Micrium

µC/OS-II Port for the ARM
(Supplement to AN-1011)

www.Micrium.com

Legend

LOW
Memory

CPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R14_sys (LR)
PC

**Stack
Growth**

HIGH
Memory

Task Stack

A red line shows data being copied.

OSTCBStkPtr

**Task's
OS_TCB**

A black line indicates a pointer.

Registers in beige are common
registers. Registers in blue are
mode specific registers.

As a convention, we will show stack
growth going from the bottom of the
page up to show that items are stacked
one on top of the other.

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

**CPU Registers
SYS Mode**

µC/OS-II Port for the ARM

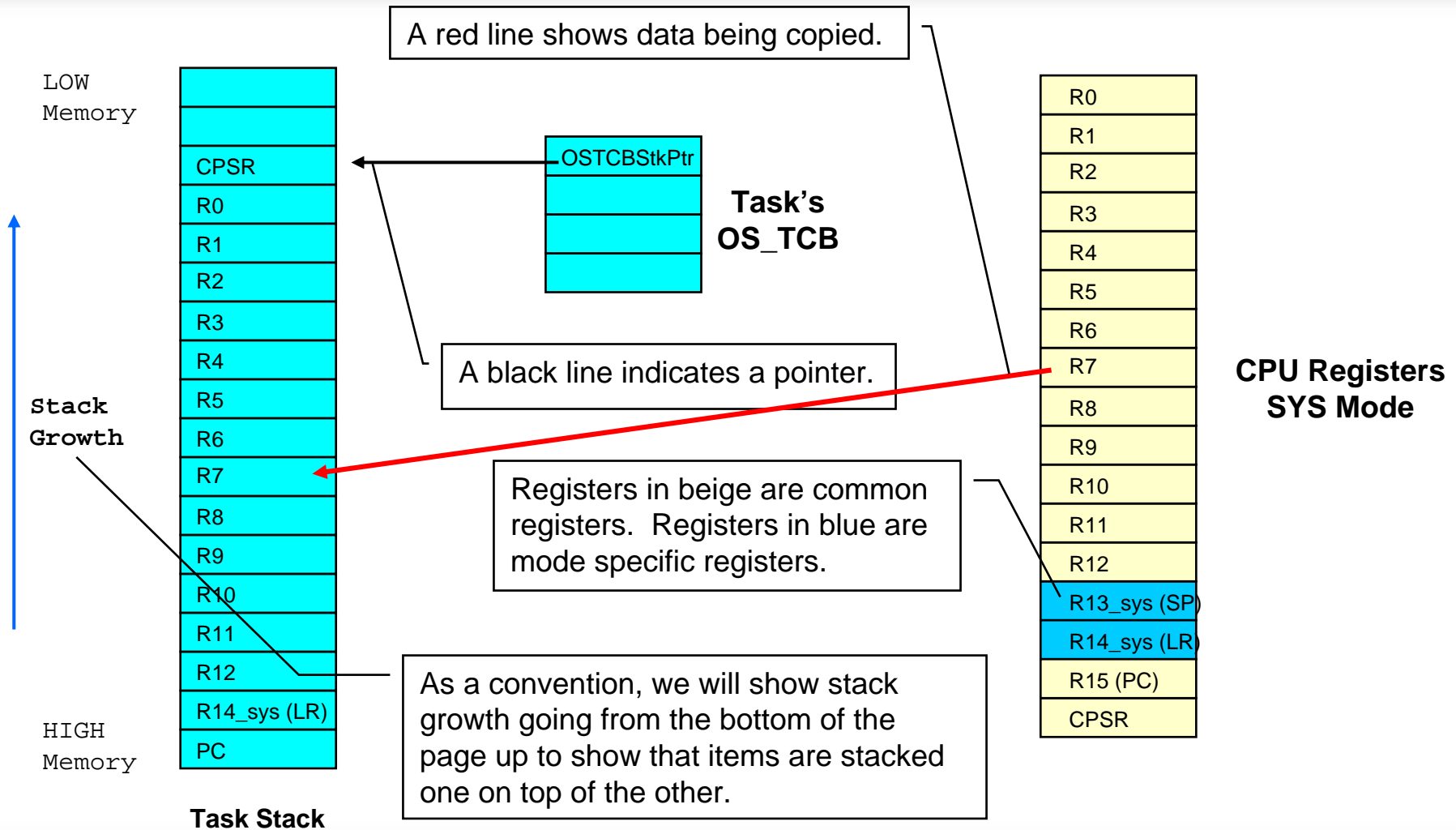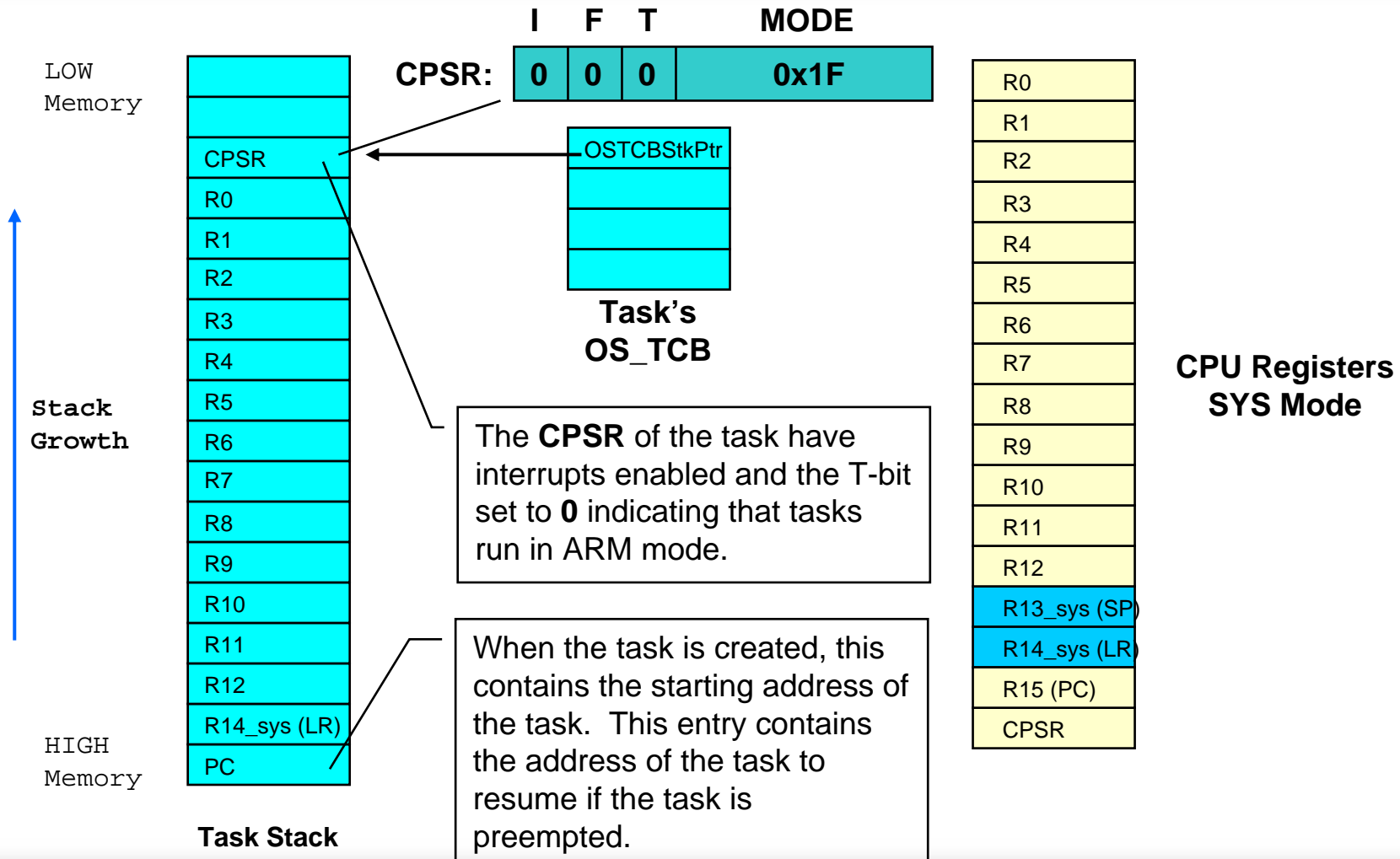# Table of Contents

Task Level Context Switch – OSCtxSw()

Servicing Interrupts – IRQ or FIQ

Interrupt Level Context Switch - OS_IntCtxSw()

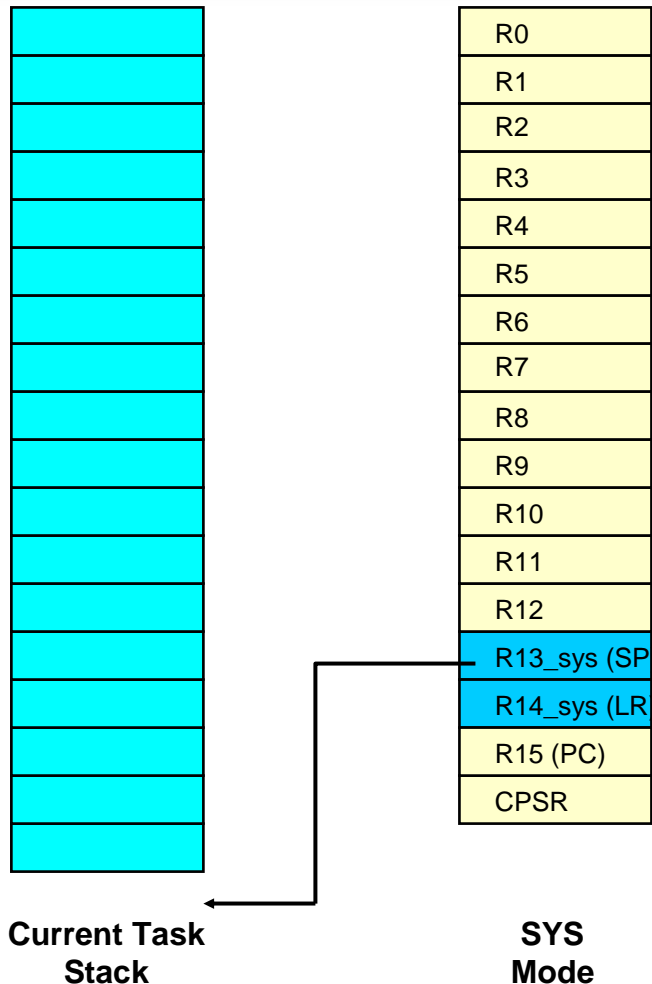µC/OS-II Port for the ARM

Task Level Context Switch
Task Stack Frame (when task is created)

CPSR:

I	F	T	MODE
0	0	0	0x1F

Task Stack

LOW Memory	
CPSR	
R0	
R1	
R2	
R3	
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R14_sys (LR)	
PC	HIGH Memory

Stack Growth

OSTCBStkPtr

Task's OS_TCB

The **CPSR** of the task have interrupts enabled and the T-bit set to **0** indicating that tasks run in ARM mode.

When the task is created, this contains the starting address of the task. This entry contains the address of the task to resume if the task is preempted.

CPU Registers SYS Mode

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

Task Level Context Switch
Task running (ARM mode)

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

Current Task Stack

SYS Mode

A task is assumed to run in ARM mode and, uses the SYS registers (Mode 0x1F).

The processor's **SP** (R13) points to a location into the current task's stack.

If a context switch needs to take place, µC/OS-II will call **OS_Sched()** which in turn calls **OS_TASK_SW()** as shown in the call tree below:

```
OSTimeDly(1)        // Delay task for 1 tick
  OS_Sched();
    OS_TASK_SW();  // Macro that invokes …
      OSCtxSw();   // … this function.
```
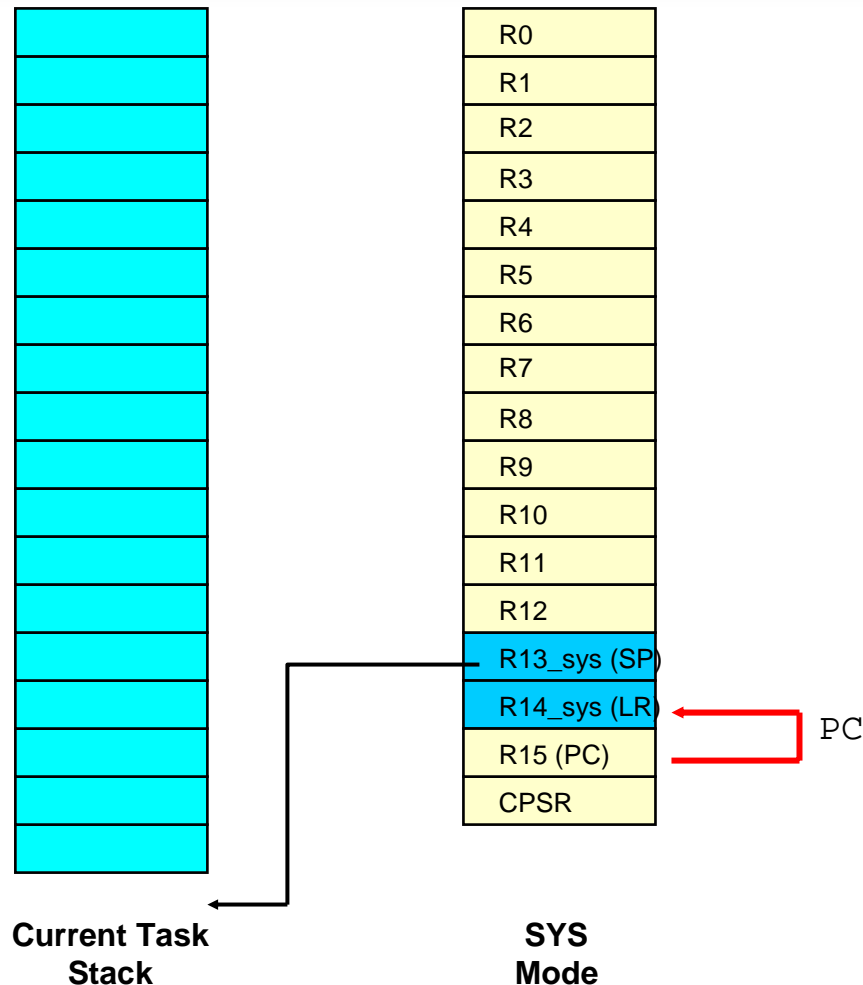
	I	F	T	MODE
CPSR:	1	1	0	0x1F

Notes:
   Interrupts are **DISABLED** during a task-level context switch. Interrupts are disabled at the beginning of **OS_Sched()** and thus the I-bit and F-bit are both set to **1**.

Task Level Context Switch
OS_TASK_SW() is invoked by the scheduler

```
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR
```

Current Task Stack

SYS Mode

PC

OS_TASK_SW() is a macros declared as follows:

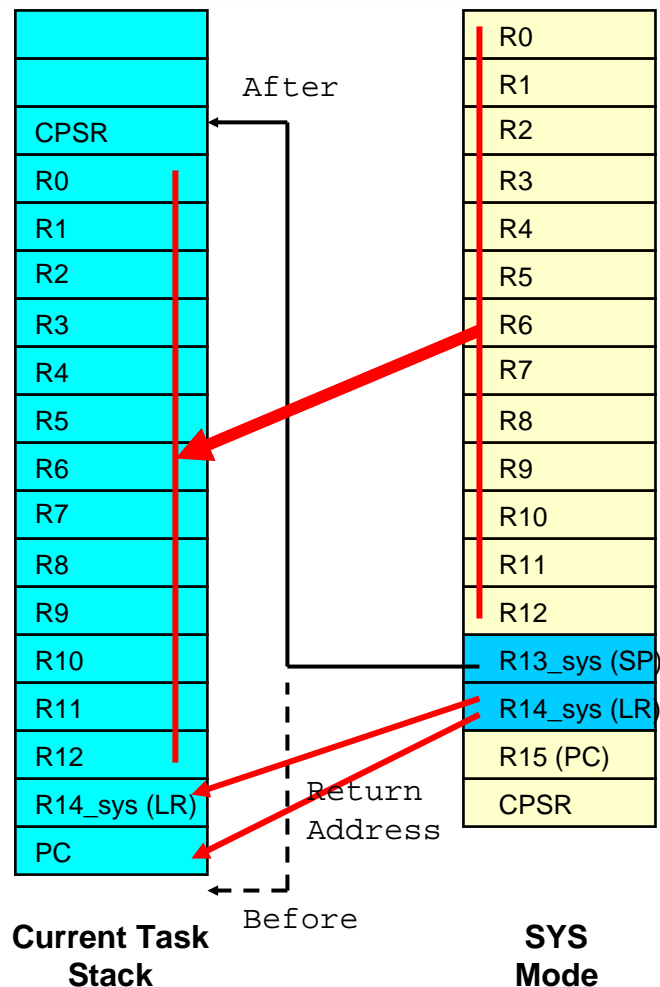**#define OS_TASK_SW() OSCtxSw()**

This macro produces the following code:

  BL OSCtxSw

… which causes the return address to be saved in the **LR** register.

I	F	T	MODE

CPSR: | 1 | 1 | 0 | 0x1F |

Task Level Context Switch
OSCtxSw() (ARM mode)

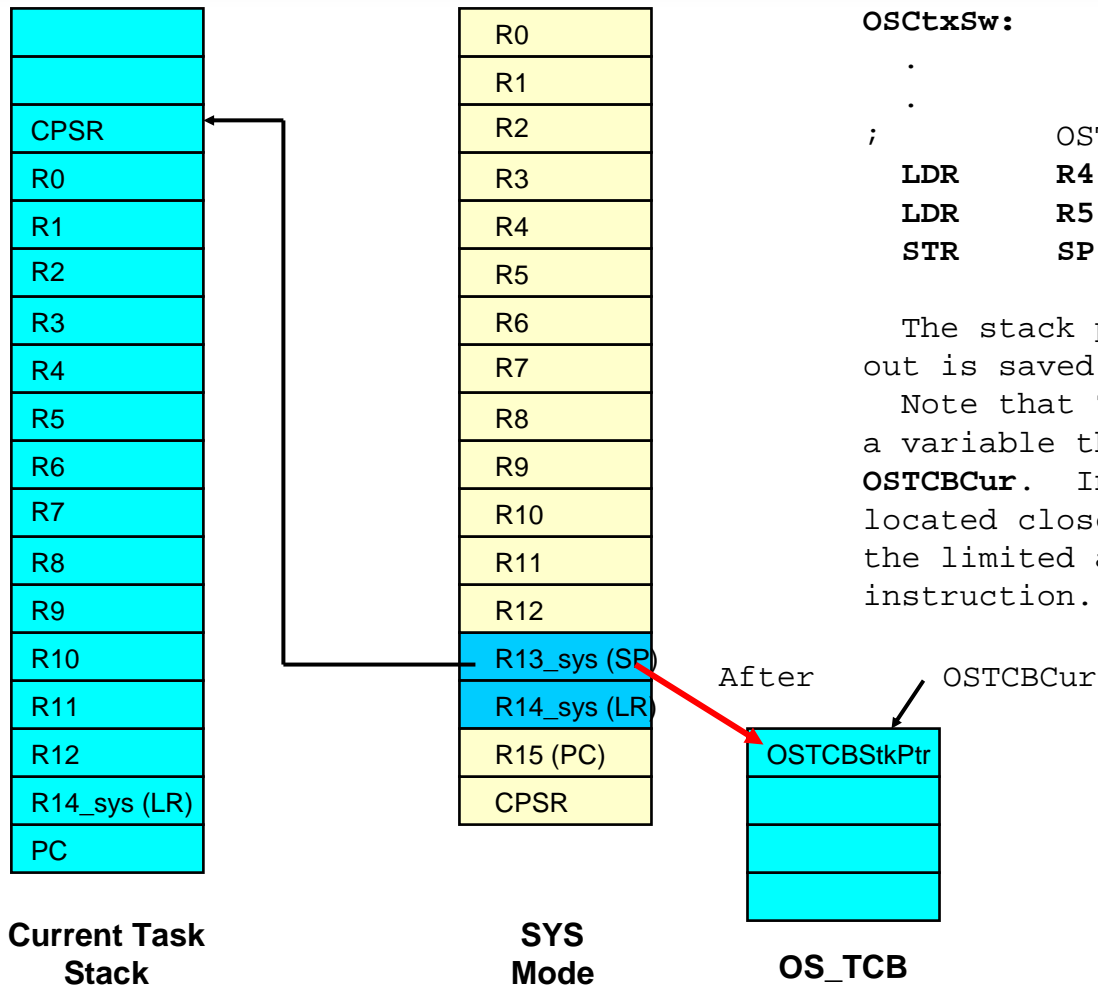

```
OSCtxSw:
    STMFD   SP!,{LR}          ; Return address
    STMFD   SP!,{R0-R12,LR}   ; Registers
    MRS     R4,CPSR           ; CPSR
    STMFD   SP!,{R4}
```

OSCtxSw() starts by saving the current task's context onto the current task's stack.

	I	F	T	MODE
CPSR:	1	1	0	0x1F

Current Task Stack

SYS Mode

µC/OS-II Port for the ARM

Task Level Context Switch
OSCtxSw() (ARM mode)

```
OSCtxSw:
     .
     .
     .
;           OSTCBCur->OSTCBStkPtr = SP
     LDR       R4,??OS_TCBCur
     LDR       R5,[R4]
     STR       SP,[R5]
```

   The stack pointer of the task being switched out is saved into the **OS_TCB** of that task.
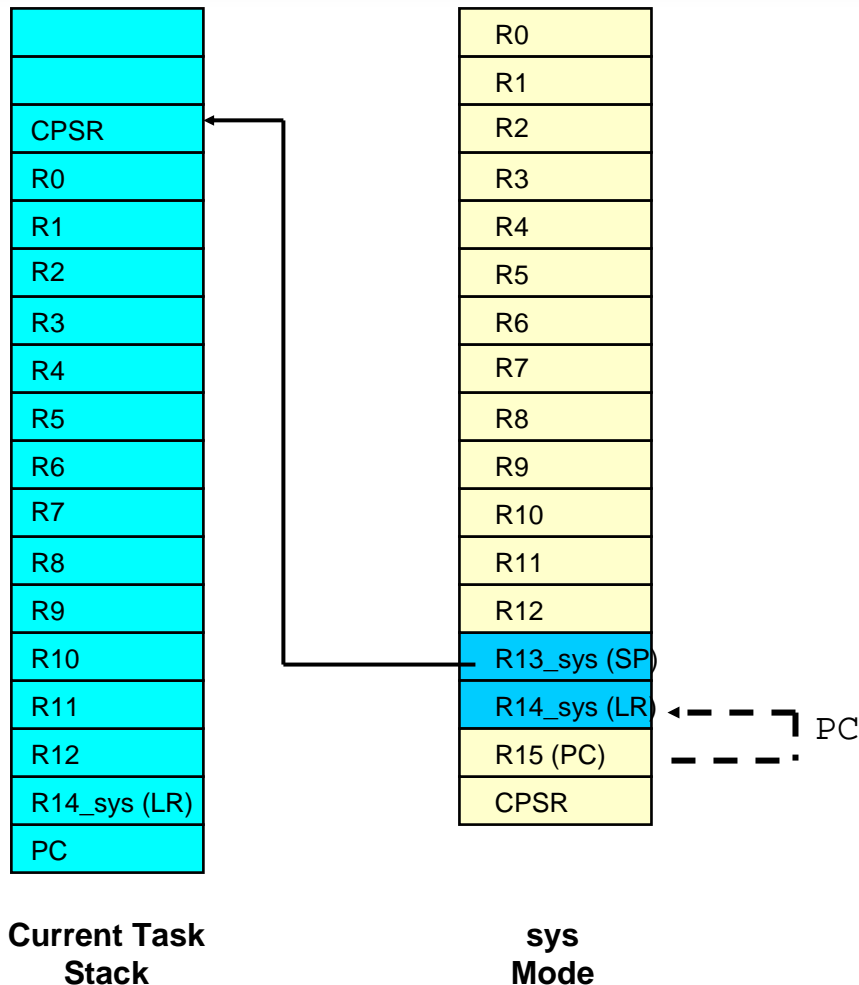   Note that **??OS_TCBCur** contains the address of a variable that contains the address of **OSTCBCur**. In fact, **??OS_TCBCur** is physically located close in memory to this code because of the limited addressing range of the **LDR** instruction.

Current Task Stack:

| CPSR |
| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R14_sys (LR) |
| PC |

Current Task Stack

SYS Mode:

| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R13_sys (SP) |
| R14_sys (LR) |
| R15 (PC) |
| CPSR |

SYS Mode

After

OSTCBCur

| OSTCBStkPtr |
| |
| |
| |

OS_TCB

	I	F	T	MODE
CPSR:	1	1	0	0x1F

Task Level Context Switch
OSCtxSw() (ARM mode)

Current Task Stack
CPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R14_sys (LR)
PC

Current Task Stack

sys Mode
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

sys Mode

PC

```
OSCtxSw:
    .
    .
    BL      OSTaskSwHook
```
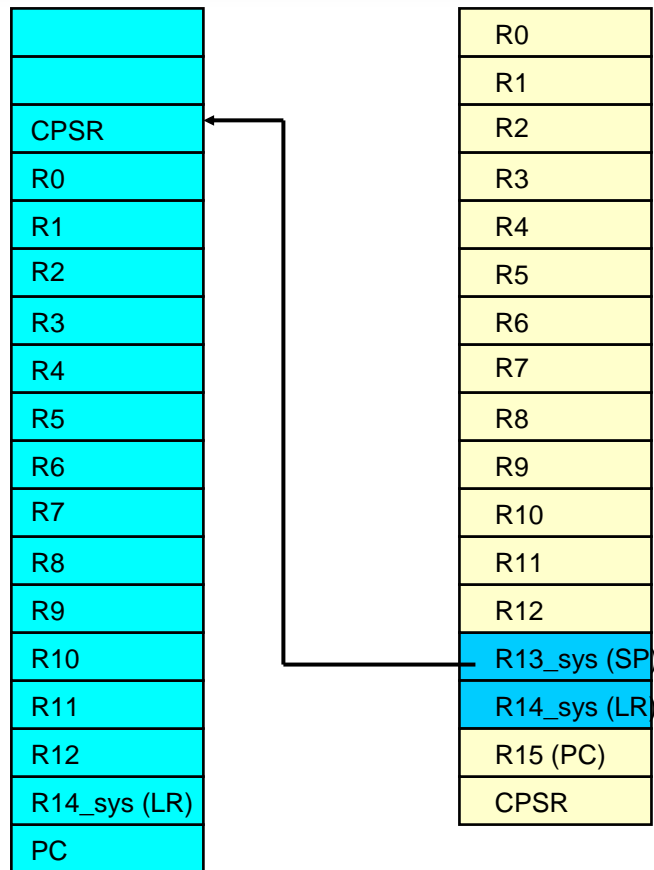
The task switch hook is called. The return address is saved in the **LR** register.

Note that **OSTaskSwHook()** is declared in **OS_CPU_C.C**.

	I	F	T	MODE
CPSR:	1	1	0	0x1F

Current Task Stack
CPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R14_sys (LR)
PC

Current Task Stack

SYS Mode
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

SYS Mode

```
OSCtxSw:
  .
  .
  .
;          OSPrioCur = OSPrioHighRdy
  LDR      R4,??OS_PrioCur
  LDR      R5,??OS_PrioHighRdy
  LDRB     R6,[R5]
  STRB     R6,[R4]
```
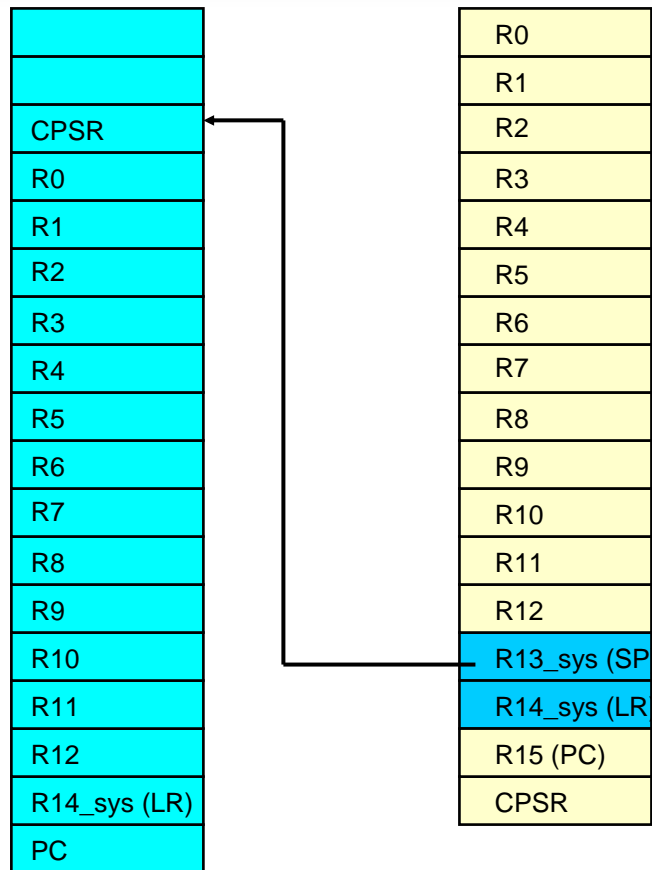
The new high priority is copied to the current priority.

	I	F	T	MODE
CPSR:	1	1	0	0x1F

Task Level Context Switch
OSCtxSw() (ARM mode)

Current Task Stack
CPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R14_sys (LR)
PC

Current Task Stack

SYS Mode
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

SYS Mode

OSTCBHighRdy

OSTCBCur

After

OS_TCB
OSTCBStkPtr

OS_TCB

```
OSCtxSw:
    .
    .
    .
;           OSTCBCur = OSTCBHighRdy
    LDR     R6,??OS_TCBHighRdy
    LDR     R4,??OS_TCBCur
    LDR     R6,[R6]
    STR     R6,[R4]
```
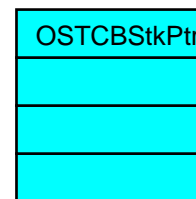
The pointer to the current **OS_TCB** is updated to point to the **OS_TCB** of the new task.

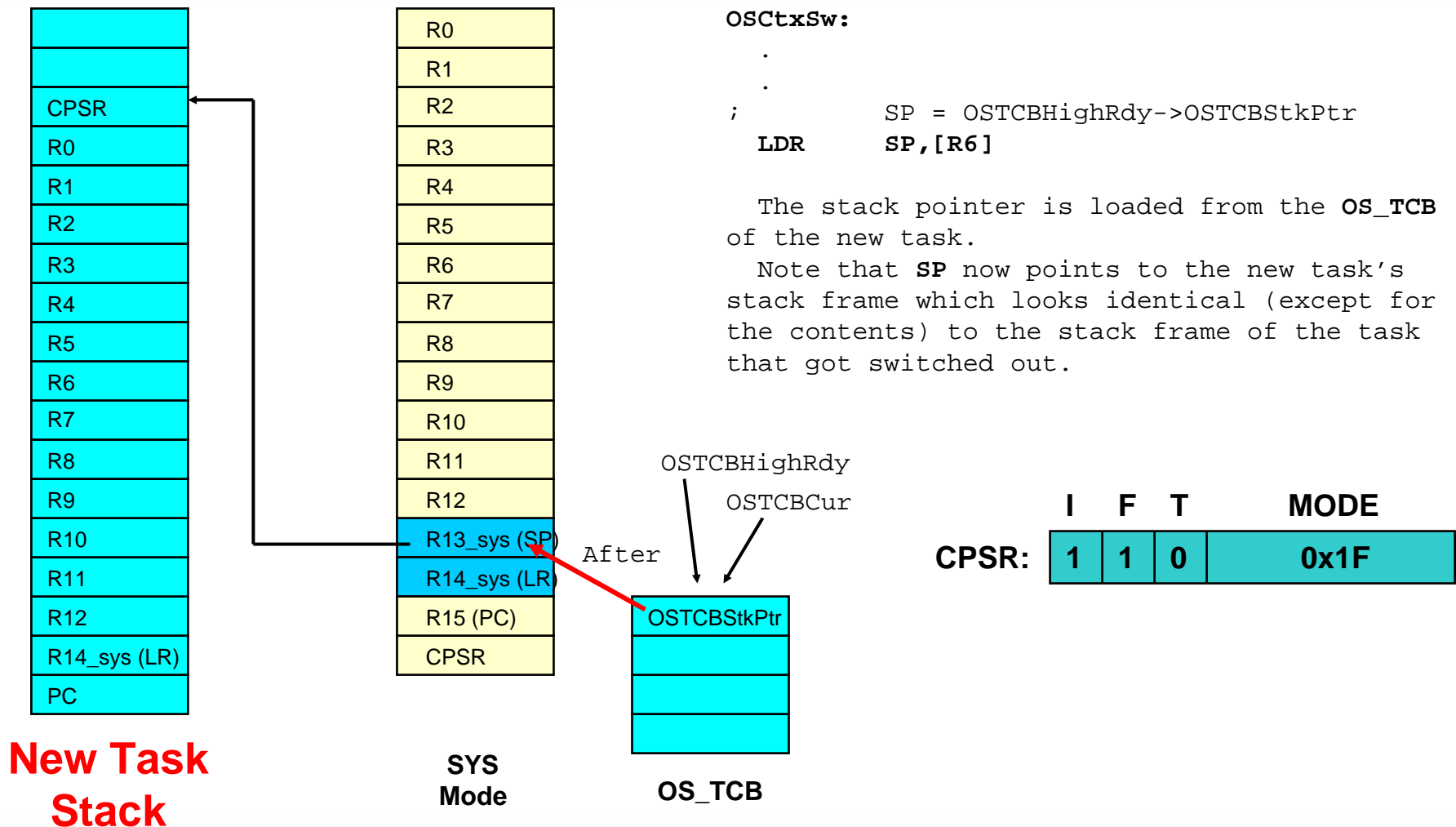|	I	F	T	MODE
CPSR:	1	1	0	0x1F

µC/OS-II Port for the ARM

Task Level Context Switch
OSCtxSw() (ARM mode)

```
OSCtxSw:
    .
    .
    .
;           SP = OSTCBHighRdy->OSTCBStkPtr
   LDR       SP,[R6]
```

The stack pointer is loaded from the **OS_TCB** of the new task.

Note that **SP** now points to the new task's stack frame which looks identical (except for the contents) to the stack frame of the task that got switched out.

New Task Stack

CPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R14_sys (LR)
PC

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

SYS Mode

After

OSTCBHighRdy
OSTCBCur

OSTCBStkPtr

OS_TCB

	I	F	T	MODE
CPSR:	1	1	0	0x1F

```
OSCtxSw:
   .
   .
   LDMFD    SP!,{R4}          ; Get CPSR
   MSR      CPSR_cxsf,R4
   LDMFD    SP!,{R0-R12,LR,PC} ; Restore registers
```

The context of the new task is pulled off the stack. After the last instruction, the CPU resumes the new task.
   Note that interrupts are still disabled when the task returns because the code returns to the scheduler. Interrupt will be re-enabled when the scheduler exits.
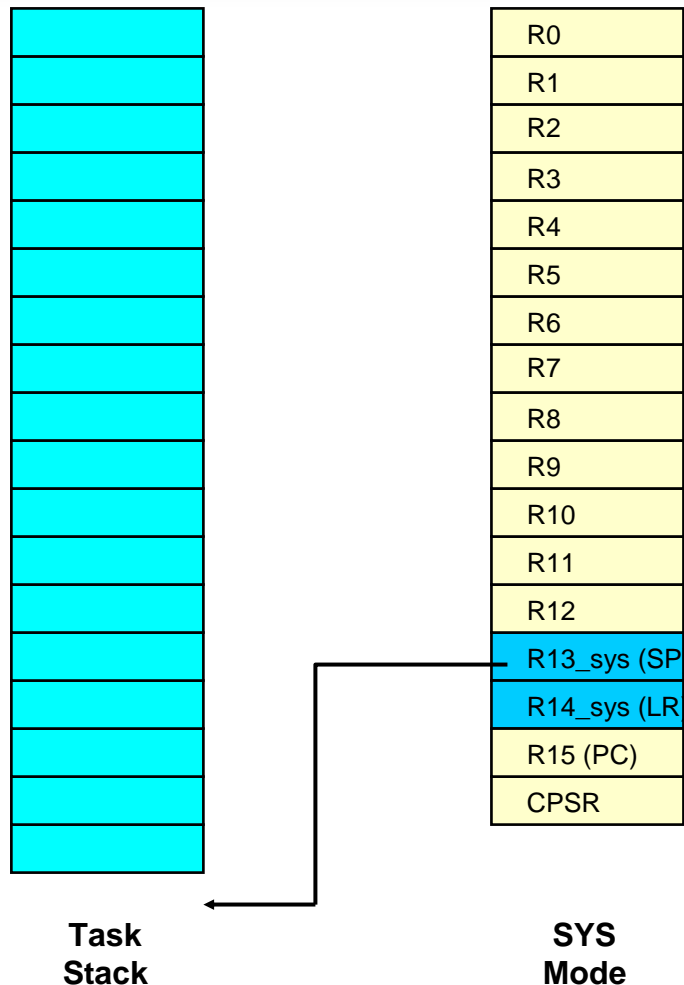
	I	F	T	MODE
CPSR:	1	1	0	0x1F

Task Level Context Switch – OSCtxSw()

Servicing Interrupts – IRQ and FIQ

Interrupt Level Context Switch – OS_IntCtxSw()

µC/OS-II Port for the ARM

Servicing Interrupts
Task running in SYS mode, ARM code

Register
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

Task Stack

SYS Mode

It is assumed that a task is running (in SYS mode) when an interrupt occurs.

The processor's **SP** (R13) points to 'some' location into the current task's stack.

The code presented here applies to both the IRQ and FIQ interrupts.

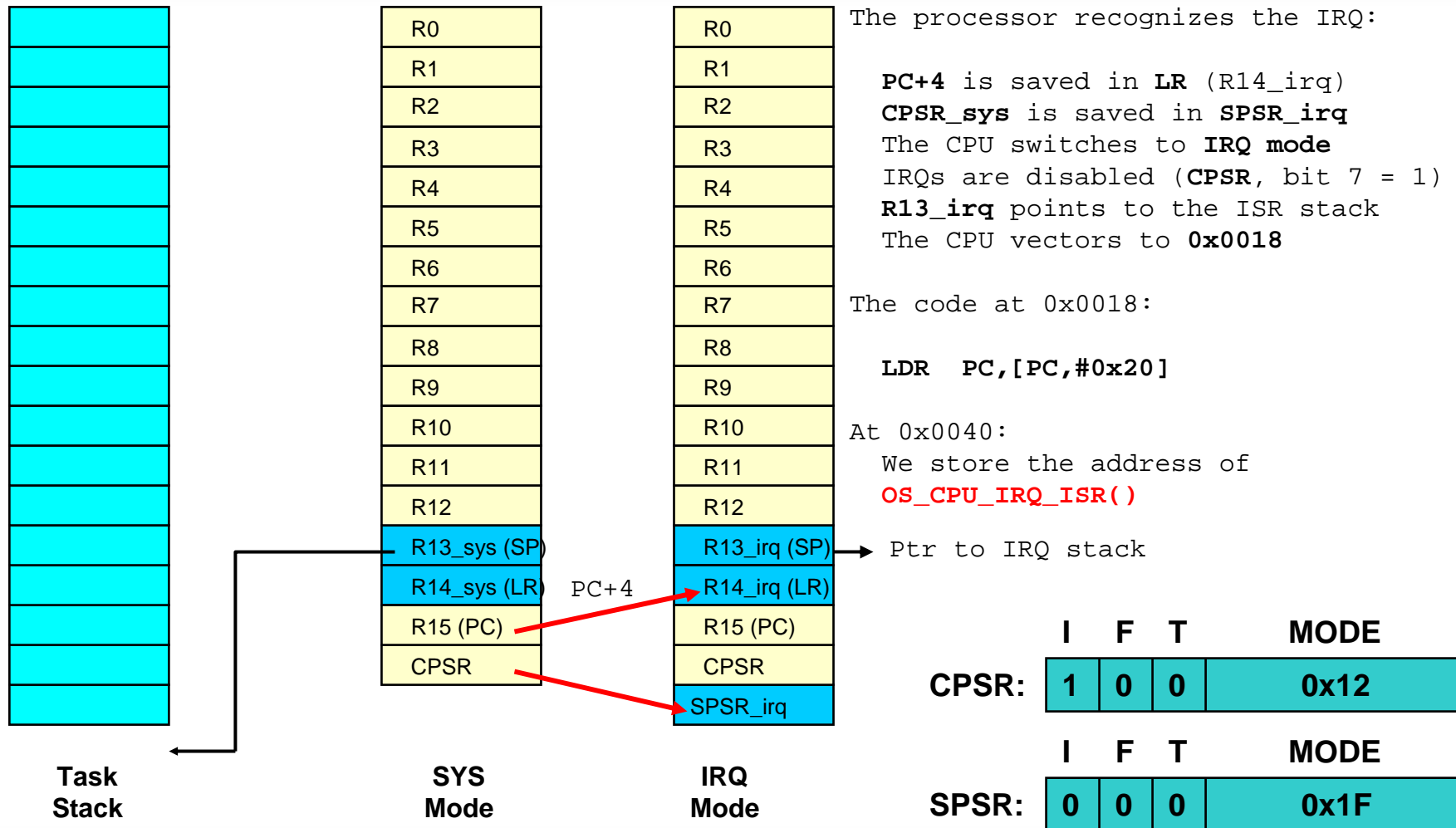The ISR is implemented as outlined in Jean J. Labrosse's book.

Specifically, your ISRs need to be written as follows:

```
Save CPU registers onto the current task's stack;
OSIntNesting++;
if (OSIntNesting == 1) {
   OSTCBCur->OSTCBStkPtr = SP;
}
Call OS_CPU_???_ISR_Handler in C;     // ??? Is IRQ or FIQ
OSIntExit();
Restore the registers from the current task's stack;
Return from interrupt;
```

	I	F	T	MODE
CPSR:	0	0	0	0x1F

Interrupt Context Switch
IRQ occurs

R0	R0
R1	R1
R2	R2
R3	R3
R4	R4
R5	R5
R6	R6
R7	R7
R8	R8
R9	R9
R10	R10
R11	R11
R12	R12
R13_sys (SP)	R13_irq (SP)
R14_sys (LR)	R14_irq (LR)
R15 (PC)	R15 (PC)
CPSR	CPSR
	SPSR_irq

Task Stack

SYS Mode

PC+4

IRQ Mode

The processor recognizes the IRQ:

PC+4 is saved in **LR** (R14_irq)
CPSR_sys is saved in **SPSR_irq**
The CPU switches to **IRQ mode**
IRQs are disabled (**CPSR**, bit 7 = 1)
R13_irq points to the ISR stack
The CPU vectors to **0x0018**

The code at 0x0018:

    LDR   PC,[PC,#0x20]

At 0x0040:
We store the address of
OS_CPU_IRQ_ISR()

Ptr to IRQ stack

	I	F	T	MODE
CPSR:	1	0	0	0x12

	I	F	T	MODE
SPSR:	0	0	0	0x1F

Interrupt Context Switch
OS_CPU_IRQ_ISR()

SYS Mode

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR
SPSR_sys

IRQ Mode

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_irq (SP)
R14_irq (LR)
R15 (PC)
CPSR
SPSR_irq

After

Before

IRQ Stack

R1
R2
R3

```
OS_CPU_IRQ_ISR
  STMFD  SP!,{R1-R3}
```

Working registers are saved onto the ISR stack because they will be used by the ISR. Only registers that will be used are saved to save clock cycles.

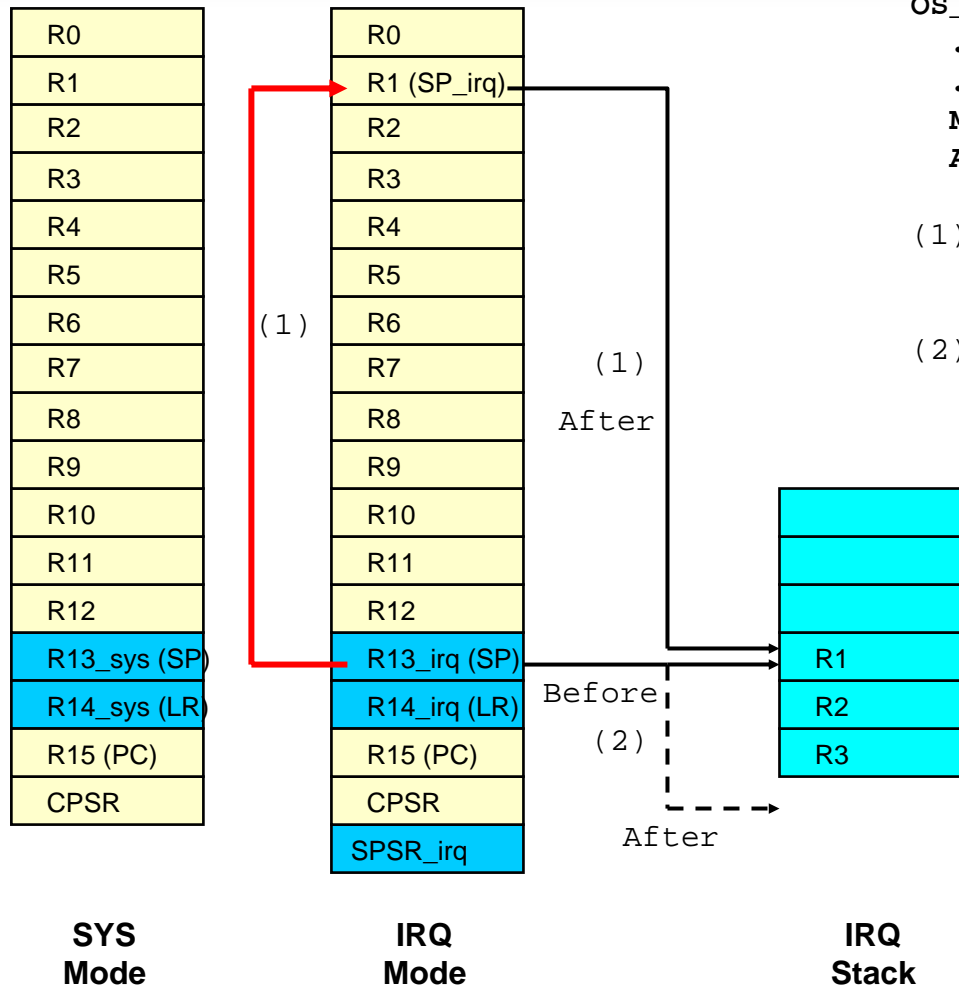|	I	F	T	MODE
CPSR:	1	0	0	0x12

	I	F	T	MODE
SPSR:	0	0	0	0x1F

Interrupt Context Switch
OS_CPU_IRQ_ISR()

```
OS_CPU_IRQ_ISR
     .
     .
   MOV  R1,SP        ; (1)
   ADD  SP,SP,#(3*4)  ; (2)
```
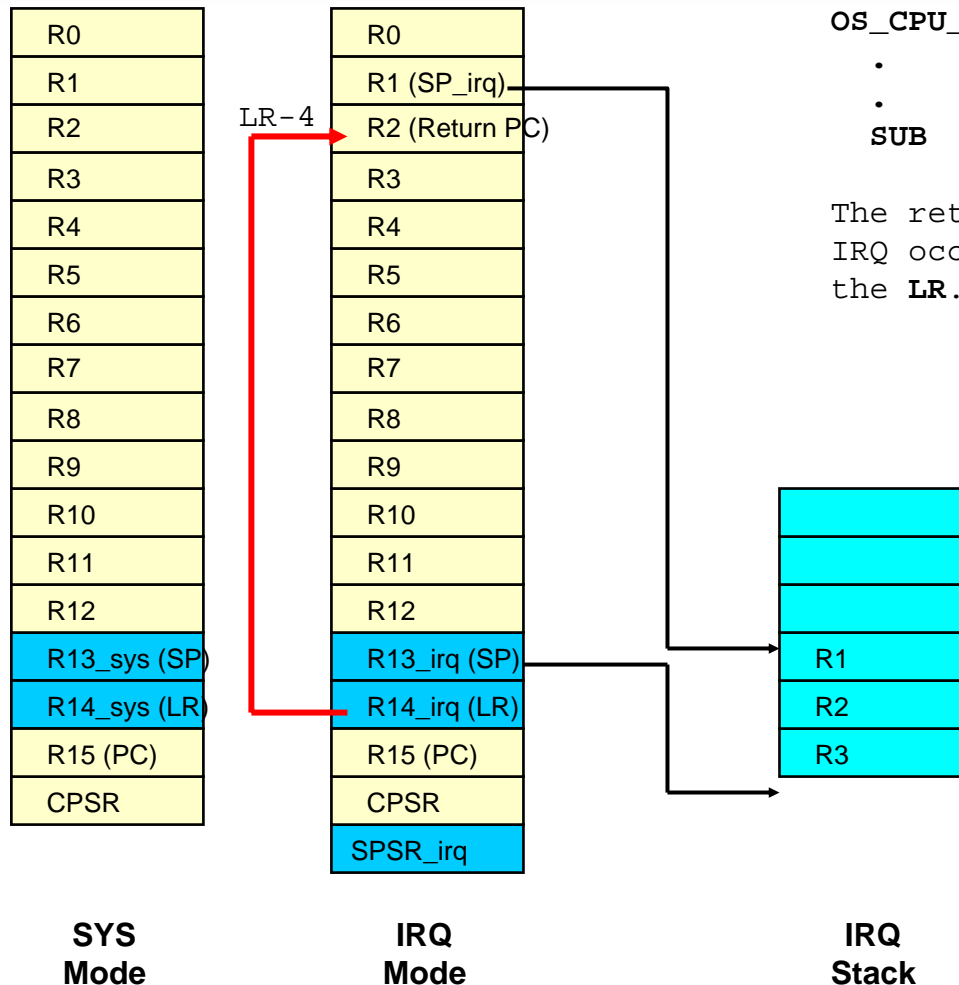
(1) The IRQ stack pointer is copied to the **R1** for future use.

(2) The IRQ stack pointer is adjusted to 'remove' the stacked data. Note that other IRQ interrupts are currently disabled so there is no danger to write over R1-R3.

SYS Mode
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

IRQ Mode
R0
R1 (SP_irq)
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_irq (SP)
R14_irq (LR)
R15 (PC)
CPSR
SPSR_irq

(1)

After

Before

(2)

After

IRQ Stack
R1
R2
R3

	I	F	T	MODE
CPSR:	1	0	0	0x12

	I	F	T	MODE
SPSR:	0	0	0	0x1F

Interrupt Context Switch
OS_CPU_IRQ_ISR()

SYS Mode
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

LR-4

IRQ Mode
R0
R1 (SP_irq)
R2 (Return PC)
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_irq (SP)
R14_irq (LR)
R15 (PC)
CPSR
SPSR_irq

IRQ Stack
R1
R2
R3

```
OS_CPU_IRQ_ISR
   .
   .
   SUB  R2,LR,#4
```

The return address is adjusted because when an IRQ occurs, the CPU saves the return **PC+4** into the **LR**.

	I	F	T	MODE
CPSR:	1	0	0	0x12

	I	F	T	MODE
SPSR:	0	0	0	0x1F

Interrupt Context Switch
OS_CPU_IRQ_ISR()

SYS Mode

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

IRQ Mode

R0
R1 (SP_irq)
R2 (Return PC)
R3 (SPSR_irq)
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_irq (SP)
R14_irq (LR)
R15 (PC)
CPSR
SPSR_irq

IRQ Stack

R1
R2
R3

```
OS_CPU_IRQ_ISR
    .
    .
    MRS  R3,SPSR
```

The **CPSR** of the interrupted task (i.e. the **SPSR_irq**) is saved for future use.

	I	F	T	MODE
CPSR:	1	0	0	0x12

	I	F	T	MODE
SPSR:	0	0	0	0x1F

Interrupt Context Switch
OS_CPU_IRQ_ISR()

SYS Mode
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

IRQ Mode
R0
R1 (SP_irq)
R2 (Return PC)
R3 (SPSR_irq)
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_irq (SP)
R14_irq (LR)
R15 (PC)
CPSR_c
SPSR_irq

0xDF

IRQ Stack
R1
R2
R3

SYS Mode

IRQ Mode

IRQ Stack

```
OS_CPU_IRQ_ISR
   .
   .
MSR  CPSR_c,#(NO_INT | SYS32_MODE)
```

The **CPSR** is changed to mode change back to SYS mode with ALL interrupts disabled.

	I	F	T	MODE
CPSR:	1	1	0	0x1F

	I	F	T	MODE
SPSR:	1	1	0	0x1F

µC/OS-II Port for the ARM

Interrupt Context Switch
OS_CPU_IRQ_ISR()

Task Stack:
- (cyan cells, empty)
- R4
- R5
- R6
- R7
- R8
- R9
- R10
- R11
- R12
- R14_sys (LR)
- Return PC

SYS Mode:
- R0
- R1 (SP_irq)
- R2 (Return PC)
- R3 (SPSR_irq)
- R4
- R5
- R6
- R7
- R8
- R9
- R10
- R11
- R12
- R13_sys (SP)
- R14_sys (LR)
- R15 (PC)
- CPSR

IRQ Stack:
- R1
- R2
- R3

(1)
(2)

```
OS_CPU_IRQ_ISR
   :
   STMFD  SP!,{R2}        ; (1) Return address
   STMFD  SP!,{R4-R12,LR} ; (2) R4 to R12
```

We now save the interrupted task's registers to its stack.

I	F	T	MODE

CPSR: | 1 | 1 | 0 | 0x1F |

Task Stack

SYS Mode

IRQ Stack

µC/OS-II Port for the ARM

Interrupt Context Switch
OS_CPU_IRQ_ISR()
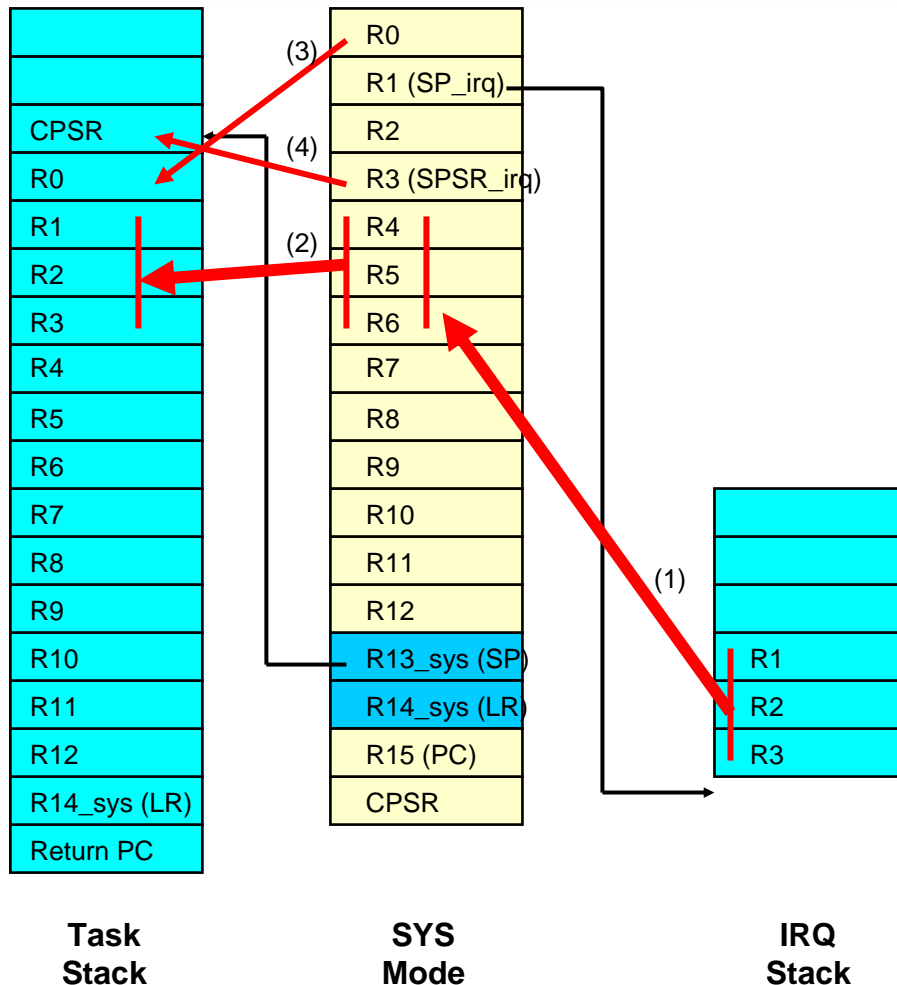

```
OS_CPU_IRQ_ISR
    :
    LDMFD   R1!,{R4-R6} ; (1) Get from IRQ stack
    STMFD   SP!,{R4-R6} ; (2) Move to task stack
    STMFD   SP!,{R0}    ; (3)

    STMFD   SP!,{R3}    ; (4) Save task's CPSR
```

We now save the remaining interrupted task registers and the interrupted task's **CPSR**.

At this point, we save the interrupted task's context onto its stack.

Task Stack

SYS Mode

IRQ Stack

I	F	T	MODE

CPSR: | 1 | 1 | 0 | 0x1F |

µC/OS-II Port for the ARM

Interrupt Context Switch
OS_CPU_IRQ_ISR()
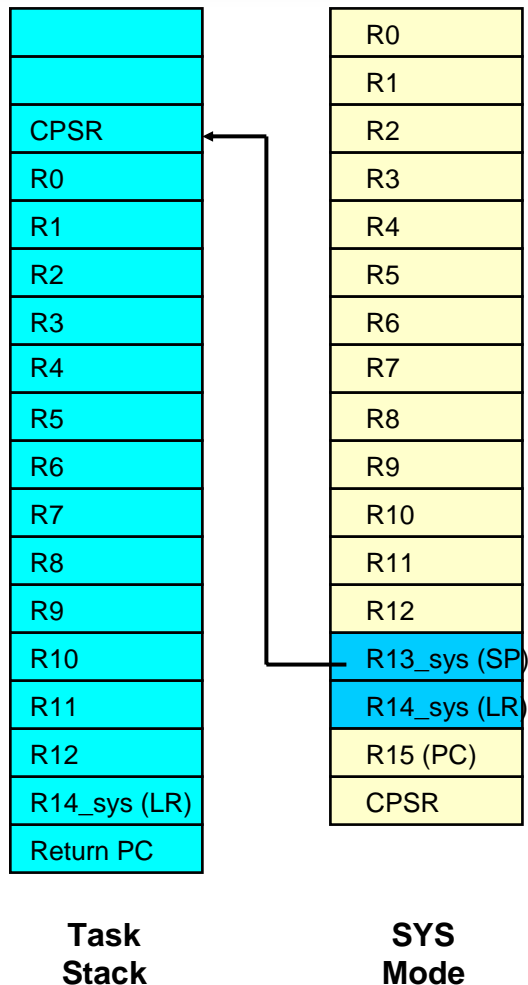

```
OS_CPU_IRQ_ISR
  :
  LDR   R0,??OSIntNesting  ; OSIntNesting++
  LDRB  R1,[R0]
  ADD   R1,R1,#1
  STRB  R1,[R0]
```

We now increment **OSIntNesting** to tell µC/OS-II that we are starting an ISR.

	I	F	T	MODE
CPSR:	1	1	0	0x1F

Interrupt Context Switch
OS_CPU_IRQ_ISR()

Task Stack
CPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R14_sys (LR)
Return PC

Task Stack

SYS Mode
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

SYS Mode
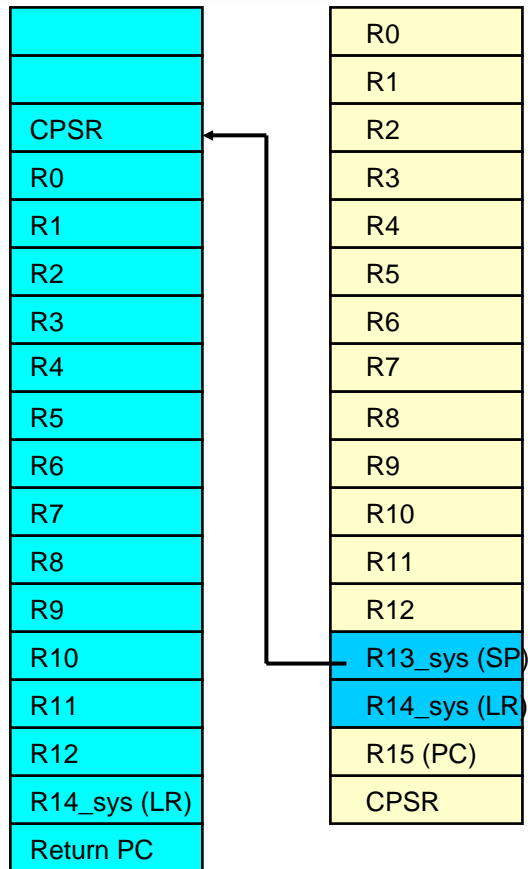
```
OS_CPU_IRQ_ISR
  :
  CMP  R1,#1            ; if (OSIntNesting == 1) {
  BNE  OS_CPU_IRQ_ISR_1
```

We now check to see if this is the first ISR and if not, we branch around the code shown on the next slide.

	I	F	T	MODE
CPSR:	1	1	0	0x1F

µC/OS-II Port for the ARM

Task Stack (cyan column, top to bottom):

CPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R14_sys (LR)
Return PC

Task Stack

SYS Mode (yellow column, top to bottom):

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

SYS Mode

After

After

OSTCBCur
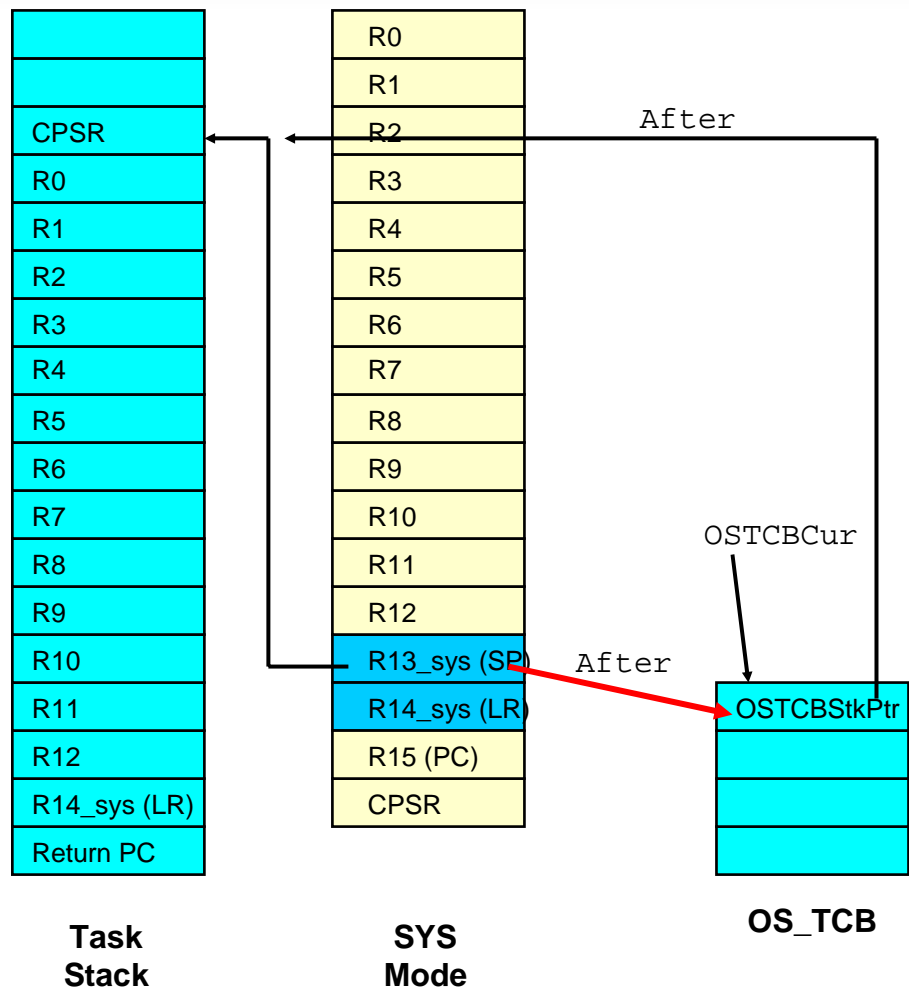
OSTCBStkPtr

OS_TCB

```
OS_CPU_IRQ_ISR
   :
;          OSTCBCur->OSTCBStkPtr = SP
   LDR    R4,??OSTCBCur
   LDR    R5,[R4]
   STR    SP,[R5]
```

If this is the first nested ISR then we save the **SP** of the current task into its **OS_TCB**.

	I	F	T	MODE
CPSR:	1	1	0	0x1F

Interrupt Context Switch
OS_CPU_IRQ_ISR()

Task Stack (cyan column, top to bottom):
CPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R14_sys (LR)
Return PC

Task Stack

SYS Mode (yellow column, top to bottom):
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

SYS Mode
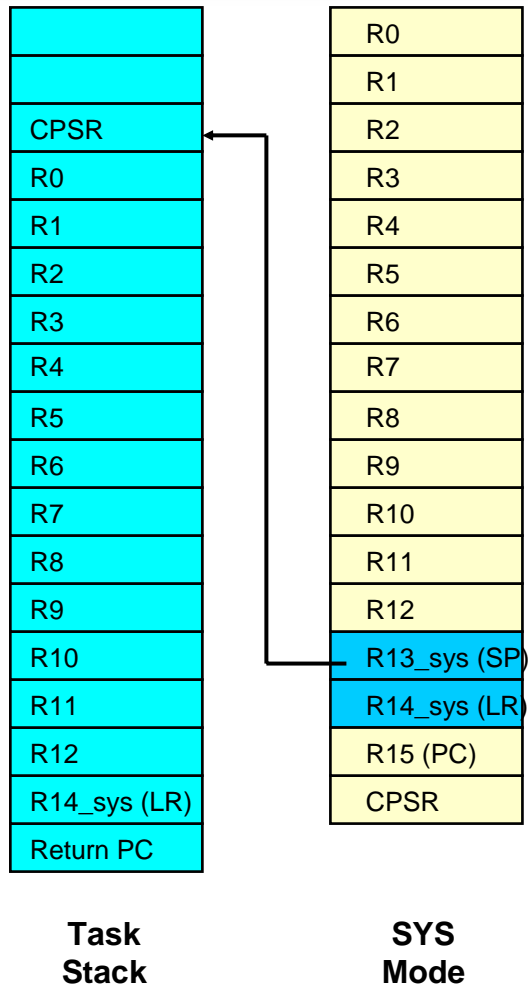
```
OS_CPU_IRQ_ISR
   :
OS_CPU_IRQ_ISR_1
   MSR    CPSR_c,#(NO_INT | IRQ32_MODE)  (1)
;
   BL     OS_CPU_IRQ_ISR_Handler          (2)
```

We now switch back to IRQ mode in order to process the ISR using the IRQ stack. This allows to reduce the RAM requirements on the task stack because all ISRs are processed on the IRQ stack.

We now call the code that will handle the ISR. We do this because it's typically easier to write this code in C instead of assembly language.

On a 32-bit bus, the CPU takes about 50 clock cycles to get to this point in the code.

I	F	T	MODE
1	1	0	0x1F

CPSR:

µC/OS-II Port for the ARM

Interrupt Context Switch
OS_CPU_IRQ_ISR()

Task Stack
CPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R14_sys (LR)
Return PC

Task Stack

SYS Mode
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR
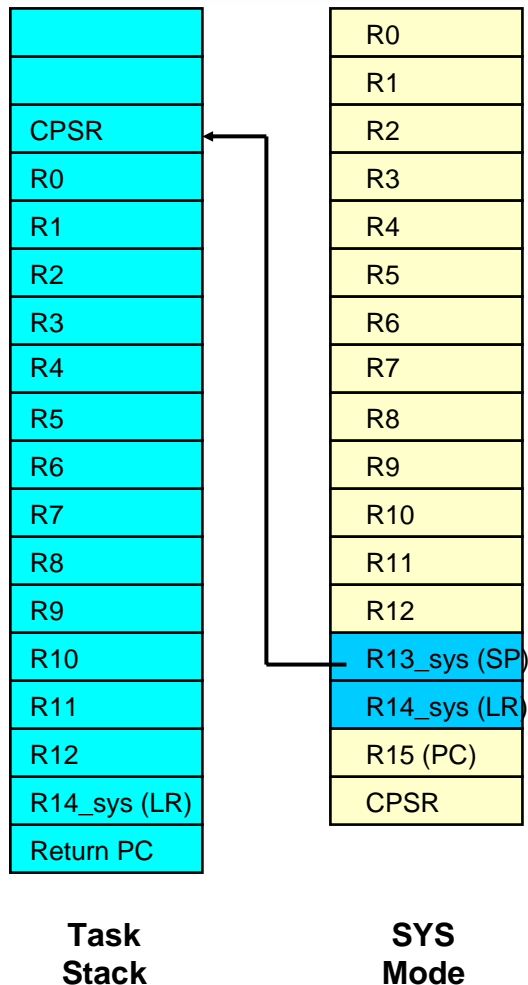
SYS Mode

```
OS_CPU_IRQ_ISR:
  :
  MSR   CPSR_c,#(NO_INT | IRQ32_MODE)   (1)

  BL    OSIntExit                       (2)
```

We now switch back to SYS mode because we are about to return to task level code.

We now call the µC/OS-II scheduler to determine whether this (or any other nested interrupt) made a higher priority task ready to run. If this is the case, **OSIntExit()** will NOT return to **OS_CPU_IRQ_ISR()** but instead, will context switch to the new, more important task (via **OSIntCtxSw()** (described later).

	I	F	T	MODE
CPSR:	1	1	0	0x1F

Task Level Context Switch
OS_CPU_IRQ_ISR()

Task Stack
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R14_svc (LR)
Return PC

Before
After

SYS Mode
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_svc (SP)
R14_svc (LR)
R15 (PC)
CPSR
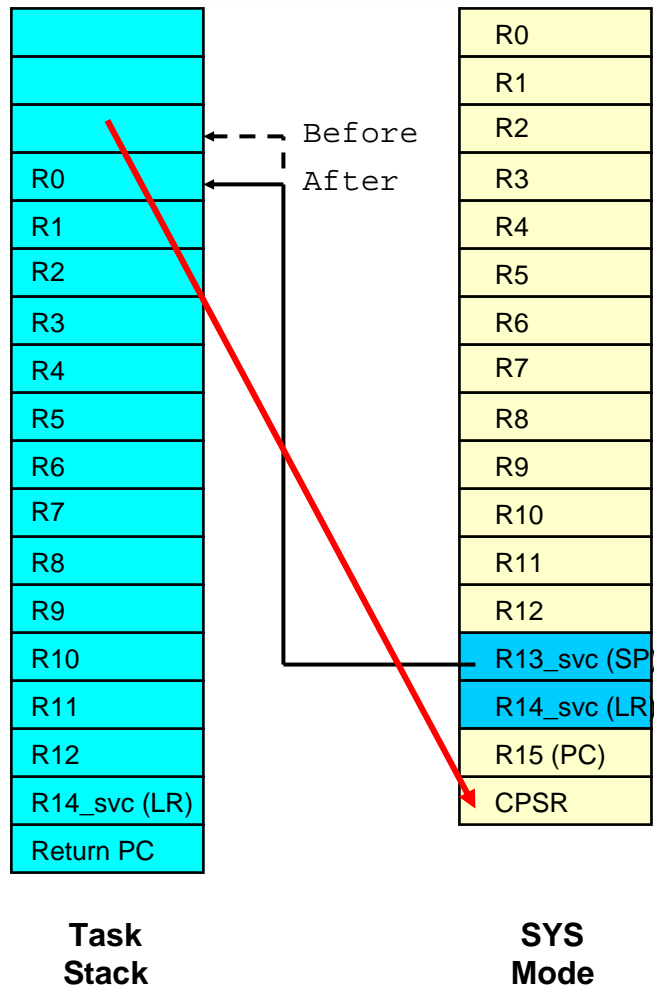
```
OS_CPU_IRQ_ISR:
     :
   LDMFD     SP!,{R4}
   MSR       CPSR_cxsf,R4
```

This code is executed if the interrupted task is still the most important task.

The **CPSR** of the interrupted task is thus retrieved from the interrupted task's stack and placed in the **CPSR** register.

You should note that interrupts are enabled because there were so prior to servicing the interrupt.

It is possible to get interrupted before we completely return to the task.

	I	F	T	MODE
CPSR:	0	0	0	0x1F

Task Level Context Switch
OS_CPU_IRQ_ISR()
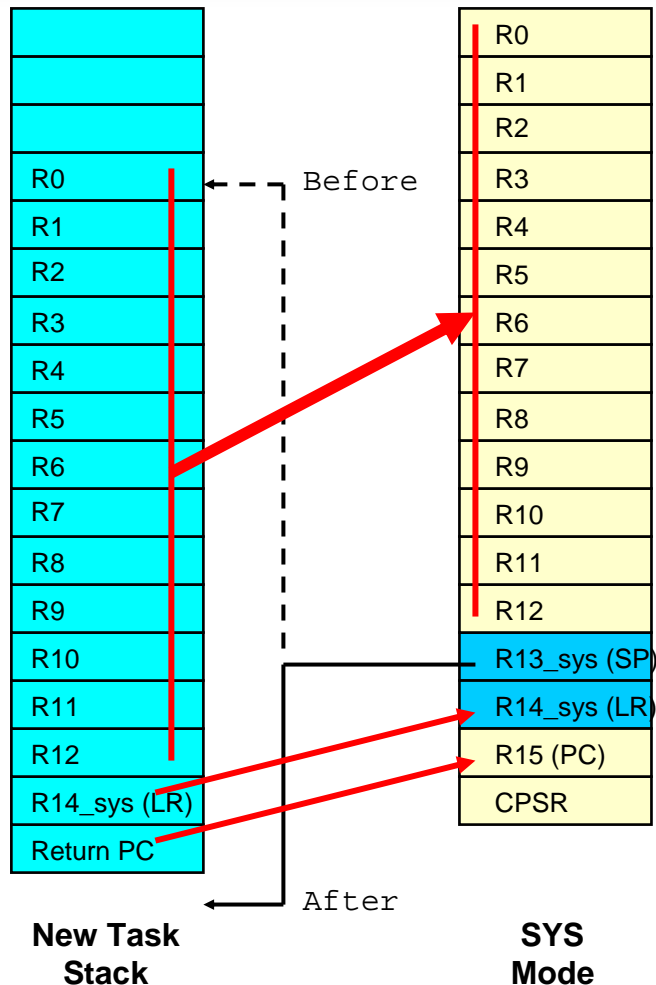

```
OSCtxSw:
    .
    .
    LDMFD    SP!,{R0-R12,LR,PC}
```

The remaining CPU registers are pulled off the stack.

After this instruction, the CPU resumes the interrupted task.
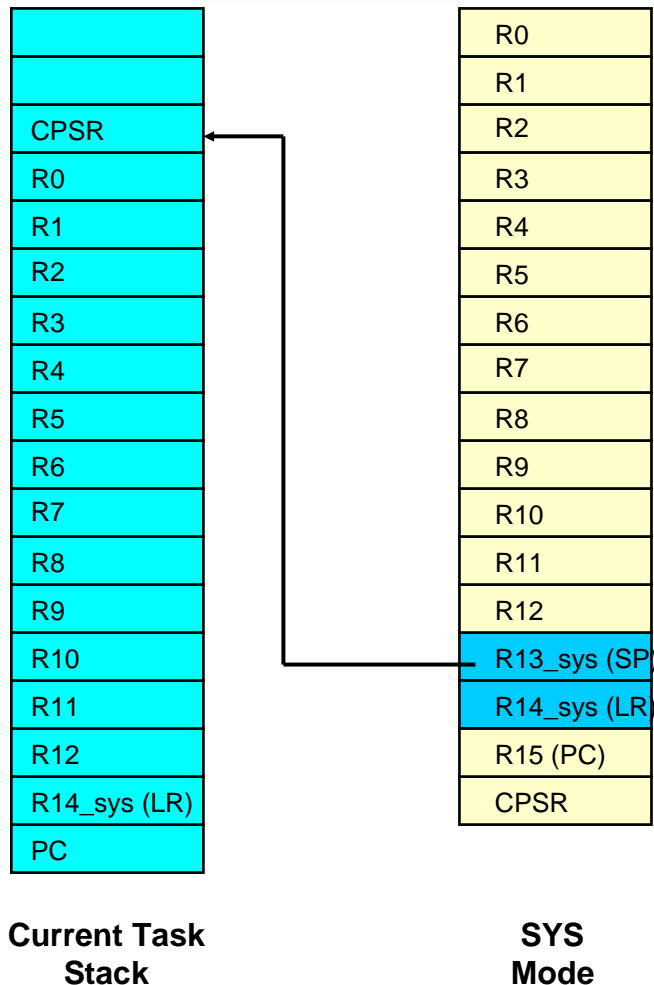
	I	F	T	MODE
CPSR:	0	0	0	0x1F

New Task Stack

SYS Mode

µC/OS-II Port for the ARM

Task Level Context Switch – OSCtxSw()

Servicing Interrupts – IRQ or FIQ

Interrupt Level Context Switch OSIntCtxSw()

Interrupt Context Switch
OSIntCtxSw()

OSIntCtxSw() is called by **OSIntExit()** if µC/OS-II determines that there is a more important task to run than the interrupted task. In this case, the CPU is in SYS mode with interrupts disabled and the SP is pointing to the interrupted task's stack.

Note that the ISR has already saved the SP into the interrupted task's OS_TCB.

	I	F	T	MODE
CPSR:	1	1	0	0x1F

Current Task Stack

CPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R14_sys (LR)
PC

SYS Mode

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

Interrupt Level Context Switch
OSIntCtxSw()

Current Task Stack:

CPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R14_svc (LR)
PC

**Current Task
Stack**

SYS Mode:

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

**SYS
Mode**
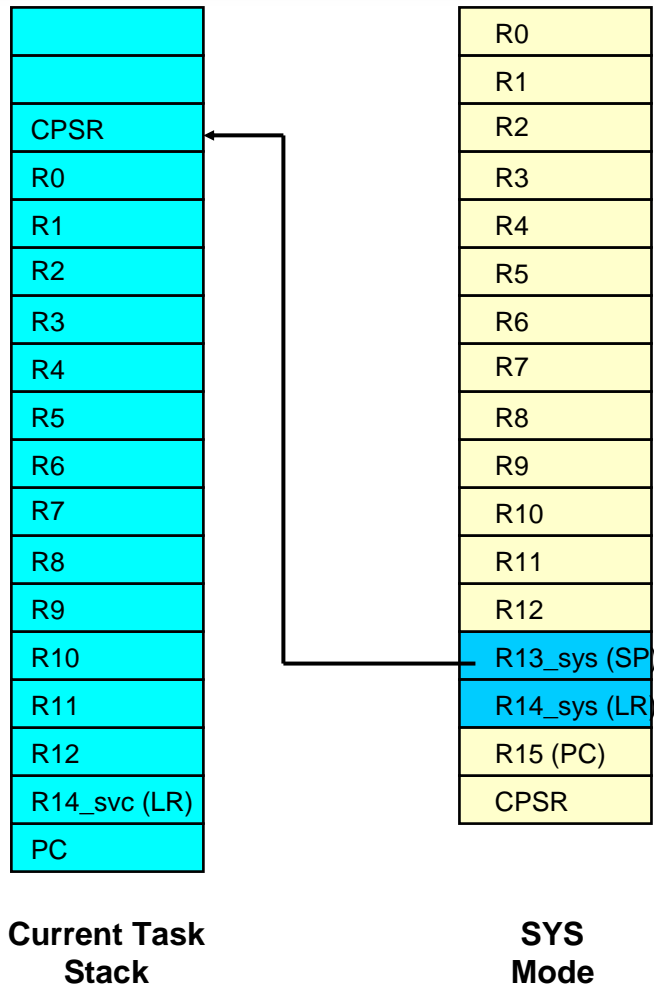
```
OSIntCtxSw:
  BL        OSTaskSwHook
```

The task switch hook is called.

I	F	T	MODE

CPSR: | 1 | 1 | 0 | 0x1F |

Current Task Stack
CPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R14_svc (LR)
PC

Current Task Stack

SYS Mode
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

SYS Mode
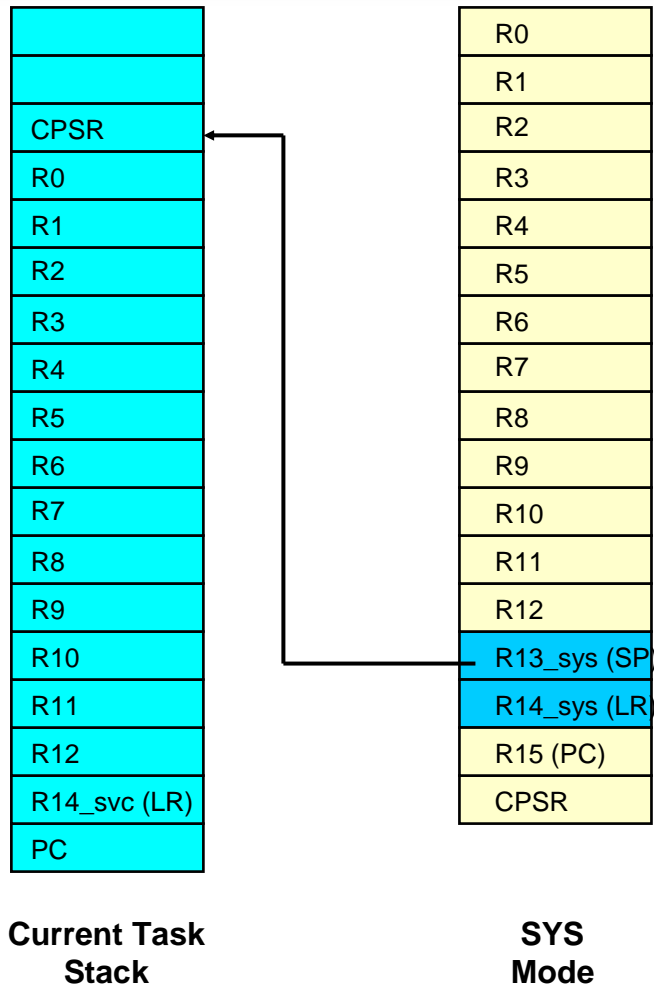
```
OSIntCtxSw:
    .
    .
    .
;           OSPrioCur = OSPrioHighRdy
    LDR     R4,??OS_PrioCur
    LDR     R5,??OS_PrioHighRdy
    LDRB    R5,[r5]
    STRB    R5,[r4]
```

The new high priority is copied to the current priority.

	I	F	T	MODE
CPSR:	1	1	0	0x1F

Current Task Stack		SYS Mode	
		R0	
		R1	
CPSR		R2	
R0		R3	
R1		R4	
R2		R5	
R3		R6	
R4		R7	
R5		R8	
R6		R9	
R7		R10	
R8		R11	
R9		R12	
R10		R13_sys (SP)	
R11		R14_sys (LR)	
R12		R15 (PC)	
R14_svc (LR)		CPSR	
PC			

Current Task Stack

SYS Mode
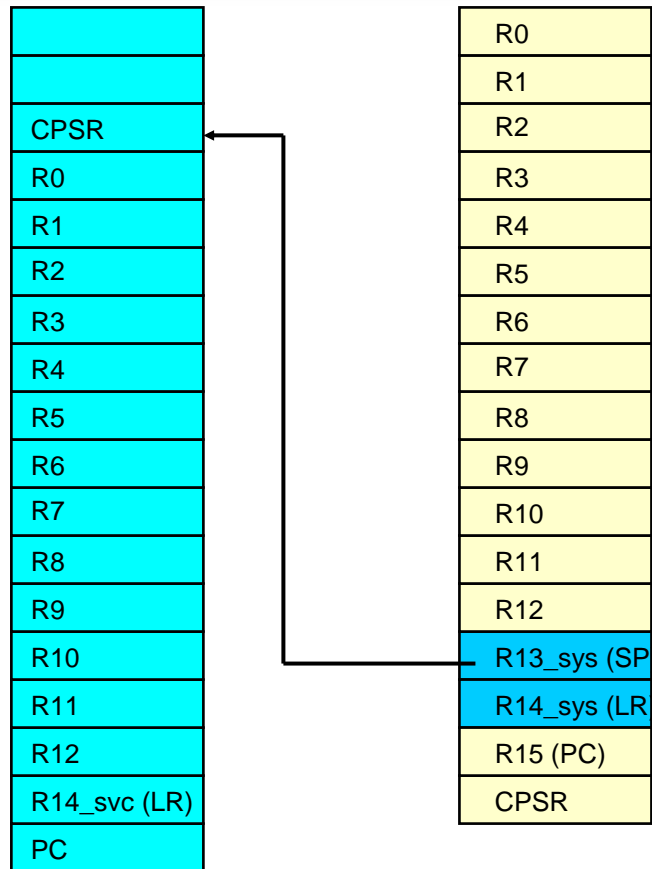
```
OSIntCtxSw:
    .
    .
    .
;           OSTCBCur = OSTCBHighRdy
    LDR     R6,??OS_TCBHighRdy
    LDR     R4,??OS_TCBCur
    LDR     R6,[R6]
    STR     R6,[R4]
```

The pointer to the current **OS_TCB** is updated to point to the **OS_TCB** of the new task.

OSTCBHighRdy

OSTCBCur

After

| OSTCBStkPtr |
| |
| |
| |

OS_TCB

	I	F	T	MODE
CPSR:	1	1	0	0x1F

Interrupt Level Context Switch
OSIntCtxSw()

New Task Stack

CPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R14_svc (LR)
PC

SYS Mode

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_sys (SP)
R14_sys (LR)
R15 (PC)
CPSR

After

OSTCBHighRdy
OSTCBCur

OS_TCB

OSTCBStkPtr

```
OSIntCtxSw:
    .
    .
    .
;           SP = OSTCBHighRdy->OSTCBStkPtr
    LDR     SP,[R6]
```

The stack pointer is loaded from the **OS_TCB** of the new task.

Note that **SP** now points to the new task's stack frame which looks identical (except for the contents) to the stack frame of the task that got switched out.

I	F	T	MODE

CPSR:

I	F	T	MODE
1	1	0	0x1F

Interrupt Level Context Switch
OSIntCtxSw()

New Task Stack: R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R14_sys (LR), PC

SYS Mode: R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13_sys (SP), R14_sys (LR), R15 (PC), CPSR

Before / After

```
OSIntCtxSw:
  .
  .
  LDMFD    SP!,{R4}
  MSR      CPSR_cxsf,R4
```

The **CPSR** of the new task is retrieved from the new task's stack and placed in the **CPSR** register.

Also note that we are assuming here that this is the first time that the new task to resume has been executed and thus interrupts are enabled for that task. If the task had been previously switched out, it's possible that the I-bit and F-bit would have been set to **1**.

I	F	T	MODE
CPSR: 0	0	0	0x1F

Interrupt Level Context Switch
OSIntCtxSw()
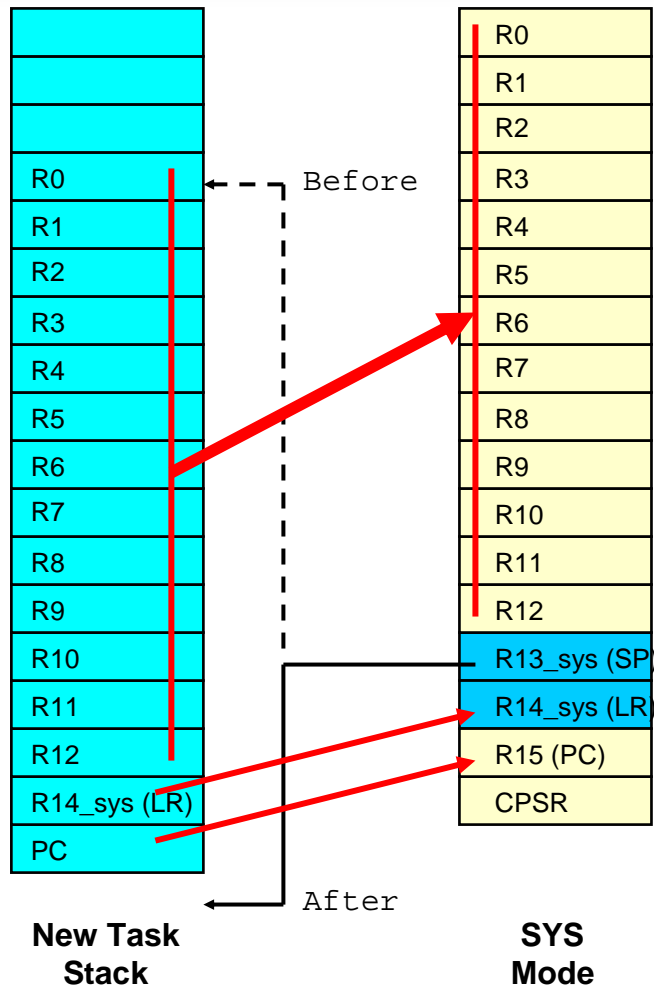

```
OSIntCtxSw:
    .
    .
    LDMFD    SP!,{R0-R12,LR,PC}
```

The remaining CPU registers are pulled off the stack. After this instruction, the CPU resumes the new task.
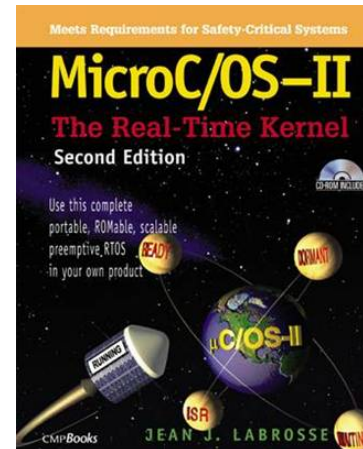
On a 32-bit bus, **OSIntCtxSw()** takes about 30 *clock cycles* to execute (excluding the call to **OSTaskSwHook()**).

	I	F	T	MODE
CPSR:	0	0	0	0x1F

New Task Stack

SYS Mode

µC/OS-II Port for the ARM

References

"µC/OS-II, The Real-Time Kernel,
    2nd Edition"

    Jean J. Labrosse, CMP Books
    ISBN 1-57820-103-9
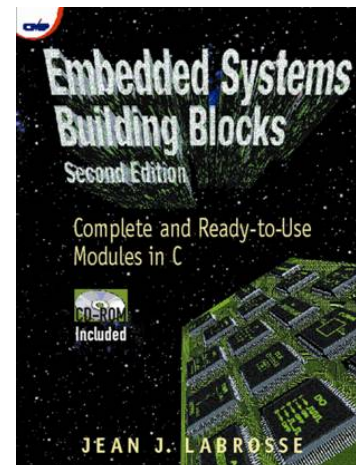
Chinese   Korean
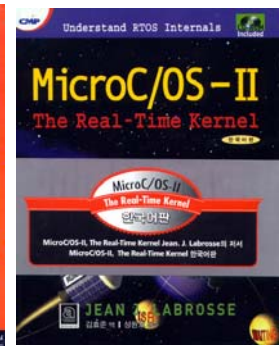
"Embedded Systems Building Blocks,

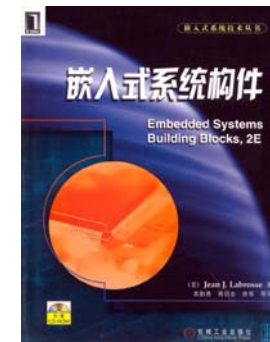   **Complete and Ready-to-Use Modules in C"**

Jean J. Labrosse, CMP Books
ISBN 0-97930-604-1

Chinese   Korean