Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

THIS SPEC IS OBSOLETE

Spec No: 002-05233

Spec Title: AN205233 - FM3 MB9B500 SERIES RT-THREAD RTOS PORTING ON MB9BF500R

Replaced by: NONE

AN205233

FM3 MB9B500 Series RT-Thread RTOS Porting on MB9BF500R

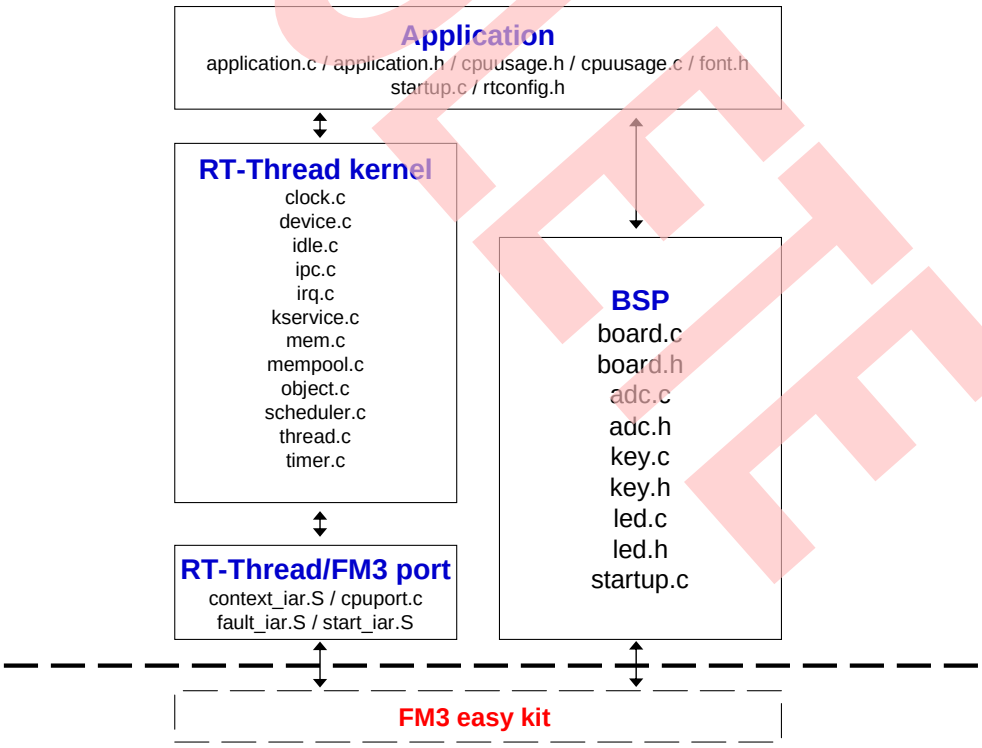
This application note describes 'official' port of RT-Thread on the FM3 easy kit.

Contents

1 Introduction

This application note describes the 'official' port of **RT-Thread** on the FM3 easy kit. Figure 1 shows a block diagram showing the relationship between the application, **RT-Thread**, the port code and the BSP (Board Support Package). Relevant sections of this application note are referenced on the figure.

Figure 1. Relationship between modules



For auto detection in background mode, LibUsbDotNet library is used which is also open source and can be found here: http://libusbdotnet.sourceforge.net

2 The Arm Cortex-M3 Processor's model

The available registers in an Arm Cortex-M3 processor are shown in Figure 2. The Arm Cortex-M3 has a total of 20 registers. Each register is 32 bit wide.

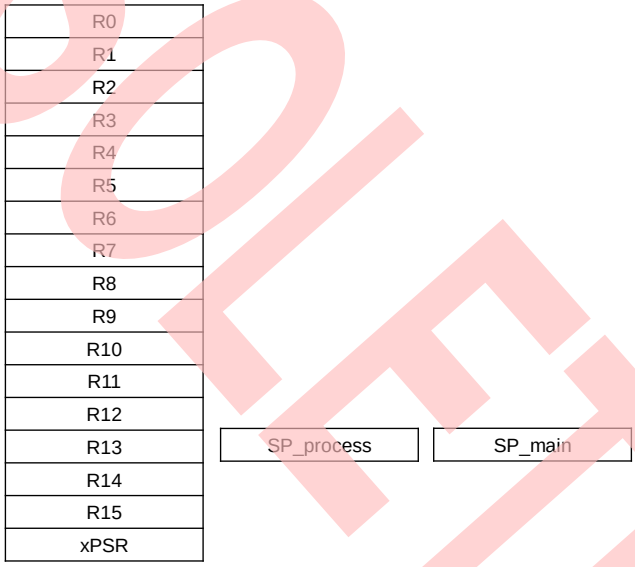
R0 - R12 R0 through R12 are general purpose registers that can be used to hold data as well as pointers.

R13 Is generally designated as the stack pointer (also called the SP) but could be the recipient of arithmetic operations. There actually two stack pointers (SP_process and SP_main) but only one is available at any given time. SP_process is used for task level code and SP_main is used for exception processing.

R14 Is called the Link Register (LR) and is used to store the contents of the PC when a Branch and Link (BL) instruction is executed. The LR allows you to return to the caller.

R15 Is dedicated to be used as the Program Counter (PC) and points to the current instruction being executed. As instructions are executed, the PC is incremented by either 2 or 4 depending on the instruction.

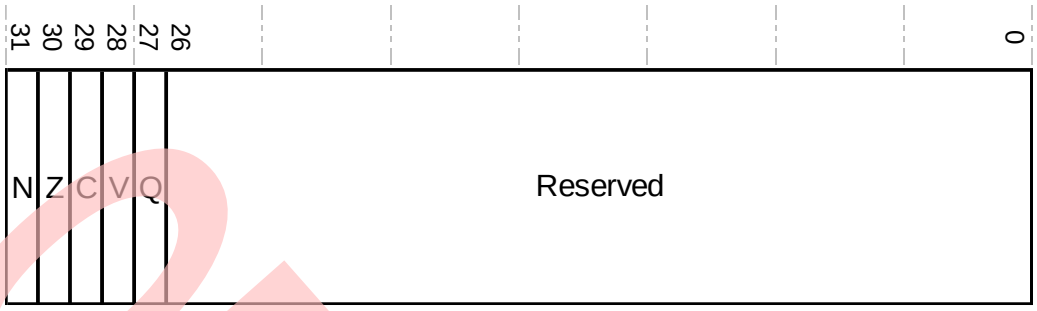
Figure 2. Arm Cortex-M3 Register Model



xPSR There are three separate registers to hold the state of the CPU: APSR, IPSR and EPSR.

The APSR contains application status such as shown in Figure 2.

Figure 3. The APSR Register



N

Bit 31 is the ‘negative’ bit and is set when the last ALU operation produced a negative result (i.e. the top bit of a 32-bit result was a one).

Z

Bit 30 is the ‘zero’ bit and is set when the last ALU operation produced a zero result (every bit of 32-bit result was zero).

C

Bit 29 is the ‘carry’ bit and is set when the last ALU operation generated a carry-out, either as a result of an arithmetic operation in the ALU or from the shifer.

V

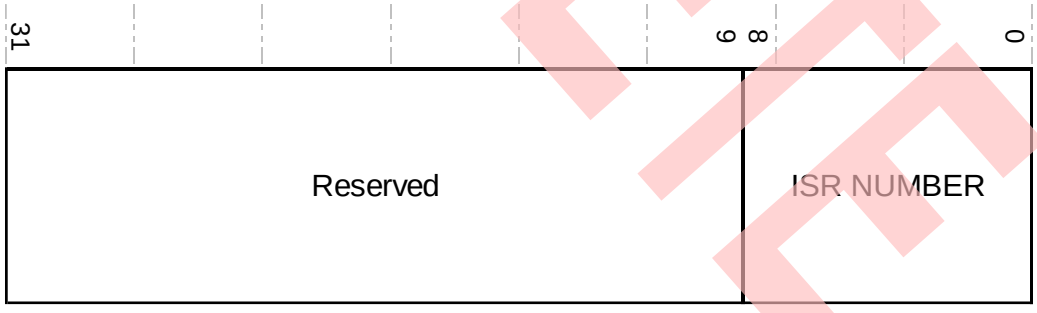
Bit 28 is the ‘overflow’ bit and is set when the last arithmetic ALU operation generated an overflow into the sign bit.

Q

Bit 27 is the sticky saturation flag.

The Interrupt PSR (IPSR) contains the ISR number of the current exception activation and is shown in Figure 4

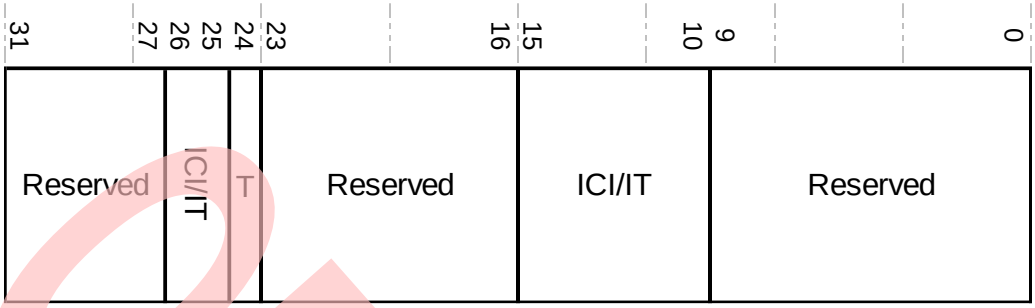
Figure 4. The IPSR Register



The Execution PSR (EPSR) contains two overlapping fields:

- The Interruptible-Continual Instruction (ICI) field for interrupted load multiple and store multiple instructions
- The execution state field for the If-Then (IT) instruction, and the T-bit (Thumb state bit).

Figure 5. The EPSR Register



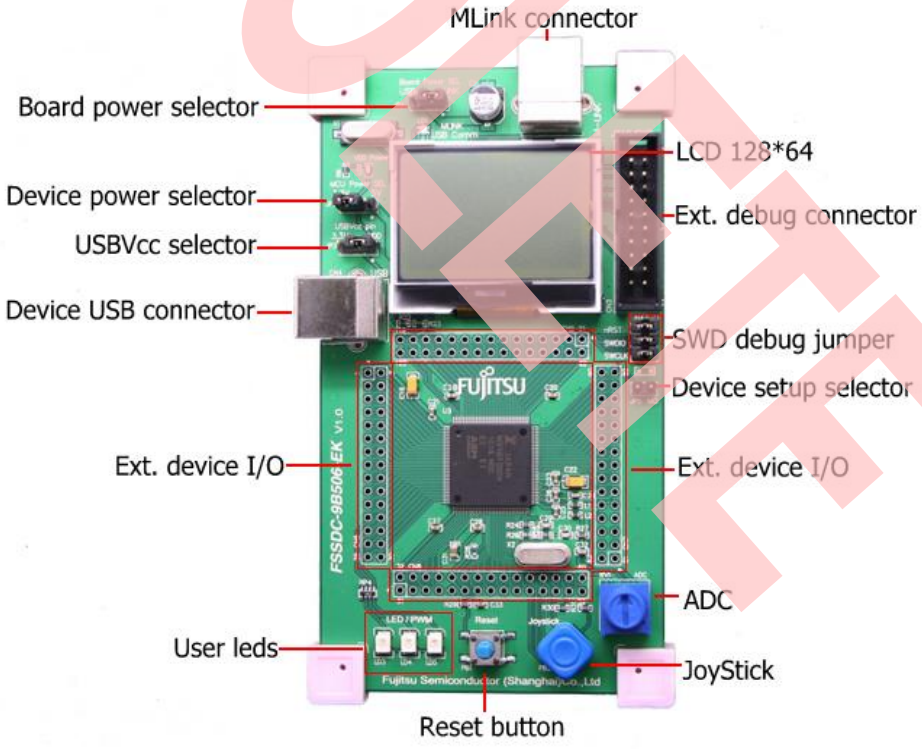
On entering an exception, the processor saves the combined information from the three status register (referred to as xPSR) onto the stack.

3 RT-Thread port for the FM3 (MB9B500) processor

We used the IAR EWARM V6.10 (Embedded Workbench for the Arm) to test the port. The EWARM contains an editor, a C/C++ compiler, an assembler, a linker/locator and the C-Spy debugger. The C-Spy debugger actually contains an Arm Cortex-M3 simulator which allows you to test code prior to run it on actual hardware.

This port is based on FM3 easy kit (FSSDC-9B506-EK, as shown in Figure 6), designed with simplified peripherals for studying MB9BF500 features.

Figure 6. FSSDC-9B506-EK



You can adapt this port provided in this application note to other Arm Cortex-M3 based compilers. The instructions (i.e. the code) should be identical and all you have to do is adapt the port to your compiler specifics.

This port assumes that you are using RT-Thread v0.4.0 or higher.

3.1 Directories and Files

The software that accompanies this application note is assumed to be placed in the following directory:

fm3_rt_thread\

the source code for the port is found in fm3\RT-Thread\fm3 directory with the following files:

start_iar.S

context_iar.S

fault_iar.S

cpuport.c

Test code and configuration files are found in their appropriate directories and are described later.

3.2 start_iar.S (Exception Vector Table)

The Arm cortex-M3 contains an exception vector table (also called the interrupt vector table) starting at address 0x0000 0000. The table can contain up to 256 entries (can be up to 1kbyte since each entry is a 32-bit pointer). Each entry in the table is a pointer to the corresponding exception or interrupt handler.

The exception vector table for the Arm Cortex-M3 is shown in table 1

Table 1. Arm Cortex-M3 Exception Vector Table

Position	Exception /Interrupt	Priority	Vector Address
0		\	0x0000 0000
1	Reset	-3 (highest)	0x0000 0004
2	Non-maskable Interrupt	-2	0x0000 0008
3	Hard Fault	-1	0x0000 000C
4	Memory Management	settable	0x0000 0010
5	Bus Fault	settable	0x0000 0014
6	Usage Fault	settable	0x0000 0018
7	Reserved	\	0x0000 001C
8	Reserved	\	0x0000 0020
9	Reserved	\	0x0000 0024
10	Reserved	\	0x0000 0028
11	SVCall	settable	0x0000 002C
12	Debug Monitor	settable	0x0000 0030
13	Reserved	\	0x0000 0034
14	**PendSV**	settable	0x0000 0038
15	**SysTick**	settable	0x0000 003C
16	INTSIR[239]	settable	0x0000 0040
17	INTSIR[238]	settable	0x0000 0044
…	…	settable	…
255	INTSIR[0]	settable	0x0000 03FC

RT-Thread uses the PendSV handler for context switching and the SysTick handler to process system ticks (i.e. clock ticks). The PendSV handler disables interrupts so that it can execute automatically.

The Arm Cortex-M3 has a built-in timer which was designed specifically for RTOS use. The timer can be configured to run at just about any tick rate. The application's BSP should set this timer to RT_TICK_PER_SECOND.

Note that it's up to the application code to setup the Exception Vector Table. To help you with this task, we created the Exception Vector Table in start_iar.S that you can edit for each project.

```
__vector_table
        DCD     sfe(CSTACK)
        DCD     __iar_program_start
        DCD     NMI_Handler                ; NMI Handler
        DCD     rt_hw_hard_fault           ; Hard Fault Handler
        DCD     MemManage_Handler          ; MPU Fault Handler
        DCD     BusFault_Handler           ; Bus Fault Handler
        DCD     UsageFault_Handler         ; Usage Fault Handler
        DCD     0                          ; Reserved
        DCD     0                          ; Reserved
        DCD     0                          ; Reserved
        DCD     0                          ; Reserved
        DCD     SVC_Handler                ; SVCall Handler
        DCD     DebugMon_Handler           ; Debug Monitor Handler
        DCD     0                          ; Reserved
        DCD     rt_hw_pend_sv              ; PendSV Handler
        DCD     rt_hw_timer_handler        ; SysTick Handler
```

3.2.1 Exception / Interrupt Handling Sequence

When the CPU invokes an exception or interrupt handler, the CPU automatically pushes the xPSR, PC, LR, R12 and R0-R3 registers onto the SP_process stack.

And then, CPU reads the vector table to extract the address of the exception/interrupt handler and updates the PC with this address. The CPU builds the exception stack frame which includes the old PC. The LR actually gets a special value that looks something like 0xFFFF FFF9. This means it is in handler mode, and when CM-3 sees this value attempt to load into the PC (as in BX LR), it recognizes that as an exception return and gets the PC from the registers saved when the exception was entered. The CPU then switches to use the SP_main stack pointer.

3.2.2 Interrupt Controllers

The Arm Cortex-M3 also comes with an integrated Nestable Vectored Interrupt Controller (NVIC).

3.2.3 Interrupt Service Routines

Interrupt Service Routines (ISRs) that need to use RT-Thread service should be written as shown in List 3-2-2 for Cypress FM3.

List 3-2-2 Interrupt Service Routines using RT-Thread services

```
void interrupt_xxx_handler(void)
{
    /* enter interrupt */
    rt_interrupt_enter();

    /* handle the interrupt */

    /* leave interrupt */
    rt_interrupt_leave();
}

void rt_interrupt_enter()
{
    rt_base_t level;

    level = rt_hw_interrupt_disable();
    rt_interrupt_nest ++;
    rt_hw_interrupt_enable(level);
}

void rt_interrupt_leave()
{
    rt_base_t level;

    level = rt_hw_interrupt_disable();
    rt_interrupt_nest --;
    rt_hw_interrupt_enable(level);
}
```

3.3 context_iar.S

A RT-Thread port requires that you write these fairly simple assembly functions:

rt_hw_interrupt_disable()

rt_hw_interrupt_enable()

rt_hw_context_switch()

rt_hw_context_switch_interrupt()

rt_hw_context_switch_to()

3.3.1 rt_hw_interrupt_disable() and rt_hw_interrupt_enable()

Use rt_hw_interrupt_disable() function to disable the interrupt. When the interrupt is disabled, that means the current task or code will not be interrupted by other event (the entire system will not respond to the external event), and it also means the current task will not be preempted.

List 3-3-1 context_iar.S, rt_hw_interrupt_disable()

```
;/*
; * rt_base_t rt_hw_interrupt_disable();
; */
EXPORT rt_hw_interrupt_disable
rt_hw_interrupt_disable:
    MRS     r0, PRIMASK
    CPSID     I
    BX     LR
```

3.3.2 rt_hw_context_switch_interrupt() and rt_hw_conext_switch()

In Fujistu FM3 port, these two functions is the same, because the normal context switch is also triggered by PendSV exception.

```
;/*
; * void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);
; * r0 --> from
; * r1 --> to
; */
EXPORT rt_hw_context_switch_interrupt
EXPORT rt_hw_context_switch
rt_hw_context_switch_interrupt:
rt_hw_context_switch:
    ; set rt_thread_switch_interrput_flag to 1
    LDR     r2, =rt_thread_switch_interrput_flag
    LDR     r3, [r2]
    CMP     r3, #1
    BEQ     _reswitch
    MOV     r3, #1
    STR     r3, [r2]

    LDR     r2, =rt_interrupt_from_thread ; set rt_interrupt_from_thread
    STR     r0, [r2]

_reswitch
    LDR     r2, =rt_interrupt_to_thread   ; set rt_interrupt_to_thread
    STR     r1, [r2]

LDR      r0, =NVIC_INT_CTRL ; trigger the PendSV exception (causes context switch)
LDR      r1, =NVIC_PENDSVSET
    STR      r1, [r0]
    BX       LR
```

3.3.3 rt_hw_context_switch_to()

It is only called by schedule at the first time.

```
;/*
; * void rt_hw_context_switch_to(rt_uint32 to);
; * r0 --> to
; */
    EXPORT rt_hw_context_switch_to
rt_hw_context_switch_to:
    LDR     r1, =rt_interrupt_to_thread
    STR     r0, [r1]

    ; set from thread to 0
    LDR     r1, =rt_interrupt_from_thread
    MOV     r0, #0x0
    STR     r0, [r1]

    ; set interrupt flag to 1
    LDR     r1, =rt_thread_switch_interrput_flag
    MOV     r0, #1
    STR     r0, [r1]

    ; set the PendSV exception priority
```

```
LDR     r0, =NVIC_SYSPRI2
LDR     r1, =NVIC_PENDSV_PRI
LDR.W   r2, [r0,#0x00]       ; read
ORR     r1,r1,r2             ; modify
STR     r1, [r0]             ; write-back

LDR     r0, =NVIC_INT_CTRL      ; trigger the PendSV exception (causes context switch)
LDR     r1, =NVIC_PENDSVSET
STR     r1, [r0]

 CPSIE   I                      ; enable interrupts at processor level
```

3.3.4 rt_hw_pend_sv()

The rt_hw_pend_sv() function is the PendSV exception handler which handles all context switching for RT-Thread. This is a recommended method for performing context switching within the Arm Cortex-M3. This is because the Arm Cortex-M3 auto-saves half of the processor context on any exception, and restores those same registers upon return from exception. The PendSV handler only needs to save R4-R11 and adjust the stack pointers. Using the PendSV exception means that context saving and restoring uses an identical method whether it's initiated from a task or occurs due to an interrupt or exception.

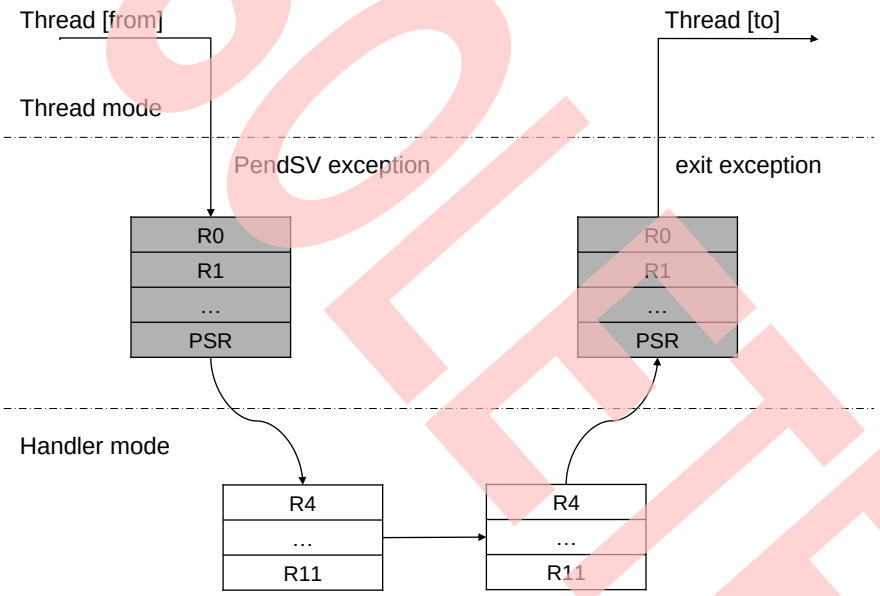
Note that you must place a pointer to rt_hw_pend_sv() in the exception vector table at the vector location 14 (based of the vector table + 4*14 or, offset 56).

```
; r0 --> swith from thread stack
; r1 --> swith to thread stack
; psr, pc, lr, r12, r3, r2, r1, r0 are pushed into [from] stack
    EXPORT rt_hw_pend_sv
rt_hw_pend_sv:
    ; disable interrupt to protect context switch
    MRS r2, PRIMASK
    CPSID  I

    ; get rt_thread_switch_interrupt_flag
    LDR r0, =rt_thread_switch_interrput_flag
    LDR r1, [r0]
    CBZ r1, pendsv_exit                 ; pendsv already handled

    ; clear rt_thread_switch_interrput_flag to 0
    MOV r1, #0x00
    STR r1, [r0]

    LDR r0, =rt_interrupt_from_thread
    LDR r1, [r0]
    CBZ r1, swtich_to_thread    ; skip register save at the first time

    MRS r1, psp                 ; get from thread stack pointer
    STMFD  r1!, {r4 - r11}      ; push r4 - r11 register
    LDR r0, [r0]
    STR r1, [r0]            ; update from thread stack pointer
```

```
swtich_to_thread
    LDR r1, =rt_interrupt_to_thread
    LDR r1, [r1]
    LDR r1, [r1]            ; load thread stack pointer

    LDMFD  r1!, {r4 - r11}          ; pop r4 - r11 register
    MSR psp, r1             ; update stack pointer

pendsv_exit
    ; restore interrupt
    MSR PRIMASK, r2

    ORR lr, lr, #0x04
    BX  lr
```

The normal context switch is shown as Figure 7

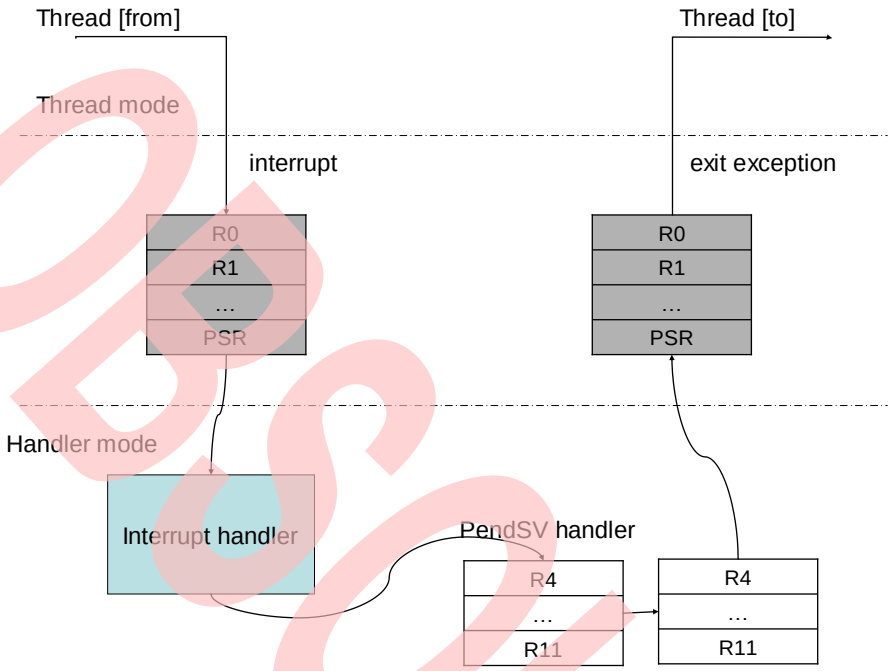
Figure 7. Normal Context Switch Flow



When running context switch (i.e. switch from Thread[from] to Thread[to]), trigger a PendSV exception by function rt_hw_context_switch(). When this exception occurred, Cortex-M3 will push PSR, PC, LR, R0-R3, R12 into the stack of current thread automatically. Then Cortex-M3 will switch to Handler mode by running rt_hw_pend_sv(). The rt_hw_pend_sv() function will restore the stack pointers of Thread[from] and Thread[to]. If the stack pointer of Thread[from] is 0, it means this is the first time for context switch and there is no need to do push stack for Thread[from]. Or if the stack pointer of Thread[from] is not equal to 0, we should push R4-R11 registers to stack; and restore the R4-R11 registers from Thread[to] stack. When the PendSV exception exits, PSR, PC, LR, R0-R3, R12 registers will be restored automatically

Context switch caused by interrupt is shown in Figure 8

Figure 8. Context Switch Caused by Interrupt Flow



When interrupt occurred, the current thread will be interrupt, and PC, PSR, R0-R3, R12 registers will be pushed into the stack of current thread, the processor mode will switch to Handle mode.

While running in the interrupt routine, if context switch needed (by calling rt_schedule() function in interrupt service route), it will check whether the processor is in the Handle mode or not by global variable rt_interrupt_nest, if rt_interrupt_nest != 0, then calling rt_hw_context_switch_ interrupt() for context switch:

In the function rt_hw_context_switch_interrupt(), it will assign stack pointer of current thread to variable rt_interrupt_from_thread, and assign stack pointer of Thread[to] to variable rt_interrupt_to_thread, then set the flag rt_thread_switch_interrupt_flag to 1.

After the last interrupt routine exit, Cortex-M3 will handle the PendSV exception, because the PendSV exception has the lowest priority.

3.4 cpuport.c

The code in listing 3-4-1 initializes the stack frame for the thread being created. The thread received an optional argument ‘parameter’. It is typical for Arm compilers (the Cortex-M3 also) to pass the first argument of a function into the R0 register. That’s why ‘parameter’ is passed in R0 when the thread is created. The initial value of most of the CPU registers is not important, so we decided to initialize them to 0 when the thread is first created but, of course, the register values will most likely change as the thread code is executed.

Listing 3-4-1 cpuport.c rt_hw_stack_init()

```
/**
 * This function will initialize thread stack
 *
 * @param tentry the entry of thread
 * @param parameter the parameter of entry
 * @param stack_addr the beginning stack address
 * @param texit the function will be called when thread exit
 *
 * @return stack address
 */
rt_uint8_t *rt_hw_stack_init(void *tentry, void *parameter,
rt_uint8_t *stack_addr, void *texit)
{
    unsigned long *stk;

    stk     = (unsigned long *)stack_addr;
    *(stk)    = 0x01000000L;                    /* PSR */
    *(--stk) = (unsigned long)tentry;           /* entry point, pc */
    *(--stk) = (unsigned long)texit;                  /* lr */
    *(--stk) = 0;                                     /* r12 */
    *(--stk) = 0;                                     /* r3 */
    *(--stk) = 0;                                     /* r2 */
    *(--stk) = 0;                                     /* r1 */
    *(--stk) = (unsigned long)parameter;        /* r0 : argument */

    *(--stk) = 0;                                     /* r11 */
    *(--stk) = 0;                                     /* r10 */
    *(--stk) = 0;                                     /* r9 */
    *(--stk) = 0;                                     /* r8 */
    *(--stk) = 0;                                     /* r7 */
    *(--stk) = 0;                                     /* r6 */
    *(--stk) = 0;                                     /* r5 */
    *(--stk) = 0;                                     /* r4 */

    /* return task's current stack address */
    return (rt_uint8_t *)stk;
}
```

4 Application

The example application is running 6 threads:

led1 thread	-- blink led1
led2 thread	-- blink led2
key thread	-- user key handler
adc thread	-- fetch ADC value and send to Application.
app thread	-- the Application thread.

4.1 Directories and Files

The software that accompanies this application note is assumed to be placed in the following directory:

fm3_rt_thread\Example\source

The source code for the application is found in the following files:

adc.c

application.c

board.c

cpuusage.c

key.c

lcd.c

led.c

startup.c

rtconfig.h

Test code and configuration files are found in their appropriate directories and are described later.

4.2 rtconfig.h

We can enable or disable some RT-Thread components with rtconfig.h in order to reduce the memory usage.

List 4-2-1 rtconfig.h

```
/* RT-Thread config file */
#ifndef __RTTHREAD_CFG_H__
#define __RTTHREAD_CFG_H__

/* RT_NAME_MAX*/
#define RT_NAME_MAX           4

/* RT_ALIGN_SIZE*/
#define RT_ALIGN_SIZE   4

/* PRIORITY_MAX */
#define RT_THREAD_PRIORITY_MAX      32

/* Tick per Second */
#define RT_TICK_PER_SECOND          100

/* SECTION: RT_DEBUG */
/* Thread Debug */
#define RT_DEBUG
#define RT_USING_OVERFLOW_CHECK
```

```
/* Using Hook */
#define RT_USING_HOOK

/* SECTION: IPC */
/* Using Semaphore */
#define RT_USING_SEMAPHORE

/* Using Mutex */
#define RT_USING_MUTEX

/* Using Event */
#define RT_USING_EVENT

/* Using MailBox */
#define RT_USING_MAILBOX

/* Using Message Queue */
#define RT_USING_MESSAGEQUEUE

/* SECTION: Memory Management */
/* Using Memory Pool Management*/
#define RT_USING_MEMPOOL

/* Using Dynamic Heap Management */
#define RT_USING_HEAP

/* Using Small MM */
#define RT_USING_SMALL_MEM

/* SECTION: Device System */
/* Using Device System */
#define RT_USING_DEVICE
/* RT_USING_UART */
#define RT_USING_UART0
#define RT_UART_RX_BUFFER_SIZE      64

/* SECTION: Console options */
#define RT_TINY_SIZE
#define RT_USING_CONSOLE
/* the buffer size of console */
#define RT_CONSOLEBUF_SIZE    128

/* SECTION: RTGUI support */
/* using RTGUI support */
//#define RT_USING_RTGUI

/* name length of RTGUI object */
#define RTGUI_NAME_MAX        16
/* support 16 weight font */
//#define RTGUI_USING_FONT16
/* support 12 weight font */
#define RTGUI_USING_FONT12
/* support Chinese font */
//#define RTGUI_USING_FONTHZ
/* use DFS as file interface */
//#define RTGUI_USING_DFS_FILERW
/* use font file as Chinese font */
/* #define RTGUI_USING_HZ_FILE */
/* use Chinese bitmap font */
```

```
//#define RTGUI_USING_HZ_BMP
/* use small size in RTGUI */
//#define RTGUI_USING_SMALL_SIZE
/* use mouse cursor */
/* #define RTGUI_USING_MOUSE_CURSOR */
#define RTGUI_DEFAULT_FONT_SIZE 12

#endif
```

4.3 startup.c

The rtthread_startup() function is the entry point of RT-Thread. We can get to know the start process of RT-Thread by look into the rtthread_startup().

It can be divided into several parts:

- Initialize the hardware
- Initialize some RT-Thread components, ex. timer, scheduler…
- Initialize device, it is used for initializing the RT-Thread device framework
- Initialize the application thread, and start the scheduler

4.3.1 List 4-3-1 startup.c, rtthread_startup()

```
/**
 * This function will startup RT-Thread RTOS.
 */
void rtthread_startup(void)
{
    /* init board */
    rt_hw_board_init();

    /* show version */
    rt_show_version();

    /* init tick */
    rt_system_tick_init();

    /* init kernel object */
    rt_system_object_init();

    /* init timer system */
    rt_system_timer_init();

#ifdef RT_USING_HEAP
    #ifdef __CC_ARM
        rt_system_heap_init((void*)&Image$$RW_IRAM1$$ZI$$Limit, (void*)FM3_SRAM_END);
    #elif __ICCARM__
        rt_system_heap_init(__segment_end("HEAP"), (void*)FM3_SRAM_END);
    #else
        /* init memory system */
        rt_system_heap_init((void*)&__bss_end, (void*)FM3_SRAM_END);
    #endif
#endif
```

```
    /* init scheduler system */
    rt_system_scheduler_init();

#ifdef RT_USING_DEVICE
    /* init all device */
    rt_device_init_all();
#endif

    /* init application */
    rt_application_init();

      /* init timer thread */
      rt_system_timer_thread_init();


    /* init idle thread */
    rt_thread_idle_init();

    /* start scheduler */
    rt_system_scheduler_start();

    /* never reach here */
    return ;
}
```


5 Conclusion

This application note presented a port for FM3 (MB9B500 series) processors. The port should be easily adapted to different compilers (the code itself should be identical). Of course, if you use RT-Thread and use the port on other actual hardware, you will need to initialize and properly handle hardware interrupts.

6 Licensing

RT-Thread RTOS is released as an open source RTOS under GNU GPLv2 license. If you intend to use RT-Thread in a commercial product, remember that you need to contact RT-Thread.org to convert the GPLv2 license to a commercial license. Your honesty is greatly appreciated.

7 References

<<RT-Thread manual>>

Document History

Document Title: AN205233 - FM3 MB9B500 Series RT-Thread RTOS Porting on MB9BF500R

Document Number: 002-05233

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	HUAL	04/01/2011	Initial release.
*A	5258872	HUAL	05/11/2016	Migrated Spansion Application Note from MCU-AN-510003-E-10 to Cypress format.
*B	6268592	HUAL	08/02/2018	Obsoleted

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709