CODE TIME TECHNOLOGIES

Abassi RTOS

Porting Document

C28X – CCS

Disclaimer

Table of Contents

List of Figures

List of Tables

1 Introduction

This document details the port of the Abassi RTOS to the C28X processor. The software suite used for this specific port is the Texas Instruments Code Composer Studio for the C28X; the version used for the port and all tests is Version 5.2.0.00069.

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_C28X_CCS.s`	RTOS assembly file for the C28X to use with Code Composer Studio
`Demo_1_TMX320_28027_CCS.c`	Demo code for the Olimex TMX320-28027 evaluation board
`Demo_3_TMX320_28027_CCS.c`	Demo code for the Olimex TMX320-28027 evaluation board
`Demo_7_TMX320_28027_CCS.c`	Demo code for the Olimex TMX320-28027 evaluation board
`AbassiDemo.h`	Build option settings for the demo code

1.2 Limitations

Only the large memory model is supported.

Abassi's standard component `OSisrInstall()`, which simplifies the attachment of interrupts handlers to sources of interrupt, is not available for this port; it is replaced by the component `OSmapISR()`. The reason is the C28X processor family interrupts may be handled through the Peripheral Interrupt Expansion (PIE) module, which uses its own look-up table in addition to the Abassi interrupt dispatcher look-up table.

NOTE: The CCS compiler defines the C28X data type `char` as 2 bytes, or 1 processor word. This has an impact on the definitions of Abassi's build options. In the Abassi User's Guide, every time "`byte`" or "`char`" are used when mentioning memory reserved by the build options, it truly means 2 bytes in the case of C28X. For example, specifying a value of 256 for the build option `OS_ALLOC_SIZE` reserves 256 words of memory; that is, 512 bytes of memory.

In this document, when "byte" is used, it really means a byte (half a C28X word). The use of byte instead of word was retained in order to keep this document consistent with all other ports documents.

2 Target Set-up

Very little is needed to configure the Code Composer Studio development environment to use the Abassi RTOS in an application. All there is to do is to add the files `Abassi.c` and `Abassi_C28X_CCS.s` in the source files of the application project, and make sure the four configuration settings (described in the following subsections) in the file `Abassi_C28X_CCS.s` are set according to the needs of the application and target device. As well, update the include file path in the C/C++ compiler preprocessor options with the location of `Abassi.h`.

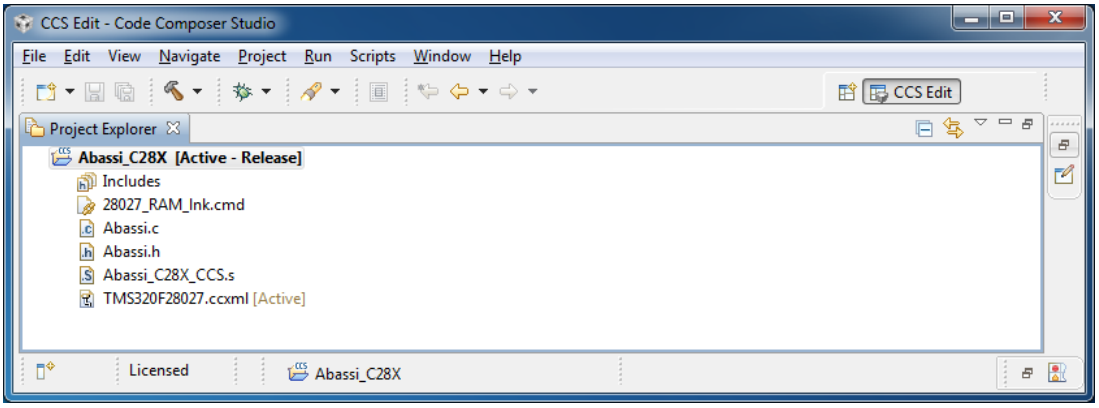

Figure 2-1 Project File List

If the application is not built though the Code Composer Studio GUI, then it is important to specify the same options on the command line for both the compiler and the assembler, as the file `Abassi_C28X_CCS.s` relies on command line options to generate the correct code. The assembler options the file `Abassi_C28X_CCS.s` requires are listed in the following table:

Table 2-1 Assembler required options

Option	Description
`-v28`	To generate C28X code
`--large_memory_model`	Code memory model

NOTE: By default, the Code Composer Studio runtime libraries are not multithread-safe, but Code Composer Studio has a special hook to make the libraries multithread-safe. The required hooks are automatically applied by attaching the Abassi internal mutex (`G_OSmutex`) during runtime in `OSstart()`.

2.1 FPU

The Abassi port for the C28X seamlessly supports the use of the floating point unit (FPU), when available on a device. If the Code Composer Studio GUI is used, then the FPU is properly handled by Abassi as long as it is enabled through in "*Build / C2000 Compiler / Processor Options / Specify floating point support (--float_support)*" menu, selecting either `fpu32` or `fpu64`.

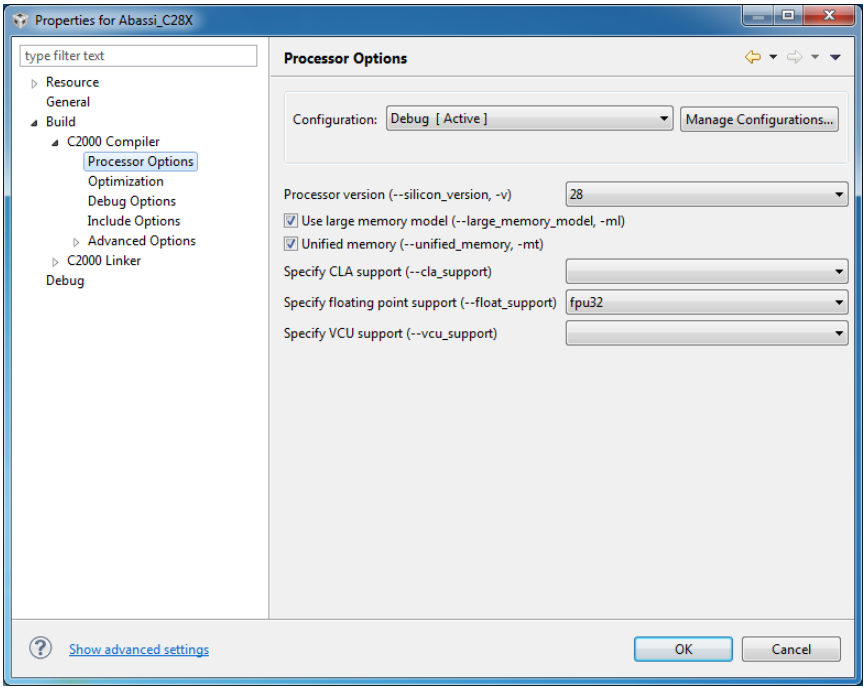

Figure 2-2 GUI enabling of the FPU

If command line assembly is used instead of the GUI, and the FPU is needed in the application, the assembler file `Abassi_C28X_CCS.s` must be informed by specifying the command line option `--float_support=fpu32` or `--float_support=fpu64`. Then the file `Abasss_C28X_CCS.s` will contain all the extra code required to deal with the FPU:

Table 2-2 Command line enabling of the FPU

```
cl2000  … --float_support=fpu32  …

Or

cl2000  … --float_support=fpu64  …
```

If the FPU is not used, i.e. when the command line option `--float_support=fpu32` or `--float_support=fpu64` is not used or not selected in the GUI, then the code in `Abassi_C28X_CCS.s` reverts to the non-FPU code.

NOTE: Abassi does not use the FPU bank of shadow registers. In other works, neither the context switch nor the interrupt dispatcher makes use of the `save` / `restore` pair of instructions. The bank of shadow registers is thus available and should ideally be used in non-nested fast interrupts when needed.

2.2 Interrupt Stack Set-up

It is possible, and is highly recommended to use a hybrid stack even when nested interrupts (Section 2.5) are not enabled 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 part of the comulative contexts of nested interrupts. This feature is controlled by the value set by the definition `OS_ISR_STACK`, located around line 25 in the file `Abassi_C28X_CCS.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 words (see Section 4 for information on stack sizing).

As supplied in the distribution, the hybrid stack feature is enabled with a size of 128 words; this is shown in the following table:

Table 2-3 Interrupt Stack Enabled

```
  .if !($defined(OS_ISR_STACK))
OS_ISR_STACK    .equ    128     ; If using a dedicated stack for the ISRs
  .endif                        ; 0 if not used, otherwise size of stack in wordss
```

Table 2-4 Interrupt Stack Disabled

```
  .if !($defined(OS_ISR_STACK))
OS_ISR_STACK    .equ    0       ; If using a dedicated stack for the ISRs
  .endif                        ; 0 if not used, otherwise size of stack in words
```

There are always 8 extra words added to the value of `OS_ISR_STACK` as 8 words are always needed for a local frame. This local frame is used when debugging and as it does not contain real data, the stack trace back feature of the debugger is invalid for the first interrupt in a nesting.

Alternatively, it is possible to overload the `OS_ISR_STACK` value set in `Abassi_C28X_CCS.s` by using the assembler command line option `--asm_define` (or `-ad`) and specifying the desired hybrid stack size. In the following example, the hybrid stack size is set to 256 words:

Table 2-5 Command line set of `OS_ISR_STACK`

```
cl2000  … --asm_define=OS_ISR_STACK=256  …
```

The hybrid stack size can also be set through the GUI, in the "*Build / C2000 Compiler / Advanced Options / Assembler Options / Pre-define assembly symbol NAME*" menu, as shown in the following figure:

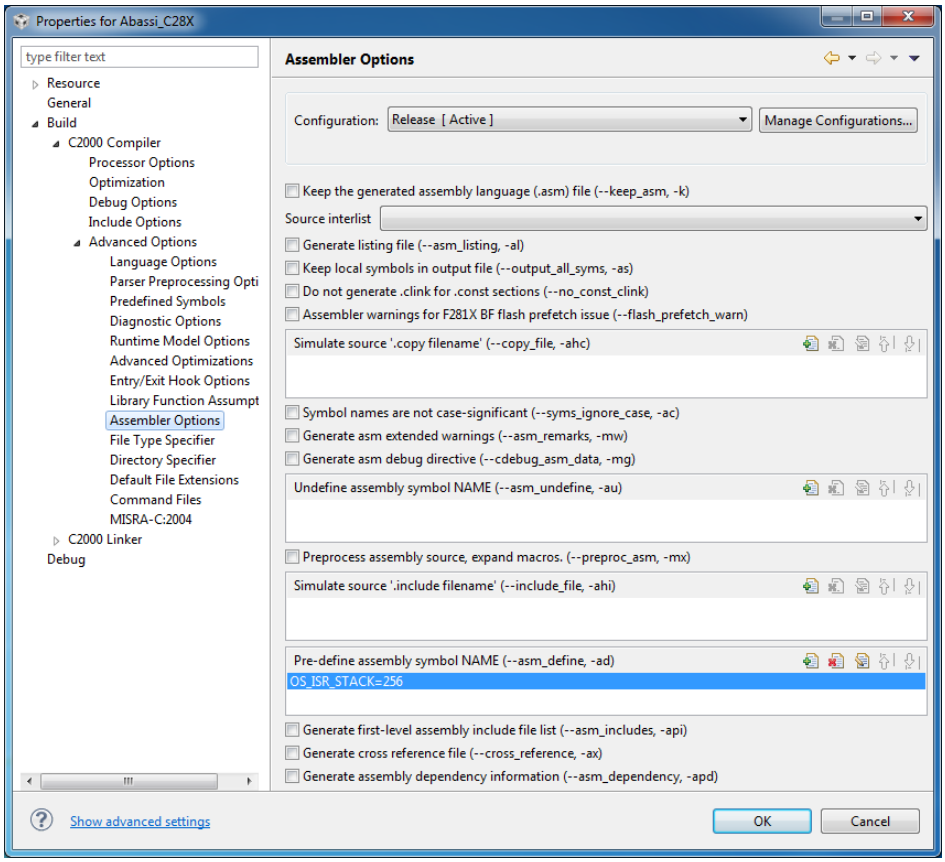

Figure 2-3 GUI set of `OS_ISR_STACK`

2.3 Status bits propagation

In the C28X status register #0, there is the `OVC/OVCU` field and the `V` flags, which are both related to arithmetic overflows. In some applications, it may be desirable to keep the overflow information local to each task. This feature is controlled by the value set by the definition `OS_KEEP_STATUS`, located around line 30 in the file `Abassi_C28X_CCS.s`. To disable this feature, meaning to propagate the overflow information across all tasks, set the definition of `OS_KEEP_STATUS` to a value of zero. To enable it, meaning to keep the overflow local to the tasks, set the definition of `OS_KEEP_STATUS` to a non-zero value. As supplied in the distribution, the overflow is propagated across all tasks; this is shown in the following table:

Table 2-6 Overflow bits non-local

```
  .if !($defined(OS_KEEP_STATUS))
OS_KEEP_STATUS      .equ    0     ; Set to non-zero to keep status local to a task
  .endif                          ; 0 if not keeping local
```

Table 2-7 Overflow bits local

```
  .if !($defined(OS_KEEP_STATUS))
OS_KEEP_STATUS      .equ    1     ; Set to non-zero to keep status local to a task
  .endif                          ; 0 if not keeping local
```

Alternatively, it is possible to overload the `OS_KEEP_STATUS` value set in `Abassi_C28X_CCS.s` by using the assembler command line option `--asm_define` (or `-ad`) and specifying the desired propagation mode. In the following example, the overflow bits are kept localized:

Table 2-8 Command line set of `OS_KEEP_STATUS`

```
cl2000  … --asm_define=OS_KEEP_STATUS=1  …
```

The propagation of the status bits can also be controlled through the GUI, in the "*Build / C2000 Compiler / Advanced Options / Assembler Options / Pre-define assembly symbol NAME*" menu, as shown in the following figure:

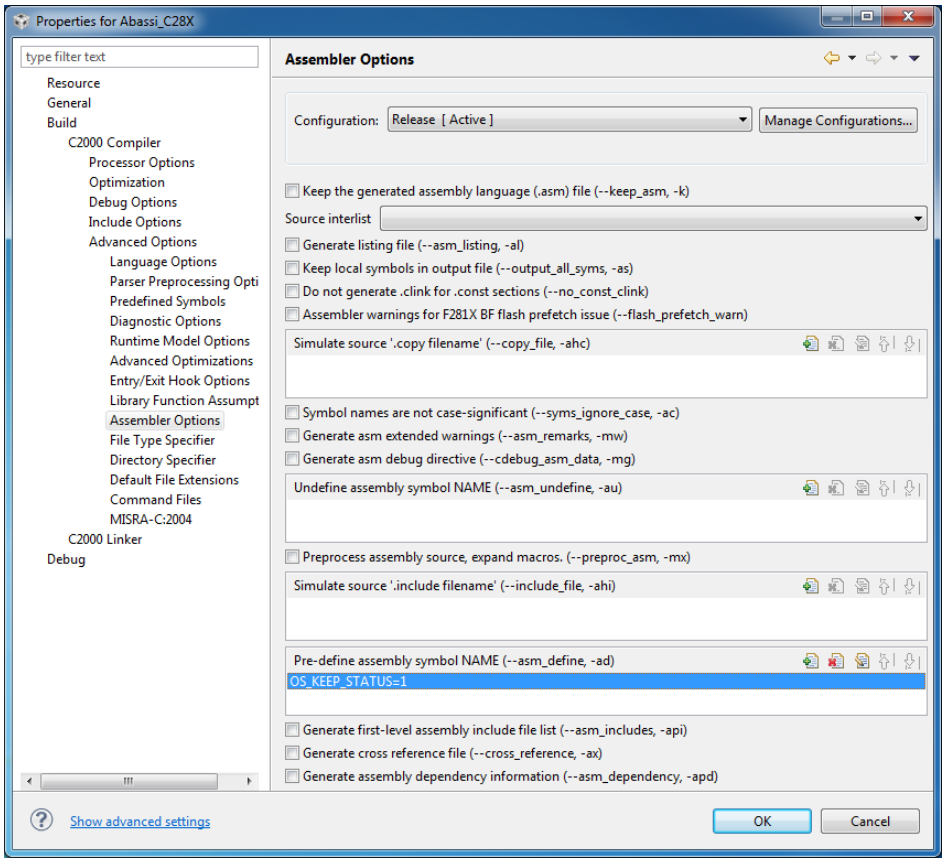

Figure 2-4 GUI set of `OS_KEEP_STATUS`

2.4 Number of interrupts

Abassi supports three different ways on how it handles the interrupts, and this is configured by the value set for the definition of the token `OS_N_INTERRUPTS`, located around line 35 in the file `Abassi_C28X_CCS.s`. More details on these three configurations are provided in Section 3. When `OS_N_INTERRUPTS` is set to a value of zero, the Peripheral Interrupt Expansion (PIE) is not used; instead the basic 32 entries interrupt table located at address 0x000000 is used. When the value of `OS_N_INTERRUPTS` is between 1 and 127 inclusive, the Peripheral Interrupt Expansion (PIE) is used with interrupt dispatcher pre-handlers. When `OS_N_INTERRUPTS` is set to a value of 128 or larger, the Peripheral Interrupt Expansion (PIE) is used but without the interrupt dispatcher pre-handlers.

As supplied in the distribution, the value assigned to `OS_N_INTERRUPTS` is 16, meaning the Peripheral Interrupt Expansion (PIE) is used with 16 interrupt dispatcher pre-handlers; this is shown in the following table:

Table 2-9 `OS_N_INTERRUPTS` (PIE with pre-handlers)

```
  .if !($defined(OS_N_INTERRUPTS))  ; Minimum number of interrupts to handle
OS_N_INTERRUPTS   .equ    16        ; ==   0: not using the PIE
  .endif                            ; >= 128: PIE without pre-handlers
                                    ; else  : PIE with pre-handlers
```

To operate the interrupt with the Peripheral Interrupt Expansion (PIE) without the interrupt dispatcher pre-handlers, assign a value of 128 or greater to `OS_N_INTERUPTS` as shown below:

Table 2-10 `OS_N_INTERRUPTS` (PIE without pre-handlers)

```
  .if !($defined(OS_N_INTERRUPTS))  ; Minimum number of interrupts to handle
OS_N_INTERRUPTS   .equ    128       ; ==   0: not using the PIE
  .endif                            ; >= 128: PIE without pre-handlers
                                    ; else  : PIE with pre-handlers
```

To operate the interrupt without the Peripheral Interrupt Expansion (PIE), assign a value of 0 to `OS_N_INTERUPTS` as shown below:

Table 2-11 `OS_N_INTERRUPTS` (no PIE)

```
  .if !($defined(OS_N_INTERRUPTS))  ; Minimum number of interrupts to handle
OS_N_INTERRUPTS   .equ    0         ; ==   0: not using the PIE
  .endif                            ; >= 128: PIE without pre-handlers
                                    ; else  : PIE with pre-handlers
```

Alternatively, it is possible to overload the `OS_N_INTERRUPTS` value set in `Abassi_C28X_CCS.s` by using the assembler command line option `--asm_define` (or `-ad`) and specifying the desired value. In the following example, the PIE is used without pre-handlers:

Table 2-12 Command line set of `OS_N_INTERRUPTS`

```
cl2000  … --asm_define=OS_N_INTERRUPTS=128  …
```

The setting of `OS_N_INTERRUPTS` can also be controlled through the GUI, in the "*Build / C2000 Compiler / Advanced Options / Assembler Options / Pre-define assembly symbol NAME*" menu, as shown in the following figure:

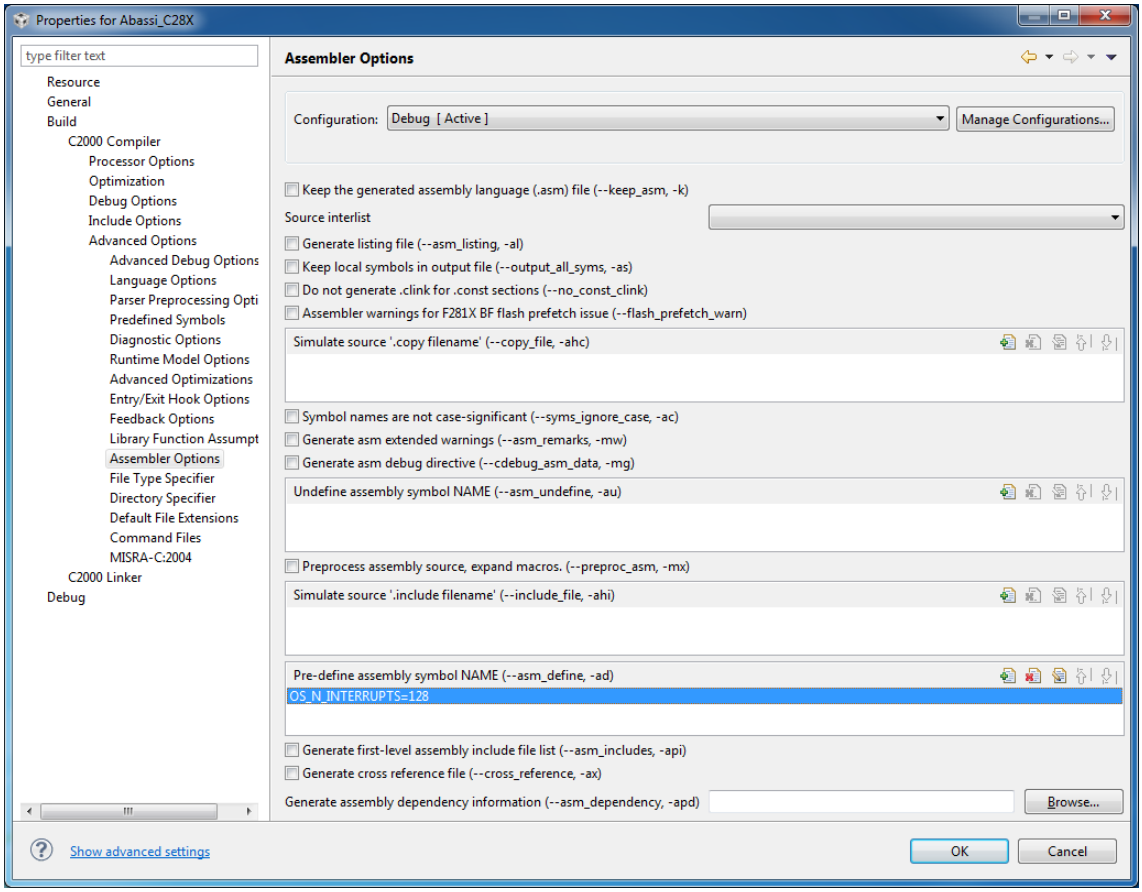

Figure 2-5 GUI set of `OS_N_INTERRUPTS`

2.5 Interrupt Nesting

The normal operation of the interrupt controller on the C28X devices is to only allow a single interrupt to operate at any time. This means when the processor is servicing an interrupt, any new interrupts, even if their priority is higher than the currently serviced interrupt level, remain pending until the processor complete the servicing of the current interrupt. The interrupt dispatcher allows the nesting of interrupts; this means an interrupt of any priority can interrupt the processing of an interrupt currently being handled, even if it is of lower priority. Nested interrupts are disabled by default. To enable nested interrupts both the build option `OS_NESTED_INTS` used by `Abassi.c` and the token `OS_NESTED_INTS` in the `Abassi_C28X_CCS.s` file, around line 40, must be set to a non-zero value, as shown in the two following tables:

Table 2-13 Nested Interrupts Enabled (C)

```
cl2000  … --define=OS_NESTED_INTS=1  …
```

Table 2-14 Nested Interrupts Enabled (ASM)

```
  .if !($defined(OS_NESTED_INTS))
OS_NESTED_INTS    .equ    1       ; == 0: Interrupt dispatcher does not nest interrupts
  .endif                          ; != 0: Interrupt dispatcher nests interrupts
```

Alternatively, it is possible to overload the `OS_NESTED_INTS` value set in `Abassi_C28X_CCS.s` by using the assembler command line option `--asm_define` (or `-ad`) and specifying the setting for the nesting of the interrupts. Even though the token name is identical to the Abassi build option, a definition passed to the compiler does not get propagated to the assembler, so the assembler option `--asm_define` (or `-ad`) must also be used. The following example shows the activation of the nesting for the interrupts in the assembly file:

Table 2-15 Command line set of `OS_NESTED_INTS`

```
cl2000  … --asm_define=OS_NESTED_INTS=1  …
```

The control of the interrupt nesting can also be also be controlled through the GUI, in the "*Build / C2000 Compiler / Advanced Options / Assembler Options / Pre-define assembly symbol NAME*" menu, as shown in the following figure:

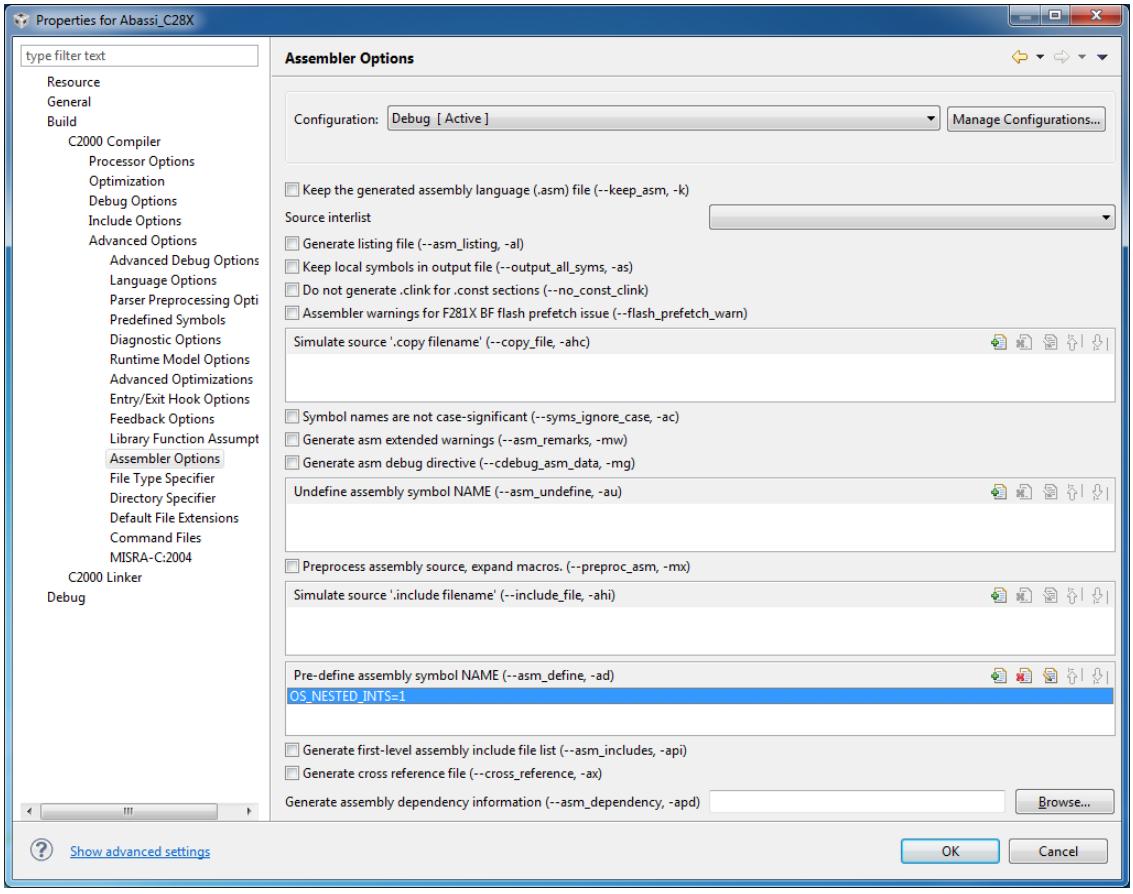

Figure 2-6 GUI set of `OS_NESTED_INTS`

The kernel is never entered as long as interrupt nesting is occurring. 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 during the time the interrupt function is using the CPU are other interrupts (if nesting is not enabled) blocked by the interrupt controller.

NOTE: The build option `OS_NESTED_INTS` must be set to a non-zero value when the token `OS_NESTED_INTS` in the file `Abassi_C28X_CCS.s` is set to a non-zero value. If the token `OS_NESTED_INTS` in the file `Abassi_C28X_CCS.s` is set to a zero value, and the build option `OS_NESTED_INTS` is non-zero, the application will properly operate, but with a tiny bit less real-time efficiency when kernel requests are performed during an interrupt.

3 Interrupts

The Abassi RTOS needs to be aware when kernel requests are performed inside or outside an interrupt context. The port for the C28X processor family offers the designer 3 different techniques to handle the interrupts to make the kernel interrupt aware. Depending on the application memory constrains, one method is more appropriate than another. For all three interrupt handling techniques, Abassi's ISR dispatcher relies on a look-up table to know what is the function attached to the interrupt being handled. Also, for all 3 methods, it is possible to bypass the Abassi interrupt dispatcher and attach a "fast interrupt" (interrupts function using the keyword `interrupt` in "C", see Section 3.3 for more details on fast interrupts).

3.1 Interrupt handling techniques

The most likely used technique to handle the interrupt is based on the Peripheral Interrupt Expansion (PIE). The PIE possesses its own interrupt vector table with 128 distinct entries. There are two options on how Abassi handles the interrupts when the PIE is involved. The first one does not use interrupt dispatcher pre-handlers, the second uses interrupt dispatcher pre-handlers.

3.1.1 PIE without pre-handlers

When the definition of `OS_N_INTERRUPTS` in the file `Abassi_C28X_CCS.s` is set to a value of 128 or greater, the interrupts are dispatched using the PIE without pre-handlers. What happens for a regular interrupt is the corresponding entry of the PIE vector table is set to the address of Abassi interrupt dispatcher, redirecting the interrupt to the dispatcher. When a fast interrupt is attached, the address of the fast interrupt function handler is set in the corresponding PIE vector table. When an interrupt occurs, the PIE uses the vector table entry to determine the address to call. For a fast interrupt, the address is the fast interrupt function handler itself. For a regular interrupt, the called address is always the interrupt dispatcher address. When called, the interrupt dispatcher uses the `PIEVECT` entry in the `PIECTRL` register of the PIE to determine which PIE vector is the source of the interrupt, and from that information the interrupt dispatcher reads its own look-up table to call the regular "C" function that was attached to the PIE vector ID through `OSmapISR()` (see Section 6.1).

All entries of PIE table vector are set to an invalid handler during the RTOS initialization upon calling `OSstart()`.

On data memory constrained applications selecting the PIE without pre-handlers is not optimal, as the interrupt dispatcher requires a 512 bytes (256 words) look-up table. The table is always dimensioned to 512 bytes, even if the application only uses a few interrupts. The PIE with pre-handler technique offers a better approach to minimize the data requirement of the interrupt dispatcher look-up table.

3.1.2 PIE with pre-handlers

When the definition of `OS_N_INTERRUPTS` in the file `Abassi_C28X_CCS.s` is set to a value greater than 0 and less than 128, the interrupts are dispatched using the PIE with pre-handlers. What happens for regular interrupt is the corresponding entry of the PIE vector table is set to the address of a pre-handler for the Abassi interrupt dispatcher, redirecting the interrupt to the pre-handler. When a fast interrupt is attached, the address of the fast interrupt function handler is set in the corresponding PIE vector table. When an interrupt occurs, the PIE uses the vector table entry to determine the address to call. For a fast interrupt, the address is the interrupt handler function itself. For a regular interrupt, the called address is the interrupt dispatcher pre-handler address. When called, the interrupt dispatcher pre-handler informs the interrupt dispatcher about the index to use to read its own look-up table to call the regular "C" function. This means there has to be unique interrupt dispatcher pre-handlers for each possible interrupt source to handle in the application. For example, if the application supports 5 sources of interrupts then the definition of `OS_N_INTERRUPTS` in the file `Abassi_C28X_CCS.s` must be set to a value of 5 or greater.

All entries of PIE table vector are set to an invalid handler during the RTOS initialization upon calling `OSstart()`.

Compared to the previous technique, the PIE without pre-handlers, it is possible to greatly reduce the data memory requirements of the interrupt dispatcher look-up table, as it can be optimally dimensioned for the application.

On program memory constrained applications selecting the PIE with pre-handlers is not optimal as the interrupt dispatcher pre-handlers add a bit of code compared to the PIE without pre-handlers technique; the more interrupts to handle, the more code is required. The PIE without pre-handler technique offers a better approach to minimize the program memory requirement.

3.1.3 PIE not used

When the definition of `OS_N_INTERRUPTS` in the file `Abassi_C28X_CCS.s` is set to a value of 0, the interrupts are not processed using the PIE. Not involving the PIE uses the very basic interrupt handling capabilities on the C28X devices, with all its limitations. The interrupt table located at program address 0x000000 is used. To map the interrupt to the table, the `ENPIE` bit in the `PIECTRL` register of the PIE is set to zero, disabling the PIE, and the `VMAP` bit in the processor status register #1 is set to 0, to use the table located at address 0x000000. The alternate table, located at address 0x003FFC0, cannot be used as it is part of the ROM, therefore not writable. The disabling of the PIE and the setting of the VMAP bit in status register #1 is performed during the RTOS initialization upon calling `OSstart()`.

What happens for regular interrupts is the corresponding entry interrupt table is set to the address of a pre-handler for the Abassi interrupt dispatcher, redirecting the interrupt to the pre-handler. When a fast interrupt is attached, the address of the fast interrupt function handler is set in the corresponding interrupt table. When an interrupt occurs, the CPU uses the interrupt table entry to determine the call address. For a fast interrupt, the address is the function itself. For a regular interrupt, the called address is the interrupt dispatcher pre-handler address. When called, the interrupt dispatcher pre-handler informs the interrupt dispatcher about the index to use to read its own look-up table to call the regular "C" function. There must be unique interrupt dispatcher pre-handlers for each interrupt source to handle in the application. As the C28X basic interrupt table has 32 entries, 32 pre-handlers are available.

All entries of interrupt table are set to invalid handler during the RTOS initialization upon calling `OSstart()`.

3.1.4 Interrupt Installer

Attaching a function to a regular interrupt is quite straightforward. All there is to do is use the RTOS component `OSmapISR()`[1] to specify the interrupt vector number, the interrupt dispatcher look-up table index, and the function to be attached to that interrupt vector number. For example, Table 3-1 shows the code required to attach the `CPU-Timer #2` interrupt to the RTOS timer tick handler (`TIMtick`). The vector number of the CPU-Timer #2 is 14 and the following example uses the second entry of the ISR dispatcher table (index #1) to hold the address of the RTOS timer function.

Table 3-1 Attaching a Function to a Regular Interrupt

```
#include "Abassi.h"

  …
  OSstart();
  …
  OSmapISR(14, 1, &TIMtick);
  /* Set-up the count reload and enable the timer interrupt */

  … /* More ISR setup */

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

[1] The component `OSisrInstall()` is not supported for the C28X port

At start-up, once `OSstart()` has been called, all interrupt handler functions and vector table entries are set to an `ESTOP0` instruction in a function named `OSinvalidISR()`. If an interrupt function is attached to an interrupt number using the `OSmapISR()` component before calling `OSstart()`, this attachment will be removed during the execution of `OSstart()`, so `OSmapISR()` 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 to ease debugging, the handling function can be set back to `OSinvalidISR()`. This is shown in Table 3-2:

Table 3-2 Invalidating an ISR handler

```
#include "Abassi.h"

   …
   /* Disable the interrupt source */
   OSmapISR(VectID, TblNmb, &OSinvalidISR);
   …
```

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

3.2 Interrupt Enabling and Acknowledgment

The ISR dispatcher does not deal with the re-enabling of the interrupt line nor does it acknowledge the response to the interrupt if the peripheral requires such a feedback. Each interrupt function handler must perform these operations. At minimum, the bit in the `IER` register corresponding to the interrupt line must be set to 1, otherwise all interrupts associated to that line will remain disabled.

3.2.1 RTOS Timer Tick interrupt re-enabling

As explained in the previous section, all interrupt handlers must re-enable the interrupt line they are associated with. In the case of the RTOS timer tick, this is always set-up for a specific `CPU-TIMER`. The selected timer is `CPU-TIMER #2`, which is already reserved for TI/RTOS use. If a different `CPU-TIMER` or external device is used as the tick source for the RTOS timer tick, the re-enabling of the interrupt line must be modified. The change affects the file `Abassi.h`. In `Abassi.h`, there is an area where all the C28X dedicated set-up is grouped; the code affected in `Abassi.h` is shown in Table 3-3 below and the line to modify is the one with " `#define OX_TIM_TICK_ACK()…`":

Table 3-3 Default RTOS timer Interrupt re-enabling

```
#elif defined(__TI_COMPILER_VERSION__) && defined(__TMS320C28XX__)
   …
   #define OX_TIM_TICK_ACK() do{asm(" or IER, #0x2000;");}while(0)
```

For example, if `CPU-TIMER #1` is used instead of `CPU-TIMER #2`, the definition of `OX_TIM_TICK_ACK()` would be replaced by:

Table 3-4 RTOS timer Tick using `CPU-Timer #1`

```
#elif defined(__TI_COMPILER_VERSION__) && defined(__TMS320C28XX__)
   …
   #define OX_TIM_TICK_ACK() do{asm(" or IER, #0x1000;");}while(0)
```

When it becomes necessary to perform more than a simple re-enabling of the interrupt line, then a re-enabling/acknowledging function should be created, as in-line assembly in "C" becomes cumbersome when trying not to corrupt the registers used by the compiler generated code. For example, if `CPU-TIMER #0` is used instead of `CPU-TIMER #1`, the definition of `OX_TIM_TICK_ACK()` would be replaced by:

Table 3-5 RTOS timer Tick using `CPU-Timer #0`

```
#elif defined(__TI_COMPILER_VERSION__) && defined(__TMS320C28XX__)
   …
   #define OX_TIM_TICK_ACK()  do{AckTimer0();}while(0)
   #ifdef __ABASSI_C__
   static void AckTimer0(void);
   static void AckTimer0(void) {
      volatile int *PieCtrlAck1 = (volatile int *)0x0CE1;
         asm(" or ier, #0x1;");    /* Re-enable INT1 line           */
         PieCtrlAck1 |= 0x1;       /* Re-enable the PIE interrupt   */
         return;
      }
   #endif
```

Here is some explanation on the modifications in Table 3-5. First, the function to re-enable the interrupts for `CPU-TIMER #0` is named `AckTimer0()` and the definition of `OX_TIM_TICK_ACK()` is set to calling the `AckTimer0()` function. The statement `#ifdef __ABASSI_C__` is only valid at the top of the file `Abassi.c`, therefore this code is only included in `Abassi.c` where the timer tick interrupt function handler is located. The statements inside the `#ifdef` / `#endif` pair simply declare the function prototype and insert the code of the function itself in `Abassi.c`. The use of the static hints to the optimizer it should in-line the function, reducing the stack usage and the CPU.

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 use the `OSmapISR()` component (see Section 6.1), making sure the second argument of `OSmapISR()` is negative. For example, attaching the ADC1 end of conversion to INT1 (priority 5) is done by assigning the ADC interrupt function handler `ADCread()` to its matching vector ID (number 32) in the PIE vector table, as shown below:

Table 3-6 Attaching a Function to a Fast Interrupt

```
#include "Abassi.h"

  …
  OSstart();
  …
  OSmapISR(32, -1, &ADCread);

  … /* More ISR setup */

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

This example uses the PIE, therefore the token `OS_N_INTERRUPTS` (see Section 2.4) required by `Abassi_C28X_CCS.s` must be set to a value greater than 0.

3.3.1 Nested Fast Interrupts

It is possible to allow interrupt nesting for fast interrupts too; all there is to do is to re-enable the interrupts with the `EINT` instruction in the fast interrupt function handler. If only fast interrupt are nested, there is no need set the build option `OS_NESTED_INTS` to a non-zero value, nor the `Abassi_C28X_CCS.s` token `OS_N_INTERRUPTS`, as fast interrupts do not use RTOS components.

As the interrupt dispatcher is not involved with fast interrupts, the global re-enabling of the interrupts must be manually done in a fast interrupt handler if nesting is desired. Either the RTOS `OSeint()` component or the CCS compiler statement `asm(" eint")` can be used for this purpose. This is shown below for the same example as used in the previous section:

Table 3-7 Fast Interrupt Nesting

```
Interrupt void FastHandler(void)
{
   IER |= 0x1;                          /* ADCINT1 #0 in PIE group 1, re-enable INT1 */
   PieCtrlRegs.PIEACK.bit.ACK1 = 0x1;   /* Re-enable it in the PIE line              */
   asm("  eint");
   …

   return
}
```

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 C28X, 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. The stack usage for the interrupt context includes the 30 bytes the processor automatically save when entering the interrupt mode.

Table 4-1 Context Save Stack Requirements

Description	Context save
Blocked/Preempted task context save	32 bytes
Blocked/Preempted task context save (`OS_KEEP_STATUS != 0`)	+2 bytes
Blocked/Preempted task context save (FPU in use)	+16 bytes
Interrupt context save	58 bytes
Interrupt context save (FPU in use)	+24 bytes
Interrupt context save (`OS_ISR_STACK != 0`)	+4 bytes

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

NOTE: The C28X processor needs alignment on 2 words for many 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 2 words.

If the hybrid interrupt stack (see Section 2.2) is enabled, then the above description changes: it is only necessary to reserve room on task stacks for a single interrupt context save and not the worst-case nesting. With the hybrid stack enabled, the second, third, and so on interrupts use the stack dedicated to the interrupts. The hybrid stack is enabled when the `OS_ISR_STACK` token in the file `Abassi_C28X_CCS.s` is set to a non-zero value (see Section 2.5).

5 Search Set-up

The Abassi RTOS build option `OS_SEARCH_FAST` offers four 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 `CPU-TIMER2` peripheral, which was set to increment 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 4 (C28X `int` are 16 bits, so 2^4), meaning a 16 bit look-up table, further searched through successive approximation. The compiler optimization for this measurement was set to High optimization (`-O4`) / Optimize for speed. The RTOS build options were set to the minimum feature set, except for option `OS_PRIO_CHANGE` set to non-zero. The presence of this extra feature provokes a small mismatch between the result for a difference of priority of 1, with `OS_SEARCH_FAST` set to zero, and the latency results in Section 7.2.

When the build option `OS_SEARCH_ALGO` is set to a negative value, indicating to use a 2-dimensional linked list search technique instead of the search array, the number of CPU is constant at 314 cycles.

Table 5-1 Search Algorithm Cycle Count

Priority	Linear search	Look-up	Approximation
1	341	397	470
2	350	406	469
3	359	415	472
4	368	424	469
5	377	433	472
6	386	442	471
7	395	451	474
8	404	401	469
9	413	410	472
10	422	419	471
11	431	428	474
12	440	437	471
13	449	446	474
14	458	455	473
15	467	464	476
16	476	414	481
17	485	423	484
18	494	432	483
19	503	441	486
20	512	450	483
21	521	459	486
22	530	468	485
23	539	477	488
24	548	427	483

The third option, when `OS_SEARCH_FAST` is set to 4, never achieves a lower CPU usage than when `OS_SEARCH_FAST` is set to 0 or 1 for about 17 priority levels. When `OS_SEARCH_FAST` is set to zero, each extra priority level to traverse requires exactly 9 CPU cycles. When `OS_SEARCH_FAST` is set to 1, each extra priority level to traverse also requires exactly 9 CPU cycles, except when the priority level is an exact multiple of 8; then there is a sharp reduction of CPU usage. When the next ready to run priority is less than 8, 16, 24, … then there is an extra 13 cycles needed, but without the 8 times 9 cycles accumulation.

The key observation, when looking at this table, is that the first option (`OS_SEARCH_FAST` set to 0) delivers better CPU performance than the second option (`OS_SEARCH_FAST` set to 1) when the search spans less than 8 priority levels. So, if an application has tasks spanning less than 8 priority levels, the build option `OS_SEARCH_FAST` should be set to 0. If an application has tasks spanning much more than 8 priority levels, the build option `OS_SEARCH_FAST` should be set to 1.

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, but 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 chip support is provided with the distribution code because the controlSUITE software library is made available for free by Texas Instruments, and this library includes a high level API for all the peripherals on the C28X devices.

6.1 OSmapISR

The standard component `OSisrInstall()` is not available for the C28X port. Instead the `OSmapISR()` C28X specific component must be used. The following section describes the usage of `OSmapISR()`.

6.1.1 OSmapISR()

Synopsis

```
#include "Abassi.h"

void OSmapISR(int VectID, int TblNmb, void (*FctPtr)(void));
```

Description

`OSmapISR()` is the component used on the C28X in replacement for the standard component `OSisrInstall`(). `OSmapISR()` attaches the interrupt handler function `FctPtr` to an interrupt. The interrupt is indicated by the argument `VectID`, which is the vector ID in the PIE when the PIE is used, or the index in the interrupt vector table when the PIE is not used. When the argument `TblNmb` is negative, the interrupt function handler is attached as a fast interrupt. When the argument `TblNmb` is non-negative and the PIE is used with the pre-handlers, `TblNmb` specifies the ISR dispatcher table look-up index to use for the interrupt. For all other cases the argument `TblNmb` is ignored.

Availability

C28X port only

Arguments

`VectID` PIE used: vector ID of the interrupt to attach the function (`FctPtr`) to.
PIE not used: index in the interrupt vector table to attach the function to.

`TblNmb` When negative (`TblNmb<0`) the function to attach (`FctPtr`) is attached as a fast interrupt handler.
When non-negative (`TblNmb>=0`):

PIE not used:	ignored:
PIE without pre-handlers;	ignored
PIE with pre-handlers:	index in ISR dispatcher look-up table to use

`FctPtr` Pointer to the function to attach to the interrupt specified by the argument `VectID`.

Returns

`void`

Component type

Function

Options

Notes

Fast interrupt function handlers must be specified with the `interrupt` "C" keyword. If the fast interrupt handler is written in assembly language, the exit of the function must be done through the `iret` instruction. When the interrupt function handler is for a regular interrupt, the "C" keyword `interrupt` must not be used.

See also

Interrupt description (Section 3)

7 Measurements

This section gives an overview of the memory requirements and the CPU latency encountered when the RTOS is used on the C28X and compiled with Code Composer Studio. The CPU cycles are exactly the CPU clock cycles, not a conversion from a duration measured on an oscilloscope then converted to a number of cycles.

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.

NOTE: The memory sizes are specified in bytes, not words (2 bytes), even though the C28X is a word-based processor.

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

1. Optimization level: 3[2]
2. Optimize for code size: Enabled

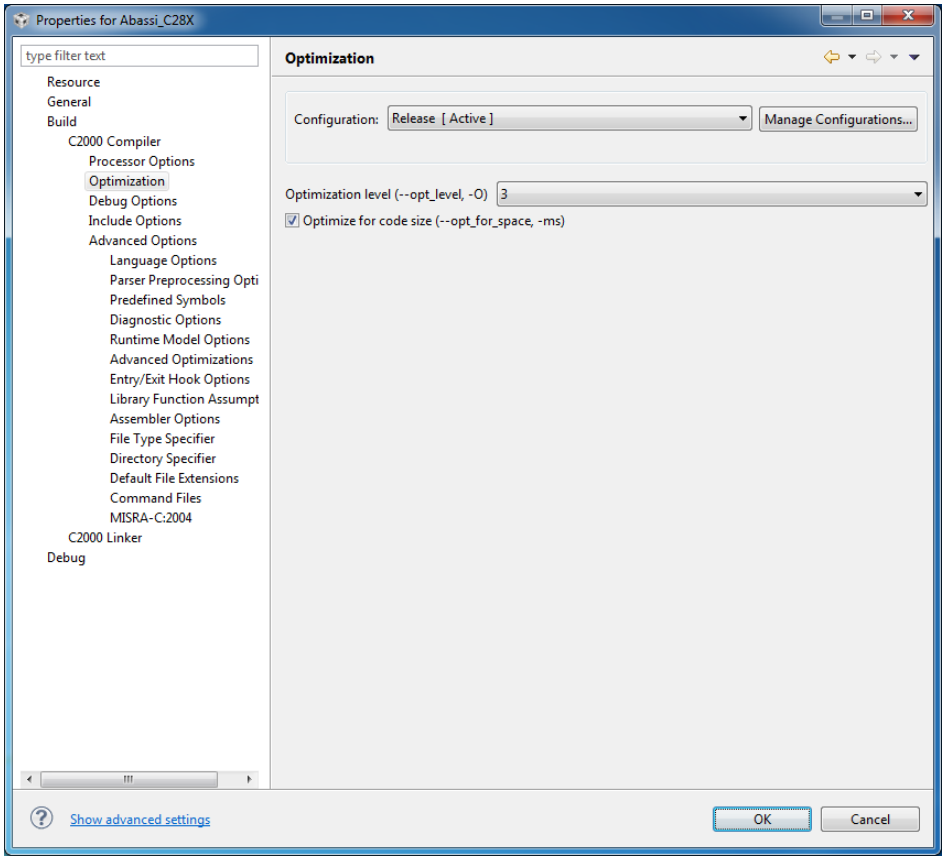

Figure 7-1 Memory Measurement Code Optimization Settings

[2] The highest optimization level on Code Composer is 4, but level 4 adds linker optimization over what optimization level 3 does. The linker optimization is not used for the memory measurements as it converts small function into in-line operations, removing these functions from the memory map, skewing the memory sizing measurements.

Table 7-1 "C" Code Memory Usage

Description	Code Size
Minimal Build	< 800 bytes
+ Runtime service creation / static memory	< 1050 bytes
+ Multiple tasks at same priority	< 1175 bytes
+ Runtime priority change + Mutex priority inheritance + FCFS + Task suspension	< 1700 bytes
+ Timer & timeout + Timer call back + Round robin	< 2325 bytes
+ Events + Mailbox	< 3050 bytes
Full Feature Build (no names)	< 3650 bytes
Full Feature Build (no name / no runtime creation)	< 3275 bytes
Full Feature Build (no names / no runtime creation) + Timer services	< 3900 bytes

Table 7-2 Added features

Description	Size
Library mutex protection	74 bytes

Table 7-3 Assembly Code Memory Usage

Description	Size
Assembly code size (OS_N_INTERRUPTS==0) Interrupt pre-handlers (32 of them)	292 bytes +256 bytes
Assembly code size (OS_N_INTERRUPTS>=128)	306 bytes
Assembly code size (0<OS_N_INTERRUPTS<128) Interrupt pre-handlers (per interrupt)	302 bytes +8 bytes
Hybrid Stack Enabled	+20 bytes
Interrupt nesting enable	+4 bytes
Status Register Preservation	+8 bytes
FPU in use	+76 bytes

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

7.2 Latency

Latency of operations has been measured on an Olimex TMX320-P28027 Evaluation board populated with a 60 MHz TMS320F28027 device. All measurements have been performed on the real platform, using the timer peripheral `CPU-TIMER2` set-up to be clocked at the same rate as the CPU. This means the interrupt latency measurements had to be instrumented to read the `CPU-TIMER2` counter value. This instrumentation can add many cycles to the measurements. The code optimization settings used for the latency measurements are:

1. Optimization level: 4
2. Optimize for code size: Disabled

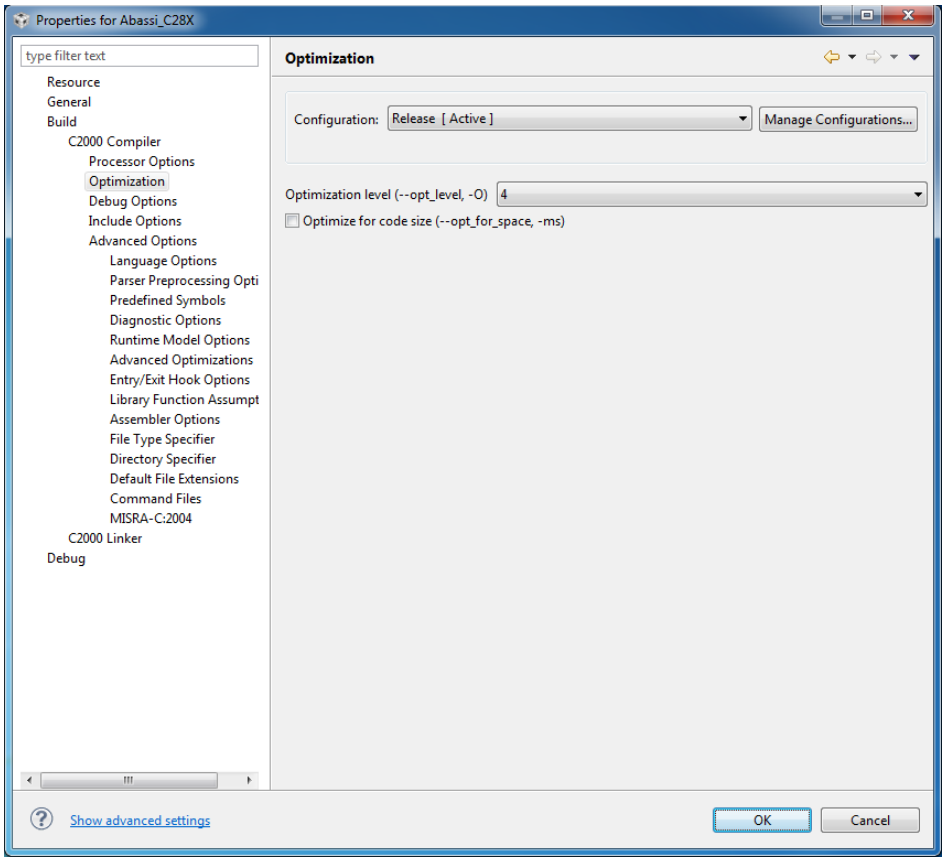

Figure 7-2 Latency Measurement Code Optimization Settings

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

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

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

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

Table 7-4 Measurement without Task Switch

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

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

Table 7-5 Measurement without Blocking

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

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

Table 7-6 Measurement with Task Switch

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

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

The 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 7-7 Measurement with Task unblocking

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

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

The fifth set of measurements counts the number of CPU cycles elapsed from the beginning of an interrupt using the component, until the task that was blocked becomes the running task and is back from the component used that blocked the task. 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 `OSmapISR()`, is simply a two line function that uses the appropriate RTOS component followed by a return.

The following tables list the results obtained, where the cycle count is measured using the `CPU-TIMER2` peripheral on the C28X. This timer is configured to increment its counter by 1 at every CPU cycle. As was the case for the memory measurements, these numbers were obtained with release version 1.122.205 of the RTOS. It is possible another released version of the RTOS may have slightly different numbers.

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. This measurement includes the call to and the return from the context switch function.

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 `CPU-TIMER1` 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 `OSmapISR()`. 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 `ST1` status register preserved, nor the interrupt nesting, in any of these tests; `OS_N_INTERRUPTS` was set to 128, meaning the ISR dispatcher pre-handlers are not used.

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

Table 7-8 Latency Measurements

Description	Minimal Features	Full Features
Semaphore posting no task switch	156 (161)	251 (257)
Semaphore waiting no blocking	174 (179)	276 (278)
Semaphore posting with task switch	247 (297)	447 (482)
Semaphore waiting with blocking	278 (262)	521 (509)
Semaphore posting in ISR with task switch	418 (462)	623 (656)
Event setting no task switch	n/a	240 (236)
Event getting no blocking	n/a	300 (300)
Event setting with task switch	n/a	459 (492)
Event getting with blocking	n/a	549 (533)
Event setting in ISR with task switch	n/a	637 (666)
Mailbox writing no task switch	n/a	330 (327)
Mailbox reading no blocking	n/a	345 (351)
Mailbox writing with task switch	n/a	536 (583)
Mailbox reading with blocking	n/a	589 (580)
Mailbox writing in ISR with task switch	n/a	730 (762)
Interrupt Latency	32	32
Interrupt overhead entering the kernel	171 (165)	176 ()
Interrupt overhead NOT entering the kernel	75	75
Context switch	42	42

NOTE: The CPU numbers involving a task switch triggered by an interrupt are dependent on which interrupt is used for the test, as the sequence of operations to acknowledge an interrupt is different depending on the source of the interrupt. Therefore these numbers should be not considered as valid for all cases.

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                          */
```