcuts). This means Atmel Studio keeps local copies of the source code and include files in each project; when files are common to multiple project this translates into each project having its own copy of the common files. As such, any change to the common files must be applied to all projects with a local copy of that file. See Section 0 for more information on how to debug the output generated by the makefiles.

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_GCC.s` in the source files of the application project. As well, update the include file path in the C/C++ compiler preprocessor options with the location of the `Abassi.h`.

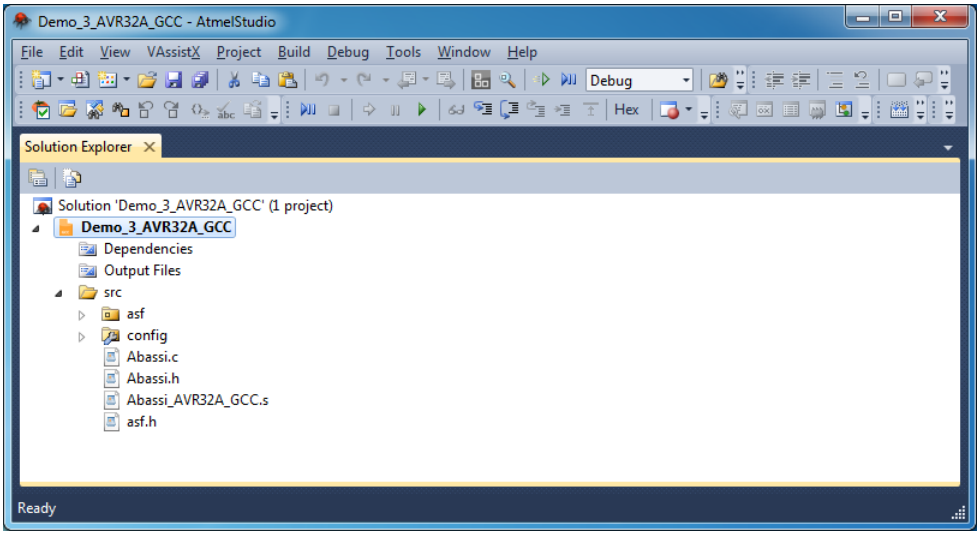

Figure 2-1 Project File List

The assembler must be configured to use "C" pre-processor statements as the file `Abassi_AVR32A_GCC.s` uses them. Using the command line, this is done with:

```
avr32-gcc -Wa, … -x assembler-with-cpp …
```

The file `exception.S` must NOT be included in the build, as all the features contained in this file are available in the file `Abassi_AVR32A_GCC.s`. The same applies to the files `intc.c` and `intc.h`; these files must not be part of the build. The interrupt API supplied by the Atmel Studio cannot be used. Instead, the RTOS supplies its own set of API for the interrupts, as the RTOS needs to be interrupt-aware. Finally, there are a few configuration settings in the files `Abassi_AVR32A_GCC.s` and `Abassi.h` that need to be set according to the target device and the needs of the application. Each of these settings is described in the following subsections.

NOTE: Some functions in the Atmel Studio run-time library are not multithread-safe. As such, library functions like `printf()`, `malloc()`, or `fopen()` can be made multithread-safe through the use of a mutex. It is also advisable to use a single mutex for all accesses to the non- multithread-safe modules, as some non- multithread-safe modules quite likely call other non- multithread-safe ones. The mutex used by the kernel, `G_OSmtx`, can be used for this purpose.

The GCC assembler does not support a command line option that would allow the definition of an assembler symbol. So, contrary to other ports for the AVR32A, modifying the value in the file `Abassi_AVR32A_GCC.s` is the only possible method to set the assembly file symbols. These configurations are still surrounded with `.ifndef` / `.endif` statements in case it becomes possible in the future to set the symbols values on the assembler command line.

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 in 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 2-1 below:

Table 2-1 INTC Configuration for "C" (Command line)

```
avr32-gcc … -DOS_AVR32A_INT_GRP=19 -DOS_AVR32A_INT_LINE=10 …
```

The number of interrupt groups and lines can also be set for the assembly code through the GUI, in the "*Toolchain / AVR32/GNU C Compiler / Symbols*" menu, as shown in the following figure:

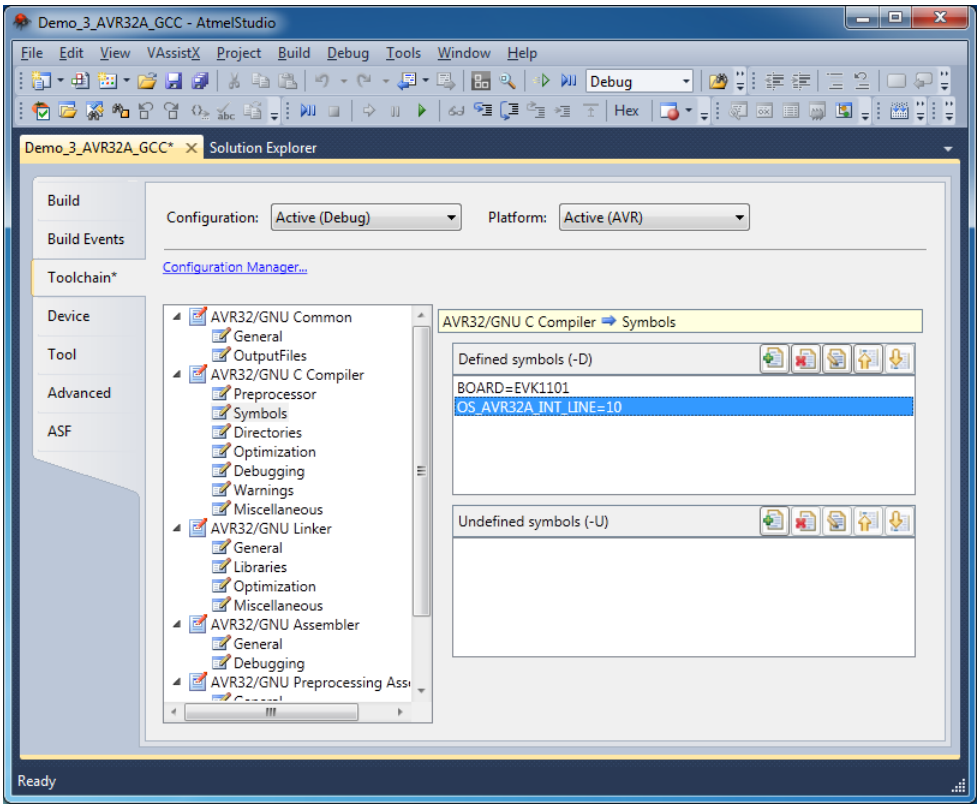

Figure 2-2 GUI set of `OS_AVR32A_INT_LINE (C)`

The line parameters must also be set in the RTOS assembly language file, and it must be set to the same value specified for the ”C” files

To specify it directly in the source file, this definition is located at around line 35 in the file `Abassi_AVR32A_GCC.s`:

Table 2-2 INTC Configuration (`Abassi_AVR32A_GCC.s`)

```
#ifndef OS_AVR32A_INT_LINE
OS_AVR32A_INT_LINE  EQU 10       /* Must be set to the same value as in Abassi.h     */
#endif
```

The values shown in the two tables 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.

Alternatively, it is possible to overload the `OS_AVR32A_INT_LINE` value in `Abassi_AVR32A_GCC.s` by using the assembler command line option `-D` and specifying the desired number of interrupts lines. In the following example, the number of interrupt lines is set to 10:

Table 2-3 Command line set of `OS_AVR32A_INT_LINE` (`Abassi_AVR32A_GCC.s`)

```
avr32-gcc -Wa, … -DOS_AVR32A_INT_LINE=10 …
```

The number of interrupt line can also be set for the assembly code through the GUI, in the “*Toolchain / AVR32/GNU Preprocessing Assembler / Symbols*” menu, as shown in the following figure:

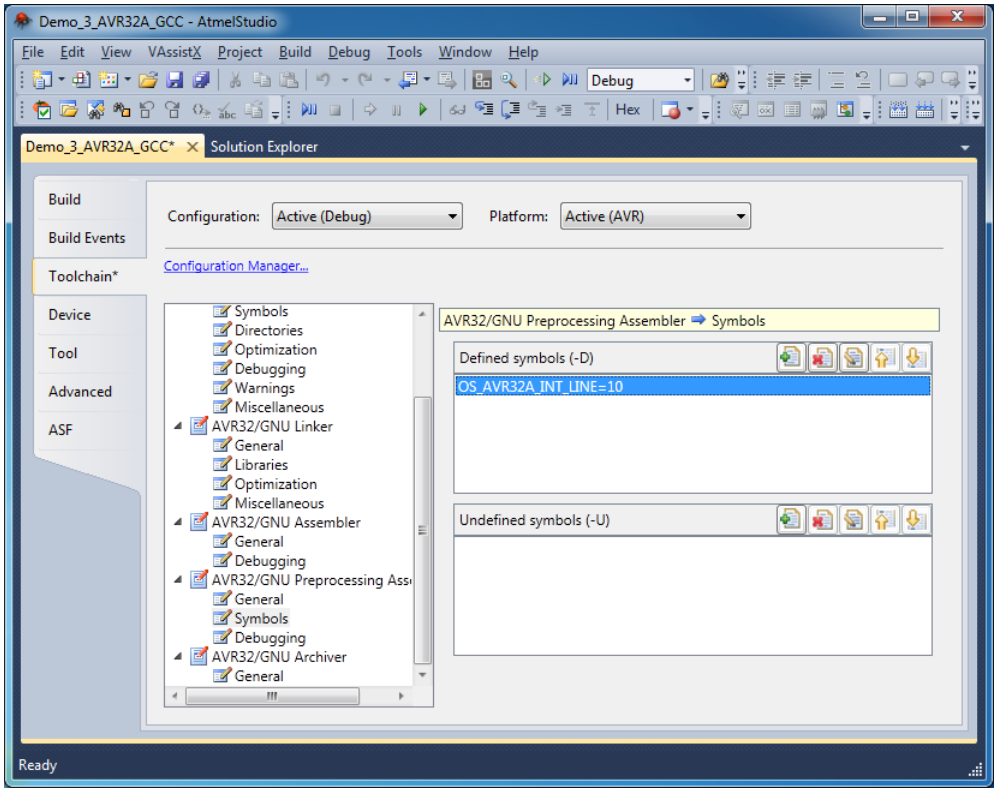

Figure 2-3 GUI set of `OS_AVR32A_INT_LINE` (ASM)

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_GCC.s`. 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 2-4 Interrupt Stack enabled

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

Table 2-5 Interrupt Stack disabled

```
.ifndef OS_ISR_STACK
  .equ OS_ISR_STACK, 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_GCC.s` 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 2-6 Command line set of `OS_ISR_STACK`

```
avr32-gcc -Wa, … -DOS_ISR_STACK=512 …
```

The number of interrupt line can also be set for the assembly code through the GUI, in the "*Toolchain / AVR32/GNU Preprocessing Assembler / Symbols*" menu, as shown in the following figure:

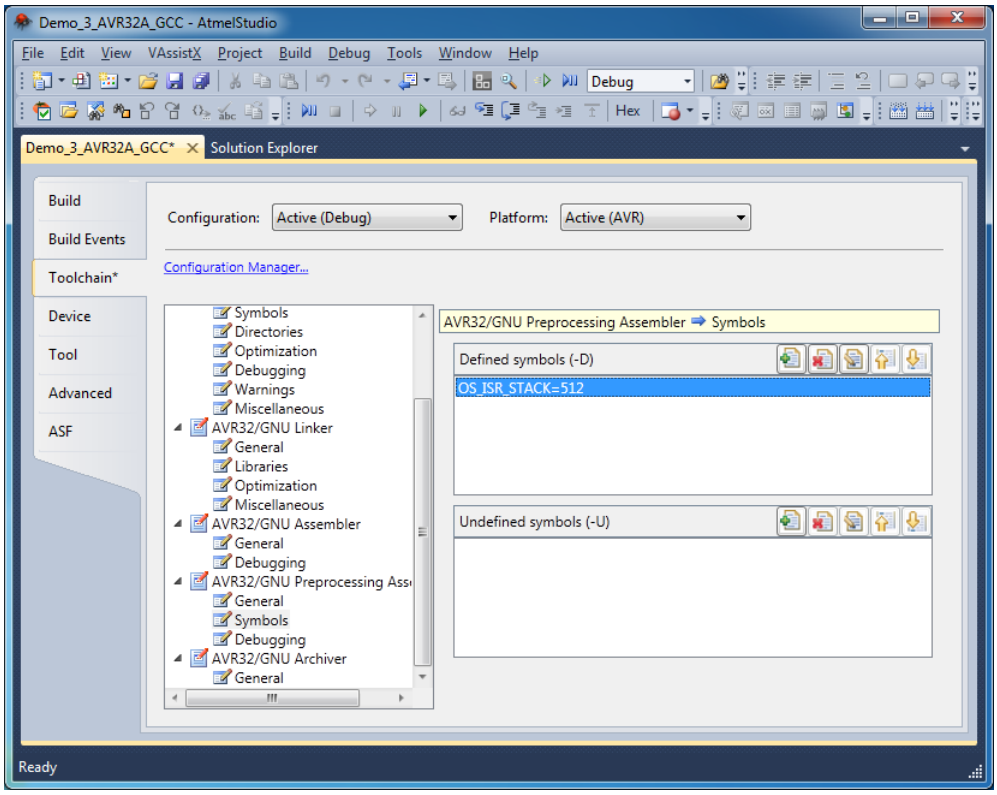

Figure 2-4 GUI set of `OS_ISR_STACK`

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 35 in the file `Absssi_AVR32A_GCC.s`, to the priority threshold at which the interrupts are mapped to fast interrupts:

Table 2-7 Fast Interrupt Configuration

```
.ifndef OS_TAST_INTS
  .equ OS_FAST_INTS, 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_GCC.s` 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 2-8 Command line set of `OS_FAST_INTS`

```
avr32-gcc -Wa, … -DOS_FAST_INTS=2 …
```

The fast interrupt priority threshold value can also be set for the assembly code through the GUI, in the "*Toolchain / AVR32/GNU Preprocessing Assembler / Symbols*" menu, as shown in the following figure:

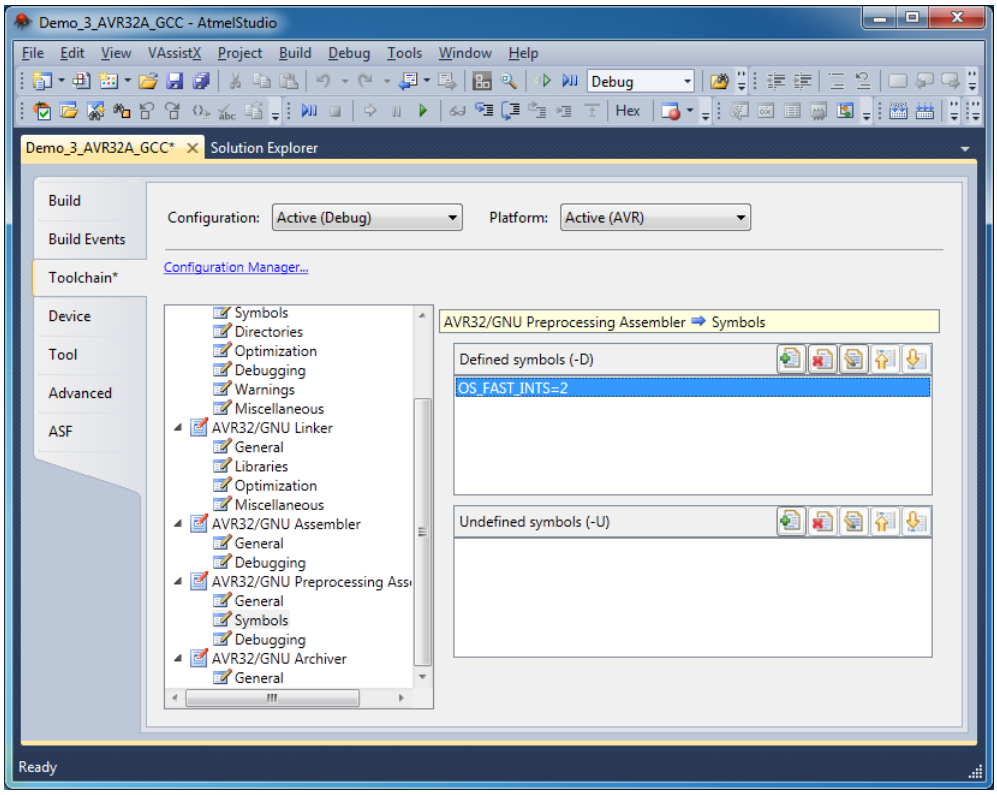

Figure 2-5 GUI set of `OS_FAST_INTS`

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 2-9 Fast Interrupts vs. Priorities

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

If an interrupt level is configured to the fast interrupt operation, all interrupts at that level are treated as fast interrupts; there is no possibility 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 0), 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 45 in the file `Abassi_AVR32A_GCC.s`. 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 2-10 Saturation Bit Configuration

```
.ifndef OS_HANDLE_SR_Q
  .equ OS_HANDLE_SR_Q, 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_GCC.s` 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 2-11 Command line set of `OS_HANDLE_SR_Q`

```
avr32-gcc -Wa, … -DOS_HANDLE_SR_Q=1 …
```

The handling of the saturation bit can also be set for the assembly code through the GUI, in the "*Toolchain / AVR32/GNU Preprocessing Assembler / Symbols*" menu, as shown in the following figure:

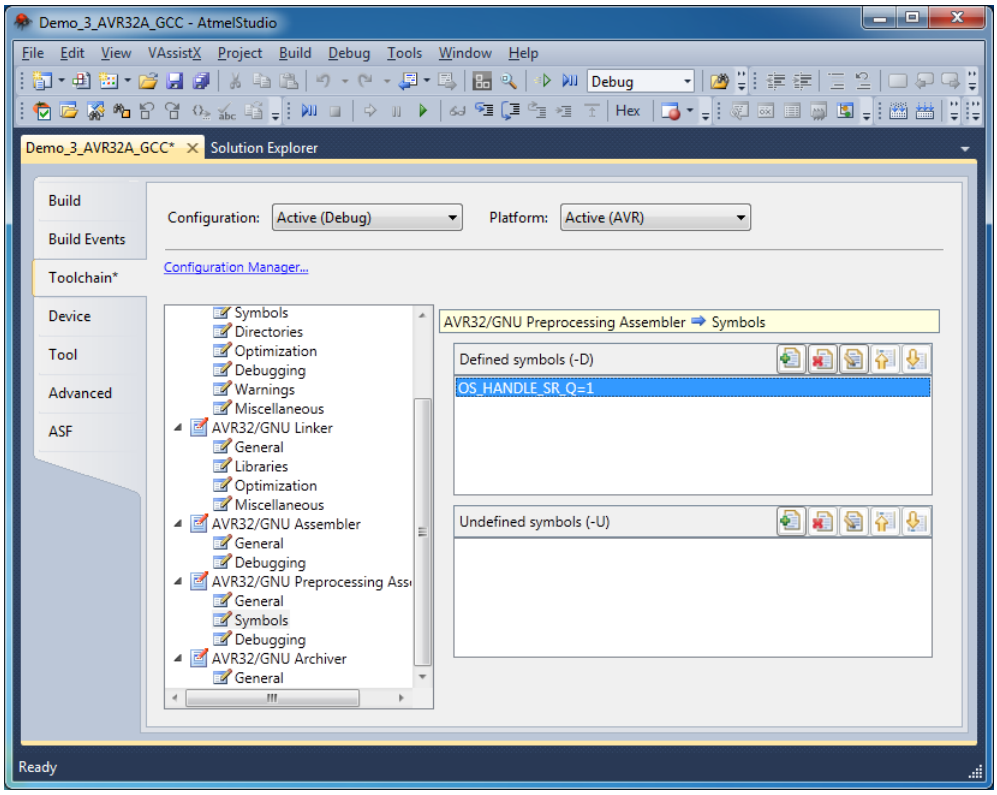

Figure 2-6 GUI set of `OS_HANDLE_SR_Q`

2.5 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_GCC.s`, 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 50 in the file `Abassi_AVR32A_GCC.s`. The distribution code enables the fix on the RETE instruction as a safety measure; this is shown in the following table:

Table 2-12 RETE Instruction Fix Configuration

```
.ifndef OS_FIX_RETE_L
  .equ OS_FIX_RETE_L, 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_GCC.s` by using the assembler command line option `-D` to enable/disable the fix. In the following example, it is enabled:

Table 2-13 Command line set of `OS_FIX_RETE_L`

```
avr32-gcc -Wa, … -DOS_FIX_RETE_L=1 …
```

The enabling / disabling of the fix can also be set for the assembly code through the GUI, in the "*Toolchain / AVR32/GNU Preprocessing Assembler / Symbols*" menu, as shown in the following figure:

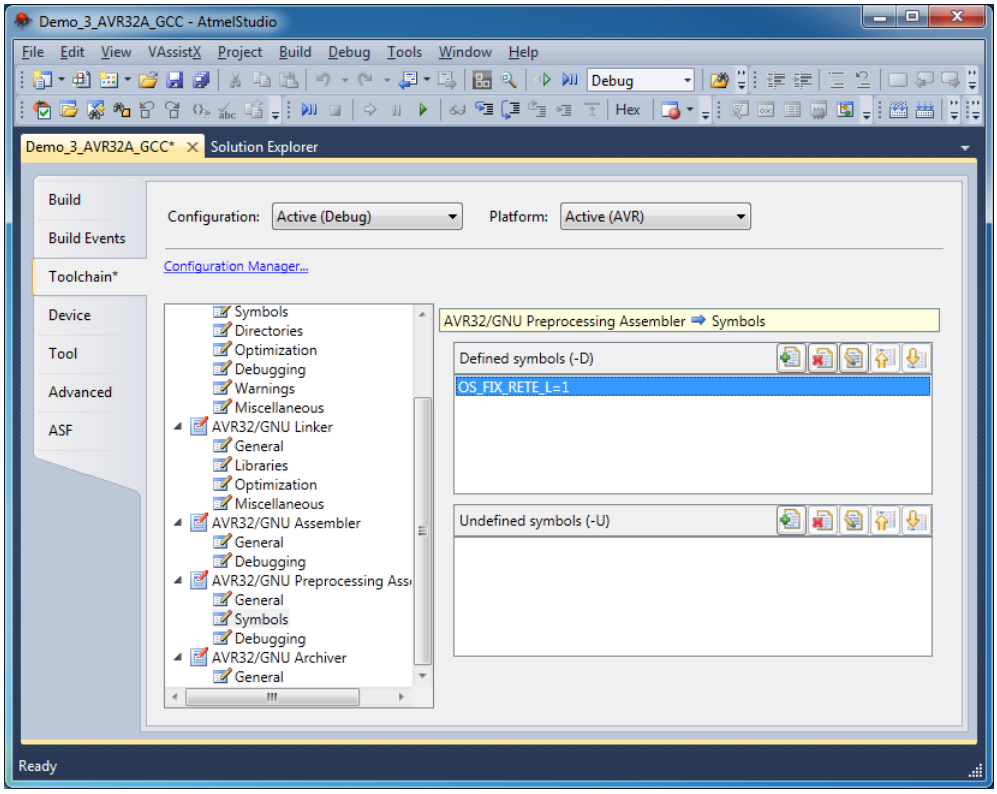

Figure 2-7 GUI set of `OS_FIX_RETE_L`

2.6 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_GCC.s`, 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 55 in the file `Abassi_AVR32A_GCC.s`. The distribution code enables this fix as a safety measure; this is shown in the following table:

Table 2-14 Interrupt Masking Fix Configuration

```
.ifndef OS_FIX_SR_GIM
  .equ OS_FIX_SR_GIM, 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_GCC.s` by using the assembler command line option `-D` to enable/disable the fix. In the following example, it is enabled:

Table 2-15 Command line set of `OS_FIX_SR_GIM`

```
avr32-gcc -Wa, … -DOS_FIX_SR_GIM=1 …
```

The enabling / disabling of the fix can also be set for the assembly code through the GUI, in the "*Toolchain / AVR32/GNU Preprocessing Assembler / Symbols*" menu, as shown in the following figure:

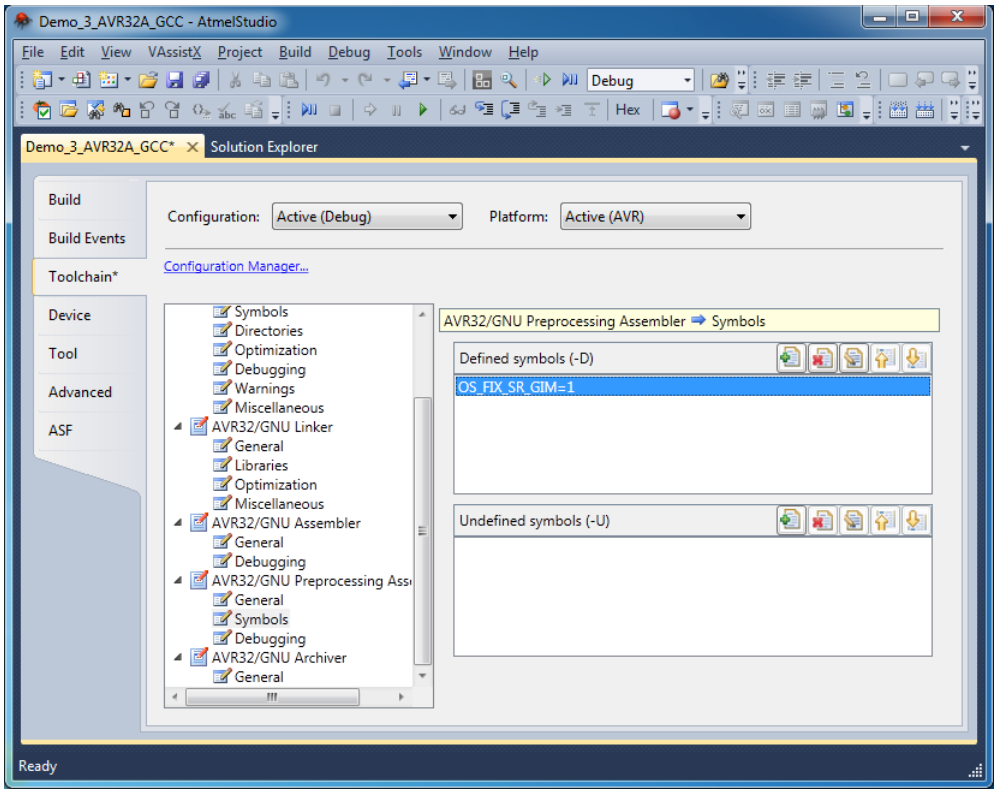

Figure 2-8 GUI set of `OS_FIX_SR_GIM`

2.7 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_GCC.s`, 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 60 in the file `Abassi_AVR32A_GCC.s`. The distribution code does not enable this fix as spurious interrupts should not happen in a well designed application:

Table 2-16 Spurious Interrupt Fix Configuration

```
.ifndef OS_SPURIOUS_ISR
  .equ OS_SPURIOUS_ISR, 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_GCC.s` by using the assembler command line option `-D` to enable/disable the fix. In the following example, it is enabled:

Table 2-17 Command line set of `OS_SPURIOUS_ISR`

```
avr32-gcc -Wa, … -DOS_SPURIOUS_ISR=1 …
```

The enabling / disabling of the fix can also be set for the assembly code through the GUI, in the "*Toolchain / AVR32/GNU Preprocessing Assembler / Symbols*" menu, as shown in the following figure:

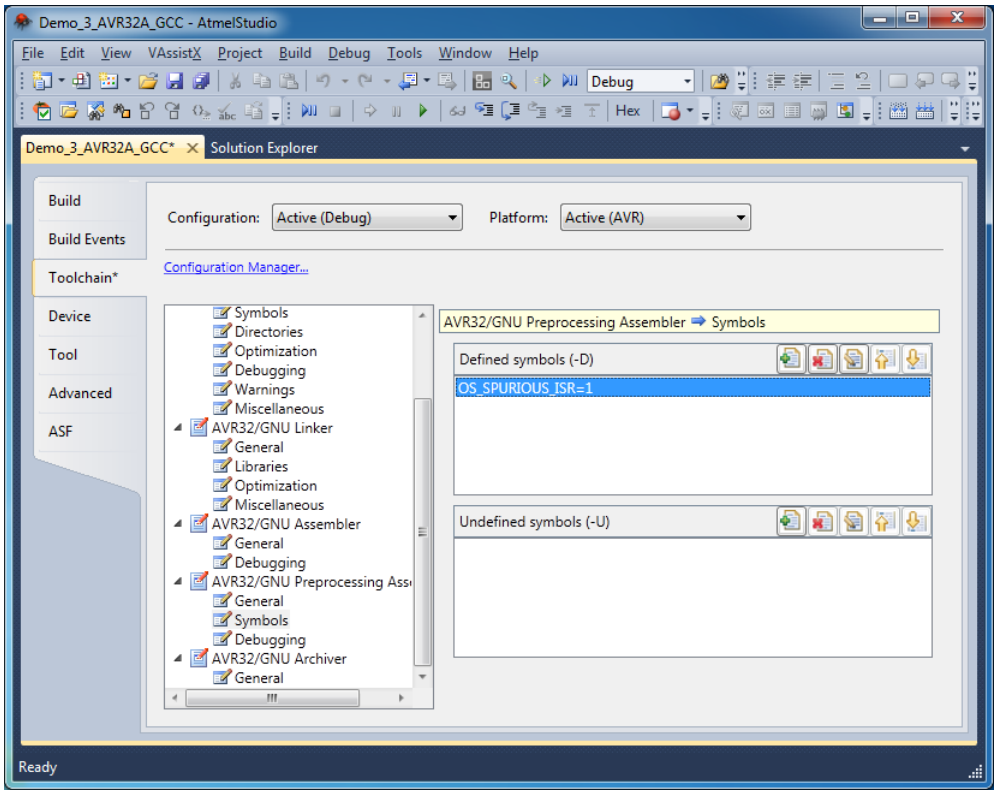

Figure 2-9 GUI set of `OS_SPURIOUS_ISR`

2.8 `.elf` Debugging

The supplied makefiles generate `.elf` files, which can be debugged at the source code level through AVR Studio. The following sections explain step by step how to load an `.elf` file and debug it.

2.8.1 Loading the `.elf` file

The output of the makefiles is an `.elf` file. The `.elf` file is loaded in the AVR studio through the following:

"*File*" → "*Open*" → "*Open Object File For Debugging*"

In the new window,

"*Select the Object File to Debug*"

Click "*Next*" and select the target part.

Click "*Finish*"

A new window will open showing all source files used to build the target application.

Click "*Finish*"

At this point, AVR Studio has internally built up the whole project and it is ready to be debugged.

2.8.2 Debugging the `.elf` file

To debug the application, at the source level, perform the following:

"*Debug*" → "*Start Debugging and Break*"

A new window, titled "*Select Tool*" will open and you select either the simulator or the JTAG used to debug on hardware. Select what fits your setup and then click "*OK*".

The code is then downloaded and the debugging can start.

When the `.elf` file is modified, e.g. changing the code and rebuilding it, it will be automatically be reloaded upon approval (a window pop-up).

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 three components are described in further 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();                              /* Global enable of all interrupts        */
```

NOTE: `OSisrInstall()` must be used with the `OS_AVR32_ISR_MAP()` 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 0), but, as described in the Atmel documentation, this fix 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, as this functionality is already available. First, the Atmel Studio provides everything required for programming the processor peripherals. Second, 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(__GNUC__) && defined(__AVR32_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(__GNUC__) && defined(__AVR32_AVR32A__)
   …
   #define OX_NESTED_INTS OS_NESTED_INTS
```

The Abassi RTOS kernel never disables interrupts, but there is 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`

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.

The AVR32A operates with a stack aligned on 4 bytes, but in the settings for the compiler the stack can be forced to remain aligned on multiple of 8 bytes. Due to the presence of this option, it is strongly advised to allocate the stack of all tasks as multiple of 8 bytes, such that stack alignment will never be an issue.

NOTE: The AVR32A processor needs alignment on 8 bytes for some instructions accessing memory. When stack memory is allocated, Abassi guarantees the 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 0) 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_GCC.s` is set to a non-zero value (see Section 0).

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 bits look-up table. Finally, the last column is when `OS_SEARCH_FAST` is set to 5 (GCC/AVR32A `int` are 32 bits, so 2^5), meaning a 32 bits look-up table further searched through successive approximation. The compiler optimization for this measurement was set to the highest level (-O3). 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 244 cycles.

Table 5-1 Search Algorithm Cycle Count

Priority	Linear search	Look-up	Approximation
1	240	269	368
2	248	275	368
3	254	281	368
4	260	287	368
5	266	293	368
6	272	299	368
7	278	305	368
8	284	277	368
9	290	285	368
10	296	291	368
11	302	297	368
12	308	303	368
13	314	309	368
14	320	315	368
15	326	321	368
16	332	288	368
17	338	296	368
18	344	302	368
19	350	308	368
20	356	314	368
21	362	320	368
22	368	326	368
23	374	332	368
24	380	299	368

When `OS_SEARCH_FAST` is set to 0, each extra priority level to traverse requires exactly 6 CPU cycles. When `OS_SEARCH_FAST` is set to 1, each extra priority level to traverse requires exactly 6 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 16 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 16 cycles needed but without the 8 times 6 cycle accumulation. Finally, the third option, when `OS_SEARCH_FAST` is set to 5, delivers a 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 383, for 64 or more it is 394).

The first observation, when looking at this table, is that the third option, when OS_SEARCH_FAST is set to 5, is less CPU efficient than the second option, the one when OS_SEARCH_FAST is set to 1. So the build option OS_SEARCH_FAST should never be set to 5, as it is the least efficient method (when the delta of priority reaches 55 to 60 or more, then the third option starts to show a lower CPU usage). The other observation is that the first option (OS_SEARCH_FAST set to 0) delivers better CPU performance than the second option (OS_SEARCH_FAST set to 1) when the search spans less than 7 to 8 priority levels. So if an application has tasks spanning less than 7 to 8 priority levels, the build option OS_SEARCH_FAST should be set to 0; for all other cases, the build option OS_SEARCH_FAST should be set to 1 (unless there are more than 50 to 55 priority levels used in the application).

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 because the Atmel Software Framework is supplied as is with the distribution code. Anyone can download this library from Atmel; Code Time Technologies provides it as is for user convenience. Everything related to the AVR32A peripherals can be dealt with through these modules.

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 Atmel Studio. 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 beta release of the RTOS and may change. One should interpret these numbers as the "very likely" numbers for the released version 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: Optimize (-O1)
2. Other optimization flags: -fdata-sections
3. All boxes unchecked except: Use assembler for pseudo instructions

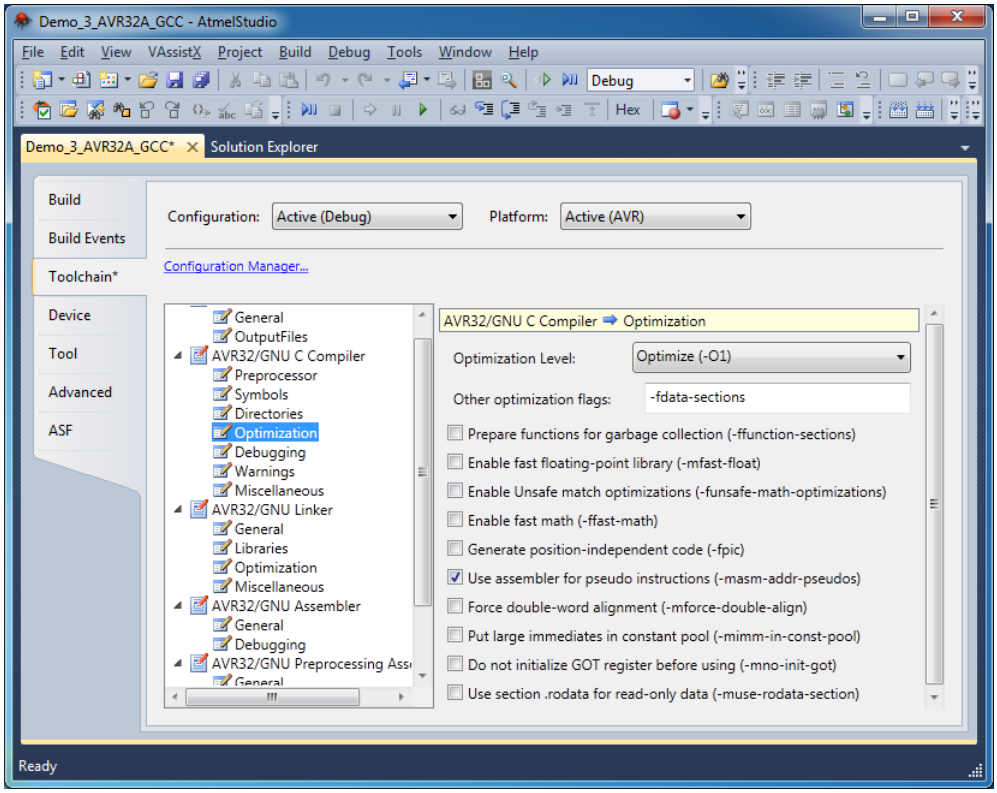

Figure 7-1 Memory Measurement Code Optimization Settings

Table 7-1 "C" Code Memory Usage

Description	Code Size
Minimal Build	< 775 bytes
+ Runtime service creation / static memory	< 1025 bytes
+ Multiple tasks at same priority	< 1100 bytes
+ Runtime priority change + Mutex priority inheritance + FCFS + Task suspension	< 1625 bytes
+ Timer & timeout + Timer call back + Round robin	< 2200 bytes
+ Events + Mailbox	< 2950 bytes
Full Feature Build (no names)	< 3650 bytes
Full Feature Build (no names / no run time creation)	< 3250 bytes
Full Feature Build (no names / no runtime creation) + Timer services module	< 3650 bytes

Table 7-2 Assembly Code Memory Usage

Description	ISR stack
Assembly code size	226 bytes
Interrupt vector table	260 bytes
ISR stack	+ 28 bytes
Fast Interrupts (independent of level enable)	+ 46 bytes
Handle Saturation bit	+ 26 bytes
Fix for RETE	+ 10 bytes
Fix for interrupt masking	+ 8 bytes
Fix for spurious interrupts	+ 60 bytes

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 crystal of 12 MHz. All measurements have been performed on the real platform, using the cycle count register as the measuring element. The code optimization settings used for the latency measurements are:

1. Debugging: None[2]
2. Optimization level: Optimize most (-O3)
3. Other optimization flags: -fdata-sections
4. All boxes unchecked except: Generate position-independent code
   Use assembler for pseudo instructions
   Put large immediates in constant pool

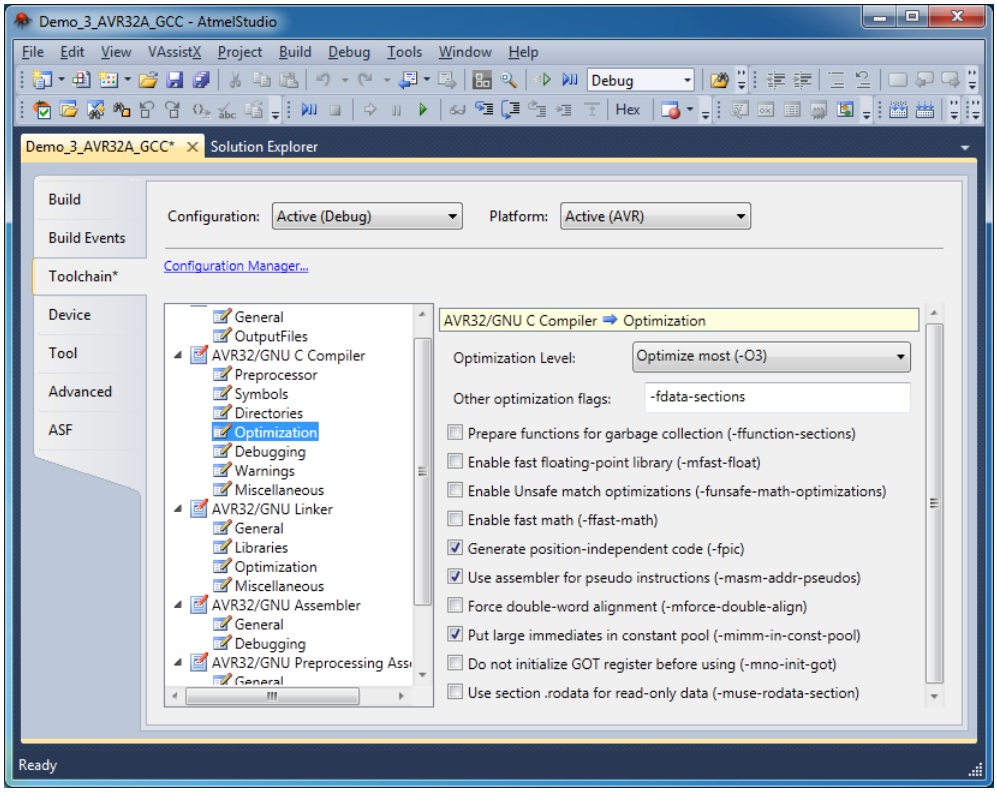

Figure 7-2 Latency Measurement Code Optimization Settings

[2] The debugging option was turned off as the debugging sometimes restricts the optimizer

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 7-3 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 7-4 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 7-5 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 7-6 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_GCC.s 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 7-7 Latency Measurements

Description	Minimal Features	Full Features
Semaphore posting no task switch	135 (143)	191 (191)
Semaphore waiting no blocking	141 (151)	201 (202)
Semaphore posting with task switch	196 (230)	305 (326)
Semaphore waiting with blocking	219 (230)	336 (336)
Semaphore posting in ISR with task switch	452 (479)	568 (587)
Event setting no task switch	n/a	186 (186)
Event getting no blocking	n/a	210 (211)
Event setting with task switch	n/a	324 (343)
Event getting with blocking	n/a	351 (351)
Event setting in ISR with task switch	n/a	586 (603)
Mailbox writing no task switch	n/a	237 (237)
Mailbox reading no blocking	n/a	243 (243)
Mailbox writing with task switch	n/a	351 (371)
Mailbox reading with blocking	n/a	399 (398)
Mailbox writing in ISR with task switch	n/a	625 (642)
Interrupt Latency	54	54
Interrupt overhead entering the kernel	256 (249)	263 (261)
Interrupt overhead NOT entering the kernel	75	75
Context switch	45	45

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: pre-emptive, when non-zero: cooperative  */
#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    2    /* Maximum number of requests in ISRs               */
#define OS_MTX_DEADLOCK     0    /* This test validates this                         */
#define OS_MTX_INVERSION    0    /* To enable protection against priority inversion  */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes   */
#define OS_NESTED_INTS      0    /* If operating with nested interrupts              */
#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          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_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 module           */
#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: pre-emptive, when non-zero: cooperative   */
#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    2    /* Maximum number of requests in ISRs                */
#define OS_MTX_DEADLOCK     0    /* This test validates this                          */
#define OS_MTX_INVERSION    0    /* To enable protection against priority inversion   */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes    */
#define OS_NESTED_INTS      0    /* If operating with nested interrupts               */
#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_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 module            */
#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: pre-emptive, when non-zero: cooperative  */
#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    32   /* Maximum number of requests in ISRs                 */
#define OS_MTX_DEADLOCK     0    /* This test validates this                           */
#define OS_MTX_INVERSION    0    /* To enable protection against priority inversion  */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes   */
#define OS_NESTED_INTS      0    /* If operating with nested interrupts                */
#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_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 module           */
#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: pre-emptive, when non-zero: cooperative   */
#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    32   /* Maximum number of requests in ISRs                */
#define OS_MTX_DEADLOCK     0    /* This test validates this                          */
#define OS_MTX_INVERSION    1    /* To enable protection against priority inversion   */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes    */
#define OS_NESTED_INTS      0    /* If operating with nested interrupts               */
#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_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 module            */
#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: pre-emptive, when non-zero: cooperative  */
#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   32    /* Maximum number of requests in ISRs                 */
#define OS_MTX_DEADLOCK    0     /* This test validates this                           */
#define OS_MTX_INVERSION   1     /* To enable protection against priority inversion  */
#define OS_NAMES           0     /* != 0 when named Tasks / Semaphores / Mailboxes   */
#define OS_NESTED_INTS     0     /* If operating with nested interrupts              */
#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_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 module           */
#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: pre-emptive, when non-zero: cooperative  */
#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   32    /* Maximum number of requests in ISRs                 */
#define OS_MTX_DEADLOCK    0     /* This test validates this                           */
#define OS_MTX_INVERSION   1     /* To enable protection against priority inversion  */
#define OS_NAMES           0     /* != 0 when named Tasks / Semaphores / Mailboxes   */
#define OS_NESTED_INTS     0     /* If operating with nested interrupts                */
#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_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 module           */
#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: pre-emptive, when non-zero: cooperative   */
#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   32    /* Maximum number of requests in ISRs                */
#define OS_MTX_DEADLOCK    0     /* This test validates this                          */
#define OS_MTX_INVERSION   1     /* To enable protection against priority inversion   */
#define OS_NAMES           0     /* != 0 when named Tasks / Semaphores / Mailboxes    */
#define OS_NESTED_INTS     0     /* If operating with nested interrupts               */
#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  100   /* 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_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 module            */
#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: pre-emptive, when non-zero: cooperative  */
#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   32    /* Maximum number of requests in ISRs                 */
#define OS_MTX_DEADLOCK    0     /* This test validates this                           */
#define OS_MTX_INVERSION   1     /* To enable protection against priority inversion  */
#define OS_NAMES           0     /* != 0 when named Tasks / Semaphores / Mailboxes   */
#define OS_NESTED_INTS     0     /* If operating with nested interrupts              */
#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_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 module           */
#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: pre-emptive, when non-zero: cooperative  */
#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   32    /* Maximum number of requests in ISRs                 */
#define OS_MTX_DEADLOCK    0     /* This test validates this                           */
#define OS_MTX_INVERSION   1     /* To enable protection against priority inversion  */
#define OS_NAMES           0     /* != 0 when named Tasks / Semaphores / Mailboxes   */
#define OS_NESTED_INTS     0     /* If operating with nested interrupts              */
#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  100   /* 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_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 module           */
#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, 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)