

CODE TIME TECHNOLOGIES

Abassi RTOS

Porting Document
MSP430/X – CCS

Copyright Information

This document is copyright Code Time Technologies Inc. ©2011,2012. All rights reserved. No part of this document may be reproduced or distributed in any form by any means, or stored in a database or retrieval system, without the written permission of Code Time Technologies Inc.

Code Time Technologies Inc. may have patents or pending applications covering the subject matter in this document. The furnishing of this document does not give you any license to these patents.

Disclaimer

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

MSP430 and Code Composer Studio are registered trademarks of Texas Instruments. All other trademarks are the property of their respective owners.

Table of Contents

1	INTRODUCTION	7
1.1	DISTRIBUTION CONTENTS	7
1.2	LIMITATIONS	7
2	TARGET SET-UP	8
2.1	INTERRUPT STACK SET-UP	9
2.2	INTERRUPT NESTING	10
2.3	OSCILLATOR CONTROL BITS PROPAGATION	12
2.4	INTERRUPT VECTOR TABLE	13
2.5	COMPILER OPTIONS	15
2.5.1	<i>Option –global_register=r4</i>	15
2.5.2	<i>Option –global_register=r5</i>	17
2.5.3	<i>.bss</i>	18
3	INTERRUPTS	19
3.1	INTERRUPT HANDLING	19
3.2	UNUSED INTERRUPTS	20
3.3	FAST INTERRUPTS	21
3.4	NESTED INTERRUPTS	25
4	STACK USAGE.....	26
5	SEARCH SET-UP	27
6	CHIP SUPPORT	30
7	MEASUREMENTS.....	31
7.1	MEMORY	31
7.2	LATENCY	35
8	APPENDIX A: BUILD OPTIONS FOR CODE SIZE	43
8.1	CASE 0: MINIMUM BUILD	43
8.2	CASE 1: + RUNTIME SERVICE CREATION / STATIC MEMORY	44
8.3	CASE 2: + MULTIPLE TASKS AT SAME PRIORITY	45
8.4	CASE 3: + PRIORITY CHANGE / PRIORITY INHERITANCE / FCFS / TASK SUSPEND	46
8.5	CASE 4: + TIMER & TIMEOUT / TIMER CALL BACK / ROUND ROBIN	47
8.6	CASE 5: + EVENTS / MAILBOXES	48
8.7	CASE 6: FULL FEATURE BUILD (NO NAMES)	49
8.8	CASE 7: FULL FEATURE BUILD (NO NAMES / NO RUNTIME CREATION)	50
8.9	CASE 8: FULL BUILD ADDING THE OPTIONAL TIMER SERVICES	51

List of Figures

FIGURE 2-1 PROJECT FILE LIST	8
FIGURE 2-2 GUI SET OF OS_ISR_STACK	10
FIGURE 2-3 GUI SET OF OS_NESTED_INTS.....	11
FIGURE 2-4 GUI SET OF OS_HANDLE_OSC.....	13
FIGURE 2-5 GUI SET OF OS_INT_VECT_SIZE.....	15
FIGURE 2-6 GUI SET OF OS_OPT_GLOBAL_R4	16
FIGURE 2-7 GUI SET OF OS_OPT_GLOBAL_R5	18
FIGURE 7-1 MEMORY MEASUREMENT CODE OPTIMIZATION SETTINGS	32
FIGURE 7-2 LATENCY MEASUREMENT CODE OPTIMIZATION SETTINGS.....	35

List of Tables

TABLE 1-1 DISTRIBUTION	7
TABLE 2-1 ASSEMBLER REQUIRED OPTIONS	8
TABLE 2-2 INTERRUPT STACK ENABLED	9
TABLE 2-3 INTERRUPT STACK DISABLED	9
TABLE 2-4 COMMAND LINE SET OF OS_ISR_STACK	9
TABLE 2-5 NESTED INTERRUPTS ENABLED.....	10
TABLE 2-6 NESTED INTERRUPTS DISABLED.....	11
TABLE 2-7 COMMAND LINE SET OF OS_NESTED_INTS	11
TABLE 2-8 OSCILLATOR BITS NOT PROPAGATED	12
TABLE 2-9 OSCILLATOR BITS PROPAGATED	12
TABLE 2-10 COMMAND LINE SET OF OS_HANDLE_OSC.....	12
TABLE 2-11 INTERRUPT VECTOR WITH 16 ENTRIES	13
TABLE 2-12 INTERRUPT VECTOR WITH 32 ENTRIES	14
TABLE 2-13 INTERRUPT VECTOR WITH 64 ENTRIES	14
TABLE 2-14 COMMAND LINE SET OF OS_INT_VECT_SIZE.....	14
TABLE 2-15 R4 USED AS A GLOBAL REGISTER.....	15
TABLE 2-16 R4 NOT USED AS A GLOBAL REGISTER.....	16
TABLE 2-17 COMMAND LINE SET OF OS_OPT_GLOBAL_R4	16
TABLE 2-18 R5 USED AS A GLOBAL REGISTER.....	17
TABLE 2-19 R5 NOT USED AS A GLOBAL REGISTER.....	17
TABLE 2-20 COMMAND LINE SET OF OS_OPT_GLOBAL_R5	17
TABLE 3-1 ATTACHING A FUNCTION TO AN INTERRUPT	19
TABLE 3-2 ATTACHING A FUNCTION TO AN INTERRUPT	19
TABLE 3-3 INVALIDATING AN ISR HANDLER.....	20
TABLE 3-4 ENTRY IN THE INTERRUPT VECTOR TABLE	20
TABLE 3-5 UNUSED INTERRUPT VECTOR ENTRY.....	20
TABLE 3-6 DO-NOTHING INTERRUPT HANDLER	21
TABLE 3-7 INTERRUPT DISPATCHER PROLOGUE	21
TABLE 3-8 INTERRUPT DISPATCHER PROLOGUE REMOVAL.....	21
TABLE 3-9 MSP430F5438 TIMERA REGULAR INTERRUPT	21
TABLE 3-10 MSP430F5438 TIMERA FAST INTERRUPT.....	21
TABLE 3-11 MSP430X FAST INTERRUPT WITH DEDICATED STACK.....	23
TABLE 3-12 MSP430 FAST INTERRUPT WITH DEDICATED STACK.....	24
TABLE 4-1 CONTEXT SAVE STACK REQUIREMENTS	26
TABLE 5-1 SEARCH ALGORITHM CYCLE COUNT	28
TABLE 7-1 “C” CODE MEMORY USAGE	33
TABLE 7-2 ADDED FEATURES.....	33
TABLE 7-3 ASSEMBLY CODE MEMORY USAGE	34
TABLE 7-4 MEASUREMENT WITHOUT TASK SWITCH.....	36
TABLE 7-5 MEASUREMENT WITHOUT BLOCKING	36
TABLE 7-6 MEASUREMENT WITH TASK SWITCH	36
TABLE 7-7 MEASUREMENT WITH TASK UNBLOCKING.....	37
TABLE 7-8 LATENCY MEASUREMENTS (MSP430)	38
TABLE 7-9 LATENCY MEASUREMENTS (SMALL CODE & SMALL DATA).....	39
TABLE 7-10 LATENCY MEASUREMENTS (LARGE CODE / SMALL DATA).....	40
TABLE 7-11 LATENCY MEASUREMENTS (LARGE CODE / RESTRICTED DATA)	41
TABLE 7-12 LATENCY MEASUREMENTS (LARGE CODE / LARGE DATA)	42
TABLE 8-1: CASE 0 BUILD OPTIONS	43
TABLE 8-2: CASE 1 BUILD OPTIONS	44
TABLE 8-3: CASE 2 BUILD OPTIONS	45
TABLE 8-4: CASE 3 BUILD OPTIONS	46
TABLE 8-5: CASE 4 BUILD OPTIONS	47
TABLE 8-6: CASE 5 BUILD OPTIONS	48

TABLE 8-7: CASE 6 BUILD OPTIONS49
TABLE 8-8: CASE 7 BUILD OPTIONS50
TABLE 8-9: CASE 8 BUILD OPTIONS51

1 Introduction

This document details the port of the Abassi RTOS to the MSP430 processor, with and without extended addressing; the processor family with extended addressing is known as the MSP430X. The software suite used for this specific port is the Texas Instruments Code Composer Studio for the MSP430; the version used for the port and all tests is Version 5.1.1.00031.

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_MSP430_CCS.s	RTOS assembly file for the MSP430 and MSP430X to use with Code Composer Studio
Demo_2X_MSP430_CCS.c	Demo code that runs on the Olimex MSP-5438STK evaluation board using the LCD
Demo_3_MSP430_CCS.c	Demo code that runs on the Olimex MSP430-P1611 evaluation board using the serial port
Demo_3X_MSP430_CCS.c	Demo code that runs on the Olimex MSP-5438STK evaluation board using the serial port
Demo_4X_MSP430_CCS.c	Demo code that runs on the Olimex MSP-5438STK evaluation board using joystick and the serial port
AbassiDemo.h	Build option settings for the demo code

1.2 Limitations

The Abassi port for the MSP430 and MSP430X with Code Composer Studio does not support the MSP430 EABI, set using the compiler/assembler/linker option `-abi=eabi`. Therefore compiler/assembler/linker option `-abi=eabi` should never be used with an application built with Abassi.

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_MSP430_CCS.s` in the source files of the application project, and make sure the configuration settings (described in the following subsections) in the file `Abassi_MSP430_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`.

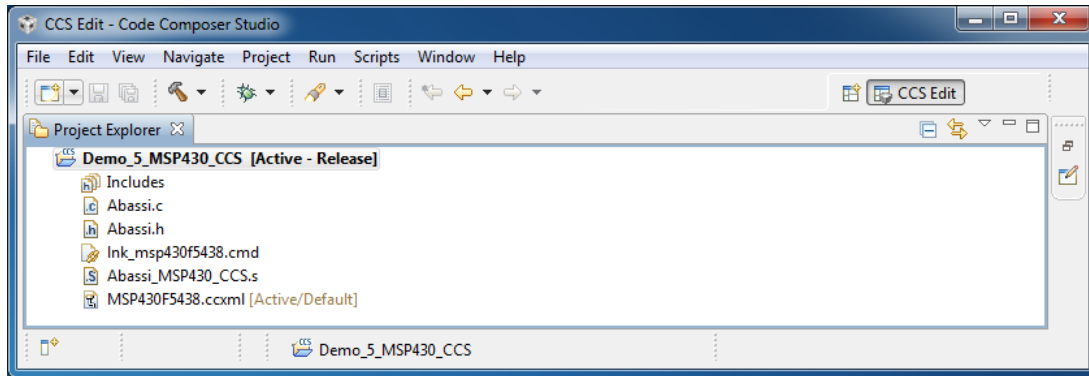


Figure 2-1 Project File List

If the application is not built through 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_MSP430_CCS.s` relies on command line options to generate the correct code. The options the file `Abassi_MSP430_CCS.s` requires are listed in the following table:

Table 2-1 Assembler required options

Option	Description
<code>-abi=coffabi</code>	Only the coff abi is supported by this port
<code>--code_model=XXX</code>	Code memory model: small or large
<code>--data_model=XXX</code>	Data memory model: small, restricted or large
<code>--define=__MSP430FXXX__</code>	The target device part number

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 applied in the file `Abassi.h` by attaching the Abassi internal mutex (`G_OSmutex`) during runtime in `OSstart()`.

NOTE: This applies when coding in assembly language:

If a hardware multiplier is available on the target device, the use of the multiplier must always be protected by disabling/enabling the interrupts. This is true even when the multiplier is not accessed inside an interrupt. The reason is that one or many task switches may be triggered by an interrupt. So, if the preemption of a task occurs when it is in the process of using the multiplier, and a newly running task also uses the multiplier, the multiplication result for the pre-empted task will be erroneous.

There is no possibility for the RTOS to protect the multiplier, as the operation to perform is set when the first operand is written to the desired “operation” register; there is no way to know which of the “operation” registers was written last, specifying the type of operation, therefore the RTOS cannot protect the multiplier registers.

The Code Composer Studio compiler generates code that protects the multiplier when using it.

2.1 Interrupt Stack Set-up

It is possible, and is highly recommended, to use a hybrid stack when nested interrupts occur in an application. Using this hybrid stack, specially dedicated to the interrupts, removes the need to allocate extra room to the stack of every task in the application to handle the interrupt nesting. This feature is controlled by the value set by the definition `OS_ISR_STACK`, located around line 30 in the file `Abassi_MSP430_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 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 128 bytes is allocated; this is shown in the following table:

Table 2-2 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 bytes
```

Table 2-3 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 bytes
```

Alternatively, it is possible to overload the `OS_ISR_STACK` value set in `Abassi_MSP430_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 bytes:

Table 2-4 Command line set of `OS_ISR_STACK`

```
c1430 ... -asm_define=OS_ISR_STACK=256 ...
```

The hybrid stack size can also be set through the GUI, in the “*Build / MSP430 Compiler / Advanced Options / Assembler Options / Pre-define assembly symbol NAME*” menu, as shown in the following figure:

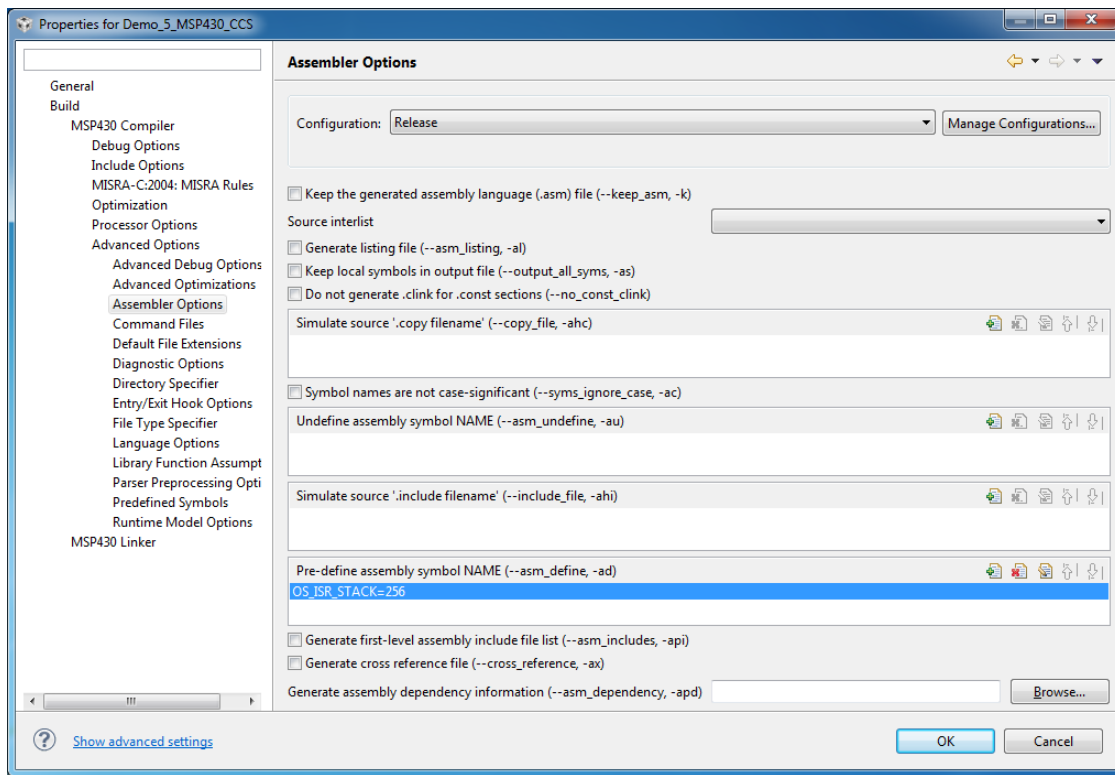


Figure 2-2 GUI set of OS_ISR_STACK

2.2 Interrupt Nesting

The normal operation of the interrupt controller on the MSP430 / MSP430X families 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 serviced interrupt level, remain pending until the processor finishes servicing the current interrupt. The Abassi interrupt dispatcher allows the nesting of interrupts; this means an interrupt of any priority can interrupt the processing of an interrupt currently being handled. Nested interrupts are enabled by setting both the build option `OS_NESTED_INTS` and the token `OS_NESTED_INTS` in the file `Abassi_MSP430_CCS.s`, around line 30, to a non-zero value, as shown in the following table:

Table 2-5 Nested Interrupts enabled

```

.if !($defined(OS_NESTED_INTS))
OS_NESTED_INTS    .equ    1    ; To allow interrupt nesting, set to non-zero
.endif            ; To not allow interrupt nesting, set to zero

```

Interrupt nesting is disabled (in other words, the interrupts operate exactly as the MSP430 / MSP430X interrupt controller operates) by setting both the build option `OS_NESTED_INTS` and the token `NESTED_INTS` to a zero value, as shown in the following table:

Table 2-6 Nested Interrupts disabled

```
.if !($defined(OS_NESTED_INTS))
OS_NESTED_INTS      .equ      0      ; To allow interrupt nesting, set to non-zero
.endif              ; To not allow interrupt nesting, set to zero
```

Alternatively, it is possible to overload the `OS_NESTED_INTS` value set in `Abassi_MSP430_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 option `--asm_define` must also be used. The following example shows the activation of the nesting for the interrupts:

Table 2-7 Command line set of `OS_NESTED_INTS`

```
cl430 ... -asm_define=OS_NESTED_INTS=1 ...
```

The interrupt nesting control can also be set through the GUI, in the “*Build / MSP430 Compiler / Advanced Options / Assembler Options / Pre-define assembly symbol NAME*” menu, as shown in the following figure:

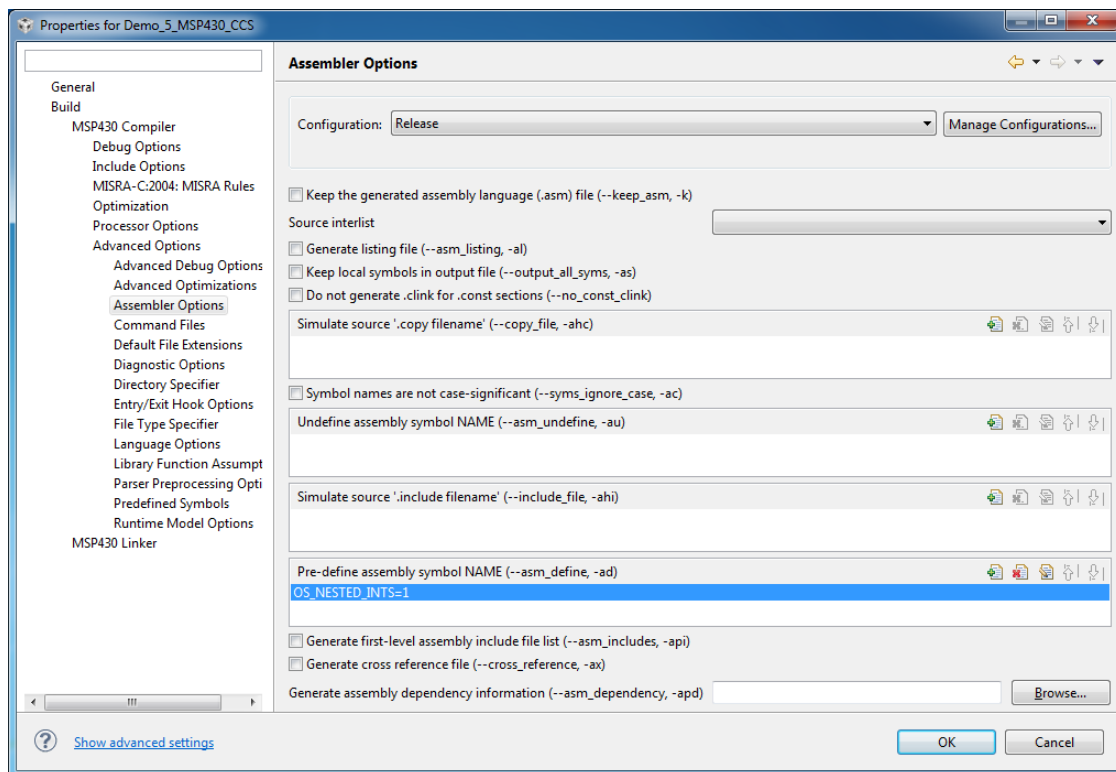


Figure 2-3 GUI set of `OS_NESTED_INTS`

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_MSP430_CCS.s` is set to a non-zero value. If the token `OS_NESTED_INTS` in the file `Abassi_MSP430_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.

2.3 Oscillator control bits propagation

In the MSP430 / MSP430X status register, there are 3 bits that control the oscillators on the device. If any of these bits is modified after the interrupts are enabled in the application, the change must be propagated across all tasks and interrupts. This feature is controlled by the value set by the definition `OS_HANDLE_OSC`, located around line 35 in the file `Abassi_MSP430_CCS.s`. To disable this feature, set the definition of `OS_HANDLE_OSC` to a value of zero. To enable it, set the definition of `OS_HANDLE_OSC` to a non-zero value. As supplied in the distribution, the oscillator control bits propagation is disabled; this is shown in the following table:

Table 2-8 Oscillator bits not propagated

```
.if !($defined(OS_HANDLE_OSC))
OS_HANDLE_OSC      .equ      0      ; Set to non-zero to propagate oscillator control bits
                              ; in SR from ISR to the background / tasks
.endif
```

Table 2-9 Oscillator bits propagated

```
.if !($defined(OS_HANDLE_OSC))
OS_HANDLE_OSC      .equ      1      ; Set to non-zero to propagate oscillator control bits
                              ; in SR from ISR to the background / tasks
.endif
```

Alternatively, it is possible to overload the `OS_HANDLE_OSC` value set in `Abassi_MSP430_CCS.s` by using the assembler command line option `--asm_define` (or `-ad`) and specifying the desired propagation mode. In the following example, the oscillator bits are propagated:

Table 2-10 Command line set of `OS_HANDLE_OSC`

```
c1430 ... -asm_define=OS_HANDLE_OSC=1 ...
```

The propagation of the oscillator bits can also be controlled through the GUI, in the “*Build / MSP430 Compiler / Advanced Options / Assembler Options / Pre-define assembly symbol NAME*” menu, as shown in the following figure:

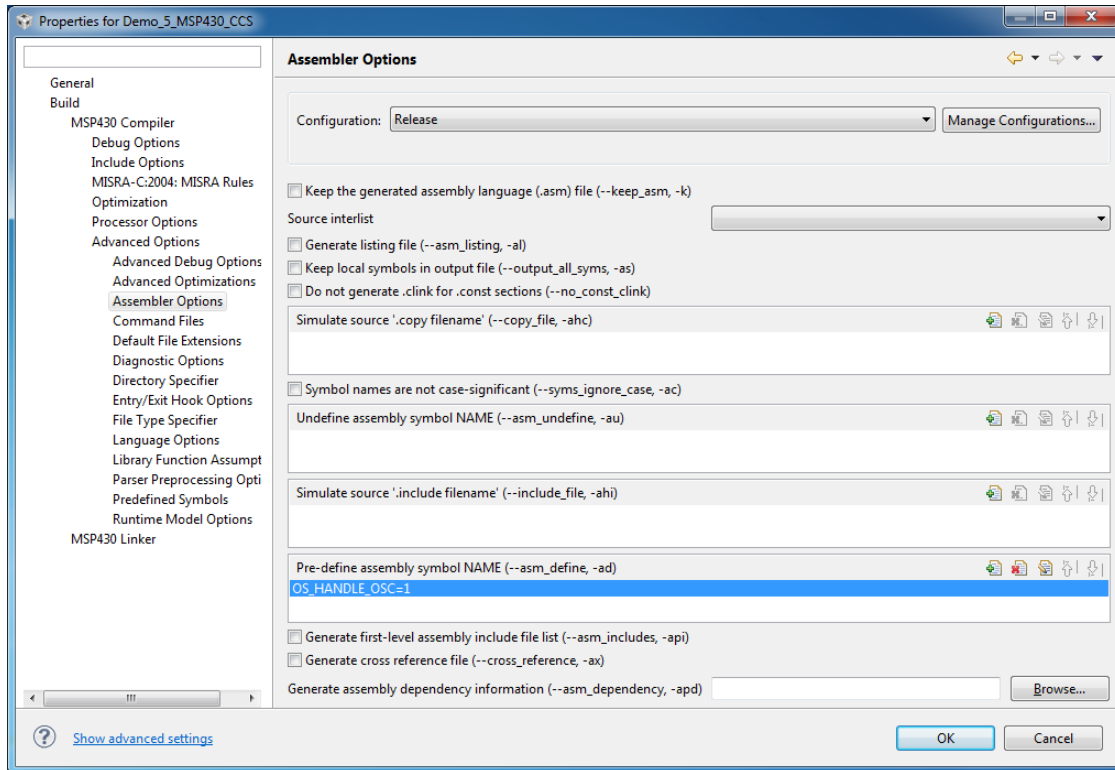


Figure 2-4 GUI set of OS_HANDLE_OSC

2.4 Interrupt vector table

There are three different flavors for the MSP430 / MSP430X interrupt table: some devices have a table capable of handling up to 16 interrupts sources, others have room for 32 interrupt sources, and others can deal with 64 interrupt sources. Abassi can support all three, but it must be configured to the correct size in order to optimize the code footprint and properly map the interrupt priority to the interrupt vector table entry. The information must be set in the file `Abassi_MSP430_CCS.s` around line 45; the token `OS_INT_VECT_SIZE` must be set to 16, for a 16 entry table, to 32, for a 32 entry table, or set to 64 for a 64 entry table:

Table 2-11 Interrupt vector with 16 entries

```
.if !($defined(OS_INT_VECT_SIZE))
OS_INT_VECT_SIZE .equ 16 ; Number of interrupts in the interrupt vector table
.endif ; Should be either 16 / 32 / 64
```

Table 2-12 Interrupt vector with 32 entries

```
.if !($defined(OS_INT_VECT_SIZE))
OS_INT_VECT_SIZE .equ 32 ; Number of interrupts in the interrupt vector table
.endif ; Should be either 16 / 32 / 64
```

Table 2-13 Interrupt vector with 64 entries

```
.if !($defined(OS_INT_VECT_SIZE))
OS_INT_VECT_SIZE .equ 64 ; Number of interrupts in the interrupt vector table
.endif ; Should be either 16 / 32 / 64
```

Alternatively, it is possible to overload the `OS_INT_VECT_SIZE` value set in `Abassi_MSP430_CCS.s` by using the assembler command line option `--asm_define` (or `-ad`) and specifying the desired interrupt vector table size. In the following example, the interrupt vector table is set to 16 entries:

Table 2-14 Command line set of `OS_INT_VECT_SIZE`

```
cl430 ... -asm_define=OS_INT_VECT_SIZE=16 ...
```

The interrupt vector table size can also be set through the GUI, in the “Build / MSP430 Compiler / Advanced Options / Assembler Options / Pre-define assembly symbol NAME” menu, as shown in the following figure:

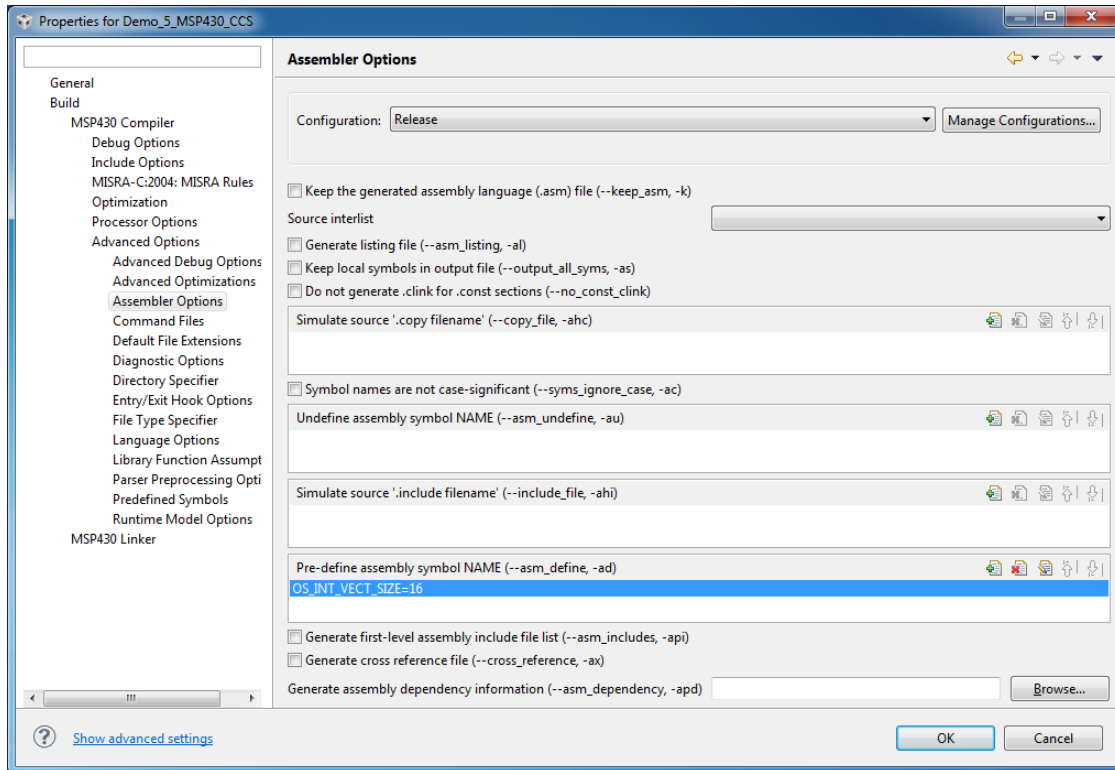


Figure 2-5 GUI set of OS_INT_VECT_SIZE

2.5 Compiler Options

This subsection describes the configuration changes to make in the `Abassi_MSP430_CCS.s` when the application is built enabling some compiler options.

2.5.1 Option `-global_register=r4`

The “C” compiler can be configured to not use the register `r4`, with the option `-global_register=r4`. This affects the implementation of the assembly-coded functions in the file `Abassi_MSP430_CCS.s`. If this option is explicitly specified in the “Runtime Model Option” in the compiler configuration menu (or on the compiler command line), then the token `OS_OPT_global_r4`, defined in the file `Abassi_MSP430_CCS.s` around line 50, must be set to a value of 1 (MUST be one), as shown in the following table:

Table 2-15 R4 used as a global register

```
.if !($defined(OS_OPT_global_r4))
OS_OPT_global_r4 .equ 1 ; Set to != 0 if "C" option -global_register=r4 is not used
.endif ; Set to 1 (ONE!!!) if "C" option -global_register=r4 is used
```

If the compiler is not configured to reserve `r4` as a global variable, leave the token `OS_OPT_global_r4` set to a zero value, as originally supplied in the distribution.

Table 2-16 R4 not used as a global register

```
.if !($defined(OS_OPT_global_r4))
OS_OPT_global_r4 .equ 0 ; Set to != 0 if "C" option -global_register=r4 is not used
.endif ; Set to 1 (ONE!!!) if "C" option -global_register=r4 is used
```

Alternatively, it is possible to overload the `OS_OPT_global_r4` value set in `Abassi_MSP430_CCS.s` by using the assembler command line option `--asm_define` (or `-ad`) and specifying the desired `r4` register mode. In the following example, the register `r4` is used as a global variable:

Table 2-17 Command line set of OS_OPT_global_r4

```
cl1430 ... -asm_define=OS_OPT_global_r4=1 ...
```

The usage of `r4` can also be set through the GUI, in the “*Build / MSP430 Compiler / Advanced Options / Assembler Options / Pre-define assembly symbol NAME*” menu, as shown in the following figure:

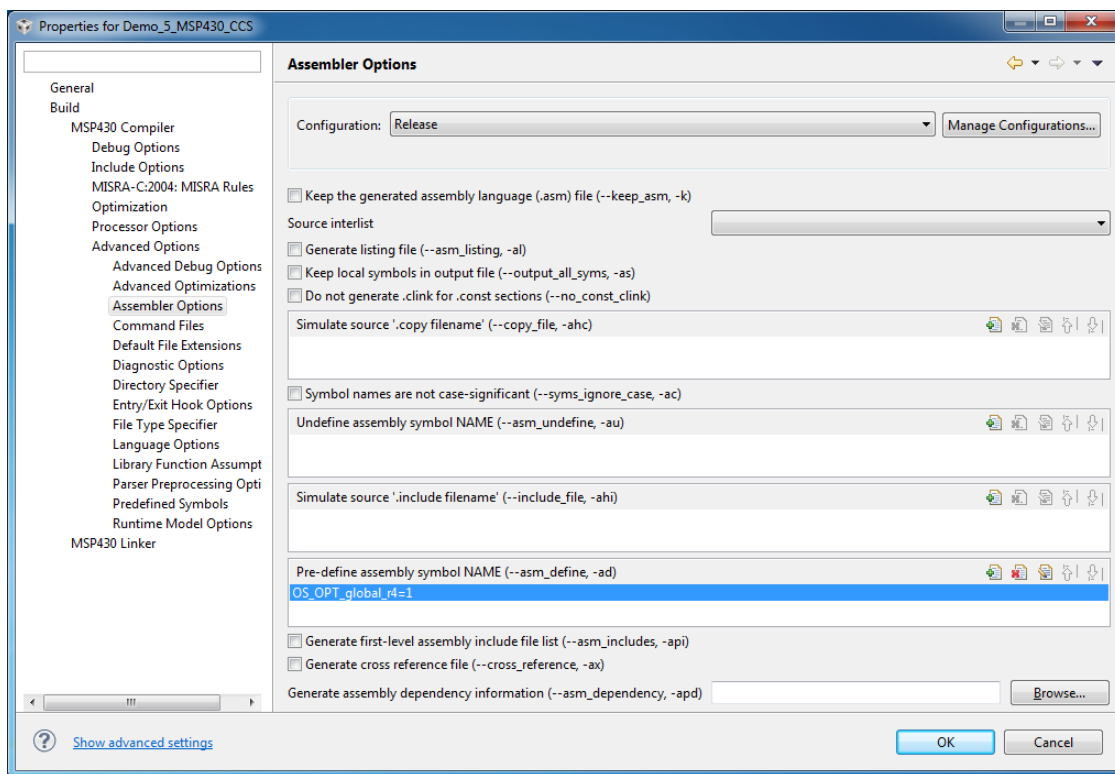


Figure 2-6 GUI set of OS_OPT_global_r4

2.5.2 Option `-global_register=r5`

The “C” compiler can be configured to not use the register `r5`, with the option `-global_register=r5`. This affects the implementation of the assembly-coded functions in the file `Abassi_MSP430_CCS.s`. If this option is explicitly specified in the “*Runtime Model Option*” in the compiler configuration menu (or on the compiler command line), then the token `OS_OPT_global_r5`, defined in the file `Abassi_MSP430_CCS.s` around line 55, must be set to a value of 1 (MUST be one), as shown in the following table:

Table 2-18 R5 used as a global register

```
.if !($defined(OS_OPT_global_r5))
OS_OPT_global_r5 .equ 1 ; Set to != 0 if "C" option -global_register=r5 is not used
#endif                ; Set to 1 (ONE!!!) if "C" option -global_register=r5 is used
```

If the compiler is not configured to reserve `r5` as a global variable, leave the token `OS_OPT_global_r5` set to a zero value, as originally supplied in the distribution.

Table 2-19 R5 not used as a global register

```
.if !($defined(OS_OPT_global_r5))
OS_OPT_global_r5 .equ 0 ; Set to != 0 if "C" option -global_register=r5 is not used
#endif                ; Set to 1 (ONE!!!) if "C" option -global_register=r5 is used
```

Alternatively, it is possible to overload the `OS_OPT_global_r5` value set in `Abassi_MSP430_CCS.s` by using the assembler command line option `--asm_define` (or `-ad`) and specifying the desired `r5` register mode. In the following example, the register `r5` is used as a regular register by the compiler:

Table 2-20 Command line set of `OS_OPT_global_r5`

```
cl430 ... -asm_define=OS_OPT_global_r5=0 ...
```

The usage of `r5` can also be set through the GUI, in the “*Build / MSP430 Compiler / Advanced Options / Assembler Options / Pre-define assembly symbol NAME*” menu, as shown in the following figure:

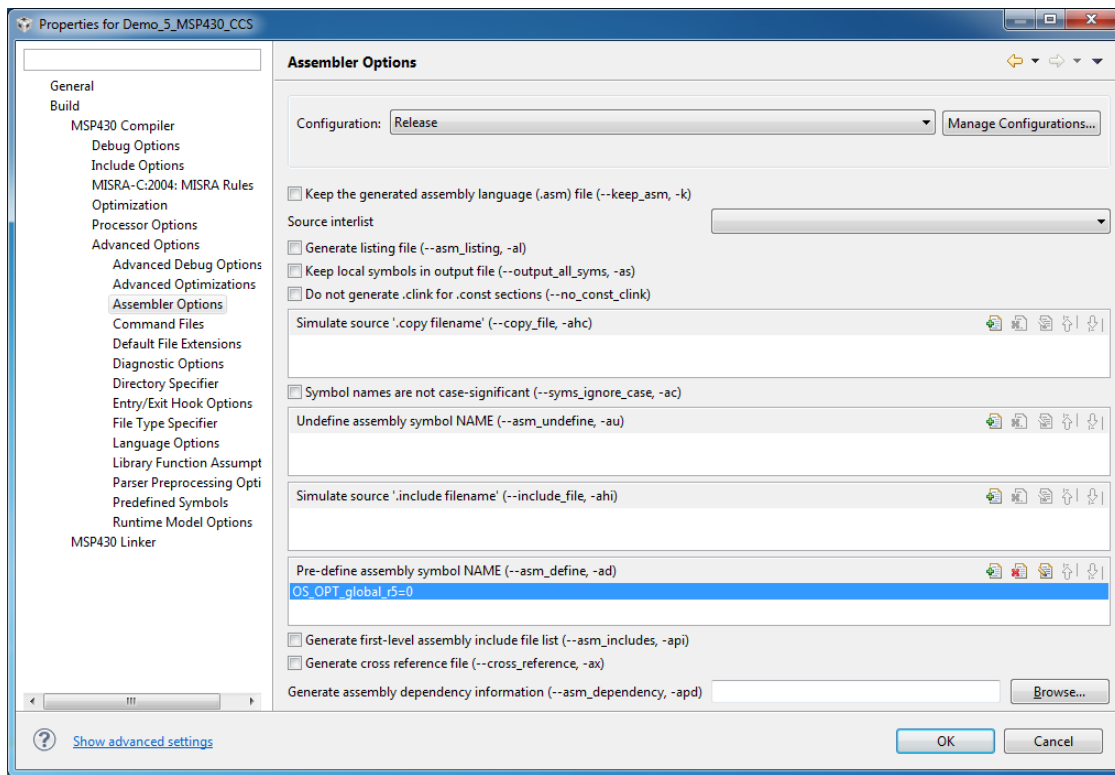


Figure 2-7 GUI set of `OS_OPT_global_r5`

2.5.3 .bss

As indicated in the Code Composer C / C++ compiler User’s Manual, the object file format COFF does not initialize to zero the contents of the `.bss` area. Having uninitialized data with random values at start-up is perfectly compliant with the C-99 standard. But, for the sole purpose of reducing the code size of the RTOS, all the `.bss` area is set to zero upon start-up using the Code Composer Studio runtime hook `_system_pre_init()`. This runtime hook was described in previous versions of the compiler User’s Manual, but starting with version 4.0, this information has disappeared, even though the standard start-up code still supports it.

As a bonus for an application built with Abassi, because the whole `.bss` area is initialized to zero, this means that all uninitialized variables in the application have a guaranteed value of zero at start-up.

3 Interrupts

The Abassi RTOS needs to be aware when kernel requests are performed inside or outside an interrupt context. Normally, when coding with Code Composer Studio, an interrupt function is specified with the `interrupt` directive. But for all interrupt sources (except for the reset), 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.

The distribution makes provision for 64 sources of interrupts, as specified by the token `OS_INT_VECT_SIZE`, in the file `Abassi_MSP430_CCS.s`. If the target device uses a 16 or 32 entry interrupt vector table, consult Section 2.4 to understand how to set Abassi to support the smaller table.

3.1 Interrupt Handling

Attaching a function to an interrupt is quite straightforward. All there is to do is use the RTOS component `OSIsrInstall()` to specify the interrupt priority and the function to be attached to that interrupt priority. For example, Table 3-1 shows the code required to attach the `TIMERA` interrupt (on a MSP430F5438) to the RTOS timer tick handler (`TIMtick`):

Table 3-1 Attaching a Function to an Interrupt

```
#include "Abassi.h"

...
OSstart();
...
OSIsrInstall(54, &TIMtick);
/* Set-up the count reload and enable SysTick interrupt */

... /* More ISR setup */

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

Alternatively, instead of using a hard coded number, the standard definition supplied by the file `mcp430.h` can be used, as shown in Table 3-2:

Table 3-2 Attaching a Function to an Interrupt

```
#include "Abassi.h"
#include <mcp430.h>

...
OSstart();
...
OSIsrInstall(TIMER0_A0_VECTOR, &TIMtick);
/* Set-up the count reload and enable SysTick interrupt */

... /* More ISR setup */

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

NOTE: Code Composer uses different definitions for these tokens compared to the IAR and GCC compilers. The latter two require a division by 2 of the definition, when Code Composer does not.

NOTE: The function to attach to an interrupt is a regular function, not one declared with the Code Composer Studio specific `interrupt` prefix statement.

NOTE: `OSIsrInstall()` uses the interrupt priority number. As an example, the non-maskable interrupt has a priority of 14 when the device uses a table of 16 interrupt, a value of 30 when the device uses a table of 32 interrupts, and 62 when the device uses a table of 64 interrupts.

At start-up, once `OSstart()` has been called, all 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 3-3:

Table 3-3 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.

3.2 Unused interrupts

The assembly file `Abassi_MSP430_CCS.s`, as supplied in the distribution, includes the prologue code for the interrupt dispatcher for all sources of interrupts. If the code memory space is becoming a bit short, removing the prologue for unused interrupts will help recover memory from that dead code.

Removing the interrupt dispatcher prologue for an unused interrupt is a three-step process. First, the unused interrupt vector must be replaced in the interrupt vector table. This table is located at around line 300, at the label `VectTbl`, and each interrupt entry is defined as shown in the following:

Table 3-4 Entry in the interrupt vector table

```
.sect .intXX
.word INTXX_handler ; Priority XX interrupt
```

The desired table entry must be attached to a do-nothing interrupt handler; it is preferable to attach a do-nothing interrupt handler in case of spurious interrupts. To attach the do-nothing interrupt handler, replace the desired vector table entry by the label `INT_NO_handler`, as shown below:

Table 3-5 Unused interrupt vector entry

```
.sect .intXX
.word INT_NO_handler ; Priority XX interrupt
```

The second step is to create the do-nothing interrupt handler. This step only needs to be performed once, as the same do-nothing handler should be re-used for all unused interrupts. The do-nothing interrupt handler

code must be located in the `.text:_isr` section. Therefore, insert the following code right after the end of the definition of the `ISR_PROLOGUE` macro, right before the `INT00_handler` label; this should be around line 455 in the file:

Table 3-6 Do-nothing interrupt handler

```
INT_NO_handler:                ; Entry point of the do-nothing ISR handler
    reti                       ; Return from the interrupt
```

The last step is to remove the unused interrupt dispatcher prologue code. Each interrupt has an interrupt dispatcher prologue, where the prologue is always defined as follows:

Table 3-7 Interrupt dispatcher prologue

```
INTXX_handler:
    ISR_PROLOGUE    XX
```

Commenting out the `ISR_PROLOGUE` line for the unused interrupt will remove the prologue code. It is not necessary to remove the label.

Table 3-8 Interrupt dispatcher prologue removal

```
INTXX_handler:
;;;    ISR_PROLOGUE    XX
```

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 that is used by the MSP430 / MSP430X processor. The beginning of the interrupt vector table is located in the file `Abassi_MSP430_CCS.s` around line 300, at the label `VectTbl`. For example, on a MSP430F5438 device, `TIMERA` is set to the priority 54. This is the entry in the table for `TIMERA` in the distribution file:

Table 3-9 MSP430F5438 TIMERA Regular Interrupt

```
.sect    .int54
.word    INT54_handler                ; Priority 54 interrupt
```

To attach a fast interrupt handler to `TIMERA`, assuming the name of the interrupt function to attach is `TIMERA_handler()`, replace the previous line, as shown in the next table:

Table 3-10 MSP430F5438 TIMERA Fast Interrupt

```
.sect    .int54
.global  TIMERA_handler
.word    TIMERA_handler                ; Priority 54 interrupt
```

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

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

NOTE: Fast interrupt handlers must use the Code Composer Studio keyword `interrupt`, or `reti` must be used when a fast interrupt is coded in assembly language.

Even if the hybrid interrupt stack feature is enabled (see Section 2.1), 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. Table 3-11 shows what this would look like, reusing the example of `TIMERA` on a MSP430F5438 device.

Table 3-11 MSP430X Fast Interrupt with Dedicated Stack

```

...
...

.int54
.word    TIMERA_preHandler          ; Priority 54 interrupt

...
...

.text

.global  TIMERA_handler

TIMERA_preHandler:
movx.a  sp, #(TIMERA_stack-4)      ; Memo current sp on the hybrid stack
movx.a  #(TIMERA_stack-4), sp      ; Set sp to the new stack
pushm.a #4, r15                    ; Context save on the hybrid stack

calla   TIMERA_handler              ; Enter the interrupt handler
; MUST BE A REGULAR "C" FUNCTION

popm.a  #4, r15                     ; Context restore
popx.a  sp                           ; Recover original sp
reti                                         ; Exit from the interrupt

...
...

.bss TIMERA_stk, TIMERA_stack_size, 1 ; Room for the fast interrupt stack

TIMERA_stack .equ    (TIMERA_stk+TIMERA_stack_size)
...

```

The same code, with unique labels, must be repeated for each of the fast interrupts. As the use of the hybrid stack creates the prologue-epilogue for the interrupt context, the function called must be a regular “C” function, not one declared with the `interrupt` directive. If the GIE (global interrupt enable) bit in the status register is not set in the interrupt function, and the nesting of interrupts is not allowed (Section 2.2), then the same hybrid stack memory can be re-used, as, by default, the MSP430X interrupt controller only allows the servicing of a single interrupt at any time.

Please note that the previous example is for the large code model. If the small code model is used, the `calla` instruction must be replaced by a `call`.

For the MSP430, using the example of `TIMERA` on a MSP430F1611 device, this would look something like:

Table 3-12 MSP430 Fast Interrupt with Dedicated Stack

```

...
...

.int06
.word    TIMERA_preHandler          ; Priority 6 interrupt

...
...

.text

.global TIMERA_handler

TIMERA_preHandler:
    mov.w sp, #(TIMERA_stack-2)    ; Memo current sp on the hybrid stack
    mov.w #(TIMERA_stack-2), sp    ; Set sp to the new stack
    push.w r15                     ; memo original stack pointer
    push.w r14                     ; Remainder of context save
    push.w r13
    pushw. r12

    call    TIMERA_handler          ; Enter the interrupt handler
                                           ; MUST BE A REGULAR "C" FUNCTION

    pop.w  r12                     ; Context restore
    pop.w  r13
    pop.w  r14
    pop.w  r15                     ; Recover original r15
    pop.w  sp                      ; Recover original sp
    reti                                     ; Exit from the interrupt

...
...

.bss TIMERA_stk, TIMERA_stack_size, 1 ; Room for the fast interrupt stack

TIMERA_stack .equ    (TIMERA_stk+TIMERA_stack_size)
...

```

The same code, with unique labels, must be repeated for each of the fast interrupts. As the use of the hybrid stack creates the prologue-epilogue for the interrupt context, the function called must be a regular “C” function, not one declared with the `interrupt` directive. If the GIE (global interrupt enable) bit in the status register is not set in the interrupt function, and the nesting of interrupts is not allowed (Section 2.2), then the same hybrid stack memory can be re-used, as, by default, the MSP430 interrupt controller only allows the servicing of a single interrupt at any time.

3.4 Nested Interrupts

The interrupt dispatcher allows the nesting of interrupts; nested interrupt means an interrupt of any priority will interrupt the processing of an interrupt currently being serviced. Refer to Section 2.2 for information on how to enable or disable interrupt nesting.

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

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 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 MSP430 / MSP430X, 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 for each of the data memory models:

Table 4-1 Context Save Stack Requirements

Processor	MSP430		MSP430X			
	Code model	Data Model	Small	Large	Restricted	Large
Blocked/Preempted task context save	16	16	16	32	32	32
Interrupt context save (no Hybrid stack)	14	14	14	24	24	24
Interrupt context save (Hybrid stack)	12	12	12	20	20	20

The numbers for the interrupt dispatcher context save include the 4 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 the interrupt handler functions. Finally, add to all this the stack required by the code implementing the task operation.

NOTE: The MSP430 processor needs alignment on 2 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 2 bytes.

If the hybrid interrupt stack (see Section 2.1) 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_MSP430_CCS.s` is set to a non-zero value (Section 2.1).

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 `TIMERA` 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 (MSP430 / MSP430X `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 (`-mf5`). 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.

NOTE: The results for the search have only been conducted using the small code and small data memory model on a MSP430X device (MSP430F5438), as it is assumed the code generated using the two other models should exhibit the same sequences of instructions.

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 239 cycles.

Table 5-1 Search Algorithm Cycle Count

Priority	Linear search	Look-up	Approximation
1	248	305	420
2	259	311	421
3	266	317	438
4	273	323	423
5	280	329	440
6	287	335	441
7	294	341	458
8	301	309	427
9	308	315	444
10	315	321	445
11	322	327	462
12	329	333	447
13	336	339	464
14	343	345	465
15	350	351	482
16	357	319	413
17	364	325	430
18	371	331	431
19	378	337	448
20	385	343	433
21	392	349	450
22	399	355	451
23	406	361	468
24	413	329	437

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 zero or 1. This is understandable, as the MSP430X possesses a barrel shifter, but the number of bits to shift is fixed by the instruction. When `OS_SEARCH_FAST` is set to zero, each extra priority level to traverse requires exactly 7 CPU cycles. When `OS_SEARCH_FAST` is set to 1, each extra priority level to traverse also requires exactly 6 CPU cycles, except when the priority level is an exact multiple of 8; then there is a sharp reduction of CPU usage. Overall, setting `OS_SEARCH_FAST` to 1 adds around 50 cycles of CPU for the search compared to setting `OS_SEARCH_FAST` to zero. But when the next ready to run priority is less than 8, 16, 24, ... then there is an extra 10 cycles needed, but without the 8 times 6 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 15 priority levels. So, if an application has tasks spanning less than 15 priority levels, the build option `OS_SEARCH_FAST` should be set to 0; for all other cases, the build option `OS_SEARCH_FAST` should be set to 1.

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 MSP430Ware software library is part of Code Composer Studio version 5, and this library includes a high level API for all the peripherals available on the MSP430 / MSP430X devices

7 Measurements

This section gives an overview of the memory requirements and the CPU latency encountered when the RTOS is used on the MSP430 / MSP430X 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 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 used for the memory measurements are:

1. Optimization level: 3¹
2. Optimize for speed: 0

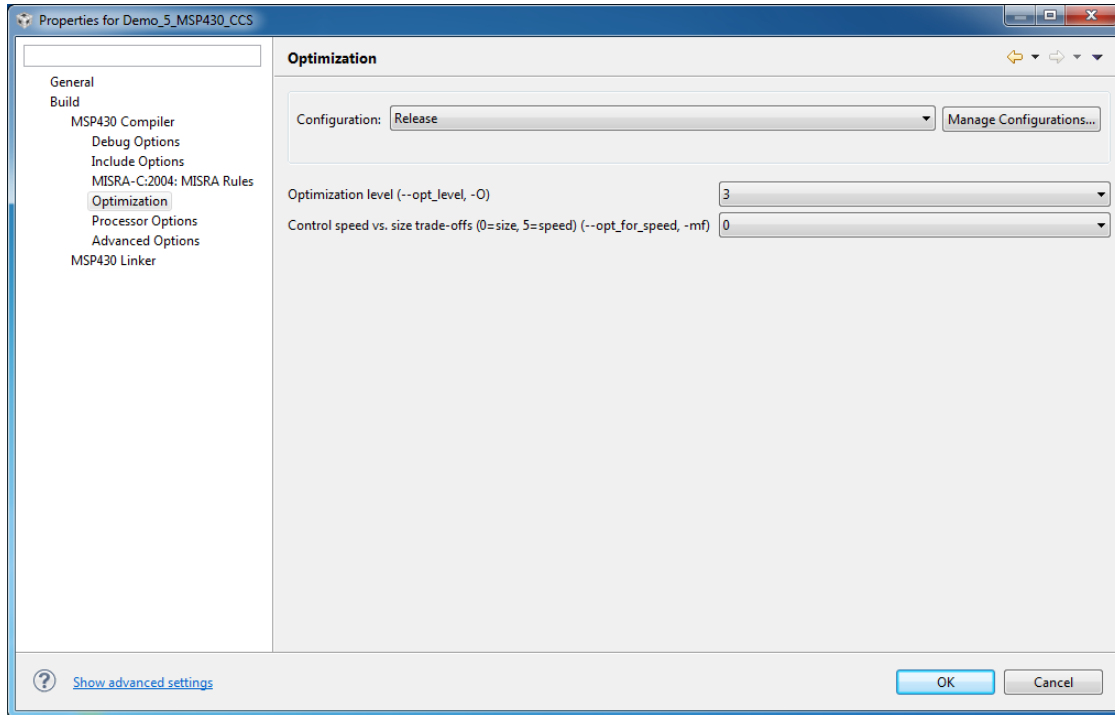


Figure 7-1 Memory Measurement Code Optimization Settings

¹ 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

Processor	MSP430		MSP430X		
	Code model	Small	Large	Large	Large
Data Model	---	Small	Small	Restricted	Large
Minimal Build	< 675	< 650	< 650	< 700	< 750
+ Runtime service creation / static memory	< 925	< 900	< 900	< 975	< 1050
+ Multiple tasks at same priority	< 1025	< 1000	< 1025	< 1100	< 1125
+ Runtime priority change + Mutex priority inheritance + FCFS + Task suspension	< 1500	< 1475	< 1550	< 1675	< 1725
+ Timer & timeout + Timer call back + Round robin	< 2050	< 2025	< 2100	< 2275	< 2300
+ Events + Mailbox	< 2750	< 2725	< 2900	< 3150	< 3225
Full Feature Build (no names)	< 3325	< 3275	< 3475	< 3750	< 3800
Full Feature Build (no names/no runtime creation)	< 2925	< 2900	< 3075	< 3350	< 3375
Full Feature Build (no names/no runtime creation) + Timer services module	< 3300	< 3275	< 3525	< 3825	< 3975

Table 7-2 Added features

Processor	MSP430		MSP430X		
	Code model	Small	Large	Large	Large
Data Model		Small	Small	Restricted	Large
Zeroing BSS	< 75	< 75	< 75	< 75	< 125
Mutex Interface for libraries	< 50	< 50	< 50	< 50	< 50

Table 7-3 Assembly Code Memory Usage

Processor	MSP430		MSP430X		
		Small	Large	Large	Large
Code model					
Data Model		Small	Small	Restricted	Large
ASM code	158	118	120	120	120
Vector Table (per interrupt)	+ 2	+ 2	+ 2	+ 2	+ 2
Interrupt prologue (per interrupt)	+ 8	+ 8	+ 10	+ 10	+ 10
Hybrid Stack Enabled	+ 10	+ 10	+ 12	+ 12	+ 12
Nested interrupts Enabled	+ 8	+ 8	+ 8	+ 8	+ 8
Oscillator bits preservation Enabled	+ 14	+ 14	+ 14	+ 14	+ 14

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 MSP430-5438STK Evaluation board populated with an 18 MHz MSP430F5438 device. All measurements have been performed on the real platform, using the timer peripheral `TIMERA` 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 `TIMERA` counter value. This instrumentation can add up to 5 or 6 cycles to the measurements. The code optimization settings used for the latency measurements are:

1. Optimization level: 4
2. Optimize for speed: 5
3. Debugging: Off

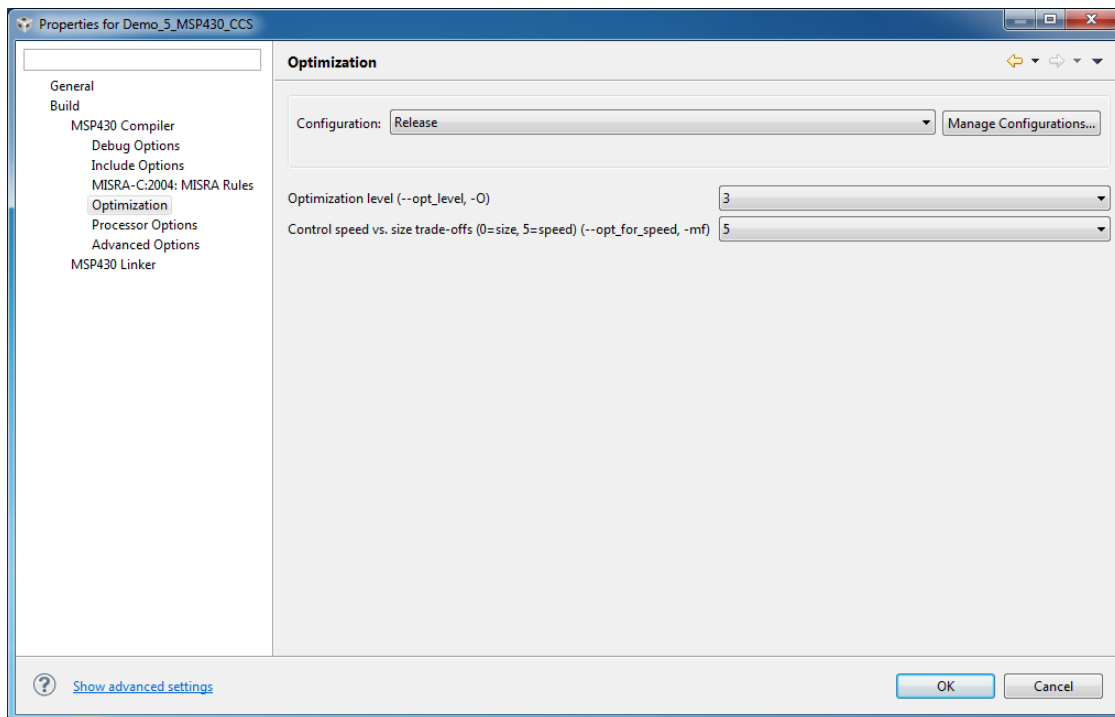


Figure 7-2 Latency Measurement Code Optimization Settings

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
    ...
}

TaskPriol()
{
    ...
    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 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);
    ...
}

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

The fifth set of measurements counts the number of CPU cycles elapsed from the beginning of an interrupt using the component, until the task that was blocked becomes the running task and is back from the component used that blocked the task. The interrupt latency measurement includes everything involved in the interrupt operation, even the cycles the processor needs to push the interrupt context before entering the interrupt code. The interrupt function, attached with `OSISRInstall()`, is simply a two line function that uses the appropriate RTOS component followed by a return.

The following tables list the results obtained, where the cycle count is measured using the `TIMERA` peripheral on the MSP430 / MSP430X. 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 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. For this measurement, the MSP430 `TIMERA` is used to trigger the interrupt and measure the elapsed time.

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 oscillator bit preservation, nor the interrupt nesting, 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 number is the latency measurements when the build option `OS_SEARCH_ALGO` is set to 0.

Table 7-8 Latency Measurements (MSP430)

Description	Minimal Features	Full Features
Semaphore posting no task switch	147 (147)	212 (212)
Semaphore waiting no blocking	152 (152)	227 (227)
Semaphore posting with task switch	250 (283)	388 (415)
Semaphore waiting with blocking	271 (267)	425 (411)
Semaphore posting in ISR with task switch	411 (438)	579 (600)
Event setting no task switch	n/a	209 (209)
Event getting no blocking	n/a	273 (273)
Event setting with task switch	n/a	413 (440)
Event getting with blocking	n/a	465 (451)
Event setting in ISR with task switch	n/a	604 (625)
Mailbox writing no task switch	n/a	279 (279)
Mailbox reading no blocking	n/a	280 (280)
Mailbox writing with task switch	n/a	460 (487)
Mailbox reading with blocking	n/a	492 (478)
Mailbox writing in ISR with task switch	n/a	651 (672)
Interrupt Latency	31	31
Interrupt overhead entering the kernel	161 (155)	191 (185)
Interrupt overhead NOT entering the kernel	58	58
Context switch	64	64

Table 7-9 Latency Measurements (Small Code & Small Data)

Description	Minimal Features	Full Features
Semaphore posting no task switch	123 (123)	181 (168)
Semaphore waiting no blocking	127 (127)	193 (178)
Semaphore posting with task switch	200 (228)	324 (334)
Semaphore waiting with blocking	219 (215)	359 (336)
Semaphore posting in ISR with task switch	334 (358)	481 (498)
Event setting no task switch	n/a	177 (162)
Event getting no blocking	n/a	233 (218)
Event setting with task switch	n/a	345 (356)
Event getting with blocking	n/a	395 (372)
Event setting in ISR with task switch	n/a	502 (520)
Mailbox writing no task switch	n/a	241 (221)
Mailbox reading no blocking	n/a	244 (226)
Mailbox writing with task switch	n/a	390 (395)
Mailbox reading with blocking	n/a	422 (397)
Mailbox writing in ISR with task switch	n/a	547 (560)
Interrupt Latency	23	23
Interrupt overhead entering the kernel	134 (130)	157 (164)
Interrupt overhead NOT entering the kernel	48	48
Context switch	43	43

Table 7-10 Latency Measurements (Large Code / Small Data)

Description	Minimal Features	Full Features
Semaphore posting no task switch	138 (138)	207 (207)
Semaphore waiting no blocking	142 (142)	219 (219)
Semaphore posting with task switch	233 (261)	368 (393)
Semaphore waiting with blocking	252 (248)	403 (392)
Semaphore posting in ISR with task switch	386 (410)	557 (570)
Event setting no task switch	n/a	203 (203)
Event getting no blocking	n/a	259 (259)
Event setting with task switch	n/a	389 (414)
Event getting with blocking	n/a	439 (428)
Event setting in ISR with task switch	n/a	578 (591)
Mailbox writing no task switch	n/a	286 (286)
Mailbox reading no blocking	n/a	287 (287)
Mailbox writing with task switch	n/a	452 (477)
Mailbox reading with blocking	n/a	486 (475)
Mailbox writing in ISR with task switch	n/a	639 (653)
Interrupt Latency	30	30
Interrupt overhead entering the kernel	153 (149)	189 (177)
Interrupt overhead NOT entering the kernel	59	59
Context switch	61	61

Table 7-11 Latency Measurements (Large Code / Restricted Data)

Description	Minimal Features	Full Features
Semaphore posting no task switch	141 (141)	215 (215)
Semaphore waiting no blocking	144 (144)	226 (226)
Semaphore posting with task switch	254 (287)	421 (441)
Semaphore waiting with blocking	274 (269)	458 (435)
Semaphore posting in ISR with task switch	432 (460)	635 (650)
Event setting no task switch	n/a	213 (213)
Event getting no blocking	n/a	270 (270)
Event setting with task switch	n/a	444 (464)
Event getting with blocking	n/a	498 (475)
Event setting in ISR with task switch	n/a	658 (673)
Mailbox writing no task switch	n/a	341 (341)
Mailbox reading no blocking	n/a	345 (345)
Mailbox writing with task switch	n/a	549 (569)
Mailbox reading with blocking	n/a	581 (558)
Mailbox writing in ISR with task switch	n/a	725 (740)
Interrupt Latency	29	29
Interrupt overhead entering the kernel	178 (163)	214 (209)
Interrupt overhead NOT entering the kernel	60	60
Context switch	66	66

Table 7-12 Latency Measurements (Large Code / Large Data)

Description	Minimal Features	Full Features
Semaphore posting no task switch	141 (141)	220 (214)
Semaphore waiting no blocking	144 (144)	231 (225)
Semaphore posting with task switch	259 (288)	426 (440)
Semaphore waiting with blocking	291 (270)	476 (434)
Semaphore posting in ISR with task switch	493 (518)	701 (706)
Event setting no task switch	n/a	218 (212)
Event getting no blocking	n/a	275 (269)
Event setting with task switch	n/a	449 (463)
Event getting with blocking	n/a	516 (474)
Event setting in ISR with task switch	n/a	724 (729)
Mailbox writing no task switch	n/a	346 (340)
Mailbox reading no blocking	n/a	350 (344)
Mailbox writing with task switch	n/a	554 (568)
Mailbox reading with blocking	n/a	599 (557)
Mailbox writing in ISR with task switch	n/a	793 (797)
Interrupt Latency	29	29
Interrupt overhead entering the kernel	234 (230)	275 (266)
Interrupt overhead NOT entering the kernel	60	60
Context switch	66	66

8 Appendix A: Build Options for Code Size

8.1 Case 0: Minimum build

Table 8-1: Case 0 build options

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

8.2 Case 1: + Runtime service creation / static memory

Table 8-2: Case 1 build options

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

8.3 Case 2: + Multiple tasks at same priority

Table 8-3: Case 2 build options

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

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

Table 8-4: Case 3 build options

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

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

Table 8-5: Case 4 build options

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

8.6 Case 5: + Events / Mailboxes

Table 8-6: Case 5 build options

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

8.7 Case 6: Full feature Build (no names)

Table 8-7: Case 6 build options

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

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

Table 8-8: Case 7 build options

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

8.9 Case 8: Full build adding the optional timer services

Table 8-9: Case 8 build options

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