Felix Daners Engineering

FreeRTOS Port for Renesas M16C62P IAR Platform

Document Information

Document State	☐ Being Processed ☐ Submitted ☒ Accepted
Document Version	1.1
Creation Date	26. September 2008
Author	Felix Daners, Felix Daners Engineering
Customer	
Customer Responsible Person	
Project Name	
Customer Project Number	
Subject	FreeRTOS Port
Last Saved Date / File	26/09/2008 / FreeRTOS_IAR_M16C62P_v1.1.doc
Number of Pages Following this Page	17

Distribution List

Customer	
Felix Daners Engineering	F. Daners

Document Approvals

Felix Daners Engineering	Date	Signature
	Date	Signature
Auftraggeber	Date	Signature
	Date	Signature

Felix Daners Engineering

Platz 3
CH-8200 Schaffhausen
T +41 (0)52 624 92 32
F +41 (0)52 624 92 31
E f.daners@swissworld.com

Felix Daners Engineering

Document Disclaimer

While every reasonable precaution has been taken in the preparation of this document, neither the author nor Felix Daners Engineering assumes responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.
The information contained in this document is believed to be accurate. However, no guarantee is provided. Use this information at your own risk.

Felix Daners Engineering

Document History

Version	Date	Changed / Reason for Change	Name
1.1	26/09/08	New Document	Felix Daners

Abstract

FreeRTOS™ is a portable, open source, mini Real Time Kernel - a free to download and royalty free RTOS that can be used in commercial applications. In this document a port to the M16C62P IAR Platform is introduced.

FreeRTOS comes in three flavours:

The FreeRTOS source code is licensed by the GNU General Public License (GPL) with an exception. The exceptions permits the source code of applications that use FreeRTOS solely through the API published on the FreeRTOS.org WEB site to remain closed source, thus permitting the use of FreeRTOS in commercial applications without necessitating that the whole application be open sourced. The exception should only be used if you wish to combine FreeRTOS with a proprietary product and you comply with the terms stated in the exception itself.
OpenRTOS is a commercially licensed version of FreeRTOS.org. The OpenRTOS license does not contain any references to the GPL.
SafeRTOS is a derivative version of FreeRTOS.org that has been analyzed, documented and tested to meet the stringent requirements of the IEC 61508 safety standard. Complete safety lifecycle documentation artefacts have been created and independently audited to verify IEC 61508 SIL 3 conformance.

For further licensing information refer to the FreeRTOS WEB site www.freertos.org.

Felix Engineering

Table of Content

List of Figures

References

[1] FreeRTOS, OpenRTOS V5.0.2 (www.freertos.org)

[2] Renesas, M16C/62P Group (M16C/62P, M16C/62PT) Hardware Manual, Rev. 2.4.1

[3] IAR, M16C/R8C IAR Assembler Reference Guide for Renesas M16C/1X–3X, 6X and R8C Series of CPU Cores.

[4] IAR, M16C/R8C IAR C/C++ Compiler Reference Guide for Renesas M16C/1X–3X, 6X, and R8C Series of CPU Cores

[5] Renesas, M16C/60, M16C/20 Series Software Manual Rev. 4.00

Compiler/Assembler Versions:

IAR Assembler for M16C 3.21A/W32 (3.21.1.4)

IAR C/C++ Compiler for M16C 3.21D/W32 (3.21.4.4)

IAR XLINK 4.60E (4.60.5.0)

Renesas M16C Simulator Debugger that comes with the C compiler package
M3T-NC30WA V.5.44 Release 00

Abbreviations

TCB	Task Control Block
RTOS	Real Time Operating System

1 Introduction

This port includes the following features:

- Separate Interrupt Stack (ISTACK), task stacks need not to save space for interrupt handlers.
- Interrupt nesting model that allows for interrupts running without RTOS interaction and a set of priority levels of RTOS handled interrupt service routines that also may nest.
- Delayed task switch until last interrupt nesting level is about to return.
- Simple compile time registration of custom interrupt service routines.
- All TCB's and stacks must be located within the 16 bit RAM address space.

2 Task Stack Layout and Creation

Figure 1 shows the layout of a task stack. When a task is created, the function `xTaskCreate` calls `pxPortInitialiseStack`.

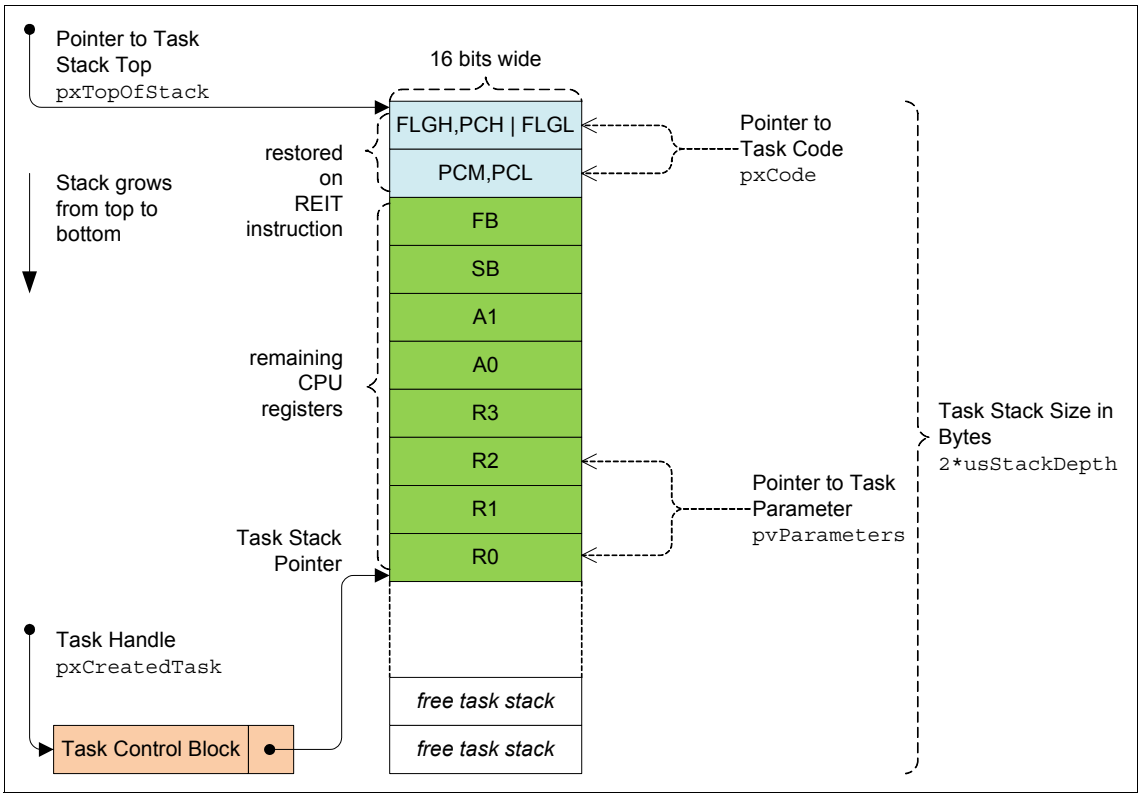

Figure 1 Task stack layout

`xTaskCreate` allocates memory for the task control block and the task stack. It then initializes both memory blocks and adds the newly created task control block to the appropriate task list.

```
portSTACK_TYPE __data16 *pxPortInitialiseStack(
            portSTACK_TYPE __data16 *pxTopOfStack,
            pdTASK_CODE pxCode,
            pdTASK_PARAM pvParameters )
{
  /* select user stack, enable interrupt, set interrupt level to kernel */
  portSTACK_TYPE flag = 0x0040 | 0x0080 | (configKERNEL_INTERRUPT_PRIORITY<<12);

  pxTopOfStack--;

  /* | FLG(H) | PC(H)  | FLG(L)          | */
  *pxTopOfStack-- = (flag & 0x00FF)
                  | (((unsigned portLONG)pxCode >> 8) & 0x00000F00)
                  | ((flag << 4) & 0xF000);

  /* | PC(M)           |  PC(L)          | */
  *pxTopOfStack-- = (portSTACK_TYPE)(((unsigned portLONG)pxCode ) & 0x0000FFFF);

  /* | FB                                | */
  *pxTopOfStack-- = (portSTACK_TYPE)0xFBFB;

  /* | SB                                | */
  *pxTopOfStack-- = (portSTACK_TYPE)0x3B3B;

  /* | A1                                | */
  *pxTopOfStack-- = (portSTACK_TYPE)0xA1A1;

  /* | A0                                | */
  *pxTopOfStack-- = (portSTACK_TYPE)0xA0A0;

  /* | R3                                | */
  *pxTopOfStack-- = (portSTACK_TYPE)0x3333;

  /* | R2                                | */
  *pxTopOfStack-- = (portSTACK_TYPE)((unsigned portLONG)pvParameters >> 16L);

  /* | R1                                | */
  *pxTopOfStack-- = (portSTACK_TYPE)0x1111;

  /* | R0                                | */
  *pxTopOfStack   = (portSTACK_TYPE)((unsigned portLONG)pvParameters & 0x0000FFFFL);

   return pxTopOfStack;
}
```

3 Task Switching Primitives

Task switching is done in five steps:

- Execute yield interrupt sequence. See `portYIELD()` and Figure 2.
- Save the context of the interrupted running task. See Figure 3.
- Select the new task to run, make it the new current task.
  Done by calling `vTaskSwitchContext()`.
- Restore the context of the new current task.
- Return from yield interrupt by executing a `REIT` instruction.

Task switching is initiated by calling `portYIELD()`. This macro uses an intrinsic function to fire the software interrupt vector `portYIELD_VECTOR`. `portYIELD_VECTOR` is defined as vector 1. On the M16C microcontroller, software interrupts are non-maskable.

```
#define portYIELD_VECTOR      1

/* this is non maskable! always yields even if I flag cleared */
#define portYIELD()                { __software_interrupt( portYIELD_VECTOR ); }
```

The M16C interrupt execution sequence is as follows:

- The CPU obtains interrupt information (interrupt number and interrupt request level) by reading address 000000h. Then, the IR bit applicable to the interrupt information is set to “0” (interrupt requested).

- The FLG register, prior to an interrupt sequence, is saved to a temporary register within the CPU.
- The I, D and U flags in the FLG register become as follows: the I flag is set to "0" (interrupt disabled), the D flag is set to "0" (single-step interrupt disabled), the U flag is set to "0" (ISP selected). However, the U flag does not change state if an `INT` instruction for software interrupt numbers 32 to 63 is executed.
- The temporary register within the CPU (with the saved FLG values) is saved to the stack.
- The PC is saved to the stack.
- The interrupt priority level of the acknowledged interrupt in IPL is set.
- The start address of the relevant interrupt routine set in the interrupt vector is stored in the PC.

The handler for the interrupt that is called from `portYIELD()` expects the U flag to be cleared during interrupt execution sequence. Therefore software interrupts with vectors 32-64 are not allowed as yield interrupt.

```
#if portYIELD_VECTOR > 31
 #error "portYIELD_VECTOR must be < 32 (we expect FLG, PCH, PCM, PCL on ISTACK)"
#endif
```

From the interrupt execution sequence described above, at entry of yield interrupt handler we find the situation on task and interrupt stack as depicted in Figure 2.

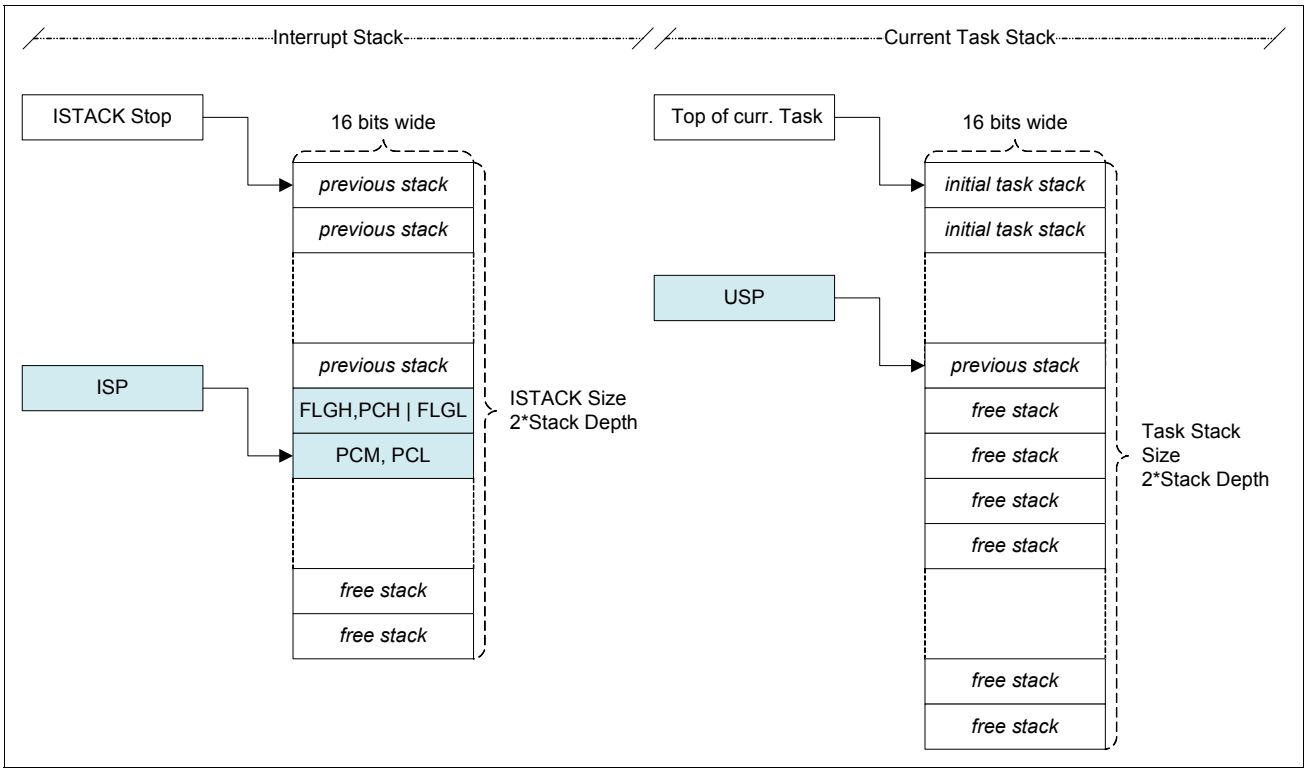

Figure 2 Interrupt- and User Stack Situation at Entry to Interrupt Service

The context of the current task is to be saved on the current task stack. This is done in four steps as also shown in Figure 3:

- The flag state and program counter saved on the interrupt stack during the interrupt execution sequence must be moved to the current task stack. This is most simply done by `PUSH` instructions with the U flag set because this implicitly allocates the required space from the task stack. (Marked with circle with number 1 in Figure 3)
- Save the remaining CPU registers on the task stack, using a `PUSHM` instruction. (2 in Figure 3)
- Now USP needs to be saved in the current task control block to be restored, when the this task is switched in again. (3 in Figure 3)
- The flag state and program counter on the interrupt stack are no longer used and need to be freed from the interrupt stack. (4 in Figure 3)
- Reactivate the interrupt stack, so interrupt service does not use task stack.

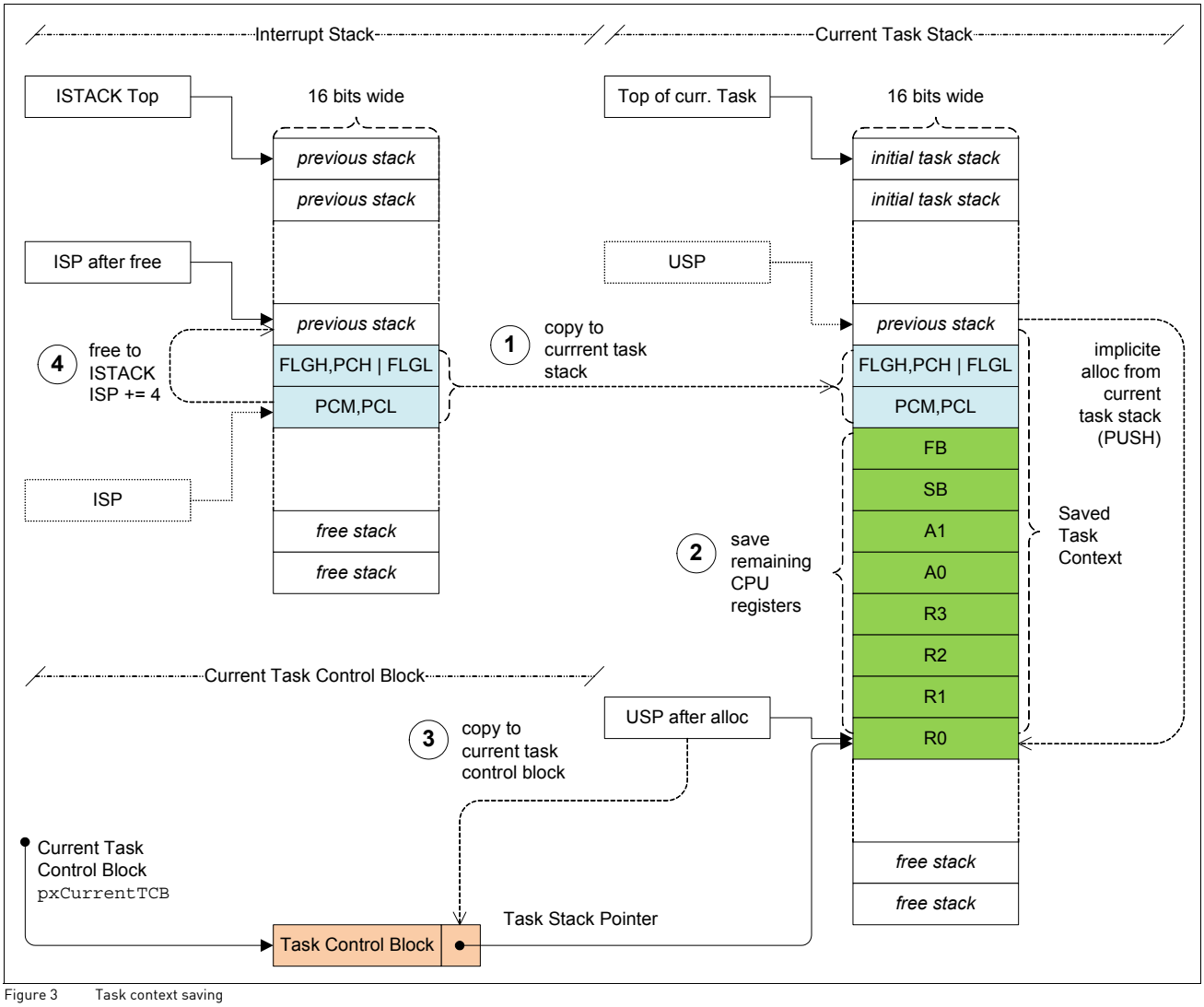

Figure 3 Task context saving

The advantage of this method is, that interrupt service routines can use their own stack (ISTACK) and do not rely on the stack of any switched out task. If the interrupt service routine used the stack of the switched out task, the interrupt stack space needs to be allocated on each task stack in addition to stack space the task requires for its own purpose. This is would not be a very economic memory usage.

Find the assembler code below which implements step 1-4 shown in Figure 3.

```
; SAVE TASK CONTEXT ----------------------------------------------------
; see stack frame in port.c
MACRO  save_context  ; Must copy saved registers from interrupt stack
                     ; to user stack and save remaining registers
                     ; we come here with U flag cleared           CYCLES
PUSHC  FB            ; save FB to interrupt stack                      2
FSET   U             ; activate the task stack                         2
PUSHC  ISP           ; Load ISP to FB, using task stack                2
POPC   FB            ; as temp                                         3

; now push PCL,PCM,PCH,FLAGL and FLAGH to task stack
PUSH.W:G  4:8[FB]    ; | FLG(H) | PC(H)  | FLG(L)          |           4
PUSH.W:G  2:8[FB]    ; | PC(M)           | PC(L)           |           4
PUSH.W:G  0:8[FB]    ; | FB(H)           | FB(L)           |           4

PUSHM  R0,R1,R2,R3,A0,A1,SB;                                          14
                     ; now save the user stack pointer in
                     ; the current task control block
                     ; the TCB is always near (16bit) memory
MOV.W  pxCurrentTCB,A0;                                               2
STC    SP,[A0];                                                       2
FCLR   U             ; activate the interrupt stack                    2
ADD.W  #6,SP         ; free from ISP the PC(L,M,H), FLG(L,H) and FB    1
                     ; (moved to the task stack before)
ENDM                 ;                                          total  42
;                                                              1.75us@24MHz
```

Selection of the task to switch in is not part of the port and is done by calling the RTOS function `vTaskSwitchContext()`.

Restoring the context of the new task is done by loading USP from the new current task control block followed by `POPM`.

```
; RESTORE TASK CONTEXT -------------------------------------------------
; see stack frame in port.c
MACRO  restore_context; task context is on user stack          CYCLES
FSET   U             ; activate the user stack                         2
                     ; set user stack pointer to current task
                     ; the TCB is always near (16bit) memory
MOV.W  pxCurrentTCB,A0;                                               2
LDC    [A0],SP       ; restore the task stack pointer                  3
POPM   R0,R1,R2,R3,A0,A1,SB,FB;                                      16
                     ; load the CPU registers from the task stack
                     ; PC(L,M,H), FLG(L,H) are restored when REIT
                     ; completes the context restore
ENDM                 ;                                          total  21
;                                                              0.85us@24MHz
```

After the context is restored, a `REIT` with U flag set returns to the switched in task.

The code for the yield handler needs to consider two cases:

- The yield interrupt has been called from an interrupt service, this means the nesting level at entry is not equal to zero.
- The yield interrupt has fired from a running task. In this case the nesting level at entry equals zero.

When the nesting level is not zero at entry, the task selection is delayed until the last interrupt returns. The yield handler only sets a flag `intTaskSwitchPending`.

When the nesting level is zero, the yield interrupt needs to save the context of the interrupted task, call the task selection routine, restore the context of the new task and execute a `REIT` instruction.

Interrupt nesting is explained later in chapter »Interrupt Nesting«.

```
        RSEG CODE:CODE:REORDER:ROOT(0)
portYieldInterrupt:
        CMP.B #0,intNesting ; do not reinable interrupts here,
                            ; we do not nest from this int.
                            ; so ++intNesting not required
        JNE portYieldInterrupt_0; If intNesting > 0, no task switch
                            ; task switch when lower priority int returns
        save_context        ;
        JSR.A vTaskSwitchContext;
        restore_context     ;
        REIT                ; return to switched in task
    portYieldInterrupt_0:   ; intNesting is not 0at entry
        MOV.W #0xFFFF,intTaskSwitchPending;
                            ; only set flag to remember new task
                            ; selection required when last ongoing
                            ; interrupt terminates
        REIT;

        ; interrupt vector for YIELD
        COMMON INTVEC:HUGECONST:ROOT(0)
        ORG portYIELD_VECTOR*4
??portYieldInterrupt??INTVEC??:
        DC24    portYieldInterrupt;
```

4 Interrupt Nesting

To support interrupt nesting with minimal overhead, interrupt nesting and task selection is controlled with the help of two variables `intTaskSwitchPending` and `intNesting`. This allows for task selection to be delayed until the last of the nested ongoing interrupts complete.

```
        ; keep track of pending task switches
        ; cleared when interrupt prologue executed with intNesting==0 at entry
        ; set from tick timer interrupt when intNesting != 0 at entry
        ; set from yield interrupt when intNesting != 0 at entry
        ; set from user isr handler with the macro portEND_SWITCHING_ISR
        RSEG DATA16_N:NEARDATA:NOROOT(1)
        EVEN
intTaskSwitchPending:
        DS16 1


        ; keep track on the interrupt nesting
        ; Incremented in interrupt entry code
        ; Decremented in interrupt exit code
        RSEG DATA16_N:NEARDATA:NOROOT(1)
        EVEN
intNesting:
        DS8 1
```

Further the eight interrupt levels of the M16C are split into three groups.

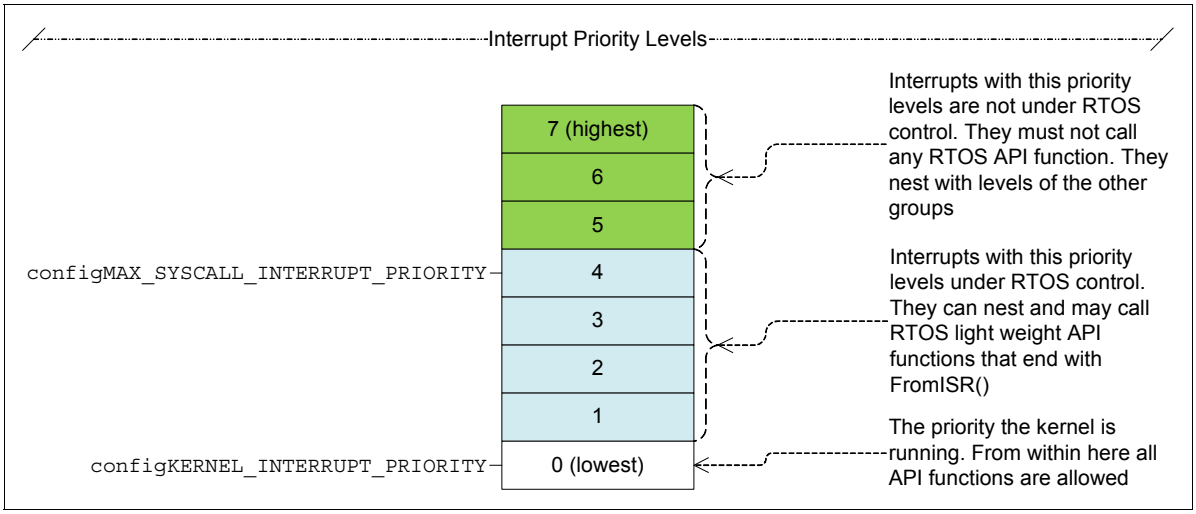

Figure 4 Interrupt priority levels

From this, we can define critical sections for the RTOS differently than just disabling all interrupts. We only need to disable interrupts that might call a RTOS API function.

```
#pragma inline=forced
  static portBASE_TYPE portSetInterruptMaskFromISR(void)
  {
    portBASE_TYPE r = __get_interrupt_level();
    __set_interrupt_level(configMAX_SYSCALL_INTERRUPT_PRIORITY);
    return r;
  }
#endif
#define portSET_INTERRUPT_MASK_FROM_ISR() portSetInterruptMaskFromISR()
#define portCLEAR_INTERRUPT_MASK_FROM_ISR( uxSavedStatusRegister ) \
{ \
  __set_interrupt_level((unsigned char)uxSavedStatusRegister);\
}


/* Critical section management. */
#define portDISABLE_INTERRUPTS()  {__set_interrupt_level(configMAX_SYSCALL_INTERRUPT_PRIORITY);}
#define portENABLE_INTERRUPTS()   {__set_interrupt_level(configKERNEL_INTERRUPT_PRIORITY);}
```

Another variant of the same concept is the `portENTER_CRITICAL portEXIT_CRITICAL` implementation:

```
/* be careful, the current state is stored on the stack! when use, make sure to have matching */
/* ENTER/EXIT pairs that do not mess up the stack. DO NOT: */
/* portENTER_CRITICAL(); */
/* { int test;  // allocated on the stack! */
/*   portLEAVE_CRITICAL(); */
/*   return;  */
/* } */

#define portENTER_CRITICAL() \
{ \
  { /* to enforce the rule, open a new scope here, close it in portEXIT_CRITICAL */ \
    asm("PUSHC FLG");\
    __set_interrupt_level(configMAX_SYSCALL_INTERRUPT_PRIORITY);\
}
#define  portEXIT_CRITICAL() \
{ \
    asm("POPC FLG");\
  } /* close the scope opened in portENTER_CRITICAL */\
}
```

With this definition, interrupts from the highest priority group are never locked out. Interrupts from the group that call the light weight API functions (`FromISR(..)`) need prologue and epilogue around the handler code that keeps track of the interrupt nesting level and postpones task selection until the nesting level reaches zero. This is depicted in Figure 5. The assembler code for prologue and epilogue you find below.

```
        ; ENTRY CODE FOR USER INTERRUPT ----------------------------------------
        ; to be executed in each user interrupt
        ; check if nested interrupt. If nested, do not save task context
        ; if iterrupt has interrupted a task, save task context
        MACRO user_interrupt_prologue
        LOCAL user_interrupt_prologue_0
        LOCAL user_interrupt_prologue_1
        INC.B intNesting;
        CMP.B #1,intNesting; intNesting was zero at entry, save task context
                           ; clear intTaskSwitchPending
        JEQ    user_interrupt_prologue_0
        PUSHM R0,R1,R2,R3,A0,A1,SB,FB; intNesting not zero,
                                     ; save context of interrupt being processed
        JMP   user_interrupt_prologue_1
      user_interrupt_prologue_0:
        MOV.W #0,intTaskSwitchPending; Interrupts are not yet enabled, save
        save_context;
      user_interrupt_prologue_1:
        MOV.W #intTaskSwitchPending,R0; Pass 16 bit address of intTaskSwitchPending
                                     ; to isr, calling convention is __simple
        FSET I; Interrupts above the one running allowed
        ENDM
```

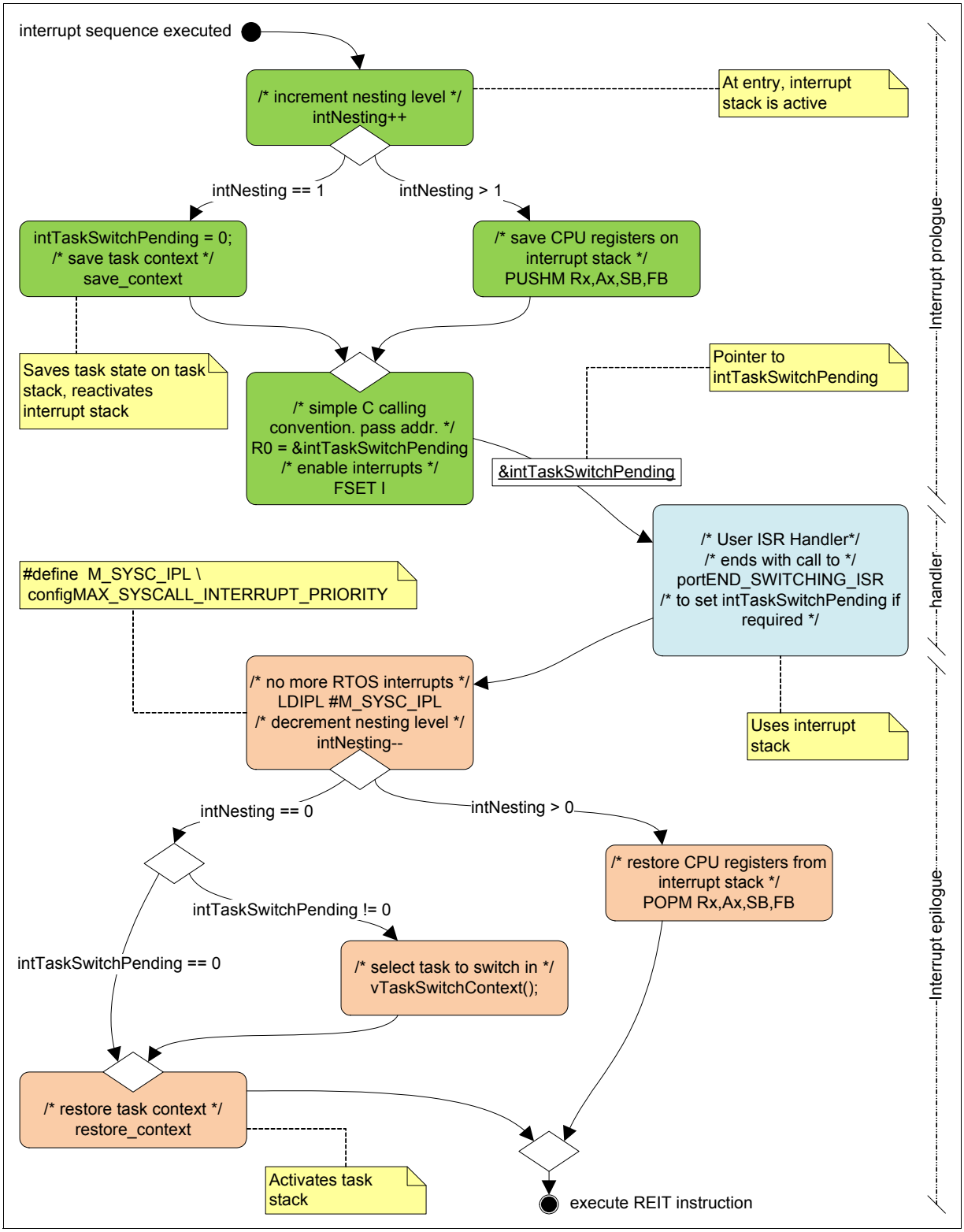

Figure 5 RTOS nested interrupt prologue and epilogue

```
; EXIT CODE FOR USER INTERRUPT ----------------------------------------
; to be executed in each user interrupt
; If nested, the previous interrupt context is on the Interrupt stack
; If not nested, the context of the task to activate needs to be
; restored on the user stack

MACRO user_interrupt_epilouge
LOCAL user_interrupt_epilouge_0
LOCAL user_interrupt_epilouge_1
LOCAL user_interrupt_epilouge_2

; this macro is immedeately after the handler function returns.
; do switch
LDIPL #configMAX_SYSCALL_INTERRUPT_PRIORITY;  No more
                                            ;  interrupts that use os
CMP.B #0,intNesting; Just to make save. This never is the case
JEQ    user_interrupt_epilouge_0
DEC.B intNesting;
user_interrupt_epilouge_0:
JNZ   user_interrupt_epilouge_1; Other interrupt is running
CMP.W #0,intTaskSwitchPending;   Last interrupt check if task
                              ;   switch is pending
JEQ user_interrupt_epilouge_2; No task switch pending
JSR  vTaskSwitchContext; Do the task switch
user_interrupt_epilouge_2:
restore_context; Restore switched in task context
REIT
user_interrupt_epilouge_1:
POPM  R0,R1,R2,R3,A0,A1,SB,FB; Restore last interrupt context
REIT
ENDM
```

5 System Tick Timer

The RTOS needs to track system time to implement task sleep delays, timeouts of blocking functions and pre-emptive multitasking. The system timer also needs to handle interrupt nesting. At every time tick the RTOS should select the next task (of same or higher task priority level). This simplifies the prologue and epilogue of the handler. For more details about interrupt nesting see chapter 4.

The following two functions install and uninstall timer M16C B0 as system tick timer.

```
/** set ms timer function */
/* This routine setups timer 1 and clears timer flag */
static void rtos_tick_timer_uninstall(void)
{
  tb0s = 0x00;      // stop timer B0 (count flag)
}

/** set ms timer function */
/* This routine setups timer B0 and clears timer flag */
static void rtos_tick_timer_install(void)
{
  /* this function is called before interrupts are enabled */

  TB0MR = 0x40;     // XXXX XXXX
                    // |||| ||++- operation mode: 00: timer 01: event counter 10:one shot timer 11: PWM
                    // |||| |+--- pulse output at pin TA4out 0: OFF 1: ON
                    // |||| +---- gate function: 0: timer counts only when TA0in is "L", 1: .. is "H"
                    // |||+------ gate function  0: not available 1: available
                    // ||+------- must always be 0 in timer mode
                    // ++-------- count source select bits: 00:f1  01:f8  10:f32 11:fc32

  TB0 = 1500-1;     // 1ms resolution @ 12MHz f8
  tb0s = 0x01;      // start timer B0 (count flag)
  TB0IC = configMAX_SYSCALL_INTERRUPT_PRIORITY;// interrupt priority level select bit 0...7
}
```

The interrupt handler function is written in assembler:

```
        ; SYSTEM TICK INTERRUPT HANDLER ---------------------------------------
        ; if interrupt nesting level is 0 at entry, do task switch
        ; if interrupt nesting level is greater 0 at entry, set
        ; intTaskSwitchPending and do increment system timer only
        RSEG CODE:CODE:REORDER:ROOT(0)
portTimerB0Interrupt:
        ;PUSHM R0;
        INC.B intNesting;
        CMP.B #1,intNesting; yes do the task switch
        JNE   portTimerB0Interrupt_0
        FSET  I; allow interrupts
        save_context;
        JSR.A vTaskIncrementTick;
        LDIPL #configMAX_SYSCALL_INTERRUPT_PRIORITY;
        MOV.B #0,intNesting; We end up here only when int nesting was zero
                           ; at entry.
                           ; Select the task to switch in, restore its context
                           ; and do the REIT to resume the task
        JSR.A vTaskSwitchContext;
        restore_context;
        REIT
    portTimerB0Interrupt_0: ; intNesting not 1, do not select task,
                            ; do not save task context
                            ; set intTaskSwitchPending so task selection
                            ; is done when interrupt epilogue of last
                            ; ongoing interrupt is executed
        MOV.W #0xFFFF,intTaskSwitchPending;
        FSET I              ; Allow interrupts, save CPU registers on ISTACK
        PUSHM R0,R1,R2,R3,A0,A1,SB,FB;
        JSR.A vTaskIncrementTick;
        LDIPL #configMAX_SYSCALL_INTERRUPT_PRIORITY;
                            ; now now more interrupt nesting
        CMP.B #0,intNesting ; Just to make save. This cannot be true
        JEQ   portTimerB0Interrupt_1;
        DEC.B intNesting;
    portTimerB0Interrupt_1:
        POPM  R0,R1,R2,R3,A0,A1,SB,FB;
        REIT


        ; interrupt vector 26 for timer B0, system tick
        COMMON INTVEC:HUGECONST:ROOT(0)
        ORG 4*26
??portTimerB0Interrupt??INTVEC_26:
        DC24    portTimerB0Interrupt;
```

6 User Interrupt Handler

User Interrupt handlers that use the RTOS lightweight API (the functions that end with `FromISR(..)`) are defined at compile time using three macros. First, the name of the handler function is to be bound to an interrupt vector. This needs to be done in the project specific `FreeRTOSConfig.h` header file.
The macros to define are `USER_ISR_VECTOR_`*nn*, where *nn* is the decimal vector number from the interval 0 to 31. Use a leading zero for vectors < 10.

```
// FreeRTOSConfig.h
// The name of the handler function for vector 30 is sl811_isr
#define USER_ISR_VECTOR_30    sl811_isr
```

This define enables a section in the port file `asm_func.s34` which defines prologue, epilogue and the vector table entry for the handler function. The assembler file has a section for each supported interrupt vector. Following the code that implements nesting prologue and epilogue as explained in chapter »Interrupt Nesting«.

```
; asm_func.s34
        ;USER INTERRUPT 30 VECTOR TABLE AND HANDLER CODE
#ifdef USER_ISR_VECTOR_30
        EXTERN USER_ISR_VECTOR_30
        PUBLIC ??USER_ISR_VECTOR??INTVEC_30

        RSEG CODE:CODE:REORDER:ROOT(0)
  userIsrVector30:
        user_interrupt_prologue;
        JSR.A USER_ISR_VECTOR_30; call the service routine
        user_interrupt_epilouge;
        REIT;

        COMMON INTVEC:HUGECONST:ROOT(0)
  ??USER_ISR_VECTOR??INTVEC_30:
        ORG 30*4;
        DC24 userIsrVector30;
        DC8  0;

        #undef USER_ISR_VECTOR_30
#endif
```

Next, the prototype for the handler can be defined in any project file using another macro:

```
// my_isr.c
portINTERRUPT_HANDLER_PROTO( sl811_isr );
```

The implementation of the handler function is done in any project file as following snipped shows:

```
// my_isr.c
portINTERRUPT_HANDLER( sl811_isr )
{
    portBASE_TYPE xHigherPriorityTaskWoken;

    /* handler code */
    …
    …
    /* call a lightweigth API function */
    xQueueSendFromISR(hQueue, &m, &xHigherPriorityTaskWoken );
    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

This shows the way the handler tells the interrupt epilogue if the task selection routine needs to be run.
The implementation is simple. When the handler is called from the port file `asm_func.s34` it passes a pointer to the variable `intTaskSwitchPending` (see chapter »Interrupt Nesting«) from that same module in R0. This meets the specification for the M16C simple calling convention as defined in [4]. Because the handler calls are enclosed with appropriate prologue and epilogue, they do not need to save any processor registers and therefore can have the attribute `__task` (see [4]).

```
#define portINTERRUPT_HANDLER_PROTO( isrFunction ) \
        __task __simple void isrFunction ( pdISR_PARAM pxSwitchRequired__ )
#define portINTERRUPT_HANDLER( isrFunction ) \
        __task __simple void isrFunction ( pdISR_PARAM pxSwitchRequired__ )
```

The value returned is set from the macro `portEND_SWITCHING_ISR`. Do not assign a value to the parameter `pxSwitchRequired__` directly, more exact do not initialize this parameter with zero in your handler. If the handler is called while another lower priority interrupt is ongoing, the lower priority interrupt could already have requested a task switch. That is why the `|=` operator is used to modify the value in the macro `portEND_SWITCHING_ISR`.

```
#define portEND_SWITCHING_ISR( xSwitchRequired ) \
        if (1) {(*pxSwitchRequired__) |= xSwitchRequired;} else ((void)0)
```

7 Starting/Stopping the OS

Starting the RTOS requires some actions:

- Install the tick timer.
- Save the flag register.
- Push the processor registers to the active stack.
- Save the user stack pointer.
- Save interrupt stack pointer.
- Set intNesting to 0. See »Interrupt Nesting«
- Do a `restore_context` as explained in »Task Switching Primitives« followed by a `REIT` instruction.

Interrupts with a priority equal or lower to `configMAX_SYSCALL_INTERRUPT_PRIORITY` (see chapter »Interrupt Nesting«) should not be on when the RTOS is about to start. Interrupts are enabled when the context of the first task is restored. See chapter »Task Stack Layout and Creation« for details about the initial task context.

```
/*
 * Setup the hardware ready for the scheduler to take control.  This generally
 * sets up a tick interrupt and sets timers for the correct tick frequency.
 */
portBASE_TYPE xPortStartScheduler( void )
{
  rtos_tick_timer_install();

  /* this enables interrupts because the initialized stack frame contains
     the flag register status */
  portStartScheduler_asm(); /* returns, when portEndScheduler_asm is called */

  return pdFALSE;
}
```

The two stack pointer and the flag register registers to save have static space in the port assembler file `asm_func.s34`.

```
        ; LOCAL VARIABLES ------------------------------------------------------
        ; keep context prior to the scheduler started
        RSEG DATA16_N:NEARDATA:NOROOT(1)
        EVEN
preStartSchedulerUSP:
        DS16 1
        RSEG DATA16_N:NEARDATA:NOROOT(1)
        EVEN
preStartSchedulerFLAGS:
        DS16 1
        RSEG DATA16_N:NEARDATA:NOROOT(1)
        EVEN
preStartSchedulerISP:
        DS16 1
```

Following the code that saves the context and switches in the first task to run.

```
        ; OS START CODE, SAVE CONTEXT PRIOR OS UP, SWITCH IN FIRST TASK --------
        ; void portStartScheduler_asm(void)
        ; this code is used to activate the first task. It returns when
        ; void portEndScheduler_asm(void) is called
        RSEG CODE:CODE:REORDER:NOROOT(0)
portStartScheduler_asm:
        PUSHM R0,R1,R2,R3,A0,A1,SB,FB; Save to the active stack
        STC   FLG, preStartSchedulerFLAGS;
        FCLR  I; stop interrupts
        FSET  U;
        STC   SP,  preStartSchedulerUSP;
        STC   ISP, preStartSchedulerISP;
        MOV.B #0,intNesting
        restore_context;
        REIT;
```

The stop code is similar but in reverse order:

- Uninstall the tick timer.
- Do a `save_context` as explained in »Task Switching Primitives«.
- Restore ISP, USP and the FLG registers from prior the call to `portStartScheduler_asm`
- Restore the CPU registers from the active stack.
- Return from the subroutine.

```
/*
 * Undo any hardware/ISR setup that was performed by xPortStartScheduler() so
 * the hardware is left in its original condition after the scheduler stops
 * executing. DO NOT CALL FROM INTERRUPT!
 */
void vPortEndScheduler( void )
{
  rtos_tick_timer_uninstall();
  portEndScheduler_asm();
}
```

Following the assembler code that switches out the task which called `vPortEndScheduler`.

```
        ; OS STOP CODE, SAVE CONTEXT OF TASK, RESTORE CONTEXT PRIOR OS UP ------
        RSEG CODE:CODE:REORDER:NOROOT(0)
        ; DO NOT CALL FROM INTERRUPT
portEndScheduler_asm:
        save_context; this activates the ISP
        FCLR  I; stop interrupts
        LDC   preStartSchedulerISP, ISP;
        FSET  U;
        LDC   preStartSchedulerUSP, SP;
        LDC   preStartSchedulerFLAGS, FLG; This activates the stack
                                         ; that was active when portStartScheduler_asm
                                         ; have been called
        POPM  R0,R1,R2,R3,A0,A1,SB,FB; restore from active stack
        RTS;
```