

CODE TIME TECHNOLOGIES

# Abassi RTOS

---

Porting Document  
ARM Cortex-M4 – IAR

## **Copyright Information**

This document is copyright Code Time Technologies Inc. ©2012-2013. 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.

IAR Embedded Workbench is a trademark owned by IAR Systems AB. ARM and Cortex are registered trademarks of ARM Limited. All other trademarks are the property of their respective owners.

## Table of Contents

<b>1</b>	<b>INTRODUCTION .....</b>	<b>6</b>
1.1	DISTRIBUTION CONTENTS .....	6
1.2	LIMITATIONS .....	6
<b>2</b>	<b>TARGET SET-UP .....</b>	<b>7</b>
2.1	OS_STACK_SIZE SET-UP .....	7
2.2	INTERRUPT STACK SET-UP .....	9
2.3	SATURATION BIT SET-UP.....	10
2.4	FPU SET-UP.....	11
2.5	MULTITHREADING.....	13
2.5.1	<i>Reentrance Protection .....</i>	<i>13</i>
2.5.2	<i>Full Multithreading Protection.....</i>	<i>14</i>
2.5.3	<i>Partial Multithreading Protection.....</i>	<i>15</i>
<b>3</b>	<b>INTERRUPTS .....</b>	<b>17</b>
3.1	INTERRUPT HANDLING .....	17
3.1.1	<i>Interrupt Table Size .....</i>	<i>17</i>
3.1.2	<i>Interrupt Installer .....</i>	<i>19</i>
3.2	INTERRUPT PRIORITY AND ENABLING .....	20
3.3	FAST INTERRUPTS.....	20
3.4	NESTED INTERRUPTS .....	23
<b>4</b>	<b>STACK USAGE.....</b>	<b>24</b>
<b>5</b>	<b>SEARCH SET-UP .....</b>	<b>25</b>
<b>6</b>	<b>CHIP SUPPORT .....</b>	<b>28</b>
<b>7</b>	<b>MEASUREMENTS.....</b>	<b>29</b>
7.1	MEMORY .....	29
7.2	LATENCY.....	31
<b>8</b>	<b>APPENDIX A: BUILD OPTIONS FOR CODE SIZE.....</b>	<b>36</b>
8.1	CASE 0: MINIMUM BUILD .....	36
8.2	CASE 1: + RUNTIME SERVICE CREATION / STATIC MEMORY .....	37
8.3	CASE 2: + MULTIPLE TASKS AT SAME PRIORITY .....	38
8.4	CASE 3: + PRIORITY CHANGE / PRIORITY INHERITANCE / FCFS / TASK SUSPEND .....	39
8.5	CASE 4: + TIMER & TIMEOUT / TIMER CALL BACK / ROUND ROBIN .....	40
8.6	CASE 5: + EVENTS / MAILBOXES .....	41
8.7	CASE 6: FULL FEATURE BUILD (NO NAMES) .....	42
8.8	CASE 7: FULL FEATURE BUILD (NO NAMES / NO RUNTIME CREATION).....	43
8.9	CASE 8: FULL BUILD ADDING THE OPTIONAL TIMER SERVICES .....	44

## List of Figures

FIGURE 2-1 PROJECT FILE LIST .....	7
FIGURE 2-2 GUI SET OF OS_STACK_SIZE .....	8
FIGURE 2-3 GUI SET OF OS_ISR_STACK .....	9
FIGURE 2-4 GUI SET OF SATURATION BIT CONFIGURATION.....	10
FIGURE 2-5 GUI ENABLING OF THE FPU .....	11
FIGURE 2-6 GUI SET OF OS_FPU_ON_OFF .....	12
FIGURE 2-7 MULTITHREAD-SAFE PROJECT FILE LIST.....	14
FIGURE 2-8 FULL MULTITHREAD PROTECTION GUI CONFIGURATION .....	15
FIGURE 3-1 GUI SET OF OS_N_INTERRUPTS .....	18
FIGURE 3-2 GUI SET OF OS_N_INTERRUPTS .....	19
FIGURE 7-1 MEMORY MEASUREMENT CODE OPTIMIZATION SETTINGS .....	29
FIGURE 7-2 LATENCY MEASUREMENT CODE OPTIMIZATION SETTINGS.....	31

## List of Tables

TABLE 1-1 DISTRIBUTION .....	6
TABLE 2-1 OS_STACK_SIZE .....	7
TABLE 2-2 COMMAND LINE SET OF OS_STACK_SIZE.....	8
TABLE 2-3 OS_ISR_STACK.....	9
TABLE 2-4 COMMAND LINE SET OF OS_ISR_STACK.....	9
TABLE 2-5 SATURATION BIT CONFIGURATION .....	10
TABLE 2-6 COMMAND LINE SET OF SATURATION BIT CONFIGURATION.....	10
TABLE 2-7 COMMAND LINE ENABLING OF THE FPU.....	11
TABLE 2-8 FPU RUN TIME ON / OFF CONFIGURATION .....	12
TABLE 2-9 COMMAND LINE SET OF OS_FPU_ON_OFF.....	12
TABLE 2-10 FULL MULTITHREAD PROTECTION COMMAND LINE CONFIGURATION .....	14
TABLE 2-11 SETTING A TASK TO BE MULTITHREAD SAFE .....	15
TABLE 3-1 ABASSI_CORTEXM4_ISR.s INTERRUPT TABLE SIZING.....	17
TABLE 3-2 COMMAND LINE SET THE INTERRUPT TABLE SIZE.....	18
TABLE 3-3 OVERLOADING THE INTERRUPT TABLE SIZING FOR ABASSI.C .....	18
TABLE 3-4 ATTACHING A FUNCTION TO AN INTERRUPT.....	19
TABLE 3-5 INVALIDATING AN ISR HANDLER.....	20
TABLE 3-6 DISTRIBUTION INTERRUPT TABLE CODE.....	21
TABLE 3-7 STM32F407 USART 1 / 2 FAST INTERRUPTS .....	21
TABLE 3-8 FAST INTERRUPT WITH DEDICATED STACK .....	22
TABLE 3-9 REMOVING INTERRUPT NESTING.....	23
TABLE 3-10 PROPAGATING INTERRUPT NESTING.....	23
TABLE 4-1 CONTEXT SAVE STACK REQUIREMENTS.....	24
TABLE 5-1 SEARCH ALGORITHM CYCLE COUNT.....	26
TABLE 7-1 “C” CODE MEMORY USAGE .....	30
TABLE 7-2 ASSEMBLY CODE MEMORY USAGE.....	30
TABLE 7-3 MEASUREMENT WITHOUT TASK SWITCH.....	32
TABLE 7-4 MEASUREMENT WITHOUT BLOCKING.....	32
TABLE 7-5 MEASUREMENT WITH TASK SWITCH .....	32
TABLE 7-6 MEASUREMENT WITH TASK UNBLOCKING.....	33
TABLE 7-7 LATENCY MEASUREMENTS FPU OFF.....	34
TABLE 7-8 LATENCY MEASUREMENTS FPU ON.....	35
TABLE 8-1: CASE 0 BUILD OPTIONS .....	36
TABLE 8-2: CASE 1 BUILD OPTIONS .....	37
TABLE 8-3: CASE 2 BUILD OPTIONS .....	38
TABLE 8-4: CASE 3 BUILD OPTIONS .....	39
TABLE 8-5: CASE 4 BUILD OPTIONS .....	40
TABLE 8-6: CASE 5 BUILD OPTIONS.....	41
TABLE 8-7: CASE 6 BUILD OPTIONS.....	42
TABLE 8-8: CASE 7 BUILD OPTIONS.....	43
TABLE 8-9: CASE 8 BUILD OPTIONS.....	44

# 1 Introduction

This document details the port of the Abassi RTOS to the ARM Cortex-M4 processor. The software suite used for this specific port is the IAR Embedded Workbench for ARM, more commonly known as EWARM; the version used for the port and all tests is Version 6.30.11.2079.

## 1.1 Distribution Contents

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

**Table 1-1 Distribution**

File Name	Description
Abassi.h	Include file for the RTOS
Abassi.c	RTOS “C” source file
Abassi_CORTEXM4_IAR.s	RTOS assembly file for the ARM Cortex-M4 to use with the IAR Embedded Workbench
Abassi_IAR_MTX_IF.c	Abassi interface functions for multithread-safe operation of the IAR DLIB.
Demo_0_STM32_P407_IAR.c	Demo code for the Olimex STM32-P407 evaluation board
Demo_2_STM32_P407_IAR.c	Demo code for the Olimex STM32-P407 evaluation board
Demo_3_STM32_P407_IAR.c	Demo code for the Olimex STM32-P407 evaluation board
Demo_4_STM32_P407_IAR.c	Demo code for the Olimex STM32-P407 evaluation board
Demo_6_STM32_P407_IAR.c	Demo code for the Olimex STM32-P407 evaluation board
AbassiDemo.h	Build option settings for the demo code

## 1.2 Limitations

To optimize reaction time of the Abassi RTOS components, it was decided to require the processor to always operate in privileged mode (which is the default start-up mode for Cortex-M microcontrollers) and to always use the main stack pointer (MSP). The start-up code supplied in the distribution fulfills these constraints and one must be careful to not change these settings in the application.

The `SVCall` interrupt (interrupt number -5 / interrupt vector number 11) is not available as it is reserved for the OS, and the Abassi RTOS uses it.

## 2 Target Set-up

Very little is needed to configure the IAR Embedded Workbench development environment to use the Abassi RTOS in an application. All there is to do is to add the files `Abassi.c` and `Abassi_CORTEXM4_IAR.s` in the source files of the application project, and make sure the five configuration settings in the file `Abassi_CORTEXM4_IAR.s` (`OS_STACK_SIZE` described in Section 2.1, `OS_ISR_STACK` described in Section 2.2, `OS_HANDLE_PSR_Q` described in Section 2.3, `OS_FPU_ON_OFF` described in Section 2.4.2.3, and `OS_N_INTERRUPTS` described in Section 3.1.1) are set according to the needs of the application. As well, update the include file path in the C/C++ compiler preprocessor options with the location of `Abassi.h`. There is no need to include a start-up file, as the `Abassi_CORTEXM4_IAR.s` file contains all the start-up operations, including the interrupt table and default exception handlers.

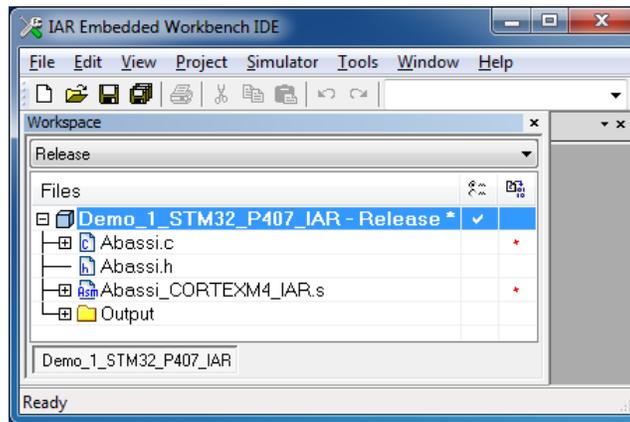


Figure 2-1 Project File List

### 2.1 OS\_STACK\_SIZE Set-up

The file `Abassi_CORTEXM4_IAR.s` contains the start-up code for “C” applications built with the IAR Embedded Workbench for the ARM that use the Abassi RTOS. There should be no other start-up file included in the project.

There is a definition used to set-up the stack size for the function `main()`, which is the highest priority task at start-up (known in Abassi as Adam&Eve). This definition is located at around line 30 in the `Abassi_CORTEXM4_IAR.s` file and is shown in the following table:

Table 2-1 OS\_STACK\_SIZE

```
#ifndef OS_STACK_SIZE
OS_STACK_SIZE EQU 1024 ; A&E (main) stack size in bytes / Set-up to your needs
#endif
```

A stack size of 1024 bytes is the value set in the distribution code; modify this value according to the needs of the application. If the value of `OS_STACK_SIZE` is set to 0, then the stack size reserved is the one specified in the linker configuration file.

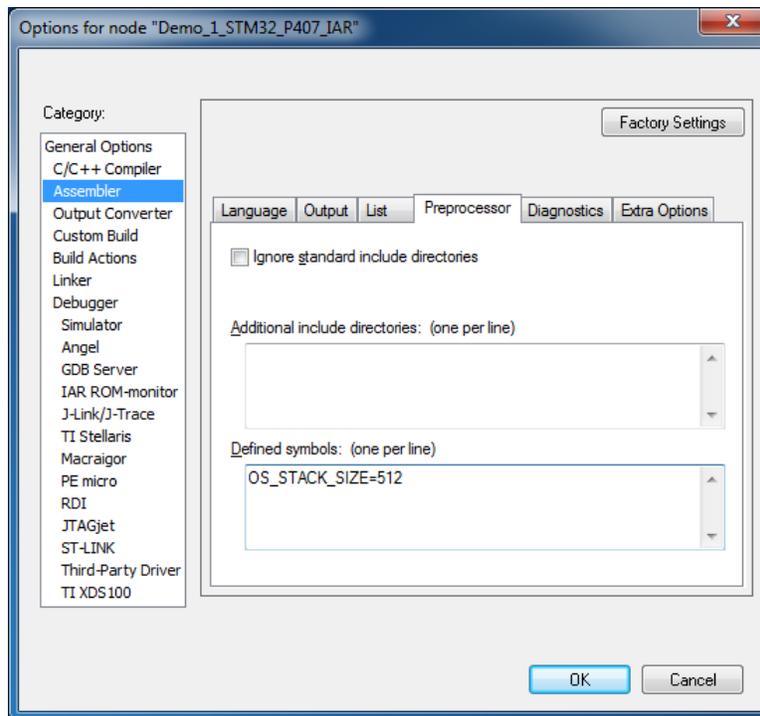
Alternatively, it is possible to overload the `OS_STACK_SIZE` value set in `Abassi_CORTEXM4_IAR.s` by using the assembler command line option `-D` and specifying the desired stack size as shown in the following example, where the stack size is set to 512 bytes:

**Table 2-2 Command line set of `OS_STACK_SIZE`**

```
iasmarm ... -DOS_STACK_SIZE=512 ...
```

When `OS_STACK_SIZE` is set to a value of zero, the stack for `main()` uses the specification from the linker configuration file.

The stack size can also be set through the GUI, in the “*Assembler / Preprocessor*” menu, as shown in the following figure:



**Figure 2-2 GUI set of `OS_STACK_SIZE`**

## 2.2 Interrupt Stack Set-up

It is possible, and is highly recommended, to use a hybrid stack when nested interrupts occur in an application. Using this hybrid stack, specially dedicated to the interrupts, removes the need to allocate extra room to the stack of every task in the application to handle the interrupt nesting. This feature is controlled by the value set by the definition `OS_ISR_STACK`, located around line 35 in the file `Abassi_CORTEXM4_IAR.s`. To disable this feature, set the definition of `OS_ISR_STACK` to a value of zero. To enable it, and specify the interrupt stack size, set the definition of `OS_ISR_STACK` to the desired size in bytes (see Section 4 for information on stack sizing). As supplied in the distribution, the hybrid stack feature is enabled and a stack size of 1024 bytes is allocated; this is shown in the following table:

**Table 2-3 OS\_ISR\_STACK**

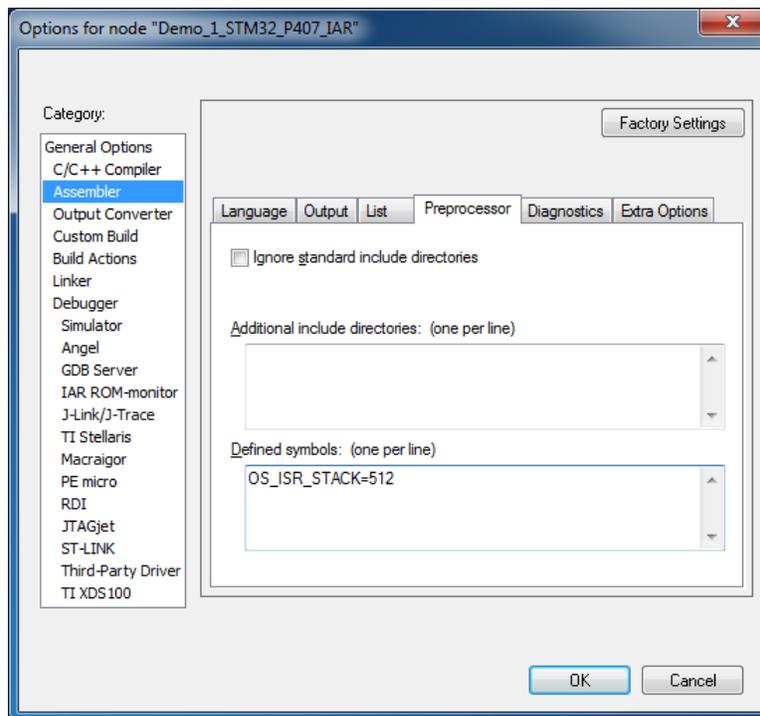
```
#ifndef OS_ISR_STACK
OS_ISR_STACK EQU 1024           ; If using a dedicated stack for the nested ISRs
#endif                          ; 0 if not used, otherwise size of stack in bytes
```

Alternatively, it is possible to overload the `OS_ISR_STACK` value set in `Abassi_CORTEXM4_IAR.s` by using the assembler command line option `-D` and specifying the desired hybrid stack size as shown in the following example, where the hybrid stack size is set to 512 bytes:

**Table 2-4 Command line set of OS\_ISR\_STACK**

```
iasmarm ... -DOS_ISR_STACK=512 ...
```

The hybrid stack size can also be set through the GUI, in the “Assembler / Preprocessor” menu, as shown in the following figure:



**Figure 2-3 GUI set of OS\_ISR\_STACK**

## 2.3 Saturation Bit Set-up

In the ARM Cortex-M4 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 it needs extra processing during a 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 in a single task, then it must be kept locally in each task. To keep the meaning of the saturation bit localized, the token `OS_HANDLE_PSR_Q` must be set to a non-zero value; to disable it, it must be set to a zero value. This is located at around line 40 in the file `Abassi_CORTEXM4_IAR.s`. The distribution code disables the localization of the Q bit, setting the token `OS_HANDLE_PSR_Q` to zero, as shown in the following table:

**Table 2-5 Saturation Bit configuration**

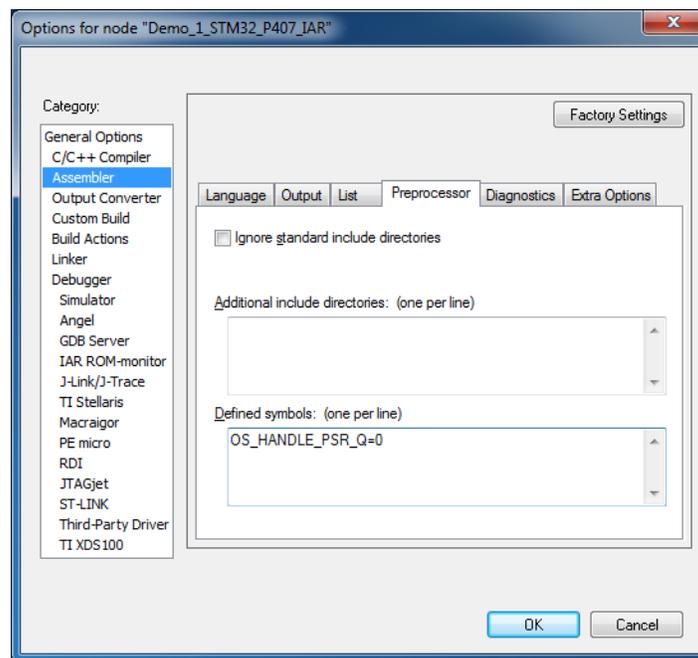
```
#ifndef OS_HANDLE_PSR_Q
OS_HANDLE_PSR_Q EQU 0           ; If we keep the Q bit (saturation) on per tasks
#endif
```

Alternatively, it is possible to overload the `OS_HANDLE_PSR_Q` value set in `Abassi_CORTEXM4_IAR.s` by using the assembler command line option `-D` and specifying the desired setting with the following:

**Table 2-6 Command line set of Saturation Bit configuration**

```
iasarm ... -DOS_HANDLE_PSR_Q=0 ...
```

The saturation bit configuration can also be set through the GUI, in the “Assembler / Preprocessor” menu, as shown in the following figure:



**Figure 2-4 GUI set of Saturation Bit configuration**

## 2.4 FPU set-up

The assembly file `Abassi_CORTEXM4_IAR.s`, depending on its configuration, handles three different types of FPU use. They are:

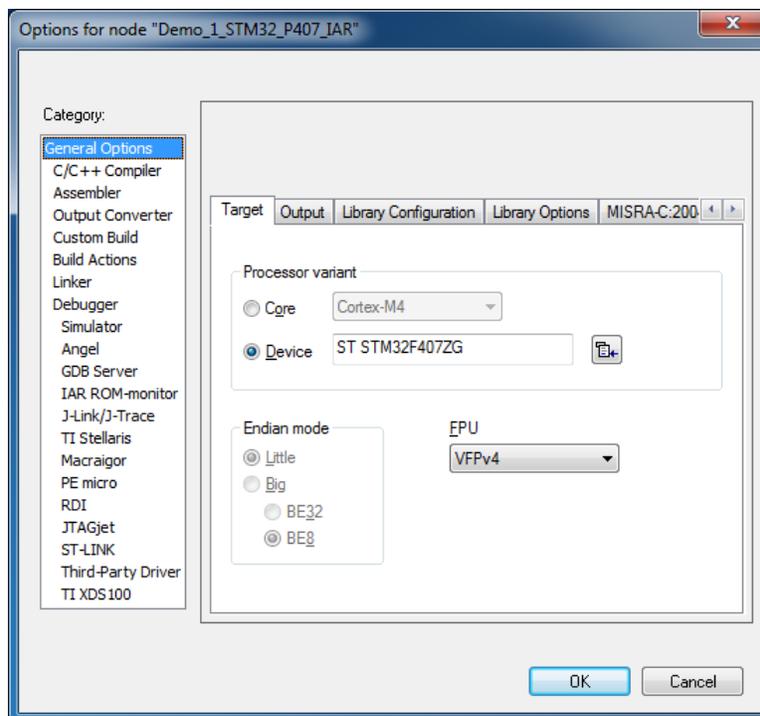
- The FPU is always disabled
- The FPU is always enabled
- The FPU is turned on and turned off during runtime

The file `Abassi_CORTEXM4_IAR.s` is aware of the enabling or disabling of the FPU by the compiler through the use of the IAR build definition `__ARMVFP__`, automatically defined when the assembler is configured to enable the FPU instructions. There are two ways to configure the assembler to support the FPU instruction. This is done on the command line with the option `--fpu`:

**Table 2-7 Command line enabling of the FPU**

```
iasarm ... --fpu=VFPv4 ...
```

The enabling of the FPU can also be performed through the GUI, in the *General Options / Target* menu, by setting the FPU to a value different than *None*.



**Figure 2-5 GUI enabling of the FPU**

When the FPU is enabled, each task can use a different configuration of the FPU (through the `FPCSR` register), as the contents of this register are part of the task context save. All tasks upon start will have their local `FPCSR` value set according to the value of `FPCSR` register upon calling `OSstart()`. This means if the application globally requires a different setting of the FPU than the default set by the compiler, the `FPCSR` must be modified before calling `OSstart()`.

It is also possible to turn on and turn off the FPU during runtime, and the ON / OFF setting is also kept on a per task basis. This means the FPU can be enabled in a set of tasks when it is not for the other tasks in the application. All tasks, upon start, will inherit the same ON / OFF state of the FPU as when `OSstart()` was called. When this feature is required, the build option `OS_FPU_ON_OFF` definition, located around line 50 in the file `Abassi_CORTEXM4_IAR.s`, must be set to a non-zero value. The distribution code does not enable the capability of turning the FPU ON and OFF during runtime, setting the token `OS_FPU_ON_OFF` to zero, as shown in the following table:

**Table 2-8 FPU run time ON / OFF configuration**

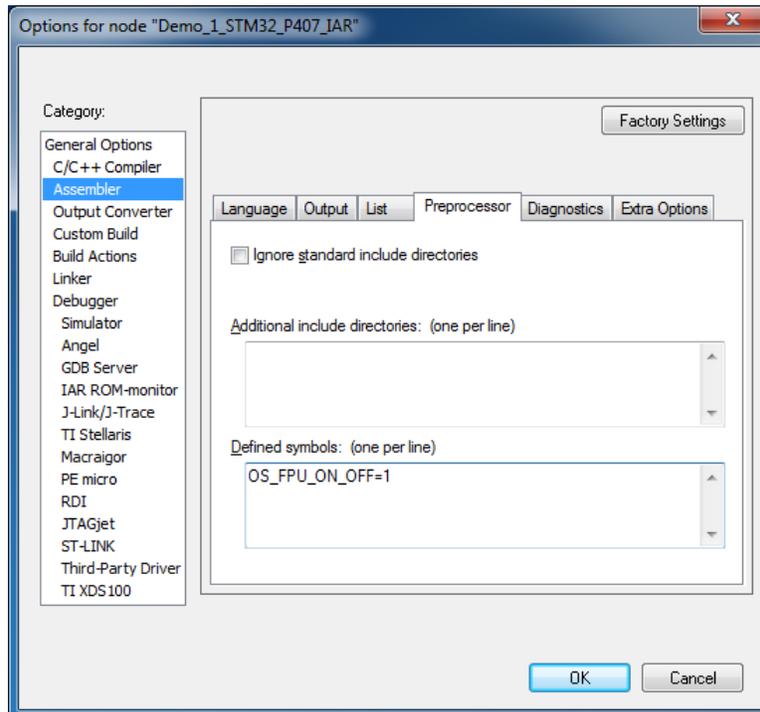
```
#ifndef OS_FPU_ON_OFF
OS_FPU_ON_OFF EQU 0 ; If the FPU can be turned ON/OFF during runtime
#endif
```

Alternatively, it is possible to overload the `OS_FPU_ON_OFF` value set in `Abassi_CORTEXM4_IAR.s` by using the assembler command line option `-D` and specifying the desired setting with the following:

**Table 2-9 Command line set of OS\_FPU\_ON\_OFF**

```
iasmarm ... -DOS_FPU_ON_OFF=1 ...
```

The indication the FPU is turned on and off during runtime can also be set through the GUI, in the “Assembler / Preprocessor” menu, as shown in the following figure:



**Figure 2-6 GUI set of OS\_FPU\_ON\_OFF**

There are two requirements to fulfill when the FPU is turned on and off during runtime. The first one, which is not related to the RTOS, but is a restriction by the Cortex-M4 core, is to never have a different enable setting of the FPU between the entry and the exit of an ISR. This means that turning ON and then OFF the FPU in an interrupt is safe. But turning it ON without turning it OFF before exiting the interrupt will crash the application. If the FPU is ON upon entry in the interrupt, and it gets turned OFF in the interrupt without being turned back ON, will trigger an access fault exception. If the FPU is turned ON and OFF in an interrupt, be aware the above stated condition may not be respected with nesting of interrupts.

The second requirement when the FPU is turned ON and OFF during runtime is that it is necessary to set the `SVCall` (Service call exception vector #11, interrupt #-5) priority to the highest level. This is configured in the System Handler Priority Register 2 (SHPR2) register. If this register is not modified, then at start-up the priority of the `SVCall` exception is set to the higher level.

**NOTE:** When the FPU is turned OFF in a task, the setting of the FPCSR will quite likely to be set back to the task start-up value upon turning ON the FPU afterward.

## 2.5 Multithreading

By default, the IAR DLIB runtime library is not multithread safe. There are two aspects to take into account when protecting the library for multithreading. The first one involves reentrance; some library functions are not reentrant, therefore two tasks accessing the same non-reentrant function at the same time can create major issues. The classic example of non-reentrant functions are the family of functions for dynamic memory allocation: e.g. `malloc()` and `free()`. As they internally use a static buffer, a few pointers, and some linked lists, if two tasks use functions that access the internals of the dynamic memory allocation at the same time, corruption could occur. Protecting the non-reentrant functions is straightforward: all there is to do is to make sure there is only a single task that can access the non-reentrant functions at any time. This is done with a mutex, as it is the perfect mechanism to guarantee exclusive access to a resource.

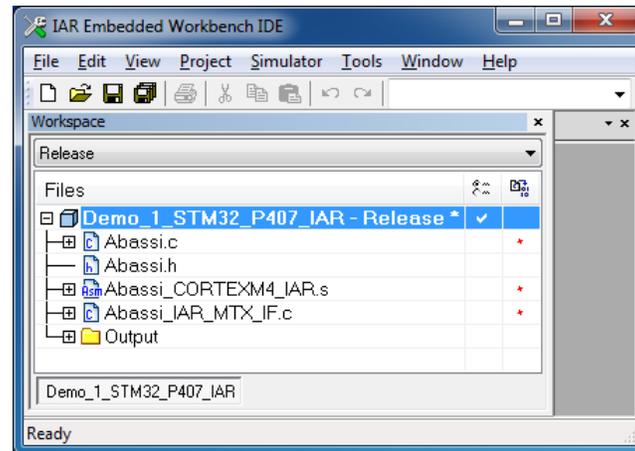
The second type of functions and variables that are not multithread safe are due to internal data used by the library, data that is truly a global resource. Examples of these are: the `errno` variable or the `locale` information; these are called TLS (Thread Local Storage) by IAR. The only efficient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

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

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

### 2.5.1 Reentrance Protection

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



**Figure 2-7 Multithread-safe Project File List**

## 2.5.2 Full Multithreading Protection

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

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

**Table 2-10 Full Multithread Protection Command Line Configuration**

```
iccarm ... -DOS_IAR_MTHREAD=1 ...
```

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

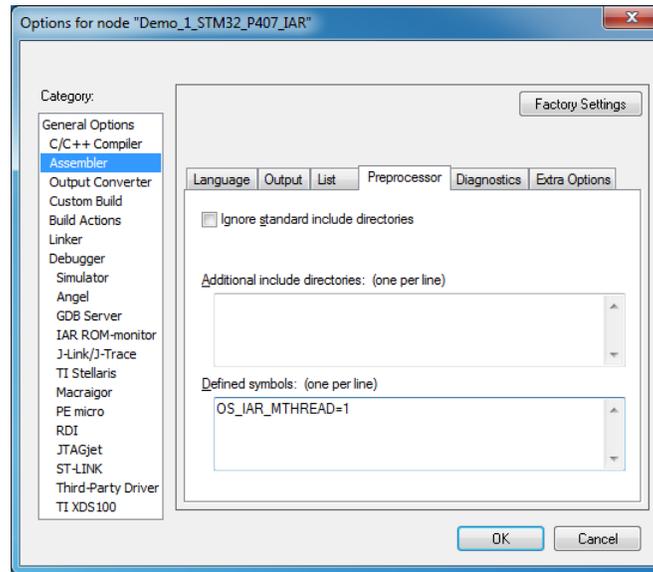


Figure 2-8 Full Multithread Protection GUI Configuration

### 2.5.3 Partial Multithreading Protection

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

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

Table 2-11 Setting a task to be multithread safe

```
#include "Abassi.h"

TSK_t *TskReent;
void _DLIB_TLS_MEMORY *Mthread;

...
/* First the task must be created
/* in the suspended state
TskReent = TSKcreate("TaskName", TskPrio, StackSize, TaskFct, 0);
/* Get memory for the TLS
Mthread = OSalloc(__IAR_DLIB_PERTHREAD_SIZE);
/* Initialize the TLS
__iar_dlib_perthread_initialize((void *) Mthread);
TskReent->XtraData[0] = (intptr_t)Mthread; /* Attach the TLS to the task
TSKresem(TskReent); /* The task may now be resumed
```

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



## 3 Interrupts

The Abassi RTOS needs to be aware when kernel requests are performed inside or outside an interrupt context. For all interrupt sources (except interrupt numbers less than -1) the Abassi RTOS provides an interrupt dispatcher, which allows it to be interrupt-aware. This dispatcher achieves two goals. First, the kernel uses it to know if a request occurs within an interrupt context or not. Second, using this dispatcher reduces the code size, as all interrupts share the same code for the decision making of entering the kernel or not at the end of the interrupt: there is no need to add a preamble / epilogue in the functions handling the interrupts.

The distribution makes provision for 241 sources of interrupts (plus the basic 15 exceptions), as specified by the token `OS_N_INTERRUPTS` in the file `Abassi_CORTEXM4_IAR.s`, and the internal default value used by `Abassi.c`. Even though the Nested Vectored Interrupt Controller (NVIC) peripheral supports a maximum of 512 interrupts on the Cortex-M4, the first 15 entries of the interrupt vector table are hard mapped to dedicated handlers (the interrupt number -1, which is attached to `SystemTick`, is not hard mapped but is handled by the ISR dispatcher). The default value of 241 interrupts is only half of the maximum number of supported exceptions.

### 3.1 Interrupt Handling

#### 3.1.1 Interrupt Table Size

Most devices do not require all 512 interrupts as they typically only handle between 64 and 128 sources of interrupts. The interrupt table can be easily reduced to recover code space, and at the same time recover the same amount of data memory, or expanded if more than 256 interrupts are supported by the target device. There are two files affected: in `Abassi_CortexM4_IAR.s`, the ARM interrupt table itself must be modified, and the value used in the file `Abassi.c`, in order to reduce the ISR dispatcher table look-up. The interrupt table size is defined by the token `OS_N_INTERRUPTS` in the file `Abassi_CortexM4_IAR.s` around line 35. For the value used by `Abassi.c`, the default value can be overloaded by defining the token `OS_N_INTERRUPTS` when compiling `Abassi.c`. The distribution table size is set to 241; that is the NVIC maximum of 256 minus the 15 hard mapped exceptions.

For example, the STM32F407 device from ST Microelectronics uses only the first 100 entries of the interrupt table (84 external interrupts plus the standard 16 exceptions). The 256 entries table can therefore be reduced to 100. The value to set in `Abassi_CortexM4_ISR.s` files is 85, which is the total of 100 entries minus 15 (there are 15 hard mapped exceptions). The changes are shown in the following table:

**Table 3-1 Abassi\_CortexM4\_ISR.s interrupt table sizing**

```

...
#ifdef OS_N_INTERRUPTS
OS_N_INTERRUPTS EQU 85
#endif
; # of entries in the interrupt table mapped to
; ISRdispatch(). Must match the definition in Abassi.h
...

```

Alternatively, it is possible to overload the `OS_N_INTERRUPTS` value set in `Abassi_CORTEXM4_IAR.s` by using the assembler command line option `-D` and specifying the desired setting with the following:

**Table 3-2 Command line set the interrupt table size**

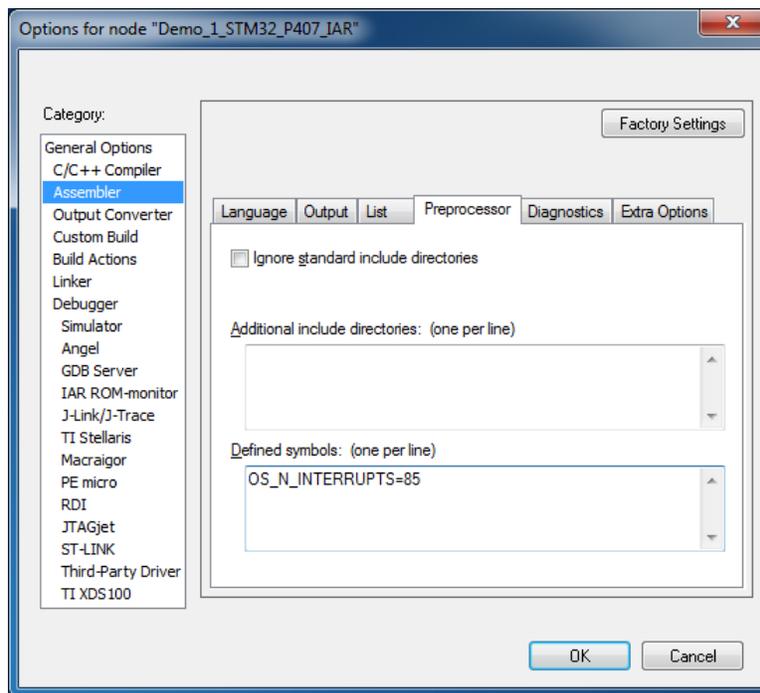
```
iasarm ... -DOS_N_INTERRUPTS=85 ...
```

The overloading of the default interrupt vector look-up table used by `Abassi.c` is done by using the compiler command line option `-D` and specifying the desired setting with the following:

**Table 3-3 Overloading the interrupt table sizing for Abassi.c**

```
iccarm ... -DOS_N_INTERRUPTS=85 ...
```

The interrupt table size used by `Abassi_CORTEXM4_IAR.s` can also be set through the GUI, in the “*Assembler / Preprocessor*” menu, as shown in the following figure:



**Figure 3-1 GUI set of `OS_N_INTERRUPTS`**

The interrupt table look-up size used by `Abassi.c` can also be overloaded through the GUI, in the “C/C++ Compiler / Preprocessor” menu, as shown in the following figure:

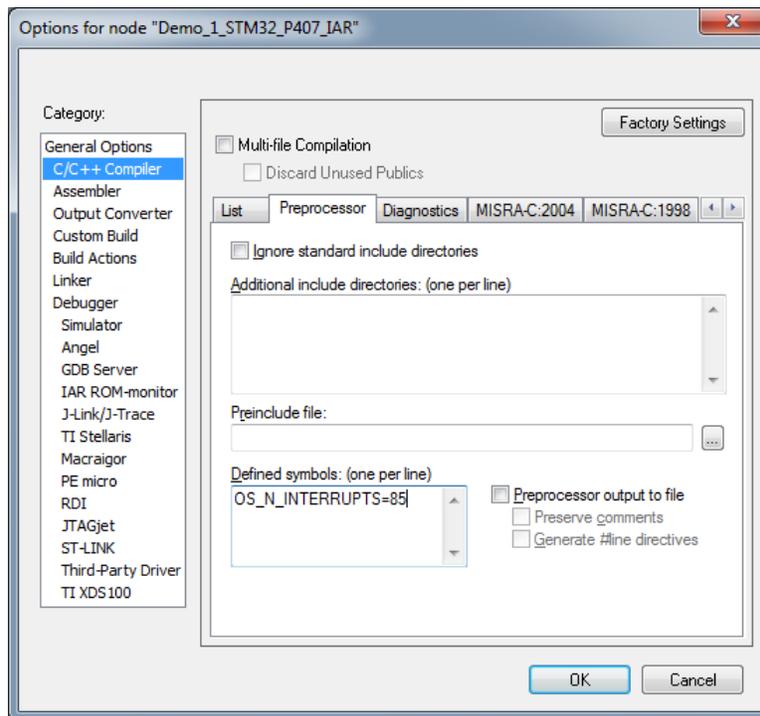


Figure 3-2 GUI set of `OS_N_INTERRUPTS`

### 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 3-4 shows the code required to attach the `SysTick` interrupt to the RTOS timer tick handler (`TIMtick`):

Table 3-4 Attaching a Function to an Interrupt

```
#include "Abassi.h"

...
OSstart();
...
OSIsrInstall(-1, &TIMtick);
/* Set-up the count reload and enable SysTick interrupt */

... /* More ISR setup */

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

**NOTE:** `OSIsrInstall()` uses the interrupt number, NOT the interrupt vector number.

At start-up, once `OSstart()` has been called, all `OS_N_INTERRUPTS` 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 ran. When an interrupt handler is removed, it is very important and necessary to first disable the interrupt source, then the handling function can be set back to `OSinvalidISR()`. This is shown in Table 3-5:

**Table 3-5 Invalidating an ISR handler**

```
#include "Abassi.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 Nested Vectored Interrupt Controller (NVIC) on the Cortex-M4 does not clear the interrupt generated by a peripheral; neither does the RTOS. If the generated interrupt is a pulse (as for the `SystemTick` interrupt), there is nothing to do to clear the interrupt request. However, if the generated interrupt is a level interrupt, the peripheral generating the interrupt must be informed to remove the interrupt request. This operation must be performed in the interrupt handler otherwise the interrupt will be re-entered over and over.

## 3.2 Interrupt Priority and Enabling

To properly configure interrupts, the interrupt priority must be set, and the peripheral configured to generate interrupts and enable them. There is no software provided to perform these operations, as this functionality is already available. First, the IAR Embedded Workbench supports the Cortex Microcontroller Software Interface Standard (CMSIS), which provides everything required to program the processor peripherals. Second, most chip manufacturers provide code to configure the specifics on their devices.

## 3.3 Fast Interrupts

Fast interrupts are supported on this port. A fast interrupt is an interrupt that never uses any component from Abassi, and as the name says, is desired to operate as fast as possible. To set-up a fast interrupt, all there is to do is to set the address of the interrupt function in the corresponding entry in the interrupt vector table used by the Cortex-M4 processor. The area of the interrupt vector table to modify is located in the file `Abassi_CORTEXM4_IAR.s` around line 100.

For example, on a ST Microelectronics STM32F407 device, USART #1 is attached to interrupt number 37 (interrupt vector number 53) and the USART #2 is attached to the interrupt number 38 (interrupt vector number 54). The code to modify is located in the macro loop that initializes the interrupt table that sets the ISR dispatcher as the default interrupt handler. All there is to do is add checks on the token holding the interrupt number, such that, when the interrupt number value matches the desired interrupt number, the appropriate address gets inserted in the table instead of the address of `ISRdispatch()`. The original macro loop code and modified one are shown in the following two tables:

**Table 3-6 Distribution interrupt table code**

```

INT_NMB SET -1
  REPT OS_N_INTERRUPTS ; Map all the external interrupts to ISRdispatch()
      DC32 ISRdispatch
INT_NMB SET INT_NMB+1
  ENDR

```

Attaching a fast interrupt handler to the USART #1 and another one to USART#2, assuming the names of the interrupt functions to attach are respectively `USART1_IRQhandler()` and `USART2_IRQhandler()`, is shown in Table 3-7:

**Table 3-7 STM32F407 USART 1 / 2 Fast Interrupts**

```

EXTERN USART1_IRQhandler
EXTERN USART2_IRQhandler

...

INT_NMB SET -1
  REPT OS_N_INTERRUPTS ; Map all the external interrupts to ISRdispatch()
      IF INT_NMB == 37 ; When is interrupt # 37, set the USART #1 handler
          DC32 USART1_IRQhandler
      ELSEIF INT_NMB == 38 ; When is interrupt # 38, set the USART #2 handler
          DC32 USART2_IRQhandler
      ELSE ; All others interrupt # mapped to ISRdispatch()
          DC32 ISRdispatch
      ENDIF
INT_NMB SET INT_NMB+1
  ENDR

...

```

It is important to add the `EXTERN` statement, otherwise there will be an error during the assembly of the file.

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

**NOTE:** A fast interrupt does not use Abassi interrupt dispatcher, therefore there is no need to use the `ISRinstall()` component for a fast interrupt.

Even if the hybrid interrupt stack feature is enabled (see Section 2.2), fast interrupts will not use that stack. This translates into the need to reserve room on all task stacks for the possible nesting of fast interrupts. To make the fast interrupts also use a hybrid interrupt stack, a prologue and epilogue must be used around the call to the interrupt handler. The prologue and epilogue code to add is almost identical to what is done in the regular interrupt dispatcher. Reusing the example of the USART #1 on the STM32F407 device, this would look something like:

**Table 3-8 Fast Interrupt with Dedicated Stack**

```

...

ELSEIF INT_NMB == 37
    DC32    USART1_preHandler        ; Set the address of the pre handler
                                           ; in the interrupt table

...
...

RSEG      .text:CODE
ALIGNROM 2
THUMB

EXTERN   USART1_IRQhandler

USART1_preHandler:
    cpsid  I                        ; Disable ISR to protect against nesting
    mov   r0, sp                    ; Memo current stack pointer
    ldr   sp, =USART1_stack         ; Stack dedicated to this fast interrupt
    cpsie I                          ; The stack is now hybrid, nesting safe
    push {r0, lr}                  ; Preserve original sp & EXC_RETURN

    bl    USART1_IRQhandler        ; Enter the interrupt handler

    pop   {r0, lr}                 ; Recover original sp & EXC_RETURN
    mov   sp, r0                   ; Recover pre-isr stack
    bx   lr                        ; Exit from the interrupt

...
...

RSEG      .noinit:DATA(3)

DS8      USART1_stack_size        ; Room for the fast interrupt stack
USART1_stack:

...

```

The same code, with unique labels, must be repeated for each of the fast interrupts.

### 3.4 Nested Interrupts

The interrupt controller allows nesting of interrupts; this means an interrupt of higher priority will interrupt the processing of an interrupt of lower priority. Individual interrupt sources can be set to one of 8 levels, where level 0 is the highest and 7 is the lowest. This implies that the RTOS build option `OS_NESTED_INTS` must be set to a non-zero value. The exception to this is an application where all enabled interrupts handled by the RTOS ISR dispatcher are set, without exception, to the same priority; then interrupt nesting will not occur. In that case, and only that case, can the build option `OS_NESTED_INTS` be set to zero. As this latter case is quite unlikely, the build option `OS_NESTED_INTS` is always overloaded when compiling the RTOS for the ARM Cortex-M4. If the latter condition is guaranteed, the overloading located after the pre-processor directive can be modified. The code affected in `Abassi.h` is shown in Table 3-9 below and the line to modify is the one with `#define OX_NESTED_INTS 1`:

**Table 3-9 Removing interrupt nesting**

```
#elif defined(__ICCARM__) && defined(__ARM_PROFILE_M__)
...
#define OX_NESTED_INTS 0 /* The ARM has 8 nested (NIVC) interrupt levels*/
```

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

**Table 3-10 Propagating interrupt nesting**

```
#elif defined(__ICCARM__) && defined(__ARM_PROFILE_M__)
...
#define OX_NESTED_INTS OS_NESTED_INTS
```

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

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

## 4 Stack Usage

The RTOS uses the tasks' stack for two purposes. When a task is blocked or ready to run but not running, the stack holds the register context that was preserved when the task got blocked or preempted. Also, when an interrupt occurs, the register context of the running task must be preserved in order for the operations performed during the interrupt to not corrupt the contents of the registers used by the task when it got interrupted. For the Cortex-M4, the context save contents of a blocked or pre-empted task is different from the one used in an interrupt, and is also different if the compiler is set to use the FPU or not. 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 (FPU OFF)	40 bytes
Interrupt dispatcher context save ( <code>OS_ISR_STACK == 0</code> ) (FPU OFF)	40 bytes
Interrupt dispatcher context save ( <code>OS_ISR_STACK != 0</code> ) (FPU OFF)	48 bytes
Blocked/Preempted task context save (FPU ON)	112 bytes
Interrupt dispatcher context save ( <code>OS_ISR_STACK == 0</code> ) (FPU ON)	120 bytes
Interrupt dispatcher context save ( <code>OS_ISR_STACK != 0</code> ) (FPU ON)	128 bytes

The numbers for the interrupt dispatcher context save include the 32 bytes (FPU OFF) or the 96 bytes (FPU ON) the processor pushes on the stack when it enters the interrupt servicing.

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

**NOTE:** The ARM Cortex-M4 processor needs alignment on 8 bytes for some instructions accessing memory. When stack memory is allocated, Abassi guarantees the alignment. This said, when sizing `OS_STATIC_STACK` or `OS_ALLOC_SIZE`, make sure to take in account that all allocation performed through these memory pools are always by block size multiple of 8 bytes.

If the hybrid interrupt stack (see Section 2.2) is enabled, then the above description changes: it is only necessary to reserve room on task stacks for a single interrupt context save (this excludes the interrupt function handler stack requirements) and not the worst-case nesting. With the hybrid stack enabled, the second, third, and so on interrupts use the stack dedicated to the interrupts. The hybrid stack is enabled when the `OS_ISR_STACK` token in the file `Abassi_CORTEXM4_IAR.s` is set to a non-zero value (see Section 2.2).

## 5 Search Set-up

The Abassi RTOS build option `OS_SEARCH_FAST` offers three different algorithms to quickly determine the next running task upon task blocking. The following table shows the measurements obtained for the number of CPU cycles required when a task at priority 0 is blocked, and the next running task is at the specified priority. The number of cycles includes everything, not just the search cycle count. The number of cycles was measured using the `SysTick` peripheral, which decrements the counter once every CPU cycle. The second column is when `OS_SEARCH_FAST` is set to zero, meaning a simple array traversing. The third column, labeled Look-up, is when `OS_SEARCH_FAST` is set to 1, which uses an 8 bit look-up table. Finally, the last column is when `OS_SEARCH_FAST` is set to 5 (IAR/Cortex-M4 `int` are 32 bits, so  $2^5$ ), meaning a 32 bit look-up table, further searched through successive approximation. The compiler optimization for these measurements was set to *Level High / Speed optimization*; the FPU is enabled for these measurements. The RTOS build options were set to the minimum feature set, except for option `OS_PRIO_CHANGE` set to non-zero. The presence of this extra feature provokes a small mismatch between the result for a difference of priority of 1, with `OS_SEARCH_FAST` set to zero, and the latency results in Section 7.2.

When the build option `OS_SEARCH_ALGO` is set to a negative value, indicating to use a 2-dimensional linked list search technique instead of the search array, the number of CPU cycles is constant at 235 cycles.

**Table 5-1 Search Algorithm Cycle Count**

Priority	Linear search	Look-up	Approximation
1	242	266	307
2	244	272	307
3	250	278	307
4	256	284	307
5	262	290	307
6	268	296	307
7	274	302	307
8	280	274	307
9	286	276	307
10	292	282	307
11	298	288	307
12	304	294	307
13	310	300	307
14	316	306	307
15	322	312	307
16	328	284	307
17	334	286	307
18	340	292	307
19	346	298	307
20	352	304	307
21	358	310	307
22	364	316	307
23	370	322	307
24	376	294	307

When `OS_SEARCH_FAST` is set to 0, each extra priority level to traverse requires exactly 6 CPU cycles. When `OS_SEARCH_FAST` is set to 1, each extra priority level to traverse requires exactly 6 CPU cycles, except when the priority level is an exact multiple of 8; then there is a sharp reduction of CPU usage. Overall, setting `OS_SEARCH_FAST` to 1 adds 28 cycles of CPU for the search compared to setting `OS_SEARCH_FAST` to zero. But when the next ready to run priority is less than 8, 16, 24, ... then there is an extra 10 cycles needed, but without the 8 times 6 cycle accumulation. Finally, the third option, when `OS_SEARCH_FAST` is set to 5, delivers a perfectly constant CPU usage, as the algorithm utilizes a successive approximation search technique (when the delta is 32 or more, the CPU cycle count is 315, for 64 or more, it is 323).

The first observation, when looking at this table, is that the first option, when `OS_SEARCH_FAST` is set to 0, is the most CPU efficient when the priority span is less than 8. For more than 8 priority spans, the second option (when `OS_SEARCH_FAST` is set to 1) is overall more CPU efficient than the third option (when `OS_SEARCH_FAST` is set to 5) for a span of around 20 priorities. So, the build option `OS_SEARCH_FAST` should never be set to 5, as it is not the most efficient method, unless the application has way more than 20 priority levels.

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

## 6 Chip Support

No custom chip support is provided with the distribution code because the IAR Embedded Workbench for the ARM supports the Cortex Microcontroller Software Interface Standard (CMSIS). Therefore, all peripherals on the Cortex-M3 can be accessed and configured through the CMSIS. Also, most device manufacturers provide code to configure the peripherals on their devices. The distribution code contains some of the manufacturer's open source libraries, e.g. STM.

## 7 Measurements

This section gives an overview of the memory requirements and the CPU latency encountered when the RTOS is used on the ARM Cortex-M4 and compiled with IAR Embedded Workbench. The CPU cycles are exactly the CPU clock cycles, as the processor typically executes one instruction at every clock transition.

### 7.1 Memory

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

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

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

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

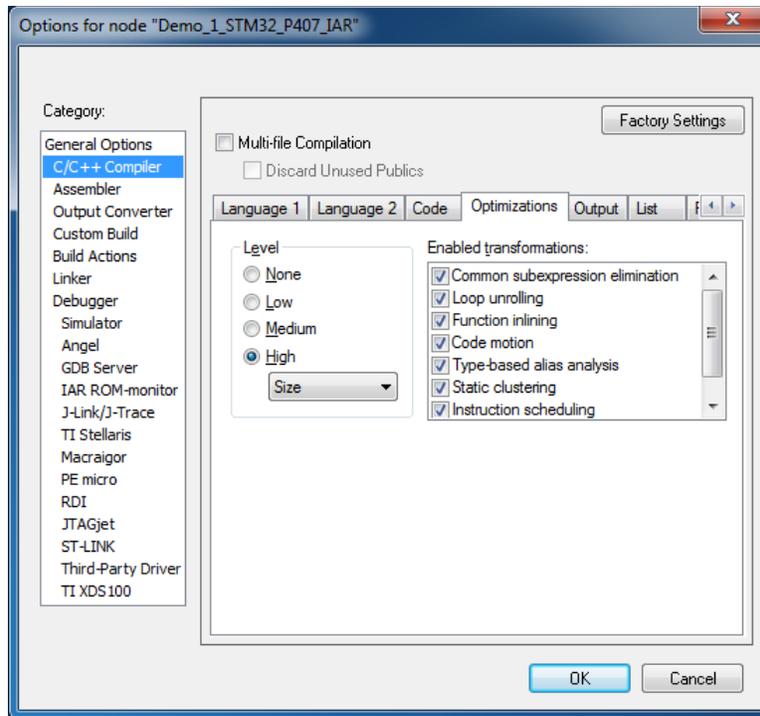


Figure 7-1 Memory Measurement Code Optimization Settings

**Table 7-1 “C” Code Memory Usage**

Description	Code Size
Minimal Build	< 650 bytes
+ Runtime service creation / static memory	< 850 bytes
+ Multiple tasks at same priority	< 900 bytes
+ Runtime priority change + Mutex priority inheritance + FCFS + Task suspension	< 1400 bytes
+ Timer & timeout + Timer call back + Round robin	< 1800 bytes
+ Events + Mailbox	< 2400 bytes
Full Feature Build (no names)	< 2900 bytes
Full Feature Build (no names / no runtime creation)	< 2600 bytes
Full Feature Build (no names / no runtime creation) + Timer services module	< 2950 bytes

**Table 7-2 Assembly Code Memory Usage**

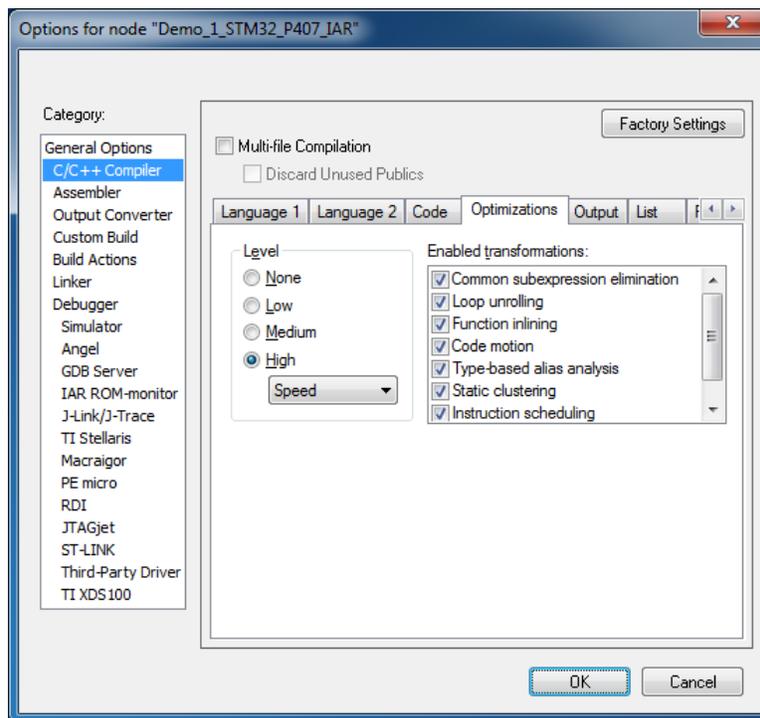
Description	Size
Assembly code size (FPU OFF)	172 bytes
Assembly code size (FPU ON)	244 bytes
Exception Handler (per handler)	+8 bytes
Vector table (per interrupt handler entry)	+4 bytes
Hybrid Stack Enabled	+16 bytes
Saturation Bit Enabled	+20 bytes
FPU runtime ON / OFF	+192 bytes

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 Code Time Technologies website.

## 7.2 Latency

Latency of operations has been measured on a Olimex STM32-P407 Evaluation board populated with a 168 MHz STM32F407 device. The clock setting for the measurement used the internal oscillator operating at 16 MHz, which allows running from the flash with 0 wait states. All measurements have been performed on the real platform, as the IAR Workbench simulator does not support the simulation of the `SysTick` timer as this is the counter used to count the cycles. 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 settings that were used for the latency measurements are:

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



**Figure 7-2 Latency Measurement Code Optimization Settings**

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

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

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

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

**Table 7-3 Measurement without Task Switch**

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

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

**Table 7-4 Measurement without Blocking**

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

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

**Table 7-5 Measurement with Task Switch**

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

TaskPriol()
{
    ...
    Start CPU cycle count
    SEMpost(...); or EVTset(...); or MBXput(...);
    ...
}
```

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

**Table 7-6 Measurement with Task unblocking**

```

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

TaskPriol()
{
    ...
    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 7-7 lists the results obtained, where the cycle count is measured using the `SysTick` peripheral on the Cortex-M4. This timer decrements its counter by 1 at every CPU cycle. As was the case for the memory measurements, these numbers were obtained using the release version 1.122.205 of the RTOS and may change in other versions.

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. But for this measurement, the STM32F407 SysTick Timer is used to trigger the interrupt and measure the elapsed time. The latency measurement includes the cycles required to acknowledge the interrupt.

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

The hybrid interrupt stack feature was not enabled, neither was the saturation bit, in any of these tests. When the FPU is on, the runtime FPU ON / OFF feature of Abassi is not enabled.

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

**Table 7-7 Latency Measurements FPU OFF**

<b>Description</b>	<b>Minimal Features</b>	<b>Full Features</b>
Semaphore posting no task switch	115 (112)	160 (153)
Semaphore waiting no blocking	118 (113)	168 (160)
Semaphore posting with task switch	177 (196)	279 (290)
Semaphore waiting with blocking	191 (184)	306 (294)
Semaphore posting in ISR with task switch	342 (359)	448 (461)
Event setting no task switch	n/a	156 (149)
Event getting no blocking	n/a	179 (171)
Event setting with task switch	n/a	293 (303)
Event getting with blocking	n/a	321 (309)
Event setting in ISR with task switch	n/a	462 (474)
Mailbox writing no task switch	n/a	199 (190)
Mailbox reading no blocking	n/a	209 (201)
Mailbox writing with task switch	n/a	339 (348)
Mailbox reading with blocking	n/a	343 (331)
Mailbox writing in ISR with task switch	n/a	508 (520)
Interrupt Latency	25	25
Interrupt overhead entering the kernel	165 (163)	169 (171)
Interrupt overhead NOT entering the kernel	46	46
Context switch	38	40

**Table 7-8 Latency Measurements FPU ON**

<b>Description</b>	<b>Minimal Features</b>	<b>Full Features</b>
Semaphore posting no task switch	115 (112)	160 (153)
Semaphore waiting no blocking	118 (113)	168 (160)
Semaphore posting with task switch	219 (238)	321 (332)
Semaphore waiting with blocking	233 (226)	348 (336)
Semaphore posting in ISR with task switch	414 (431)	520 (533)
Event setting no task switch	n/a	156 (149)
Event getting no blocking	n/a	179 (171)
Event setting with task switch	n/a	335 (345)
Event getting with blocking	n/a	363 (351)
Event setting in ISR with task switch	n/a	534 (546)
Mailbox writing no task switch	n/a	199 (190)
Mailbox reading no blocking	n/a	209 (201)
Mailbox writing with task switch	n/a	381 (390)
Mailbox reading with blocking	n/a	385 (373)
Mailbox writing in ISR with task switch	n/a	580 (592)
Interrupt Latency	49	49
Interrupt overhead entering the kernel	195 (193)	199 (201)
Interrupt overhead NOT entering the kernel	70	70
Context switch	82	82

## 8 Appendix A: Build Options for Code Size

### 8.1 Case 0: Minimum build

Table 8-1: Case 0 build options

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

## 8.2 Case 1: + Runtime service creation / static memory

**Table 8-2: Case 1 build options**

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

### 8.3 Case 2: + Multiple tasks at same priority

Table 8-3: Case 2 build options

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

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

Table 8-4: Case 3 build options

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

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

Table 8-5: Case 4 build options

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

## 8.6 Case 5: + Events / Mailboxes

**Table 8-6: Case 5 build options**

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

## 8.7 Case 6: Full feature Build (no names)

**Table 8-7: Case 6 build options**

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

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

Table 8-8: Case 7 build options

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

## 8.9 Case 8: Full build adding the optional timer services

Table 8-9: Case 8 build options

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