CODE TIME TECHNOLOGIES

mAbassi RTOS

Porting Document

SMP / ARM Cortex-A9 – Xilinx SDK (GCC)

Copyright Information

This document is copyright Code Time Technologies Inc. ©2013-2018. All rights reserved. No part of this document may be reproduced or distributed in any form by any means, or stored in a database or retrieval system, without the written permission of Code Time Technologies Inc.
Code Time Technologies Inc. may have patents or pending applications covering the subject matter in this document. The furnishing of this document does not give you any license to these patents.

Disclaimer

Code Time Technologies Inc. provides this document "AS IS" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.
Code Time Technologies Inc. does not warrant that the contents of this document will meet your requirements or that the document is error-free. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the document. Code Time Technologies Inc. may make improvements and/or changes in the product(s) and/or program(s) described in the document at any time. This document does not imply a commitment by Code Time Technologies Inc. to supply or make generally available the product(s) described herein.

ARM and Cortex are registered trademarks of ARM Limited. Xilinx and Zynq is a registered trademark of Xilinx Inc. Sourcery CodeBench is a registered trademark of Mentor Graphics. All other trademarks are the property of their respective owners.

Table of Contents

List of Figures

List of Tables

1 Introduction

This document is a complement to the user guide and it details the port of the SMP / BMP multi-core mAbassi RTOS to the ARM Cortex-A9 multi-core processor, commonly known as the Arm9 MPcore. The software suite used for this specific port is the Xilinx Software Development Kit (SDK) from Xilinx Inc.; the version used for the port and all tests is Version 14.6 build SDK_P.68d. The GCC tool chain used for compilation, assembly and linking is the Mentor Graphics Sourcery CodeBench Lite version 2012.09-105, which is supplied as an integral part of the Xilinx SDK environment.

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
`mAbassi.h`	RTOS include file
`mAbassi.c`	RTOS "C" source file
`ARMv7_SMP_L1_L2_GCC.s`	L1 and L2 caches, MMU, and SCU set-up module for the MPcore A9 / GCC
`cmsis_os.h`	Optional CMSIS V 3.0 RTOS API include file
`cmsis_os.c`	Optional CMSIS V 3.0 RTOS API source file
`mAbassi_SMP_CORTEXA9_GCC.s`	RTOS assembly file for the SMP ARM Cortex-A9 to use with the GCC tool chain
	Many demos

1.2 Limitations

The RTOS reserves the `SWI` (software interrupts) numbers 0 to 6 when mAbassi is configured to operate in user mode. A hook is made available for the application to use the `SWI`, as long as the numbers used are above 6. This keeps mAbassi compatible with the ARM semi-hosting protocol.

1.3 Features

Depending on the selected build configuration, the application can operate either in privileged or user mode. Operating in privileged mode eliminates almost all the code areas that disable interrupts, as SWIs are not required to access privileged registers or peripherals (when the processor processes a SWI, the interrupts are disabled). Selecting to run in privileged mode also generates more real-time optimal code.

Fast Interrupts (FIQ) are not handled by the RTOS, and are left untouched by the RTOS to fulfill their intended purpose of interrupts not requiring kernel access. Only the interrupts mapped to the IRQ interrupt are handled by the RTOS.

The hybrid stack is not available in this port, as ARM's GIC (Generic Interrupt Controller) does not support nesting of the interrupts (except FIQ nesting the IRQ). The ARM Cortex-A9 intrinsically supports exactly the same functionality delivered by mAbassi's hybrid stack. This is because the interrupts (IRQ) use a dedicated stack when in this processor mode.

The assembly file does not use the `BL` *addr* when calling a function. This was chosen to allow the assembly file to access the whole 4 GB address space.

The VFPv3 or VPFv3D16, and NEON floating-point peripherals are supported by this port, and their registers are saved as part of the task context save and the interrupt context save.

2 Target Set-up

Very little is needed to configure the Xilinx Software Development Kit environment to use the mAbassi RTOS in an application. All there is to do is to add the files `mAbassi.c` and `mAbassi_SMP_CORTEXA9_GCC.s` in the `makefile` of the application project, and make sure the configuration settings in the file `mAbassi_SMP_CORTEXA9_GCC.s` (described in the following sub-sections) match to the needs of the application. As well, update the include file path in the C/C++ compiler preprocessor options with the location of `mAbassi.h`. There is no need to include a start-up file, as the file `mAbassi_SMP_CORTEXA9_GCC.s` takes care of all the start-up operations required for an application to operate on a multi-core processor.

NOTE: The document describes the settings for different platforms, many of which are not Xilinx devices. The reason for this is the assembly file `mAbassi_SMP_CORTEX_A9_GCC.s`, used for the GCC tool chain, is shared amongst multiple development platforms. Therefore any information related to non-Xilinx devices should not be used and can simply be ignored.

2.1 Build Options (GUI)

In the following section, all build option examples are given using command line commands, as they would be used in a makefile. It is quite likely the application project will be based on the SDK GUI and not a custom makefile. All build options, either for the compiler, the assembler, or for both, can also be defined through the SDK GUI. Taking for example the build option `OS_N_CORE`, it can be passed to the compiler using the GUI by moving across the following menus / options:

Properties → C/C++ Build → Settings → ARM gcc compiler → Symbols

By clicking on the *Add...* icon and inserting the statement, for example, `OS_N_CORE=2` in the pop-up box. This is shown in the following figure:

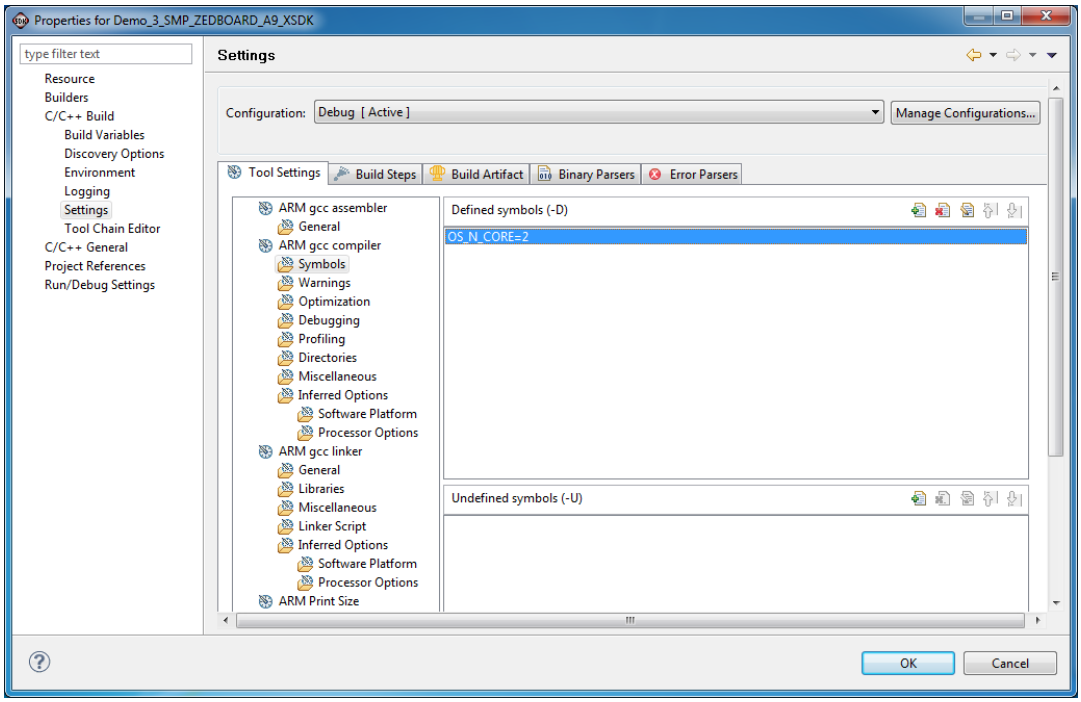

Figure 2-1 : "C" build option setting

In the case of the assembler, the default command for assembly is `arm-xilinx-eabi-gcc`, which is the compiler and not the assembler itself (`arm-xilinx-eabi-as`). This means the compiler must be informed about the fact it has to pass the information to the assembler. This is done through the `–Wa` command line option of the compiler. For example, to set the build option `OS_N_CORE` for the assembler, the command line option to use is shown in the following table:

Table 2-1 Assembly build options passed through the compiler

```
arm-xilinx-eabi-gcc … -Wa,--defsym -Wa,OS_N_CORE=2 …
```

This is specified using the GUI by moving across the following menus / options:

Properties → C/C++ Build → Settings → ARM gcc assembler → General

And inserting information in the *Assembler Flags* box as shown in the following figure:

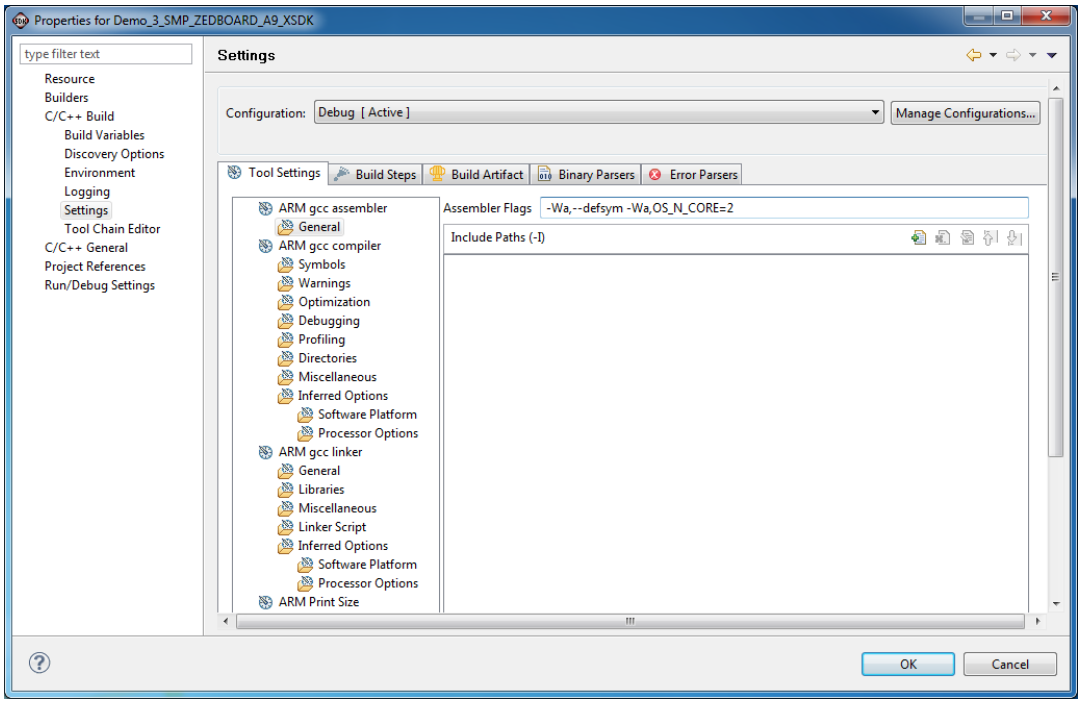

Figure 2-2 : "ASM" build option setting (through "GCC")

Instead of using the compiler for assembly, it is possible to use the assembler directly. To make the GUI use the assembler directly, go through the following menus / options:

Properties → C/C++ Build → Settings → ARM gcc assembler

And replace `arm-xilinx-eabi-gcc` to `arm-xilinx-eabi-as` in the *Command* box. This is shown in the following figure:

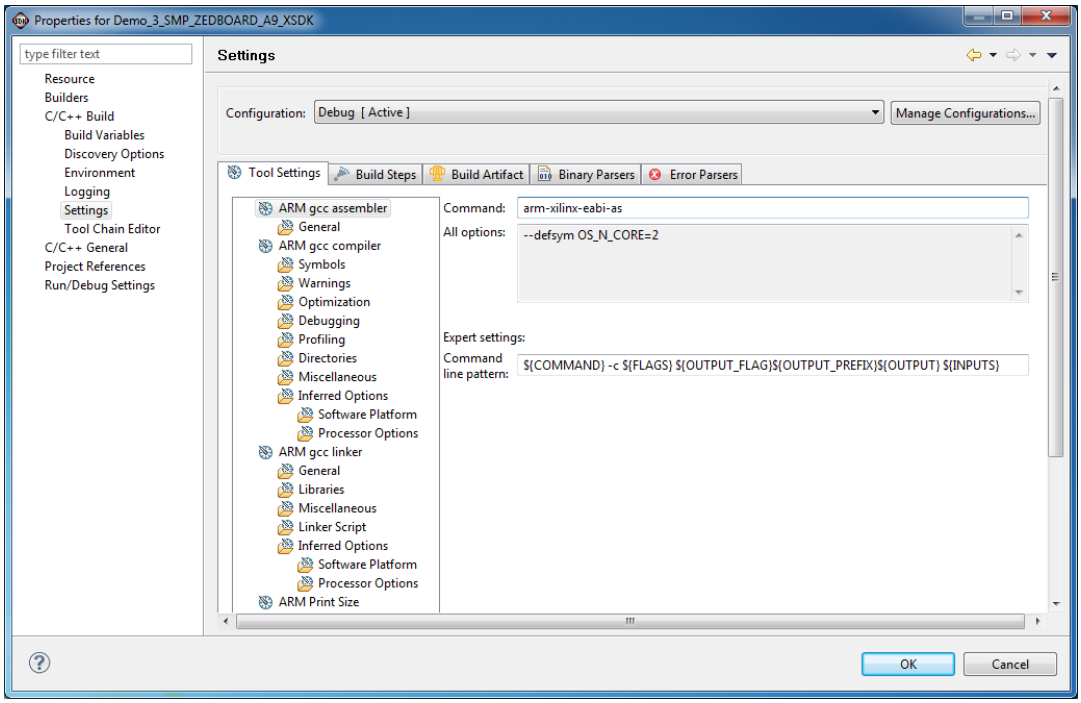

Figure 2-3 : Using the assembler in the GUI

If the assembler is used in the GUI instead of the compiler, there is no need to use the `–Wa` option, as this option is only needed when using the compiler. For example, to set the build option `OS_N_CORE` for the assembler, the command line option to use is shown in the following table:

Table 2-2 Assembly build options passed through the assembler

```
arm-xilinx-eabi-as … --defsym OS_N_CORE=2 …
```

This is specified using the GUI by moving across the following menus / options:

Properties → C/C++ Build → Settings → ARM gcc assembler → General

And inserting the information in the *Assembler Flags* box as shown in the following figure:

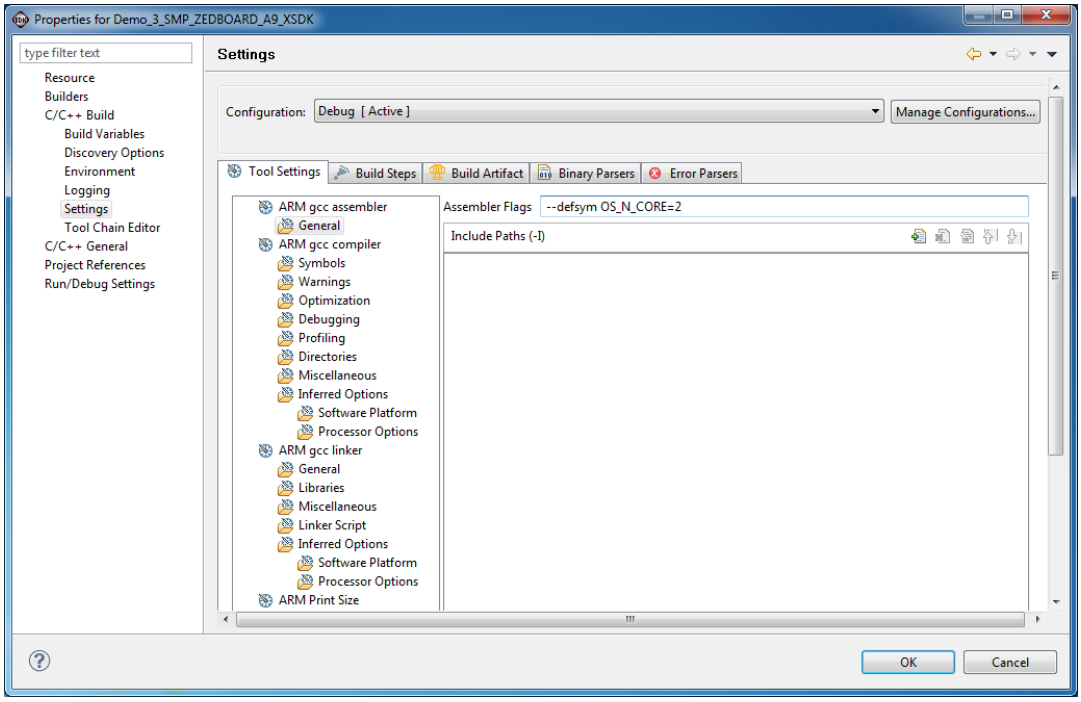

Figure 2-4 : "ASM" build option setting (through "AS")

2.2 Target Device

Each manufacturer uses a different method to release from reset the cores other than core #0. As such, the start-up code must to be tailored for each target device. This information is specified in the assembly file with the value assigned to the token `OS_PLATFORM`. At the time of writing this document, the following platforms are supported:

Table 2-3 OS_PLATFORM valid settings

Target Platform	`OS_PLATFORM` value
Altera / Arria V Soc FPGA	`0xAAC5 (Same as Cyclone V)`
Altera / Cyclone V Soc FPGA	`0xAAC5 (Same as Arria V)`
Texas Instruments / OMAP 4460	`0x4460`
Xilinx / Zynq XC7Z020	`0x7020`
Freescale i.MX6	`0xFEE6`

If in the future if there are platforms that are not listed in the above table, the numerical values assigned to the platform are specified in comments in the file `mAbassi_SMP_CORTEXA9_GCC.s`., right beside the internal definition of `OS_PLATFORM` (around line 85).

To select the target platform, all there is to do is to change the numerical value associated with the token `OS_PLATFORM` located around line 85 in the file `mAbassi_SMP_CORTEXA9_GCC.s`. By default, the target platform is the Xilinx Zynq XC7Z020, therefore `OS_PLATFORM` is assigned the numerical value `0x7020`. The following table shows how to set the target platform to the Altera Cyclone V, which is assigned the numerical value `0xAAC5`:

Table 2-4 `OS_PLATFORM` modification

```
#ifndef OS_PLATFORM
.ifndef OS_PLATFORM
  .equ OS_PLATFORM, 0xAAC5
.endif
#endif
```

Alternatively, it is possible to overload the `OS_PLATFORM` value set in `mAbassi_SMP_CORTEXA9_GCC.s` by using the assembler command line option `--defsym` and specifying the target platform numerical value:

Table 2-5 Command line set of `OS_PLATFORM`

```
arm-xilinx-eabi-as  … --defsym OS_PLATFORM=0xAAC5  …
```

2.3 Stacks Set-up

The Cortex-A9 processor uses 6 individual stacks, which are selected according to the processor mode. The following table describes each stack and the build token used to define the size of the associated stack:

Table 2-6 Stack Size Tokens

Description	Token Name
User / System mode	`N/A`
Supervisor mode	`OS_SUPER_STACK_SIZE`
Abort mode	`OS_ABORT_STACK_SIZE`
Undefined mode	`OS_UNDEF_STACK_SIZE`
Interrupt mode	`OS_IRQ_STACK_SIZE`
Fast Interrupt mode	`OS_FIQ_STACK_SIZE`

The User / System mode stack is defined as the symbol `__cs3_stack` (or `__stack` depending on the target platform) in the linker command. That linker-set stack is assigned to the Adam & Eve task, which is the element of code executing upon start-up on core #0. The other cores start by entering the functions `COREstartN()`, which are fully described in the mAbassi User's Guide [R1], and their stack sizes are defined as part of the mAbassi standard build options.

The other stack sizes are individuality controlled by the values set by the `OS_?????_STACK_SIZE` definitions, located between lines 50 and 70 in the file `mAbassi_SMP_CORTEXA9_GCC.s`. To not reserve a stack for a processor mode, all there is to do is to set the definition of `OS_?????_STACK_SIZE` to a value of zero (0). To specify the stack size, the definition of `OS_?????_STACK_SIZE` is set to the desired size in bytes (see Section 4 for more information on stack sizing). Note that each core on the device (up to the number specified by the build option `OS_N_CORE`) will use the same stack sizes for a processor mode: this is the value assigned to the token `OS_?????_STACK_SIZE`. This equal distribution of stack size may not be optimal; if a non-equal distribution is required, contact Code Time Technologies for additional information on the code modifications involved.

Alternatively, it is possible to use definitions of the stack extracted from the linker command file. When the value assigned to a stack definition token `OS_?????_STACK_SIZE` is set to -1, the stack uses the size and the data section reserved in the linker command file; then no memory is reserved for this stack by the file `Abassi_SMP_CORTEXA9_GCC.s`. Contrary to setting the definition token to a positive value, the stack size defined in the linker is the total stack size shared by all cores (except for `OS_STACK_SIZE` / `__stack`). . This means for 2 cores, each core will be allocated half the stack size defined in the linker command file; for 4 cores, each core will be allocated a quarter of the stack size defined in the linker command file. The names of the base of the stack (which is at the highest memory address of data section) and the names of their sizes are listed in Table 2-7.

Table 2-7 Linker Command file Stack Tokens

Description	Stack Name	Stack Size Name
User / System mode	`__cs3_stack`	`_STACK_SIZE`
Supervisor mode	`__supervisor_stack`	`_SUPERVISOR_STACK_SIZE`
Abort mode	`__abort_stack`	`_ABORT_STACK_SIZE`
Undefined mode	`__undef_stack`	`_UNDEF_STACK_SIZE`
Interrupt mode	`__irq_stack`	`_IRQ_STACK_SIZE`[1]
Fast Interrupt mode	`__fiq_stack`	`_FIQ_STACK_SIZE`

As supplied in the distribution, all stack size tokens are assigned the value of -1 meaning all stack information is supplied by the linker command file.

To modify the size of a stack, taking the IRQ stack for example and reserving a stack size of 256 bytes for each core for the IRQ processor mode, all there is to do is to change the numerical value associated with the token; this is shown in the following table:

Table 2-8 `OS_IRQ_STACK_SIZE` modification

```
#ifndef OS_IRQ_STACK_SIZE
.ifndef OS_IRQ_STACK_SIZE
  .equ OS_IRQ_STACK_SIZE, 256
.endif
#endif
```

[1] For versions pre-1.64.62: the size of the IRQ stack is the same as the one for `main()`. This setting originally comes from the standard Xilinx Zynq linker command file definitions. It is retained because the file `mAbassi_SMP_CORTEX_GCC.s` is common to all GCC tool chains. If necessary, it is straightforward to change.

Alternatively, it is possible to overload the `OS_?????_STACK_SIZE` value set in `mAbassi_SMP_CORTEXA9_GCC.s` by using the assembler command line option `--defsym` and specifying the desired stack size as shown in the following example, where the IRQ stack size is set to 512 bytes:

Table 2-9 Command line set of `OS_IRQ_STACK_SIZE`

```
arm-xilinx-eabi-as  … --defsym OS_IRQ_STACK_SIZE=512  …
```

A third way to allocate the different stacks and specify their sizes is by setting to a value of -2 the token `OS_?????_STACK_SIZE`. When this is done, the stacks memory and their sizes are supplied through external arrays and variables. The stack arrays must be dimensioned to: *#Core * stacksize* bytes; the run-time set-up of the stacks makes sure the stacks are aligned according to the Cortex-A9 requirements. The variable specifying the stack sizes indicates the number of bytes per core and not the number of bytes in the stack array.

Table 2-10 Variables Stack Names

Description	Stack Name	Stack Size Name
User / System mode	`n/a`	`n/a`
Supervisor mode	`G_SUPER_stack`	`G_SUPER_stackSize`
Abort mode	`G_ABORT_stack`	`G_ABORT_stackSize`
Undefined mode	`G_UNDEF_stack`	`G_UNDEF_stackSize`
Interrupt mode	`G_IRQ_stack`	`G_IRQ_stackSize`
Fast Interrupt mode	`G_FIQ_stack`	`G_FIQ_stackSize`

2.4 Number of cores

When operating the mAbassi RTOS on a platform, the RTOS needs to be configured for the number of cores it has access to, or will use. This number is most of the time the same as the number of cores the device has, but it also can be set to a value less than the total number of cores on the device, but not larger obviously. This must be done for both the `mAbassi.c` file and the `mAbassi_SMP_CORTEXA9_GCC.s` file, through the setting of the build option `OS_N_CORE`. In the case of the file `mAbassi.c`, `OS_N_CORE` is one of the standard build options. In the case of the file `mAbassi_SMP_CORTEXA9_GCC.s`, to modify the number of cores, all there is to do is to change the numerical value associated to the token definition of `OS_N_CORE`, located around line 35; this is shown in the following table. By default, the number of cores is set to 2.

Table 2-11 `OS_N_CORE` modification

```
#ifndef OS_N_CORE
.ifndef OS_N_CORE
  .equ OS_N_CORE, 4
.endif
#endif
```

Alternatively, it is possible to overload the OS_N_CORE value set in mAbassi_SMP_CORTEXA9_GCC.s by using the assembler command line option --defsym and specifying the required number of cores as shown in the following example, where the number of cores is set to 4:

Table 2-12 Command line set of OS_N_CORE (ASM)

```
arm-xilinx-eabi-as  … --defsym OS_N_CORE=4  …
```

Exactly the same value of OS_N_CORE as specified for the assembler must be specified for the compiler; a mismatch between the assembly and "C" definition either generates a link error if the assembly file value is less than the "C" value or will freeze the application if the assembly file value is larger than the "C" value. In the following example, the number of cores is set to 4 for the "C" files:

Table 2-13 Command line set of OS_N_CORE (C)

```
arm-xilinx-eabi-gcc  … -D OS_N_CORE=4  …
```

NOTE: mAbassi can be configured to operate as the single core Abassi by setting OS_N_CORE to 1, or setting OS_MP_TYPE to 0 or 1. When configured for single core on the Cortex-A9 MPCore, the single core application always executes on core #0.

2.5 Privileged mode

It is possible to configure mAbassi for the Cortex-A9 MPCore to make the application execute in either the USER processor mode (un-privileged) or in the SYS processor mode (privileged). Having the application executing in the SYS processor mode (privileged) delivers two main advantages over having it executing in the USER mode (un-privileged). The first one is, when in the USER mode, Software interrupts (SWI) are needed to read or write the registers and peripherals that are only accessible in privileged mode. Having to use SWI disables the interrupts during the time a SWI is processed. The second advantage of executing the application in SYS mode is again related to the SWI, but this time it is one of CPU efficiency: the code required to replace the functionality of the SWI is much smaller, therefore less CPU is needed to execute the same operation.

There is no fundamental reason why an application should be executing in the un-privileged mode with mAbassi. First, even though the mAbassi kernel is a single function, it always executes within the application context. There are no service requests, alike the Arm9 SWI, involved to access the kernel. And second, mAbassi was architected to be optimal for embedded application, where the need to control accesses to peripherals or other resources, as in the case of a server level OS, is not applicable.

Only the file mAbassi_SMP_CORTEXA9_GCC.s (ARM_SMP_L1_L2_GCC.s, release version 1.69.69 and upward, also needs this information) requires the information if the application executes in the privileged mode or not. By default, the distribution file sets the processor to operate in privileged mode. To select if the application executes in privileged mode or not, all there is to do is to change the value associated to the definition of the token OS_RUN_PRIVILEGE, located around line 40. Associating a numerical value of zero to the build option configures mAbassi to let the application execute in the USER processor mode, which is un-privileged:

Table 2-14 `OS_RUN_PRIVILEGE` modification

```
#ifndef OS_RUN_PRIVILEGE
.ifndef OS_RUN_PRIVILEGE
  .equ OS_RUN_PRIVILEGE, 0
.endif
#endif
```

Alternatively, it is possible to overload the `OS_RUN_PRIVILEGE` value set in `mAbassi_SMP_CORTEXA9_GCC.s` by using the assembler command line option `--defsym` and specifying the desired mode of execution as shown in the following example, where the selected mode is non-privilege:

Table 2-15 Command line set of `OS_RUN_PRIVILEGE`

```
arm-xilinx-eabi-as  … --defsym OS_RUN_PRIVILEGE=0  …
```

2.6 L1 & L2 Cache Set-up

The build option `OS_USE_CACHE` controls if the MPcore L1 and L2 caches, the memory management unit (MMU) and the snoop control unit (SCU) are configured and enabled. Setting the token `OS_USE_CACHE` to a non-zero value configures and enables the caches, MMU and SCU; setting it to a value of zero (0) disables the caches, MMU and SCU. The `OS_USE_CACHE` token is defined around line 120 in the file `mAbassi_SMP_CORTEXA9_GCC.s` and is set to enable (non-zero) by default. This is shown in the following table:

Table 2-16 `OS_USE_CACHE` set-up

```
#ifndef OS_USE_CACHE
.ifndef OS_USE_CACHE
  .equ OS_USE_CACHE, 0
.endif
#endif
```

Alternatively, it is possible to overload the `OS_USE_CACHE` value set in `mAbassi_SMP_CORTEXA9_GCC.s` by using the assembler command line option `--defsym` and specifying the desired mode of execution as shown in the following example, where the selected mode is non-privileged:

Table 2-17 Command line set of `OS_USE_CACHE`

```
arm-xilinx-eabi-as  … --defsym OS_USE_CACHE=0  …
```

NOTE: When the caches are used, there is a dependency on the external function `COREcacheON()`. This function is located in the file `ARMv7_SMP_L1_L2_GCC.s` file, which is optional.

2.7 Saturation Bit Set-up

In the ARM Cortex-A9 status register, there is a sticky bit to indicate if an arithmetic saturation or overflow has occurred during a DSP instruction; this is the Q flag in the status register (bit #27). By default, this bit is not kept localized at the task level, as extra processing is required during the task context switch 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 for a single task, then it must be kept local to each task. To keep the value of the saturation bit localized, the token `OS_HANDLE_PSR_Q` must be set to a non-zero value; to disable the localization, it must be set to a zero value. This is located at around line 50 in the file `mAbassi_SMP_CORTEXA9_GCC.s`. The distribution code disables the localization of the Q bit, setting the token `HANDLE_PSR_Q` to zero, as shown in the following table:

Table 2-18 Saturation Bit configuration

```
#ifndef OS_HANDLE_PSR_Q
.ifndef OS_HANDLE_PSR_Q
  .equ OS_HANDLE_PSR_Q, 0
.endif
#endif
```

Alternatively, it is possible to overload the `OS_HANDLE_PSR_Q` value set in `mAbassi_SMP_CORTEXA9_GCC.s` by using the assembler command line option `--defsym` and specifying the desired setting (here is to keep localized) with the following:

Table 2-19 Command line set of Saturation Bit configuration

```
arm-xilinx-eabi-as  … --defsym OS_HANDLE_PSR_Q=1 …
```

2.8 Spinlock implementation

All multi-core SMP RTOSes require the use of spinlocks to provide a short time exclusive access to shared resources. mAbassi internally uses two functions to lock and unlock the spinlocks, `CORElock()` and `COREunlock()` (refer to mAbassi User's Guide [R1]), and these functions are implemented in the `mAbassi_SMP_CORTEXA9_GCC.s` file. The spinlocks can be implemented using 3 different techniques:

- Pure software spinlock
- Using the `LDREX` and `STREX` instructions
- Using a hardware spinlock register

The difference between the 3 types of spinlocks can be resumed as follows:

- A pure software spinlock disables the interrupts for a short time but does not depend on any hardware resources nor peripherals
- The `LDREX`/`STREX` based spinlock does not disable the interrupts but the ARM Snoop Control Unit (SCU) must be enabled, which implies the D-Cache must be configured and enabled which in turn requires the Memory Management Unit (MMU) to be configured and enabled
- The hardware register spinlock does not disable the interrupts, uses less code than the pure software spinlock, but it uses more code than the `LDREX`/`STREX` based spinlock. This type of spinlock can only be used f the device has custom spinlock registers

The type of spinlock to use is specified with the token the token `OS_SPINLOCK`. It must be set to zero (0) for a pure software spinlock, one (1) for the `LDREX`/`STREX` based spinlock, or, for a hardware register base spinlock, to the same value as the token `OS_PLATFORM` (Section 2.2) when the target device supports hardware spinlocks.

NOTE: If the token OS_SPINLOCK is set to one (1) for the LDREX / STREX based spinlock and the cache is not enabled (Section 2.6), an assembly time error message is issued.

The distribution code uses the pure software spinlock, meaning the OS_PLATFORM token is set to zero, as shown in the following table. This is located around line 90 in the file mAbassi_SMP_CORTEXA9_GCC.s:

Table 2-20 OS_SPINLOCK specification

```
#ifndef OS_SPINLOCK
.ifndef OS_SPINLOCK
  .equ OS_SPINLOCK, 0
.endif
#endif
```

It is possible to overload the OS_SPINLOCK value set in mAbassi_SMP_CORTEXA9_GCC.s by using the assembler command line option --defsym and specifying the type of spinlock as shown in the following example, where the spinlock is set to use the LDREX/STREX pair of instructions:

Table 2-21 Command line set of OS_SPINLOCK

```
arm-xilinx-eabi-as  … --defsym OS_SPINLOCK=1 …
```

Some target platforms have sets of hardware spinlock registers, e.g. Texas Instruments OMAP 4460. When the selected spinlock implementation is based on the hardware spinlock registers, mAbassi reserves a total of 4 spinlock registers. By default, the spinlock register indexes #0 to #5 are used by mAbassi and as such must not be used by anything else. If mAbassi needs to co-exist with other applications that are already using one or more registers in this set of 4, it is possible to make mAbassi use a different set of indexes. The base index of the set of 4 registers is specified with the value assigned to the token OS_SPINLOCK_BASE. The assembly file and the "C" file use same token and they must be set to the same value. As an example to make mAbassi use the hardware registers #22 to #27, all there is to do is to change the numerical value associated to the OS_SPINLOCK_BASE token, located around line 100; this is shown in the following table:

Table 2-22 OS_SPINLOCK_BASE modification

```
#ifndef OS_SPINLOCK_BASE
.ifndef OS_SPINLOCK_BASE
  .equ OS_SPINLOCK_BASE, 22
.endif
#endif
```

It is also possible to overload the OS_SPINLOCK_BASE value set in mAbassi_SMP_CORTEXA9_GCC.s by using the assembler command line option --defsym and specifying the required base register index as shown in the following example:

Table 2-23 Command line set of OS_SPINLOCK_BASE (ASM)

```
arm-xilinx-eabi-as  … --defsym OS_SPINLOCK_BASE=22 …
```

The same numerical value must also be provided to the "C" file. This is show in the following table:

Table 2-24 Command line set of OS_SPINLOCK_BASE (C)

```
arm-xilinx-eabi-gcc  … --DOS_SPINLOCK_BASE=22 …
```

There is a possible race condition when using spinlocks on a target processor with 3 or more cores. This is the corner case when the tasks running on 3 cores or more are all trying non-stop to lock and unlock the same spinlock. When this condition arises, it is possible that the same 2 cores always get the spinlock, starving the other one(s). This is not an issue with mAbassi but it is due to the fact the memory accesses (S/W spinlock or H/W spinlock) and the cache give access based on the core numbering and not in a round robin or random fashion. To randomize the core given the spinlock, a random delay can be added when the number of cores (OS_N_CORES) is greater than 2. The delay is added when the build option OS_SPINLOCK_DELAY is set to a non-zero value and OS_N_CORE is greater than 2; by default, the token OS_SPINLOCK_DELAY is set to a non-zero value. As for other tokens, the numerical value associated to the OS_SPINLOCK_DELAY token, located around line 90, can be changed as shown in the following table:

Table 2-25 OS_SPINLOCK_DELAY modification

```
#ifndef OS_SPINLOCK_DELAY
.ifndef OS_SPINLOCK_DELAY
  .equ OS_SPINLOCK_DELAY, 0
.endif
#endif
```

It is also possible to overload the OS_SPINLOCK_DELAY value set in mAbassi_SMP_CORTEXA9_GCC.s by using the assembler command line option --defsym and specifying the required base register index as shown in the following example:

Table 2-26 Command line set of OS_SPINLOCK_DELAY (ASM)

```
arm-xilinx-eabi-as  … --defsym OS_SPINLOCK_DELAY=0 …
```

2.9 Spurious Interrupt

The Cortex-A9 interrupt controller (GIC) generates an interrupt when a spurious interrupt is detected. The interrupt number for a spurious interrupt is 1023 and it cannot be disabled. When mAbassi is configured to handle less than 1024 interrupt sources (set by the build option `OS_N_INTERRUPTS`), then if a spurious interrupt occurs, the function pointer of the handler read from the interrupt table is invalid, as the interrupt table is not large enough to hold an entry for it. The build option `OS_SPURIOUS_INT` (new in version 1.66.65) can be set to inform the interrupt dispatcher to ignore interrupt #1023. The special handling of the spurious interrupt is enabled when the build option `OS_SPURIOUS_INT` is set to a non-zero value; by default, the token `OS_SPURIOUS_INT` is set to a non-zero value, enabling the special handling. As for other tokens, the numerical value associated to the `OS_SPURIOUS_INT` token, located around line 130, can be changed as shown in the following table:

Table 2-27 `OS_SPURIOUS_INT` modification

```
#ifndef OS_SPURIOUS_INT
.ifndef OS_SPURIOUS_INT
  .equ OS_SPURIOUS_INT, 0
.endif
#endif
```

It is also possible to overload the `OS_SPURIOUS_INT` value set in `mAbassi_SMP_CORTEXA9_GCC.s` by using the assembler command line option `--defsym` and specifying the required base register index as shown in the following example:

Table 2-28 Command line set of `OS_SPURIOUS_INT` (ASM)

```
arm-xilinx-eabi-as  … --defsym OS_SPURIOUS_INT=0 …
```

Unless the build option `OS_N_INTERRUPTS` is set to 1024, the build option `OS_SPURIOUS_INT` should never be set to a value of zero. The real-time and code size impact of including the spurious interrupt special handling is very minimal: it adds 1 instruction when using 32-bit ARM instructions and 2 instructions with Thumb2.

2.10 Thumb2

The assembly support file (`mAbassi_SMP_CORTEXA9_GCC.s`) is by default using 32-bit ARM instructions. The build option `OS_ASM_THUMB` (new in version 1.66.66) can be set to use Thumb2 instructions instead. The use of Thumb2 instructions is enabled when the build option `OS_ASM_THUMB` is set to a non-zero value; by default, the token `OS_ASM_THUMB` is set to a non-zero value, enabling the special handling. As for other tokens, the numerical value associated to the `OS_ASM_THUMB` token, located around line 135, can be changed as shown in the following table:

Table 2-29 `OS_ASM_THUMB` modification

```
#ifndef OS_ASM_THUMB
.ifndef OS_ASM_THUMB
  .equ OS_ASM_THUMB, 1
.endif
#endif
```

It is also possible to overload the `OS_ASM_THUMB` value set in `mAbassi_SMP_CORTEXA9_GCC.s` by using the assembler command line option `--defsym` and specifying the required base register index as shown in the following example:

Table 2-30 Command line set of `OS_ASM_THUMB` (ASM)

```
arm-xilinx-eabi-as  … --defsym OS_ASM_THUMB=1 …
```

NOTE: Never use the `--thumb` command line option with the `mAbassi_SMP_CORTEXA9_GCC.s` file.

2.11 VFP / NEON set-up

When a processor is coupled with a floating-point peripheral and this VFP is used by the application code, then mAbassi must be informed it must preserve the VFP register during the task context switch and the interrupt context save. As such, the type of VFP mAbaasi must handle needs to match the one the compiler is set to use. A mismatch will quite likely make the application misbehave. The type of floating-point peripheral is specified to Abassi through the assembly file `mAbassi_SMP_CORTEXA9_GCC.s`, which depending on its configuration can handle three different types of floating-point peripherals. They are:

- No VFP coprocessor
- VFPv2 / VFPv3D16 / Neon with 16 registers
- VFPv3 / Neon with 32 registers

The type of FPU to support is specified in the `mAbassi_SMP_CORTEXA9_GCC.s` file, with the build option `OS_VFP_TYPE`. The value assigned to `OS_VFP_TYPE` is the number of 64 bit registers the VFP possesses. The valid values for `OS_VFP_TYPE` are:

Table 2-31 `OS_VFP_TYPE` values

FPU	`OS_VFP_TYPE` value
No FPU	`0`
VFPv2	`16`
VFPv3D16	`16`
VFPv3	`32`
Neon	`32`

By default, the token `OS_VFP_TYPE` is set to a value of 32 indicating to mAbassi to support a floating-point peripheral with 32 64-bit registers. The following table shows the default setting of the token `OS_VFP_TYPE` in file `mAbassi_CORTEXA9_GCC.s` around line 105:

Table 2-32 `OS_VFP_TYPE` setting

```
#ifndef OS_VFP_TYPE
.ifndef OS_VFP_TYPE
  .equ OS_VFP_TYPE, 32
.endif
#endif
```

The VFP type can be changed directly in the file `mAbassi_CORTEXA9_GCC.s`, by changing the assigned numerical value. Also, the value internally assigned in `mAbassi_CORTEXA9_GCC.s` can be overloaded through the command line option to the assembler.

The following 3 tables show the command line setting for each of the supported floating-point units:

Table 2-33 Command line selection of no VFP

```
arm-xilinx-eabi-as  … --defsym OS_VFP_TYPE=0 …
```

Table 2-34 Command line selection of VFP with 16 registers

```
arm-xilinx-eabi-as  … --defsym OS_VFP_TYPE=16 …
```

Table 2-35 Command line selection of VFP with 32 registers

```
arm-xilinx-eabi-as  … --defsym OS_VFP_TYPE=32…
```

2.12 Multithreading

For a complete description of the multi-threading protection mechanisms, refer to the Library reentrance protection document [R4].

NOTE: When the application is compiled using the Thumb instruction set then the reentrance and the multithreading protections are not supported by the Red Hat's Newlib. This is not a limitation of mAbassi but a restriction of the pre-compiled library itself. A compile time error message is generated if the Thumb instruction set is used when the reentrance and multithreading protections are enabled.

The build option can be set directly in the assembly file; this is located at around line 115 in the file `mAbassi_SMP_CORTEXA9_GCC.s`. The distribution code disables the multithreading and reentrance protections, setting the token `OS_NEWLIB_REENT` to zero, as shown in the following table:

Table 2-36 Assembly file multithread configuration

```
#ifndef OS_NEWLIB_REENT
.ifndef OS_NEWLIB_REENT
  .equ OS_NEWLIB_REENT, 0
.endif
#endif
```

To enable the full multithreading and reentrance protections, set the value of `OS_NEWLIB_REENT` to a positive value as shown in the following table:

Table 2-37 Assembly file multithread protection enabling

```
#ifndef OS_NEWLIB_REENT
.ifndef OS_NEWLIB_REENT
  .equ OS_NEWLIB_REENT, 1
.endif
#endif
```

Alternatively, it is possible to overload the `OS_NEWLIB_REENT` value set in the file `mAbassi_SMP_CORTEXA9_GCC.s` by using the assembler command line option `--defsym` and specifying the desired setting with the following:

Table 2-38 Command line set of multithread configuration

```
arm-xilinx-eabi-as  … --defsym OS_NEWLIB_REENT=1 …
```

The exact same definition of `OS_NEWLIB_REENT` as the one specified for the assembler (either directly or through the command line) must be must be given to the compiler. This can be done with the command line option `-D` and specifying the setting with the following:

Table 2-39 Command line set of multithread configuration

```
arm-xilinx-eabi-gcc  … -DOS_NEWLIB_REENT=1 …
```

A little more than one (1) kilobyte per task is needed to support multithreading protection. This extra memory is not an integral part of the task descriptor; instead a pointer in the task descriptor holds the location of this per task extra memory. The one (1) kilobyte per task is allocated using the component `OSalloc()`, which means either enough memory must be reserved with `OS_ALLOC_SIZE` or, if `OS_ALLOC_SIZE` is set to zero, enough heap area must be reserved for `malloc()`.

2.13 Performance Monitoring

New in version 1.69.69, is the performance monitoring add-on. This facility relies on a fine resolution timer / counter and this timer / counter is set-up and handled in the file `mAbassi_SMP_CORTEXA9_GCC.s`. There are 3 build options related to the performance monitoring timer / counter:

- `OS_PERF_TIMER_BASE`
- `OS_PERF_TIMER_DIV`
- `OS_PERF_TIMER_ISR`

All Cortex-A9 MPCores have a global timer / counter and it is a natural candidate to be used by the performance monitoring add-on. To use the global timer / counter, set the build option `OS_PERF_TIMER_BASE` to a value of 0. If the pre-scaling value of the global timer / counter is set in the application, then set the build option `OS_PERF_TIMER_DIV` to a value of 0. If the prescaler is not set by the application, then set build option `OS_PERF_TIMER_DIV` to the desired prescaler value. For the global timer / counter, the build option `OS_PERF_TIMER_ISR` is ignored as it does not requires interrupts as it is a 64-bit timer (64 bits is the size of the counters used by the performance monitoring add-on).

For other counter / timer options, please look directly into the file `mAbassi_SMP_CORTEXA9_GCC.s`, searching for the `OS_PERF_TIMER_BASE` token. There is a very detailed table explaining all the offerings and specifying the values to set the 3 related build options to for each the timer / counter supported.

2.14 Code Sourcery / Yagarto

There are two main trunks of the GCC for ARM; one is maintained by Mentor Graphics and is named Code Sourcery, the other was originally named Yagarto, with Atollic, for example, using this variant. There are small differences between the two that are related to the start-up code and the "C" library. Both variants are supported as indicated through the token `OS_CODE_SOURCERY`. By default, this token is set to a non-zero value to fulfill the start-up and library requirements of the Code Sourcery variant. If the Yagarto variant is used instead, the token `OS_CODE_SOURCERY` must be set to a value of zero; this is shown in the following table where the code snippet is located around line 125:

Table 2-40 `OS_CODE_SOURCERY` modification

```
#ifndef OS_CODE_SOURCERY
.ifndef OS_CODE_SOURCERY
  .equ OS_CODE_SOURCERY, 0
.endif
#endif
```

It is also possible to overload the `OS_SPINLOCK_DELAY` value set in `mAbassi_SMP_CORTEXA9_GCC.s` by using the assembler command line option `--defsym` and specifying the required base register index as shown in the following example:

Table 2-41 Command line set of `OS_CODE_SOURCERY` (ASM)

```
arm-xilinx-eabi-as  … --defsym OS_CODE_SOURCERY=0 …
```

The same numerical value must also be provided to the "C" file. This is show in the following table:

Table 2-42 Command line set of `OS_CODE_SOURCERY` (C)

```
arm-xilinx-eabi-gcc  … --DOS_CODE_SOURCERY=0 …
```

2.15 Section Naming

It is possible to select between 3 type of linker section naming for the code and data from the RTOS assembly file `mAbassi_SMP_CORTEXA9_GCC.s`. The build option `OS_SECT_NAMES` controls the naming. If this build option is not defined or defined and set to a zero value, the code, R/O data and R/W sections are respectively named `.text`, `.data` and `.bss`. If the build option `OS_SECT_NAMES` is defined and set to a negative value, the sections are named `.text.mabassi`, `.data.mabassi` and `.bss.mabassi`. If instead the built option is defined and set to a positive value, the sections are named `.text.FctName`, `.data.`*`FctName`* and `.bss.`*`FctName`*, where *`FctName`* is the name of the function (this is identical to the way the compiler names the section).

3 Interrupts

The mAbassi RTOS needs to be aware when kernel requests are performed inside or outside an interrupt context. For all IRQ sources the mAbassi 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.

The distribution makes provision for 256 sources of interrupts, as specified by the token `OX_N_INTERRUPTS` in the file `mAbassi.h`[2], and the value of `OX_N_INTERRUPTS` is the internal default value used by `mAbassi.c`. Even though the Generic Interrupt Controller (GIC) peripheral supports a maximum of 1020 interrupts, it was decided to set the distribution value to 256, as this seems to be a typical maximum supported by the different devices on the market.

3.1 Interrupt Handling

3.1.1 Interrupt Table Size

Most devices do not require all 256 interrupts, as they may only handle between 64 and 128 sources of interrupts; or some very large device may require more than 256. The interrupt table can be easily reduced to recover data space. All there is to do is to define the build option `OS_N_INTERRUPTS` (`OS_N_INTERRUPTS` is used to overload mAbassi internal value of `OX_N_INTERRUPTS`) to the desired value. This can be done by using the compiler command line option `-D` and specifying the desired setting with the following:

Table 3-1 Command line set the interrupt table size

```
arm-xilinx-eabi-gcc … -D=OS_N_INTERRUPTS=49 …
```

3.1.2 Interrupt Installer

Attaching a function to a regular interrupt is quite straightforward. All there is to do is use the RTOS component `OSisrInstall()` to specify the interrupt number and the function to be attached to that interrupt number. For example, Table 3-2 shows the code required to attach the private timer interrupt on an ARM MPcore processor (Interrupt ID #29) to the RTOS timer tick handler (`TIMtick`):

Table 3-2 Attaching a Function to an Interrupt

```
#include “mAbassi.h”

   …
   OSstart();
   …
   GICenable(29, 128, 1);                  /* Timer set mid prioirty edge triggered  */
   OSisrInstall(29, &TIMtick);

   /* Set-up the count reload and enable private timer interrupt and start the timer */

   … /* More ISR setup */

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

[2] This is located in the port-specific definition area.

At start-up, once `OSstart()` has been called, all `OS_N_INTERRUPTS` (truly `OX_N_INTERRUPTS` if not overloaded) interrupt handler functions are set to a "do nothing" function, named `OSinvalidISR()`. If an interrupt function is attached to an interrupt number using the `OSisrInstall()` component before calling `OSstart()`, this attachment will be removed by `OSstart()`, so `OSisrInstall()` should never be used before `OSstart()` has executed. 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 Table 3-3:

Table 3-3 Invalidating an ISR handler

```
#include "mAbassi.h"

   …
   /* Disable the interrupt source */
   OSisrInstall(Number, &OSinvalidISR);
   …
```

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

The interrupt number, as reported by Generic Interrupt Controller (GIC), is acknowledged by the ISR dispatcher, but the dispatcher does not remove the request by a peripheral if the peripheral generate a level interrupt. The removal of the interrupt request must be performed within the interrupt handler function.

One has to remember the mAbassi interrupt table is shared across all the cores. Therefore, if the same interrupt number is used on multiple cores, but the processing is different amongst the cores, a single function to handle the interrupt must be used in which the core ID controls the processing flow. The core ID is obtained through the `COREgetID()` component of mAbassi. One example of such situation is if the private timer is used on each of two cores, but each core private timer has a different purpose, e.g.:

1- on one core, it is the RTOS timer base

2- on the other core, it is the real-time clock tick.

At the application level, when the core ID is used to select specific processing, a critical region exists that must be protected by having the interrupts disabled (see mAbassi User's Guide [R1]). But within an interrupt handler, as nested interrupts are not supported for the Cortex-A9, there is no need to add a critical region protection, as interrupts are disabled when processing an interrupt.

3.2 Fast Interrupts

Fast interrupts are supported on this port as the FIQ interrupts. The ISR dispatcher is designed to only handle the IRQ interrupts. A default do-nothing FIQ handler is supplied with the distribution; the application can overload the default handler (Section 7.2).

3.3 Nested Interrupts

Interrupt nesting, other than a FIQ nesting an IRQ, is not supported on this port. The reason is simply based on the fact the Generic Interrupt Controller is not a nested controller. Also, supporting nesting on this processor architecture becomes real-time inefficient as the processor interrupt context save is not stack based, but register bank based.

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 Cortex-A9, 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	48 bytes
Blocked/Preempted task context save / VFP enable (`OS_FPU_TYPE != 0`)	+112 bytes
Interrupt dispatcher context save (User Stack)	64 bytes
Interrupt dispatcher context save (User Stack) / 16 - VFP (`OS_FPU_TYPE == 16`)	+136 bytes
Interrupt dispatcher context save (User Stack) / 32 - VFP (`OS_FPU_TYPE == 32`)	+264 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, add to all this the stack required by the code implementing the task operation, or the interrupt operation.

NOTE: The ARM Cortex-A9 processor needs alignment on 8 byes for some instructions accessing memory. When stack memory is allocated, mAbassi 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.

5 Memory Configuration

The mAbassi kernel is not a kernel entered though a service request, such as the SWI on the Cortex-A9. The kernel is a regular function, protected against re-entrance or multiple core entrance. The kernel code executes as part of the application code, with the same processor mode and access privileges.

6 Search Set-up

The search results are identical to the single core Cortex-A9 port as Abassi and mAbassi use the same code for the search algorithm. Please refer to the single core Cortex-A9 port document [R2] for the measurements.

7 API

The ARM Cortex-A9 supports multiple types of exceptions. Defaults exception handlers are supplied with the distribution code, but each one of them can be overloaded by an application specific function. The default handlers are simply an infinite loop (except FIQ, which is a do-nothing with return from exception). The choice of an infinite loop was made as this allows full debugging, as all registers are left untouched by the defaults handlers. The following sub-sections describe each one of the default exception handlers.

7.1 DATAabort_Handler

Synopsis

```
#include "mAbassi.h"

void DATAabort_Handler(void);
```

Description

`DATAabort_Handler()` is the exception handler for a data abort fault. In the distribution code, this is implemented as an infinite loop. If the application needs to perform special processing when a data fault occurs, all there is to do is to include a function with the above function prototype, and it will overload the supplied data abort handler. As this is an exception, the return must be performed with a "`subs pc, lr, #8`".

Availability

Always.

Arguments

`void`

Returns

`void`

Component type

Function

Options

Notes

This is an exception function, executing in the abort processor mode. This means the abort stack is in use instead of the user stack, and the IRQ interrupts are disabled.

See also

`FIQ_Handler()` (Section 7.2)
`PFabort_Handler()` (Section 7.3)
`SWI_Handler()` (Section 7.4)
`Undef_Handler()` (Section 7.5)

7.2 FIQ_Handler

Synopsis

```
#include "mAbassi.h"

void FIQ_Handler(void);
```

Description

`FIQ_Handler()` is the handler for a fast interrupt request. In the distribution code, this is implemented as a return only. If the application needs to handle fast interrupts, all there is to do is to include a function with the above function prototype and it will overload the supplied fast interrupt handler. As this is an exception, the return must be performed with a "`subs pc, lr, #4`".

Availability

Always.

Arguments

`void`

Returns

`void`

Component type

Function

Options

Notes

This is an exception function, executing in the FIQ processor mode. This means the FIQ stack is in use instead of the user stack, and the FIQ are now disabled and IRQ interrupts are also disabled.

See also

`DATAabort_Handler()` (Section 7.1)
`PFabort_Handler()` (Section 7.3)
`SWI_Handler()` (Section 7.4)
`Undef_Handler()` (Section 7.5)

7.3 PFabort_Handler

Synopsis

```
#include "mAbassi.h"

void PFabort_Handler(void);
```

Description

`PFabort_Handler()` is the exception handler for a pre-fetch abort fault. In the distribution code, this is implemented as an infinite loop. If the application needs to perform special processing when a pre-fetch fault occurs, all there is to do is to include a function with the above function prototype and it will overload the supplied data abort handler. As this is an exception, the return must be performed with a "`subs pc, lr, #4`".

Availability

Always.

Arguments

`void`

Returns

`void`

Component type

Function

Options

Notes

This is an exception function, executing in the abort processor mode. This means the abort stack is in use instead of the user stack, and the IRQ interrupts are disabled.

See also

`DATAabort_Handler()` (Section 7.1)
`FIQ_Handler()` (Section 7.2)
`SWI_Handler()` (Section 7.4)
`Undef_Handler()` (Section 7.5)

7.4 SWI_Handler

Synopsis

```
#include "mAbassi.h"

void SWI_Handler(int SWInmb);
```

Description

`SWI_Handler()` is the exception handler for software interrupts that are not handled or reserved by mAbassi. The number of the software interrupt is passed through the function argument `SWInmb`. This is a regular function; do not use the exception instruction "`movs pc, lr`".

Availability

Always.

Arguments

`SWInmb`	Number of the software interrupt. The interrupt numbers 0 to 7 must not be used by the application as they are used / reserved by the RTOS.

Returns

`void`

Component type

Function

Options

Notes

This is a regular function, but executing in the supervisor processor mode. This means the supervisor stack is in use instead of the user stack, and the IRQ interrupts are disabled.

See also

`DATAabort_Handler()` (Section 7.1)
`FIQ_Handler()` (Section 7.2)
`FPabort_Handler()` (Section 7.3)
`Undef_Handler()` (Section 7.5)

7.5 Undef_Handler

Synopsis

```
#include "mAbassi.h"

void Undef_Handler(void);
```

Description

`Undef_Handler()` is the exception handler for a undefined instruction fault. In the distribution code, this is implemented as an infinite loop. If the application needs to perform special processing when an undefined instruction fault occurs, all there is to do is to include a function with the above function prototype, and it will overload the supplied undefined instruction abort handler. As this is an exception, the return must be performed with a "`movs pc, lr`".

Availability

Always.

Arguments

`void`

Returns

`void`

Component type

Function

Options

Notes

This is an exception function, executing in the abort processor mode. This means the abort stack is in use instead of the user stack, and the IRQ interrupts are disabled.

See also

`DATAabort_Handler()` (Section 7.1)
`FIQ_Handler()` (Section 7.2)
`PFabort_Handler()` (Section 7.3)
`SWI_Handler()` (Section 7.4)

8 Chip Support

No custom chip support is provided with the distribution because most device manufacturers provide a BSP, i.e. code to configure the peripherals on their devices. The distribution code contains some of the manufacturer's open source libraries.

Basic support for the Generic Interrupt Controller (GIC) is provided in this port as SMP/BMP multi-core on the Cortex-A9 MPCore device requires the use of interrupts. The following sub-sections describe the two support components.

8.1 GICenable

Synopsis

```
#include "mAbassi.h"

void GICenable(int IntNmb, int Prio, int Edge);
```

Description

`GICenable()` is the component used to enable an interrupt number (called ID*nn* in the literature) on the Generic Interrupt Controller (GIC). The interrupt configuration is always applied to the core on which `GICenable()` is executing.

Availability

Always.

Arguments

`IntNmb`	Interrupt number to enable
`Prio`	Priority of the interrupt `0` : highest priority `255` : lowest priority
`Edge`	Edge or level detection `== 0` : level detection `!= 0` : edge detection

Returns

`void`

Component type

Function

Options

Notes

On the Cortex-A9 MPCore, some GIC registers are local to the core, while others are global across all cores. Care must be taken when using `GICenable()`.
When the interrupt number (argument `IntNmb`) is non-negative, then the GIC is programmed to target the interrupt to the core it's currently operating on. If the interrupt number is negative, then the interrupt number `–IntNmb` is targeted to all cores.
The function `GICenable()` is implemented in assembly language, in the file `mAbassi_SMP_CORTEXA9_GCC.s` as it avoids supplying 2 port files in the distribution. If the supplied functionality does not fulfill the application needs, `GICenable()` can be overloaded by adding a new `GICenable()` function in the application. As the supplied assembly function is declared weak, it will not be included during the link process. The equivalent "C" code of the distribution implementation is supplied in comments in the assembly file.

See also

`GICinit()` (Section 8.2)

8.2 GICinit

Synopsis

```
#include "mAbassi.h"

void GICinit(void);
```

Description

`GICinit()` is the component used to initialize the Generic Interrupt Controller (GIC) for the needs of mAbassi. It must be used after using the `OSstart()` component and before `GICenable()` and / or `OSeint()` components. Also, it must be used in every `COREstartN()` function.
Consult the mAbassi User guide for more information on this topic [R1].

Availability

Always.

Arguments

`void`

Returns

`void`

Component type

Function

Options

Notes

The function `GICinit()` is implemented in assembly language, in the file `mAbassi_SMP_CORTEXA9_GCC.s` as it avoids supplying 2 port files in the distribution. If the supplied functionality does not fulfill the application needs, `GICinit()` can be overloaded by adding a new `GICinit()` function in the application. As the supplied assembly function is declared weak, it will not be included during the link process. The equivalent "C" code of the distribution implementation is supplied in comments in the assembly file.

See also

`GICenable()` (Section 8.1)

9 Measurements

This section provides an overview of the memory requirements encountered when the RTOS is used on the Arm9 and compiled with Mentor's Code Sourcery tool chain. Latency measurements are provided, but one should remember CPU latency latencies are highly dependent on 3 factors. It first depends on how many cores are used; it also depends on the type of load balancing, i.e. if mAbassi is configured in SMP or BMP, and if the load balancing algorithm is the True or the Packed one. All these possible configurations are one part of the complexity. A second part of the complexity is where the task switch was detected and on which core(s) the task switch will occur due to that change of state. Finally, the third factor is if a core is already executing in the kernel when another needs to enter the kernel. Any combination of these dynamic factors affects differently the CPU latency of mAbassi. The specific configuration and run-time conditions are described in the latency subsection (Section 9.2)

9.1 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 runtime 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 1.34.36 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. Debugging model: Off[3] (option –g not specified)
2. Optimization level: `-Os`
3. Target `-arch=armv7-a –mtune=cortex-a9 –mcpu=cortex-a9`
4. FPU `-mfloat-abi-soft`

[3] Debugging is turned off as it restricts the optimizer.

Table 9-1 "C" Code Memory Usage

Description	Thumb Size	32-Bit Size
Minimal Build	< 1425 bytes	< 2175 bytes
+ Runtime service creation / static memory	< 1650 bytes	< 2475 bytes
+ Runtime priority change + Mutex priority inheritance + FCFS + Task suspension	< 1675 bytes	< 2475 bytes
+ Timer & timeout + Timer call back + Round robin	< 2175 bytes	< 3225 bytes
+ Events + Mailbox	< 3000 bytes	< 4425 bytes
Full Feature Build (no names)	< 3600 bytes	< 5500 bytes
Full Feature Build (no name / no runtime creation)	< 3775 bytes	< 6550 bytes
Full Feature Build (no names / no runtime creation) + Timer services module	< 4275 bytes	< 6050 bytes
OS_NEWLIB_REENT > 0	n/a	+1048 bytes
OS_NEWLIB_REENT < 0	n/a	+304 bytes
True SMP (OS_MP_TYPE == 2)	+0 bytes	+0 bytes
Packed SMP (OS_MP_TYPE == 3)	~ +100 bytes	~ +100 bytes
True BMP (OS_MP_TYPE == 4)	~ +300 bytes	~ +400 bytes
Packed BMP (OS_MP_TYPE == 5)	~ +350 bytes	~ +475 bytes

The selection of load balancing type affects the "C" code size. The added memory requirements are indicated as approximate because depending on the build option combination, the kernel code is different. As such, the optimizer does not deliver the same code size.

Table 9-2 Assembly Code Memory Usage

Description	Size
Assembly code size (non-privilege / >1 core)	1308 bytes
Assembly code size (non-privilege / ==1 core)	788 bytes
Assembly code size (privilege / >1 core)	1040 bytes
Assembly code size (privilege / ==1 core)	656 bytes
VFPv3	+116 bytes
VFPv3D16	+108 bytes
Saturation Bit Enabled	+36 bytes
`GICinit()` (>1 core)	+144 bytes
`GICinit()` (=1 core)	+84 bytes
`GICenable()` (>1 core)	+276 bytes
`GICenable()` (=1 core)	+276 bytes
`OS_NEWLIB_REENT > 0`	+84 bytes
`OS_NEWLIB_REENT < 0`	+104 bytes
Altera Cyclone V Support	+264 bytes
TI OMAP 4460 Support	+16 bytes
Xilinx Zynq Support	+84 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.

9.2 Latency

Latency of operations has been measured on an Altera Cyclone V Evaluation board populated with a 800MHz dual-core Cortex-A9. All measurements have been performed on the real platform. This means the interrupt latency measurements had to be instrumented to read the `SysTick` counter value. This instrumentation can add up to 5 or 6 cycles to the measurements. The code optimization setting that was used for the latency measurements is `-O3`, which optimizes the code generated for the best speed. The debugging option was turned off as the debugging sometimes restricts the optimizer. All operations are performed on core #0; the FPU was enabled (VFPv3 / Neon) and the type of multi-processing was set to true SMP (`OS_MP_TYPE` set to 2). The cache is enabled and the spinlock type is the one using LDREX/STREX. All measurements shown are the resulting average of the last 128 runs out of 256. This averaging is done as the presence of the cache does not guarantee a deterministic operation of the test suite. One must remember the latencies measured apply to the test suite; any other application specific latencies depend if the mAbassi code and data are or are not in the cache.

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 9-3 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 9-4 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 9-5 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 9-6 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. 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 two line function that uses the appropriate RTOS component followed by a return.

Table 9-7 lists the results obtained, where the cycle count is measured using core #0 private timer. This timer decrements its counter by 1 at every 4 CPU cycle

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 branch to the address specified in the interrupt vector table

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 9-7 Latency Measurements

Description	Minimal Features	Full Features
Semaphore posting no task switch	224 (232)	258 (262)
Semaphore waiting no blocking	228 (252)	298 (330)
Semaphore posting with task switch	444 (452)	504 (524)
Semaphore waiting with blocking	416 (416)	476 (516)
Semaphore posting in ISR with task switch	624 (640)	710 (746)
Event setting no task switch	n/a	260 (260)
Event getting no blocking	n/a	396 (404)
Event setting with task switch	n/a	556 (576)
Event getting with blocking	n/a	628 (676)
Event setting in ISR with task switch	n/a	716 (746)
Mailbox writing no task switch	n/a	320 (337)
Mailbox reading no blocking	n/a	346 (312)
Mailbox writing with task switch	n/a	596 (596)
Mailbox reading with blocking	n/a	520 (540)
Mailbox writing in ISR with task switch	n/a	762 (820)
Interrupt Latency	140	140
Context switch	48	48

10 Appendix A: Build Options for Code Size

10.1 Case 0: Minimum build

Table 10-1: 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_MP_TYPE          2    /* SMP vs BMP and load balancing selection          */
#define OS_MTX_DEADLOCK     0    /* This test validates this                         */
#define OS_MTX_INVERSION    0    /* To enable protection against priority inversion  */
#define OS_N_CORE           2    /* Number of cores to handle                        */
#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        1    /* 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_STACK_START      256  /* Stack sie of the start-up / ilde functions       */
#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                          */
```

10.2 Case 1: + Runtime service creation / static memory + Multiple tasks at same priority

Table 10-2: 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    32   /* Maximum number of requests in ISRs               */
#define OS_MP_TYPE          2    /* SMP vs BMP and load balancing selection          */
#define OS_MTX_DEADLOCK     0    /* This test validates this                         */
#define OS_MTX_INVERSION    0    /* To enable protection against priority inversion  */
#define OS_N_CORE           2    /* Number of cores to handle                        */
#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_STACK_START      256  /* Stack sie of the start-up / ilde functions       */
#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                          */
```

10.3 Case 2: + Priority change / Priority inheritance / FCFS / Task suspend

Table 10-3: 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             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_MP_TYPE          2    /* SMP vs BMP and load balancing selection            */
#define OS_MTX_DEADLOCK     0    /* This test validates this                           */
#define OS_MTX_INVERSION    1    /* To enable protection against priority inversion    */
#define OS_N_CORE           2    /* Number of cores to handle                          */
#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_STACK_START      256  /* Stack sie of the start-up / ilde functions         */
#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                            */
```

10.4 Case 3: + Timer & timeout / Timer call back / Round robin

Table 10-4: 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_MP_TYPE         2     /* SMP vs BMP and load balancing selection          */
#define OS_MTX_DEADLOCK    0     /* This test validates this                           */
#define OS_MTX_INVERSION   1     /* To enable protection against priority inversion  */
#define OS_N_CORE          2     /* Number of cores to handle                          */
#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_STACK_START     256   /* Stack sie of the start-up / ilde functions       */
#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                            */
```

10.5 Case 4: + Events / Mailboxes

Table 10-5: 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_MP_TYPE         2     /* SMP vs BMP and load balancing selection          */
#define OS_MTX_DEADLOCK    0     /* This test validates this                           */
#define OS_MTX_INVERSION   1     /* To enable protection against priority inversion  */
#define OS_N_CORE          2     /* Number of cores to handle                          */
#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_STACK_START     256   /* Stack sie of the start-up / ilde functions       */
#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                          */
```

10.6 Case 5: Full feature Build (no names)

Table 10-6: 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          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_MP_TYPE         2     /* SMP vs BMP and load balancing selection          */
#define OS_MTX_DEADLOCK    0     /* This test validates this                           */
#define OS_N_CORE          2     /* Number of cores to handle                          */
#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_STACK_START     256   /* Stack sie of the start-up / ilde functions       */
#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                          */
```

10.7 Case 6: Full feature Build (no names / no runtime creation)

Table 10-7: 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_MP_TYPE         2     /* SMP vs BMP and load balancing selection          */
#define OS_MTX_DEADLOCK    0     /* This test validates this                           */
#define OS_MTX_INVERSION   1     /* To enable protection against priority inversion  */
#define OS_N_CORE          2     /* Number of cores to handle                          */
#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_STACK_START     256   /* Stack sie of the start-up / ilde functions       */
#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                          */
```

10.8 Case 7: Full build adding the optional timer services

Table 10-8: 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_MP_TYPE         2     /* SMP vs BMP and load balancing selection          */
#define OS_MTX_DEADLOCK    0     /* This test validates this                           */
#define OS_MTX_INVERSION   1     /* To enable protection against priority inversion  */
#define OS_N_CORE          2     /* Number of cores to handle                          */
#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_STACK_START     256   /* Stack sie of the start-up / ilde functions       */
#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                          */
```

11 References

[R1] mAbassi RTOS – User Guide, available at http://www.code-time.com
[R2] Abassi Port – Cortex-A9, available at http://www.code-time.com
[R3] NewLib documentation, available at http://sourceware.org/newlib
[R4] Abassi RTOS – Library Re-entrance Protection, available at http://www.code-time.com