CODE TIME TECHNOLOGIES

Abassi RTOS

Porting Document

ARM Cortex-M0 – Keil Suite

`

Disclaimer

Table of Contents

List of Figures

List of Tables

1 Introduction

This document details the port of the Abassi RTOS to the ARM Cortex-M0 processor. The software suite used for this specific port is the MDK-ARM Microcontroller Development Kit, more commonly known as Keil μVision4; the version used for the port and all tests is V4.50.0.0.

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_CORTEXM0_KEIL.s`	RTOS assembly file for the ARM Cortex-M0 to use with the MDK-ARM
`Demo_2_BB_LPC11U24_KEIL.c`	Demo code that runs on the NGX LPC11U24 evaluation board
`Demo_3_BB_LPC11U24_KEIL.c`	Demo code that runs on the NGX LPC11U24 evaluation board
`Demo_6_BB_LPC11U24_KEIL.c`	Demo code that runs on the NGX LPC11U24 evaluation board
`AbassiDemo.h`	Build option settings for the demo code

1.2 Limitations

To optimize the reaction time of the Abassi RTOS components, it was decided to require the processor to always operate in privileged mode (which is the default 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 Keil µVision4 development environment to use the Abassi RTOS in an application. All there is to do is to add the files `Abassi.c` and `Abassi_CORTEXM0_KEIL.s` in the source files of the application project, and make sure the four configuration settings in the file `Abassi_CORTEXM0_KEIL.s` (`OS_HEAP_SIZE` and `OS_STACK_SIZE` as described in Section 2.1, `OS_ISR_STACK` as described in Section 2.2, and `OS_N_INTERRUPTS` as 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`.

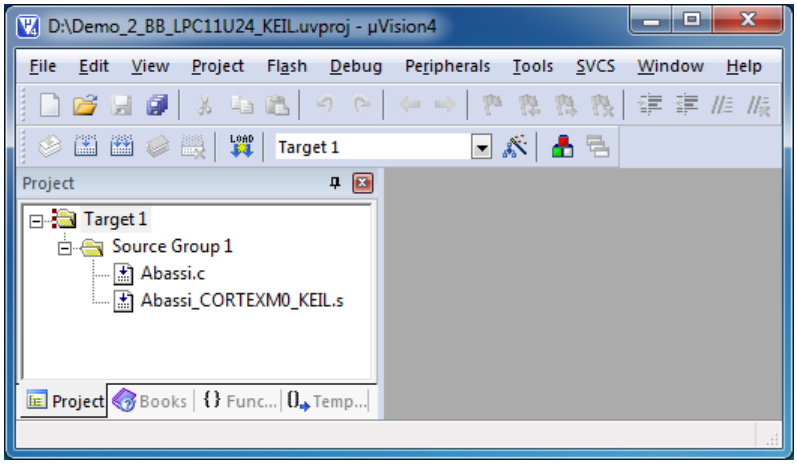

Figure 2-1 Project File List

2.1 OS_HEAP_SIZE and OS_STACK_SIZE Set-up

The file `Abassi_CORTEXM0_KEIL.s` contains the start-up code for "C" applications built with the Keil µVision development suite that use the Abassi RTOS. There should be no other start-up file included in the project.

There are two definitions that are used to set-up the heap size (memory used by `malloc()`) and the stack size for the function `main()`, which is the highest priority task at start-up (known in Abassi as Adam&Eve). These definitions are located at around line 30 in the `Abassi_CORTEXM0_KEIL.s` file and are shown in the following table:

Table 2-1 `OS_STACK_SIZE` and `OS_HEAP_SIZE`

```
  IF (:DEF: OS_HEAP_SIZE) == {FALSE}
OS_HEAP_SIZE  EQU 4096          ; Heap size (malloc()) in bytes / Set-up to your needs
  ENDIF


  IF (:DEF: OS_STACK_SIZE) == {FALSE}
OS_STACK_SIZE EQU 1024          ; A&E stack size in bytes / Set-up to your needs
  ENDIF
```

A heap size of 4096 bytes and a stack size of 1024 bytes are the values set in the distribution code; modify these values according to the needs of the application.

Alternatively, it is possible to overload the values of `OS_HEAP_SIZE` and `OS_STACK_SIZE` set in `Abassi_CORTEXM0_KEIL.s` by using the assembler command line option `-predefine` and specifying the desired heap size and stack size as shown in the following example, where the heap size is set to 2048 bytes, and the stack size is set to 512 bytes:

Table 2-2 Command line set of Heap and Stack sizes

```
armasm … -predefine "OS_HEAP_SIZE SETA 2048" -predefine "OS_STACK_SIZE SETA 512" …
```

The heap and stack sizes can also be set through the GUI, in the "*Asm*" menu, as shown in the following figure:

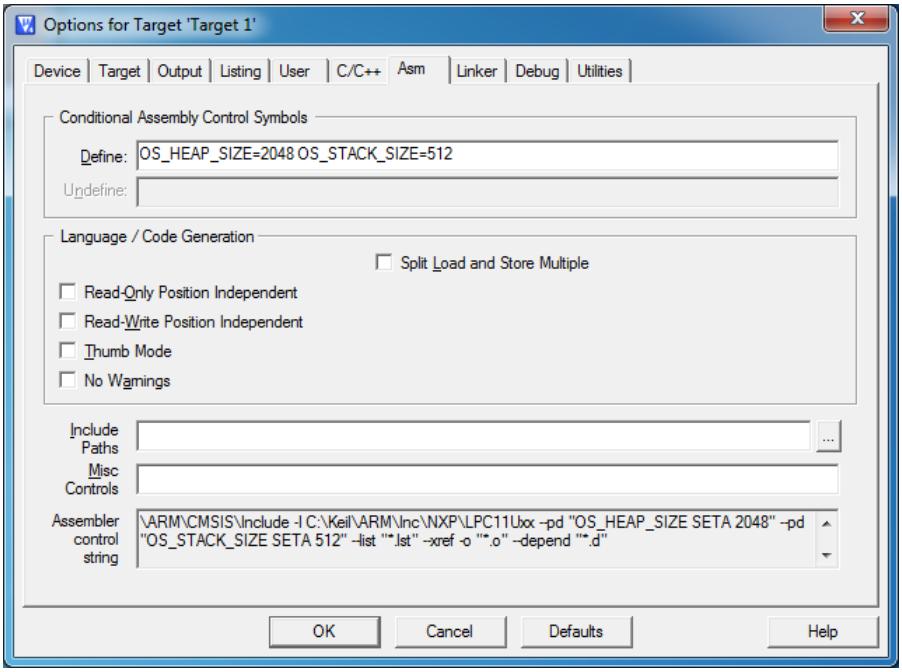

Figure 2-2 GUI set of Heap and Stack sizes

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_CORTEXM0_KEIL.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`

```
  IF (:DEF: OS_ISR_STACK) == {FALSE}
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 values of `OS_ISR_STACK` set in `Abassi_CORTEXM0_KEIL.s` by using the assembler command line option `-predefine` 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`

```
armasm … -predefine "OS_ISR_STACK SETA 512" …
```

The hybrid stack size can also be set through the GUI, in the "*Asm*" menu, as shown in the following figure:

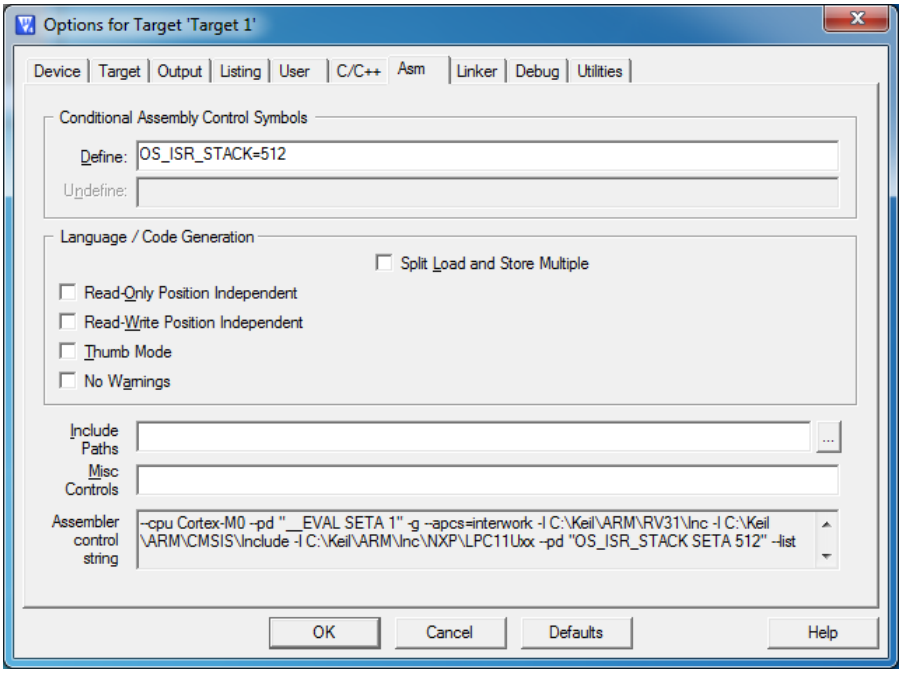

Figure 2-3 GUI set of `OS_ISR_STACK`

2.3 Multithreading protection

By default, the Keil "C" 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; a few library functions are not reentrant, therefore two tasks accessing the same function at the same time can create major issues. A good example of non-reentrant functions are the dynamic memory allocation, `malloc()` and `free()`. As they internally use a static buffer, a few pointers and some linked lists, if two tasks 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 function 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 function and variables that are not multithread safe are due to internal data used by the library; data that is truly a global resource. Such examples of these are: the `errno` variable or the `locale` information. The only efficient way to protected these functions and variables against multithreading is to have the library setup to use a per task internal static data. There are multiple ways to implement the data swapping, but fundamentally, if the library does not provided such a swapping mechanism, it becomes cumbersome to solve the issue. It would require manually swapping the contents, by copying the individual internal static variables of the library at every task switch.

Keil's standard library fully supports mechanisms to make the library multithread safe. The MicroLIB does not have such a mechanism. The following sub-sections describe how to make each of the two libraries multithread safe.

2.3.1 Standard Library Multithreading Protection

The Keil standard library (not the MicroLIB, see Section 2.3.2 for the MicroLIB) can be set to be completely protected against reentrance and also be multithread-safe. The type of multithreading protection is selected according to the definition of the build option `OS_KEIL_REENT`; this is not a standard build option, as it only is used with the Keil development suite on ARM processors. If this build option is not defined, or if it is defined with a value of zero, the library is neither protected against reentrance nor multithreading. If the build option is positive, the library is fully multithread-safe and protected against reentrance for every task in the application. If the build option value is negative, only user-selected tasks that are configured access the library in a multithread-safe fashion; the library still remains protected against reentrance for all tasks.

2.3.1.1 Full Protection

For full multithreading protection of the standard library, all there is to do is to define the build option `OS_KEIL_REENT` with a positive value. The build option `OS_KEIL_REENT` for the multithreading protection must be given to the compiler. This can be done with the command line option `-D` and specifying the setting with the following:

Table 2-5 Command line set of multithread configuration

```
armcc … -DOS_KEIL_REENT=1 …
```

The multithreading configuration can also be set through the GUI, in the "*C/C++*" menu, as shown in the following figure:

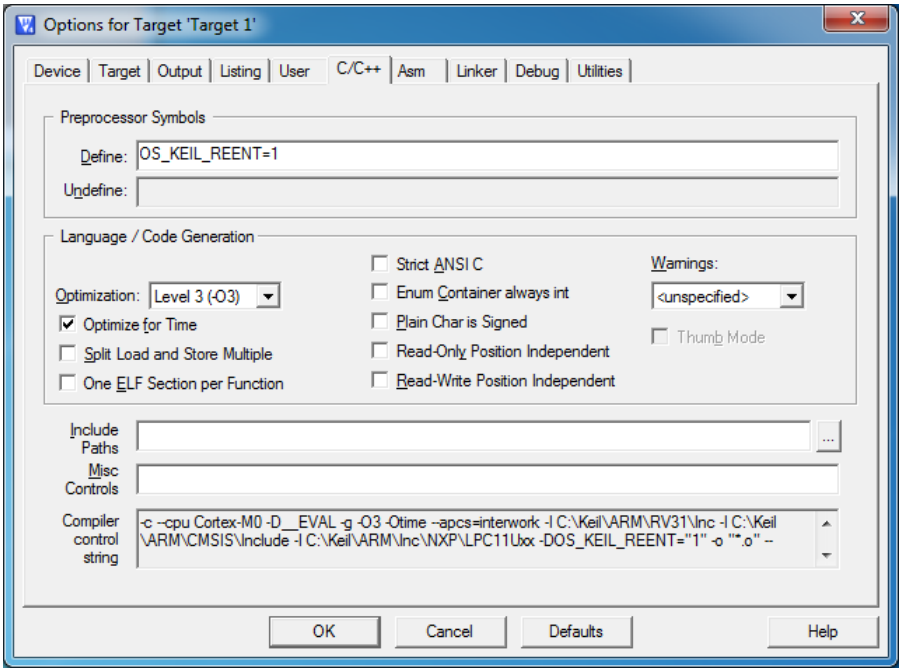

Figure 2-4 GUI set of `OS_KEIL_REENT`

2.3.1.2 Partial Protection

The use of full multithread protection for the library requires 96 bytes of extra data memory for each task in the application. The extra memory required is not due to Abassi, but it is the amount of memory the library requires to hold all its internal static data. It may not be desirable to use multithread protection for all tasks, or on data memory restricted applications it may be impossible to use full multithreading protection. Setting the build option `OS_KEIL_REENT` to a negative value allows the designer to select the tasks where multithreading protection is required. The library modules that are non-reentrant are still protected by a mutex; only the static area of the library becomes under control. The build option `OS_KEIL_REENT` is set the same way as described in the previous section, only it must be set to a negative value for partial protection.

Partial multithreading means that only the tasks that are set up to use the library in a multithread safe manner will require the 96 bytes block of extra data memory. Not only is memory needed for the library internal data, but if file I/O is used in the task, more memory is also needed for the buffering of the file or stream. It may be good practice to use the standard library function `setbuf()`, or `setvbuf()` to tailor each stream buffer size.

If a task uses none of the library multithread unsafe static data, then the task does not need to access the library internal data in an exclusive manner, so there is no need to reserve and assign the memory block of 96 bytes of data memory. If a task uses the library multithread unsafe static data, but it is the only task using that data, there is still no need to make the library multithread safe for that task. Only when two or more tasks use the same internal data of the library do these tasks need to access the library in a multithread safe manner.

For more information on which library functions and/or variables are non-reentrant and/or multithread unsafe, refer to Section 2.3.2.

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

Table 2-6 Setting a task to use re-entrant library

```
#include "Abassi.h"

TSK_t *TskReent
int ReentData[96/sizeof(int)];

…
                                                /* First the task must be created    */
                                                /* in the suspended state            */
TskReent = TSKcreate("TaskName", TskPrio, StackSize, TaskFct, 0);

memset(&ReentData[0], sizeof(ReentData), 0);    /* Buffer must be set to zero        */

TskReent->XtraData[0] = (intptr_t)&ReentData;   /* Attach the libspace to the task   */

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

The declaration "`int ReentData[96/sizeof(int)];`" can be replaced by a dynamic memory allocation of `(size_t)96`. If a task does not require access to the library in a multithread safe manner, the above code is not required.

2.3.2 MicroLIB Multithreading Protection

Contrary to the standard library, the MicroLIB does not offer internal support for multithreading protection. Some functions in the Keil C MicroLIB runtime library are not reentrant. If these functions are only used in one task, then there will be no problems. But if they are used by more than one task, they need to be protected by an Abassi mutex. The preferred way is to re-use the `G_OSmutex` for all multithread unsafe functions, as this will avoid deadlocks. Therefore, non-reentrant functions must be manually protected with a mutex.

For the multithread unsafe functions and/or variables, there is no simple way to make these functions or variables multithread safe.

Figure 2-5 shows the page in the µVision help that describes all the functions that are multithread unsafe. When there is mention of `_mutex_*`, it means the function is not reentrant and must be protected by a mutex. When there is mention of `__user_libspace` or `__user_perthread_libspace`, it indicates the function, or variable, is not multithread safe, as it relies on static data.

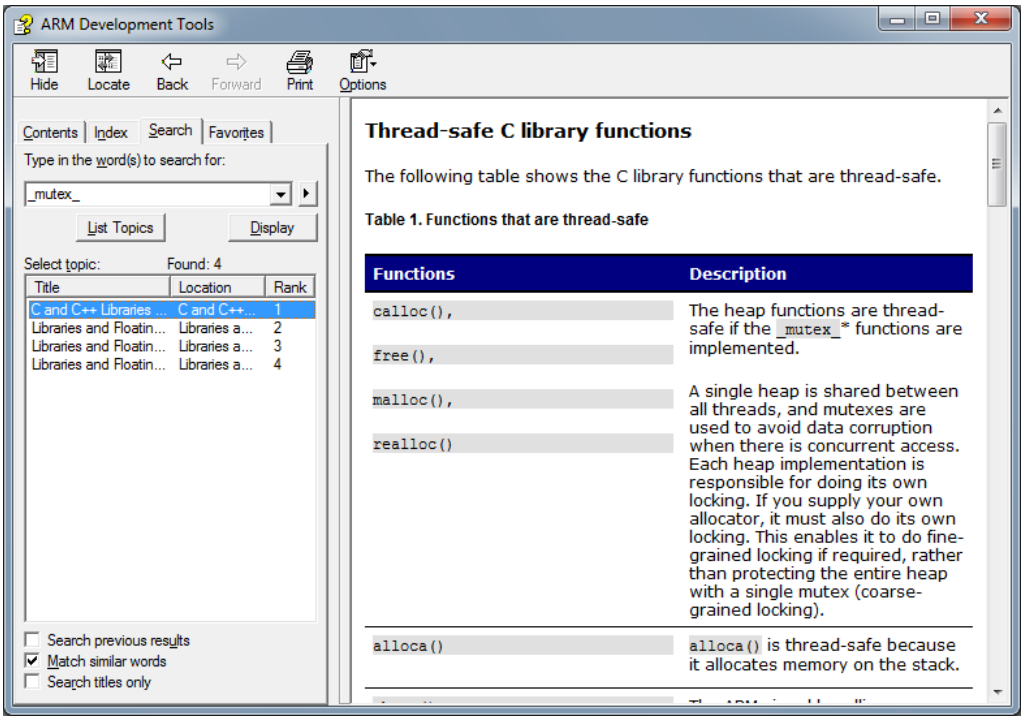

Figure 2-5 C Library Help

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, as specified by the token `OS_N_INTERRUPTS` in the file `Abassi_CORTEXM0_KEIL`.s, and the internal default value used by `Abassi.c`. Even though the Nested Vectored Interrupt Controller (NVIC) peripheral supports a maximum of 256 interrupts on the Cortex-M0, the first 15 entries of the interrupt vector table are hard mapped to dedicated handlers (the interrupt number -1, which is attached to `SysTick`, is not hard mapped but is handled by the ISR dispatcher).

3.1 Interrupt Handling

3.1.1 Interrupt Table Size

Most devices do not require all 256 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. There are two files affected: in `Abassi_CORTEXM0_KEIL.s`, the ARM interrupt table itself must be shrunk, 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_CORTEXM0_KEIL.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 LPC11U24 device from NXP uses only the first 48 entries of the interrupt table (32 external interrupts plus the standard 16 exceptions). The 256 entries table can therefore be reduced to 48. The value to set in `Abassi_CORTEXM0_KEIL.s files` is 33, which is the total of 48 entries minus 15 (there are 15 hard mapped exceptions). The changes are shown in the following table:

Table 3-1 `Abassi_CORTEXM0_KEIL.s` interrupt table sizing

```
  …

  IF (:DEF: OS_N_INTERRUPTS) == {FALSE} ; # of entries in the interupt table mapped to
OS_N_INTERUPTS    EQU 33                ; ISRdispatch()
  ENDIF

   …
```

Alternatively, it is possible to overload the `OS_N_INTERRUPTS` value set in `Abassi_CORTEXM0_KEIL.s` by using the assembler command line option `-predefine` and specifying the desired setting with the following:

Table 3-2 Command line set the interrupt table size

```
armasm … -predefine "OS_N_INTERRUPTS SETA 33"  …
```

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`

```
armcc … -DOS_N_INTERRUPTS=33 …
```

The interrupt table size used by `Abassi_CORTEXM0_KEIL.s` can also be set through the GUI, in the "*Asm*" menu, as shown in the following figure:

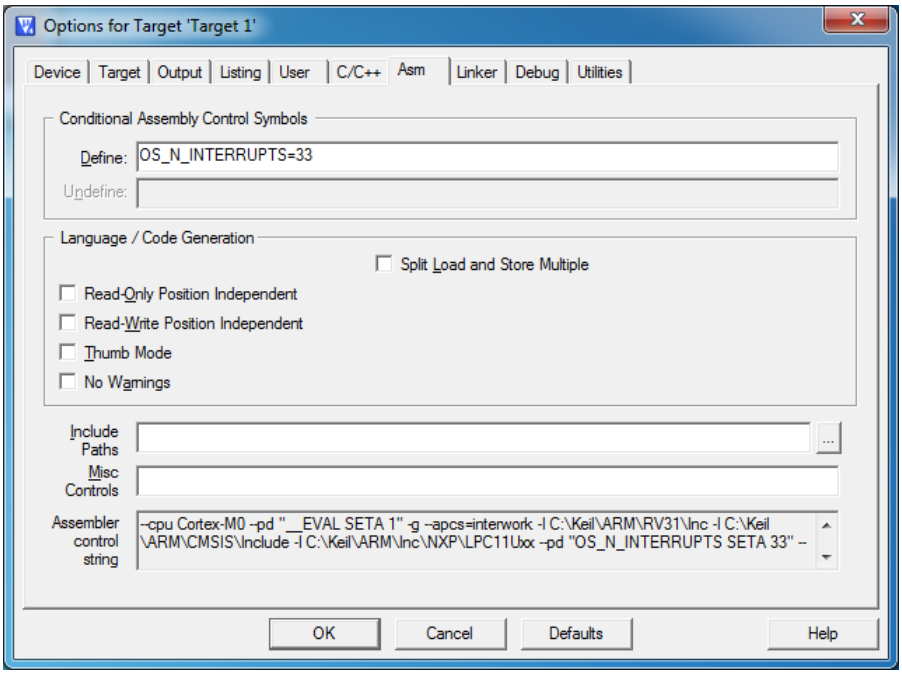

Figure 3-1 GUI set of the interrupt table size

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

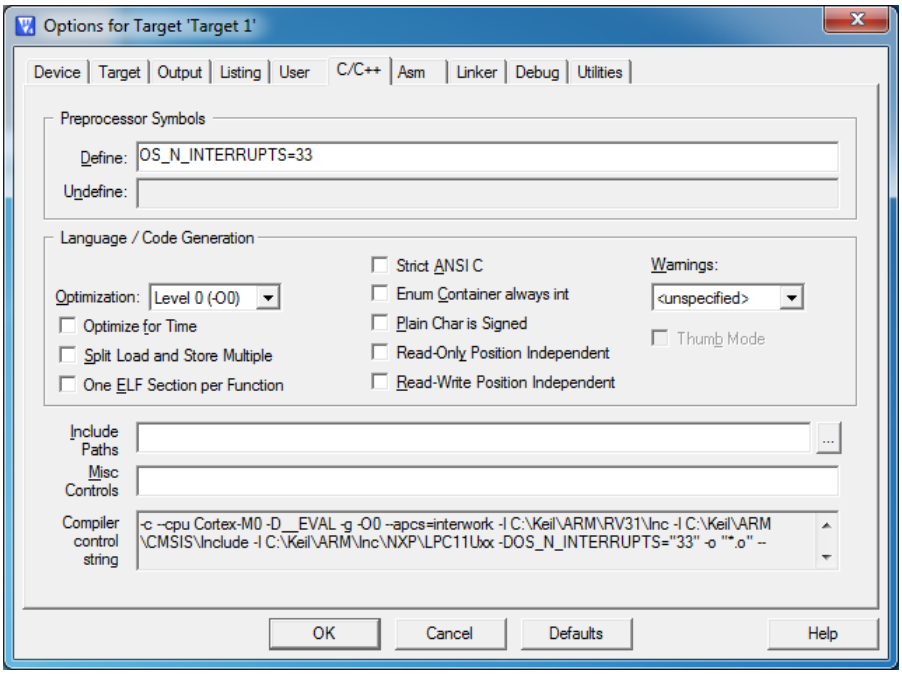

Figure 3-2 GUI set of the interrupt table size

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-M0 does not clear the interrupt generated by a peripheral; neither does the RTOS. If the generated interrupt is a pulse (as for the `SysTick` 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, Keil µVision4 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-M0 processor. The area of the interrupt vector table to modify is located in the file `Abassi_CORTEXM0_KEIL.s` around line 90. For example, on a Texas Instruments LPC11U24 device, TIMER16 #0 is attached to interrupt number 16 (interrupt vector number 32) and the TIMER16 #1 is attached to the interrupt number 17 (interrupt vector number 33). 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

```
   GBLA  INT_NMB                        ; Interrupt number in the loop
INT_NMB  SETA   -1                      ; Can't use < as < is unsigned
   WHILE  INT_NMB != (OS_N_INTERRUPTS-1); Map all external interrupts to ISRdispatch()
      DCD    ISRdispatch
INT_NMB   SETA  INT_NMB+1
   WEND
```

Attaching a fast interrupt handler to the UART #0 and another one to UART #1, assuming the names of the interrupt functions to attach are respectively `UART0_IRQhandler()` and `UART1_IRQhandler()`, is shown in the following table:

Table 3-7 LPC11U24 UART 0 / 1 Fast Interrupts

```
    EXTERN  TIMER16_0_IRQhandler
    EXTERN  TIMER16_1_IRQhandler

    …

   GBLA  INT_NMB                        ; Interrupt number in the loop
INT_NMB  SETA   -1                      ; Can't use < as < is unsigned
   WHILE  INT_NMB != (OS_N_INTERRUPTS-1); Map all external interrupts to ISRdispatch()
     IF INT_NMB == 16                   ; When is interrupt #16, set TIMER16 #0 handler
        DCD    TIMER16_0_IRQhandler
     ELSEIF INT_NMB == 17               ; When is interrupt #16, set TIMER16 #1 handler
        DCD    TIMER16_1_IRQhandler
     ELSE                               ; All others interrupt # set to ISRdispatch()
        DCD    ISRdispatch
     ENDIF
INT_NMB   SETA  INT_NMB+1
   WEND

    …
```

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.

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 TIMER #0 on the LPC11U24 device, this would look something like:

Table 3-8 Fast Interrupt with Dedicated Stack

```
    …

    ELSEIF INT_NMB == 16
       DCD     Timer16_0_preHandler         ; Set the address of the pre handler
                                            ; in the interrupt table
    …
    …

    THUMB
    ALIGN
    AREA    |.text|, CODE, READONLY

    EXTERN  TIMER16_0_handler

Timer16_0_preHandler

    cpsid   I                               ; Disable ISR to protect against nesting
    mov     r0, sp                          ; Memo current stack pointer
    ldr     sp, =TIMER16_0_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      TIMER16_0_handler               ; 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

    …

    ALIGN
    AREA    HEAP, NOINIT, READWRITE, ALIGN=3

    SPACE   TIMER16_0_stack_size            ; Room for the fast interrupt stack
TIMER16_0_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-M0. 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(__CC_ARM)
   …
  #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(__CC_ARM)
   …
  #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 processor 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-M0, 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	`40 bytes`
Interrupt dispatcher context save (`OS_ISR_STACK == 0`)	`40 bytes`
Interrupt dispatcher context save (`OS_ISR_STACK != 0`)	`48 bytes`

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

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

NOTE: The ARM Cortex-M0 processor needs alignment on 8 bytes for some instructions accessing memory. When stack memory is allocated, Abassi guarantees the alignment. This said, when sizing `OS_STATIC_STACK` or `OS_ALLOC_SIZE`, make sure to take in account that all allocation performed through these memory pools are by block size multiple of 8 bytes.

If the hybrid interrupt stack (see Section 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_CORTEXM0_KEIL.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 (Keil/Cortex-M0 `int` are 32 bits, so 2^5), meaning a 32 bit look-up table, further searched through successive approximation. The compiler optimization for this measurement was set to Level 3 (`-O3`), optimized for time. The 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 268 cycles.

Table 5-1 Search Algorithm Cycle Count

Priority	Linear search	Look-up	Approximation
1	277	312	353
2	281	320	353
3	289	328	353
4	297	336	353
5	305	344	353
6	313	352	353
7	321	360	353
8	329	317	353
9	337	321	353
10	345	329	353
11	353	337	353
12	361	345	353
13	369	353	353
14	377	361	353
15	385	369	353
16	393	326	353
17	401	330	353
18	409	338	353
19	417	346	353
20	425	354	353
21	433	362	353
22	441	370	353
23	449	378	353
24	457	335	353

When `OS_SEARCH_FAST` is set to 0, each extra priority level to traverse requires exactly 8 CPU cycles. When `OS_SEARCH_FAST` is set to 1, each extra priority level to traverse requires exactly 8 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 39 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 8 cycles needed, but without the 8 times 8 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 363, for 64 or more, it is 373).

The only real observation, when looking at this table, is that the third option, when OS_SEARCH_FAST is set to 1, is almost all the time either less CPU efficient than the first option, the one when OS_SEARCH_FAST is set to 0, or less efficient than the third option OS_SEARCH_FAST is set to 5. So, the build option OS_SEARCH_FAST should never be set to 1, as it is the least efficient method. The other observation is that the first option (OS_SEARCH_FAST set to 0) delivers better CPU performance than the third option (OS_SEARCH_FAST set to 5) when the search spans less than 10 priority levels. So, if an application has tasks spanning less than 10 priority levels, the build option OS_SEARCH_FAST should be set to 0; for all other cases, the build option OS_SEARCH_FAST should be set to 5.

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

6 Chip Support

No custom chip support is provided with the distribution code because the Keil µVision suite supports the Cortex Microcontroller Software Interface Standard (CMSIS). Therefore, all standard peripherals on the Cortex-M can be accessed 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 NXP.

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-M0 and compiled with Keil's µVision4. The CPU cycles are exactly the CPU clock cycles, as the processor executes one instruction at every clock transition.

7.1 Memory

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

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

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

1. Optimization: Level 3 (`-O3`)
2. Optimize for Time: Disabled
3. Split Load and Store Multiple: Disabled

All other options are disabled, as they do not affect the code generated.

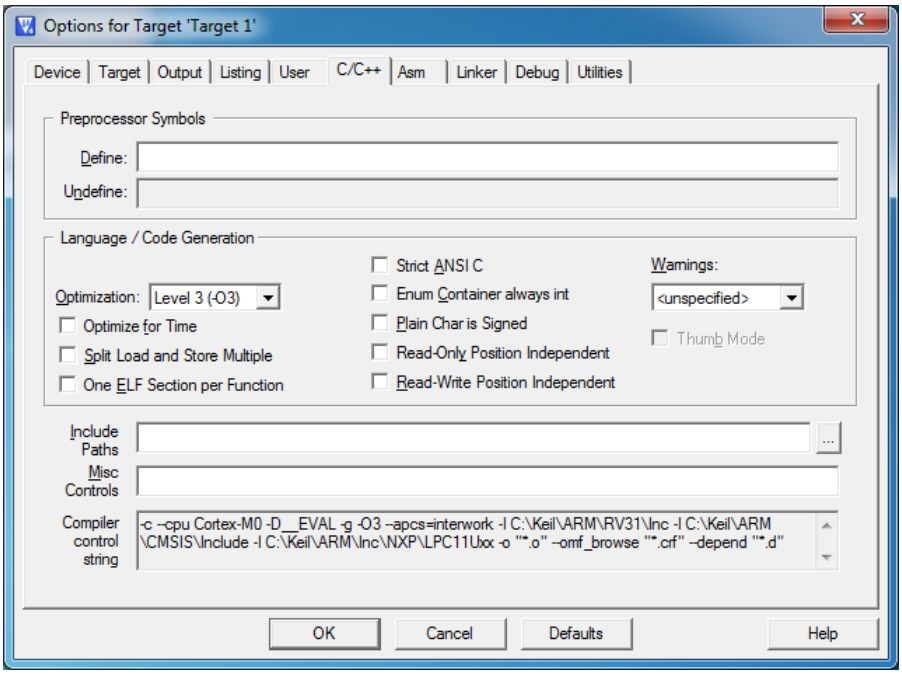

Figure 7-1 Memory Measurement Code Optimization Settings

Table 7-1 "C" Code Memory Usage

Description	Code Size
Minimal Build	< 725 bytes
+ Runtime service creation / static memory	< 900 bytes
+ Multiple tasks at same priority	< 975 bytes
+ Runtime priority change + Mutex priority inheritance + FCFS + Task suspension	< 1500 bytes
+ Timer & timeout + Timer call back + Round robin	< 1950 bytes
+ Events + Mailbox	< 2525 bytes
Full Feature Build (no names)	< 3000 bytes
Full Feature Build (no names / no runtime creation)	< 2675 bytes
Full Feature Build (no names / no runtime creation) + Timer services module	< 3025 bytes

Table 7-2 Assembly Code Memory Usage

Description	Size
Assembly code size	300 bytes
Vector table (per interrupt handler entry)	+4 bytes
Hybrid Stack Enabled	+16 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 NGX LPC11U24 development board populated with a 48 MHz LPC11U24 device. For the purpose of the latency measurements, the device was clocked at 24 MHz in order to operate the Flash at full clock rate, eliminating the insertion of wait states. All measurements have been performed on the real platform, with the `SysTick` timer 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 3 (`-O3`)
2. Optimize for Time: Enable
3. Split Load and Store Multiple: Disabled

All other options are disabled, as they do not affect the efficiency of the code generated.

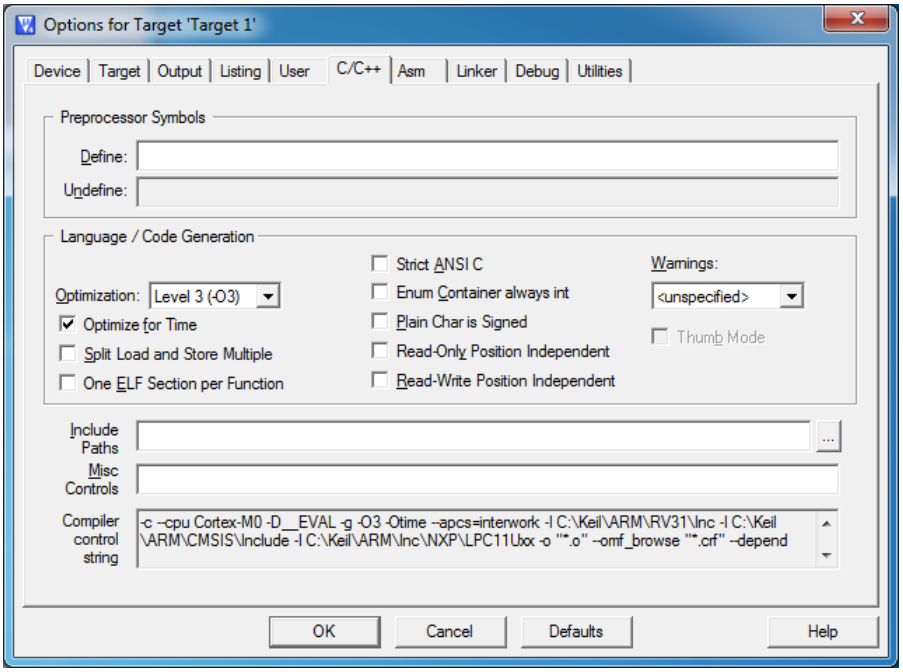

Figure 7-2 Latency Measurement Code Optimization Settings

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
    …
}

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

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

Table 7-6 Measurement with Task unblocking

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

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

The fifth set of measurements counts the number of CPU cycles elapsed from the beginning of an interrupt using the component, until the task that was blocked becomes the running task and is back from the component used that blocked the task. 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-M0. This timer decrements its counter by 1 at every CPU cycle. As was the case for the memory measurements, these numbers were obtained with a beta release of the RTOS. It is possible the released version of the RTOS may have slightly different numbers.

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

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 numbers are the latency measurements when the build option `OS_SEARCH_ALGO` is set to 0.

Table 7-7 Latency Measurements

Description	Minimal Features	Full Features
Semaphore posting no task switch	144 (143)	218 (220)
Semaphore waiting no blocking	147 (145)	231 (233)
Semaphore posting with task switch	232 (262)	376 (403)
Semaphore waiting with blocking	251 (247)	406 (409)
Semaphore posting in ISR with task switch	478 (507)	636 (661)
Event setting no task switch	n/a	214 (216)
Event getting no blocking	n/a	254 (256)
Event setting with task switch	n/a	398 (425)
Event getting with blocking	n/a	424 (427)
Event setting in ISR with task switch	n/a	659 (684)
Mailbox writing no task switch	n/a	275 (277)
Mailbox reading no blocking	n/a	279 (281)
Mailbox writing with task switch	n/a	450 (477)
Mailbox reading with blocking	n/a	459 (462)
Mailbox writing in ISR with task switch	n/a	706 (731)
Interrupt Latency	45	45
Interrupt overhead entering the kernel	246 (245)	260 (258)
Interrupt overhead NOT entering the kernel	68	68
Context switch	55	53

8 Appendix A: Build Options for Code Size

8.1 Case 0: Minimum build

Table 8-1: Case 0 build options

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

8.2 Case 1: + Runtime service creation / static memory

Table 8-2: Case 1 build options

```
#define OS_ALLOC_SIZE       0    /* When !=0, RTOS supplied OSalloc                    */
#define OS_COOPERATIVE      0    /* When 0: pre-emptive, when non-zero: cooperative  */
#define OS_EVENTS           0    /* If event flags are supported                       */
#define OS_FCFS             0    /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK       0    /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE     0    /* Type of logging to use                             */
#define OS_MAILBOX          0    /* If mailboxes are used                              */
#define OS_MAX_PEND_RQST    2    /* Maximum number of requests in ISRs                 */
#define OS_MIN_STACK_USE    0    /* Not minimizing the ernel stack usage               */
#define OS_MTX_DEADLOCK     0    /* This test validates this                           */
#define OS_MTX_INVERSION    0    /* To enable protection against priority inversion  */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes   */
#define OS_NESTED_INTS      0    /* If operating with nested interrupts                */
#define OS_PRIO_CHANGE      0    /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN         2    /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME        0    /* Support multiple tasks with the same priority    */
#define OS_ROUND_ROBIN      0    /* Use round-robin, value specifies period in uS    */
#define OS_RUNTIME          1    /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO      0    /* If using a fast search                             */
#define OS_STARVE_PRIO      0    /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX   0    /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX  0    /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX   0    /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX       0    /* If !=0 how many mailboxes                          */
#define OS_STATIC_NAME      0    /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM       5    /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK     128  /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK      5    /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_TASK_SUSPEND     0    /* If a task can suspend another one                */
#define OS_TIMEOUT          0    /* !=0 enables blocking timeout                       */
#define OS_TIMER_CB         0    /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV        0    /* !=0 includes the timer services module           */
#define OS_TIMER_US         0    /* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG     0    /* If tasks have arguments                            */
```

8.3 Case 2: + Multiple tasks at same priority

Table 8-3: Case 2 build options

```
#define OS_ALLOC_SIZE       0    /* When !=0, RTOS supplied OSalloc                    */
#define OS_COOPERATIVE      0    /* When 0: pre-emptive, when non-zero: cooperative  */
#define OS_EVENTS           0    /* If event flags are supported                       */
#define OS_FCFS             0    /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK       0    /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE     0    /* Type of logging to use                             */
#define OS_MAILBOX          0    /* If mailboxes are used                              */
#define OS_MAX_PEND_RQST    32   /* Maximum number of requests in ISRs                 */
#define OS_MIN_STACK_USE    0    /* Not minimizing the ernel stack usage               */
#define OS_MTX_DEADLOCK     0    /* This test validates this                           */
#define OS_MTX_INVERSION    0    /* To enable protection against priority inversion  */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes   */
#define OS_NESTED_INTS      0    /* If operating with nested interrupts                */
#define OS_PRIO_CHANGE      0    /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN         20   /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME        1    /* Support multiple tasks with the same priority    */
#define OS_ROUND_ROBIN      0    /* Use round-robin, value specifies period in uS    */
#define OS_RUNTIME          1    /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO      0    /* If using a fast search                             */
#define OS_STARVE_PRIO      0    /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX   0    /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX  0    /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX   0    /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX       0    /* If !=0 how many mailboxes                          */
#define OS_STATIC_NAME      0    /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM       5    /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK     128  /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK      5    /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_TASK_SUSPEND     0    /* If a task can suspend another one                */
#define OS_TIMEOUT          0    /* !=0 enables blocking timeout                       */
#define OS_TIMER_CB         0    /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV        0    /* !=0 includes the timer services module           */
#define OS_TIMER_US         0    /* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG     0    /* If tasks have arguments                            */
```

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

Table 8-4: Case 3 build options

```
#define OS_ALLOC_SIZE       0    /* When !=0, RTOS supplied OSalloc                    */
#define OS_COOPERATIVE      0    /* When 0: pre-emptive, when non-zero: cooperative  */
#define OS_EVENTS           0    /* If event flags are supported                       */
#define OS_FCFS             1    /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK       0    /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE     0    /* Type of logging to use                             */
#define OS_MAILBOX          0    /* If mailboxes are used                              */
#define OS_MAX_PEND_RQST    32   /* Maximum number of requests in ISRs                 */
#define OS_MIN_STACK_USE    0    /* Not minimizing the ernel stack usage               */
#define OS_MTX_DEADLOCK     0    /* This test validates this                           */
#define OS_MTX_INVERSION    1    /* To enable protection against priority inversion  */
#define OS_NAMES            0    /* != 0 when named Tasks / Semaphores / Mailboxes   */
#define OS_NESTED_INTS      0    /* If operating with nested interrupts                */
#define OS_PRIO_CHANGE      1    /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN         20   /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME        1    /* Support multiple tasks with the same priority    */
#define OS_ROUND_ROBIN      0    /* Use round-robin, value specifies period in uS    */
#define OS_RUNTIME          1    /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO      0    /* If using a fast search                             */
#define OS_STARVE_PRIO      0    /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX   0    /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX  0    /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX   0    /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX       0    /* If !=0 how many mailboxes                          */
#define OS_STATIC_NAME      0    /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM       5    /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK     128  /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK      5    /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_TASK_SUSPEND     1    /* If a task can suspend another one                */
#define OS_TIMEOUT          0    /* !=0 enables blocking timeout                       */
#define OS_TIMER_CB         0    /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV        0    /* !=0 includes the timer services module           */
#define OS_TIMER_US         0    /* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG     0    /* If tasks have arguments                            */
```

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

Table 8-5: Case 4 build options

```
#define OS_ALLOC_SIZE      0     /* When !=0, RTOS supplied OSalloc                   */
#define OS_COOPERATIVE     0     /* When 0: pre-emptive, when non-zero: cooperative   */
#define OS_EVENTS          0     /* If event flags are supported                      */
#define OS_FCFS            1     /* Allow the use of 1st come 1st serve semaphore     */
#define OS_IDLE_STACK      0     /* If IdleTask supplied & if so, stack size          */
#define OS_LOGGING_TYPE    0     /* Type of logging to use                            */
#define OS_MAILBOX         0     /* If mailboxes are used                             */
#define OS_MAX_PEND_RQST   32    /* Maximum number of requests in ISRs                */
#define OS_MIN_STACK_USE   0     /* Not minimizing the ernel stack usage              */
#define OS_MTX_DEADLOCK    0     /* This test validates this                          */
#define OS_MTX_INVERSION   1     /* To enable protection against priority inversion   */
#define OS_NAMES           0     /* != 0 when named Tasks / Semaphores / Mailboxes    */
#define OS_NESTED_INTS     0     /* If operating with nested interrupts               */
#define OS_PRIO_CHANGE     1     /* If a task priority can be changed at run time     */
#define OS_PRIO_MIN        20    /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0    */
#define OS_PRIO_SAME       1     /* Support multiple tasks with the same priority     */
#define OS_ROUND_ROBIN     100000/* Use round-robin, value specifies period in uS     */
#define OS_RUNTIME         1     /* If create Task / Semaphore / Mailbox at run time  */
#define OS_SEARCH_ALGO     0     /* If using a fast search                            */
#define OS_STARVE_PRIO     0     /* Priority threshold for starving protection        */
#define OS_STARVE_RUN_MAX  0     /* Maximum Timer Tick for starving protection        */
#define OS_STARVE_WAIT_MAX 0     /* Maximum time on hold for starving protection      */
#define OS_STATIC_BUF_MBX  0     /* when OS_STATIC_MBOX != 0, # of buffer element     */
#define OS_STATIC_MBX      0     /* If !=0 how many mailboxes                         */
#define OS_STATIC_NAME     0     /* If named mailboxes/semaphore/task, size in char   */
#define OS_STATIC_SEM      5     /* If !=0 how many semaphores and mutexes            */
#define OS_STATIC_STACK    128   /* if !=0 number of bytes for all stacks             */
#define OS_STATIC_TASK     5     /* If !=0 how many tasks (excluding A&E and Idle)    */
#define OS_TASK_SUSPEND    1     /* If a task can suspend another one                 */
#define OS_TIMEOUT         1     /* !=0 enables blocking timeout                      */
#define OS_TIMER_CB        10    /* !=0 gives the timer callback period               */
#define OS_TIMER_SRV       0     /* !=0 includes the timer services module            */
#define OS_TIMER_US        50000 /* !=0 enables timer & specifies the period in uS    */
#define OS_USE_TASK_ARG    0     /* If tasks have arguments                           */
```

8.6 Case 5: + Events / Mailboxes

Table 8-6: Case 5 build options

```
#define OS_ALLOC_SIZE      0     /* When !=0, RTOS supplied OSalloc                    */
#define OS_COOPERATIVE     0     /* When 0: pre-emptive, when non-zero: cooperative  */
#define OS_EVENTS          0     /* If event flags are supported                       */
#define OS_FCFS            1     /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK      0     /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE    0     /* Type of logging to use                             */
#define OS_MAILBOX         0     /* If mailboxes are used                              */
#define OS_MAX_PEND_RQST   32    /* Maximum number of requests in ISRs                 */
#define OS_MIN_STACK_USE   0     /* Not minimizing the ernel stack usage               */
#define OS_MTX_DEADLOCK    0     /* This test validates this                           */
#define OS_MTX_INVERSION   1     /* To enable protection against priority inversion  */
#define OS_NAMES           0     /* != 0 when named Tasks / Semaphores / Mailboxes   */
#define OS_NESTED_INTS     0     /* If operating with nested interrupts                */
#define OS_PRIO_CHANGE     1     /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN        20    /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME       1     /* Support multiple tasks with the same priority    */
#define OS_ROUND_ROBIN     100000/* Use round-robin, value specifies period in uS    */
#define OS_RUNTIME         1     /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO     0     /* If using a fast search                             */
#define OS_STARVE_PRIO     0     /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX  0     /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX 0     /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX  0     /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX      0     /* If !=0 how many mailboxes                          */
#define OS_STATIC_NAME     0     /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM      5     /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK    128   /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK     5     /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_TASK_SUSPEND    1     /* If a task can suspend another one                */
#define OS_TIMEOUT         1     /* !=0 enables blocking timeout                       */
#define OS_TIMER_CB        10    /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV       0     /* !=0 includes the timer services module           */
#define OS_TIMER_US        50000 /* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG    0     /* If tasks have arguments                            */
```

8.7 Case 6: Full feature Build (no names)

Table 8-7: Case 6 build options

```
#define OS_ALLOC_SIZE      0     /* When !=0, RTOS supplied OSalloc                    */
#define OS_COOPERATIVE     0     /* When 0: pre-emptive, when non-zero: cooperative  */
#define OS_EVENTS          1     /* If event flags are supported                       */
#define OS_FCFS            1     /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK      0     /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE    0     /* Type of logging to use                           */
#define OS_MAILBOX         1     /* If mailboxes are used                            */
#define OS_MAX_PEND_RQST   32    /* Maximum number of requests in ISRs               */
#define OS_MIN_STACK_USE   0     /* Not minimizing the ernel stack usage             */
#define OS_MTX_DEADLOCK    0     /* This test validates this                         */
#define OS_MTX_INVERSION   1     /* To enable protection against priority inversion  */
#define OS_NAMES           0     /* != 0 when named Tasks / Semaphores / Mailboxes   */
#define OS_NESTED_INTS     0     /* If operating with nested interrupts              */
#define OS_PRIO_CHANGE     1     /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN        20    /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME       1     /* Support multiple tasks with the same priority    */
#define OS_ROUND_ROBIN     -100000 /* Use round-robin, value specifies period in uS  */
#define OS_RUNTIME         1     /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO     0     /* If using a fast search                           */
#define OS_STARVE_PRIO     -3    /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX  -10   /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX -100  /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX  100   /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX      2     /* If !=0 how many mailboxes                        */
#define OS_STATIC_NAME     0     /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM      5     /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK    128   /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK     5     /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_TASK_SUSPEND    1     /* If a task can suspend another one                */
#define OS_TIMEOUT         1     /* !=0 enables blocking timeout                     */
#define OS_TIMER_CB        10    /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV       0     /* !=0 includes the timer services module           */
#define OS_TIMER_US        50000 /* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG    1     /* If tasks have arguments                          */
```

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

Table 8-8: Case 7 build options

```
#define OS_ALLOC_SIZE      0     /* When !=0, RTOS supplied OSalloc                    */
#define OS_COOPERATIVE     0     /* When 0: pre-emptive, when non-zero: cooperative  */
#define OS_EVENTS          1     /* If event flags are supported                       */
#define OS_FCFS            1     /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK      0     /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE    0     /* Type of logging to use                           */
#define OS_MAILBOX         1     /* If mailboxes are used                            */
#define OS_MAX_PEND_RQST   32    /* Maximum number of requests in ISRs               */
#define OS_MIN_STACK_USE   0     /* Not minimizing the ernel stack usage             */
#define OS_MTX_DEADLOCK    0     /* This test validates this                         */
#define OS_MTX_INVERSION   1     /* To enable protection against priority inversion  */
#define OS_NAMES           0     /* != 0 when named Tasks / Semaphores / Mailboxes   */
#define OS_NESTED_INTS     0     /* If operating with nested interrupts              */
#define OS_PRIO_CHANGE     1     /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN        20    /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME       1     /* Support multiple tasks with the same priority    */
#define OS_ROUND_ROBIN     -100000 /* Use round-robin, value specifies period in uS  */
#define OS_RUNTIME         0     /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO     0     /* If using a fast search                           */
#define OS_STARVE_PRIO     -3    /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX  -10   /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX -100  /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX  0     /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX      0     /* If !=0 how many mailboxes                        */
#define OS_STATIC_NAME     0     /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM      0     /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK    0     /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK     0     /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_TASK_SUSPEND    1     /* If a task can suspend another one                */
#define OS_TIMEOUT         1     /* !=0 enables blocking timeout                     */
#define OS_TIMER_CB        10    /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV       0     /* !=0 includes the timer services module           */
#define OS_TIMER_US        50000 /* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG    1     /* If tasks have arguments                          */
```

8.9 Case 8: Full build adding the optional timer services

Table 8-9: Case 8 build options

```
#define OS_ALLOC_SIZE      0     /* When !=0, RTOS supplied OSalloc                    */
#define OS_COOPERATIVE     0     /* When 0: pre-emptive, when non-zero: cooperative  */
#define OS_EVENTS          1     /* If event flags are supported                       */
#define OS_FCFS            1     /* Allow the use of 1st come 1st serve semaphore    */
#define OS_IDLE_STACK      0     /* If IdleTask supplied & if so, stack size         */
#define OS_LOGGING_TYPE    0     /* Type of logging to use                           */
#define OS_MAILBOX         1     /* If mailboxes are used                            */
#define OS_MAX_PEND_RQST   32    /* Maximum number of requests in ISRs               */
#define OS_MIN_STACK_USE   0     /* Not minimizing the ernel stack usage             */
#define OS_MTX_DEADLOCK    0     /* This test validates this                         */
#define OS_MTX_INVERSION   1     /* To enable protection against priority inversion  */
#define OS_NAMES           0     /* != 0 when named Tasks / Semaphores / Mailboxes   */
#define OS_NESTED_INTS     0     /* If operating with nested interrupts              */
#define OS_PRIO_CHANGE     1     /* If a task priority can be changed at run time    */
#define OS_PRIO_MIN        20    /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0   */
#define OS_PRIO_SAME       1     /* Support multiple tasks with the same priority    */
#define OS_ROUND_ROBIN     -100000 /* Use round-robin, value specifies period in uS  */
#define OS_RUNTIME         0     /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO     0     /* If using a fast search                           */
#define OS_STARVE_PRIO     -3    /* Priority threshold for starving protection       */
#define OS_STARVE_RUN_MAX  -10   /* Maximum Timer Tick for starving protection       */
#define OS_STARVE_WAIT_MAX -100  /* Maximum time on hold for starving protection     */
#define OS_STATIC_BUF_MBX  100   /* when OS_STATIC_MBOX != 0, # of buffer element    */
#define OS_STATIC_MBX      2     /* If !=0 how many mailboxes                        */
#define OS_STATIC_NAME     0     /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM      5     /* If !=0 how many semaphores and mutexes           */
#define OS_STATIC_STACK    128   /* if !=0 number of bytes for all stacks            */
#define OS_STATIC_TASK     5     /* If !=0 how many tasks (excluding A&E and Idle)   */
#define OS_TASK_SUSPEND    1     /* If a task can suspend another one                */
#define OS_TIMEOUT         1     /* !=0 enables blocking timeout                     */
#define OS_TIMER_CB        10    /* !=0 gives the timer callback period              */
#define OS_TIMER_SRV       1     /* !=0 includes the timer services module           */
#define OS_TIMER_US        50000 /* !=0 enables timer & specifies the period in uS   */
#define OS_USE_TASK_ARG    1     /* If tasks have arguments                          */
```