Micrium

µC/OS-II for ARM Processors

(Supplement to AN-1014)

ARM and Thumb Mode

www.Micrium.com

Legend

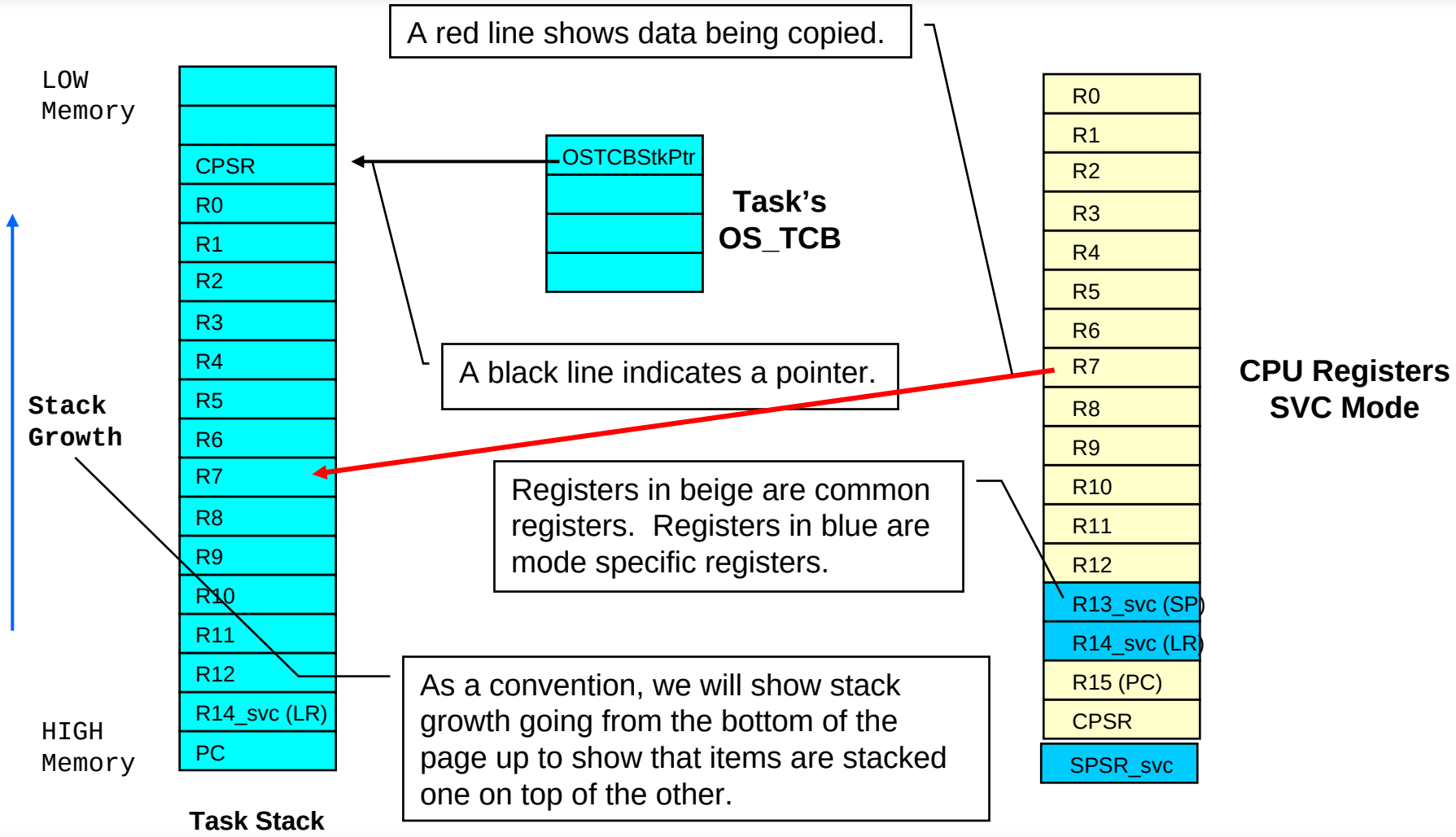
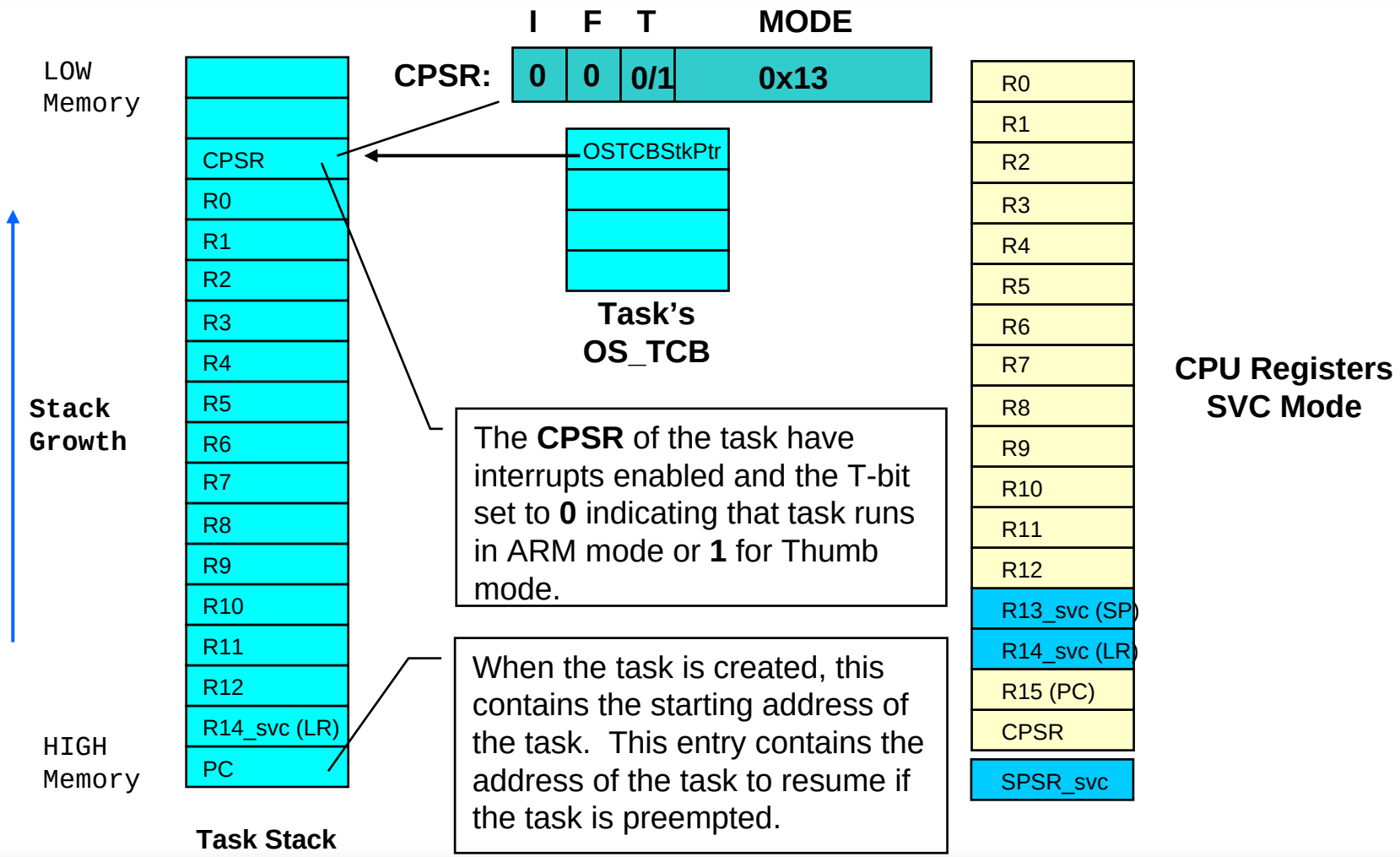


Table of Contents

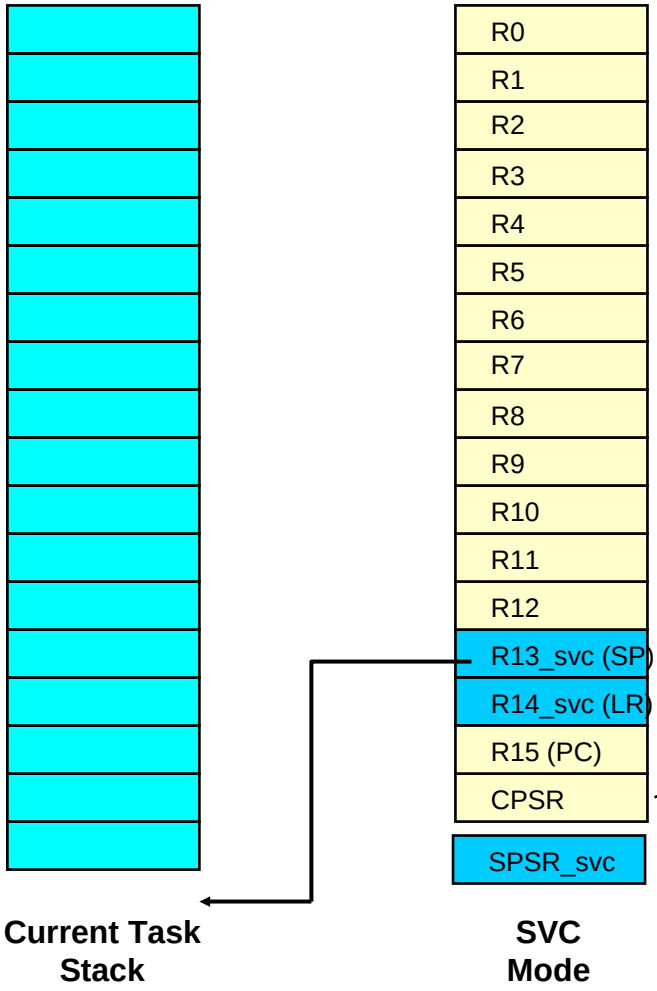
Task Level Context Switch

Task Stack Frame (when task is created)



Task Level Context Switch

Task running



A task is assumed to run (ARM or Thumb mode), uses the SVC registers (Mode = 0x13).

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:

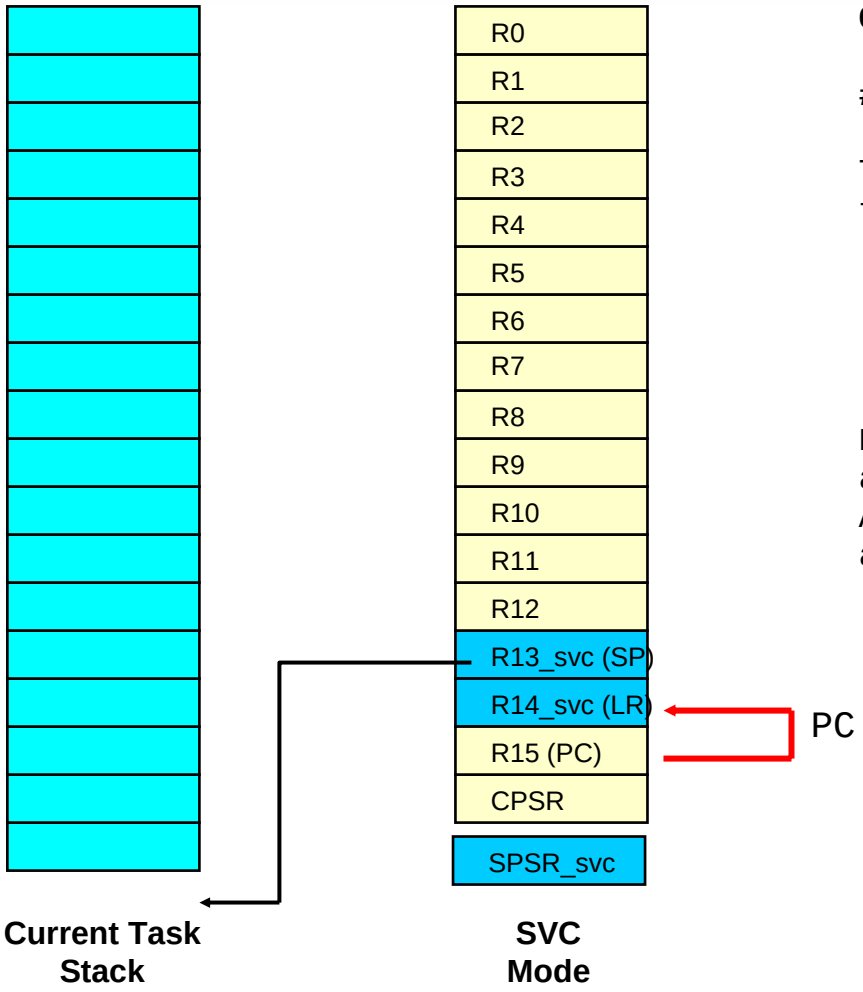
```
OSTimeDly(1)          // Delay task for 1 tick
  OS_Sched();
    OS_TASK_SW();     // Macro that invokes …
      OSCtxSw();      // … this function.
```

	I	F	T	MODE
CPSR:	1	1	0/1	0x13

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



OS_TASK_SW() is a macros declared as follows:

```
#define  OS_TASK_SW()  OSCtxSw()
```

This macro produces something similar to the following code:

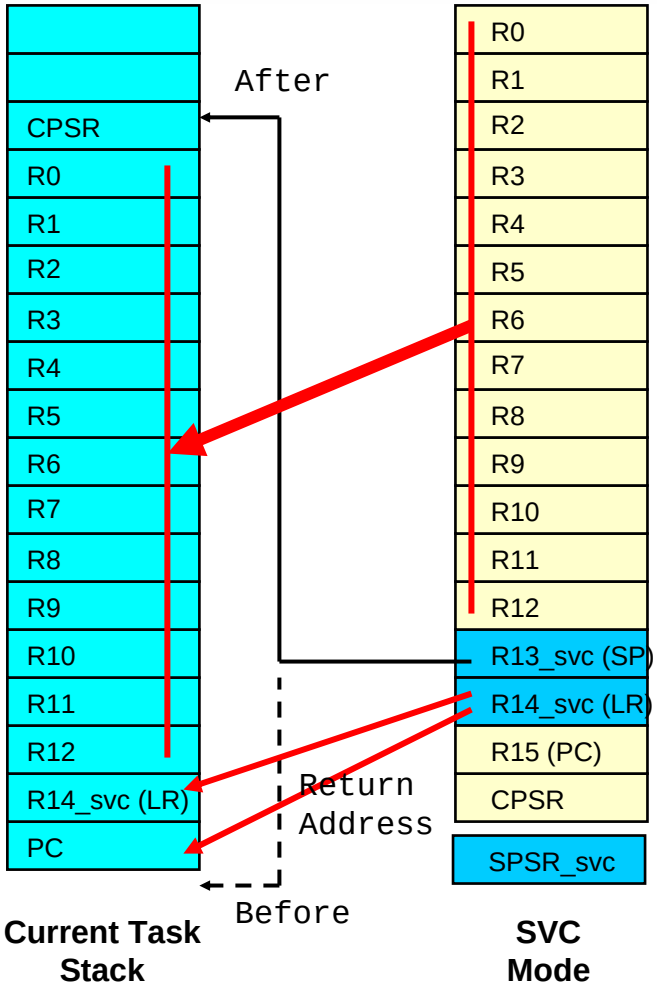
```
  LDR  R0,??OS_CtxSw
  MOV  LR,PC
  BX   R0
```

Note that the compiler generates interwork code and thus, µC/OS-II can either be compiled for ARM or Thumb mode. OSCtxSw() is actually assumed to be code in ARM mode.

	I	F	T	MODE
CPSR:	1	1	0/1	0x13

Task Level Context Switch

OSCtxSw() (ARM mode)




```
OSCtxSw:
  STMFD   SP!, {LR}       ; Push return address
  STMFD   SP!, {LR}
  STMFD   SP!, {R0-R12}   ; Push registers
  MRS     R4,  CPSR       ; Push current CPSR
  TST     LR, #1          ; See if called from
                          ;   Thumb mode
  ORRNE   R4,  R4, #0x20  ; If yes,
                          ;   Set the T-bit
  STMFD   SP!, {R4}
```

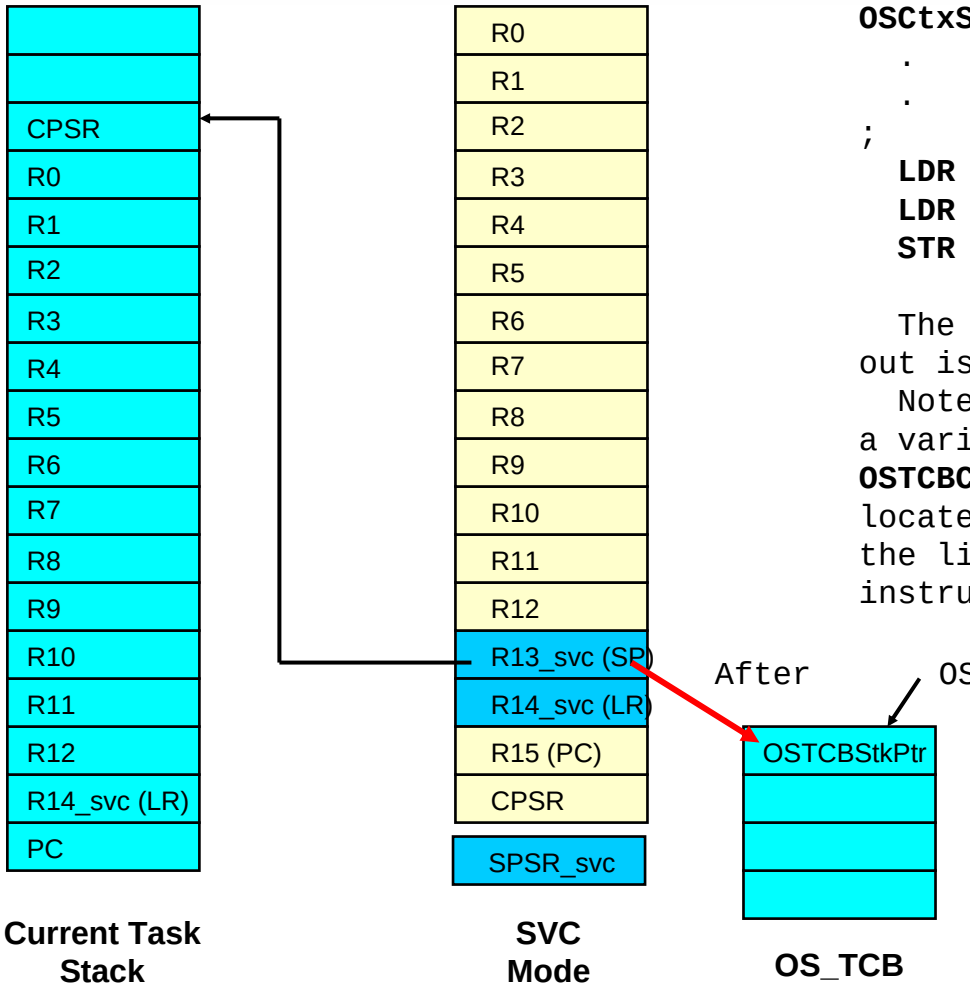
OSCtxSw() starts by saving the current task's context onto the current task's stack.

You should note that the least significant bit of the **LR** is either set to 1 (indicating that OSCtxSw() was called from Thumb mode) or 0 (indicating it was called from an ARM mode function). If called by a Thumb function, we set the T bit in the saved CPSR so that we can resume Thumb mode when the task is resumed.

	I	F	T	MODE
CPSR:	**1**	**1**	**0**	**0x13**

Task Level Context Switch

OSCtxSw() (ARM mode)




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

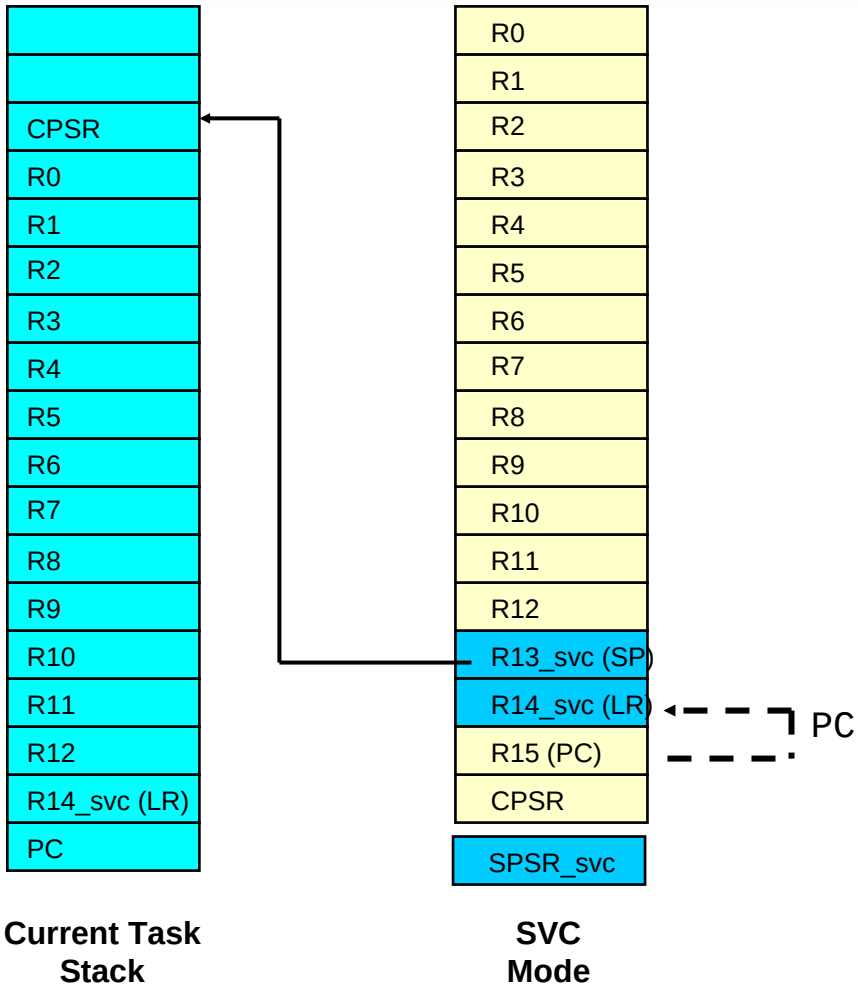
The stack pointer of the task being switched out is saved into the **OS_TCB** of that task.

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.

	I	F	T	MODE
CPSR:	**1**	**1**	**0**	**0x13**

Task Level Context Switch

OSCtxSw() (ARM mode)




```
OSCtxSw:
   .
   .
   LDR   R0, ??OS_TaskSwHook
   MOV   LR,PC
   BX    R0
```

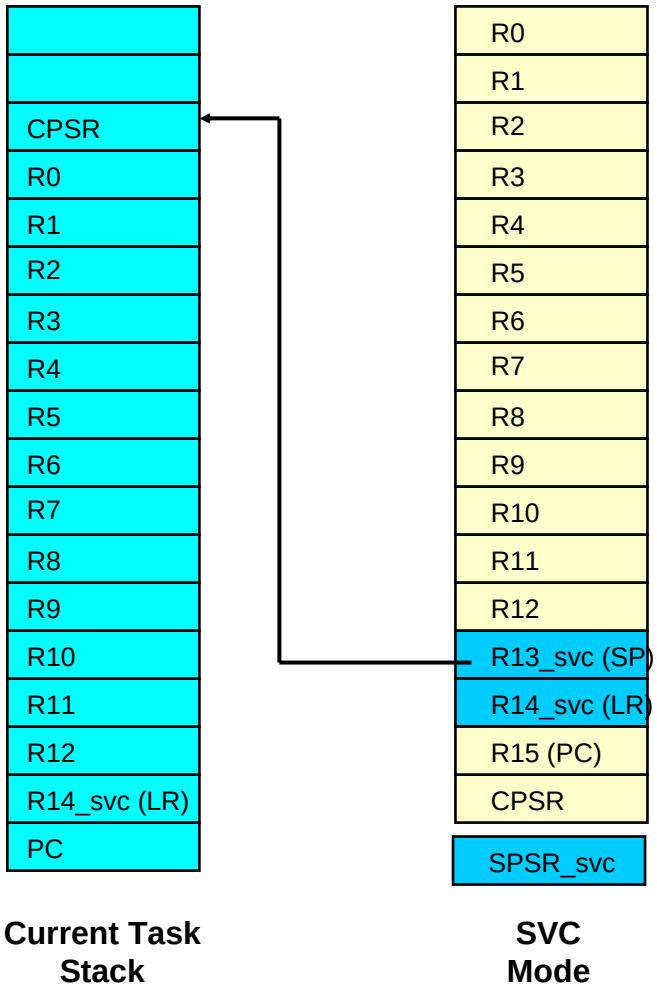
The task switch hook is called using a BX instruction allowing OSTaskSwHook() to be an ARM or Thumb mode function. The return address is saved in the **LR** register.

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

	I	F	T	MODE
CPSR:	**1**	**1**	**0**	**0x13**

Task Level Context Switch

OSCtxSw() (ARM mode)



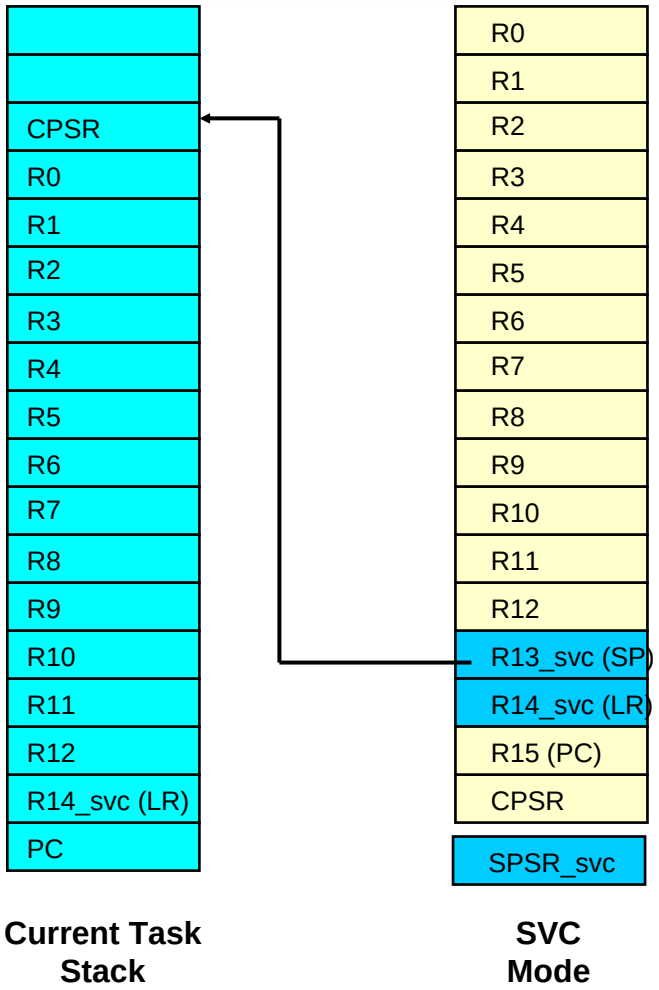

```
OSCtxSw:
   .
   .
;          OSPrioCur = OSPrioHighRdy
  LDR      R4,??OS_PrioCur
  LDR      R5,??OS_PrioHighRdy
  LDRB     R6,[R5]
  STRB     R6,[R4]
```

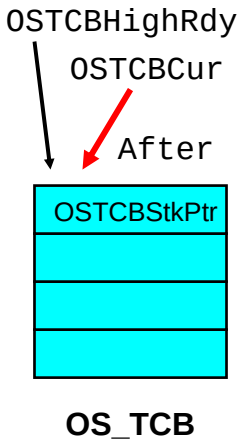
The new high priority is copied to the current priority.

	I	F	T	MODE
CPSR:	1	1	0	0x13

Task Level Context Switch

OSCtxSw() (ARM mode)





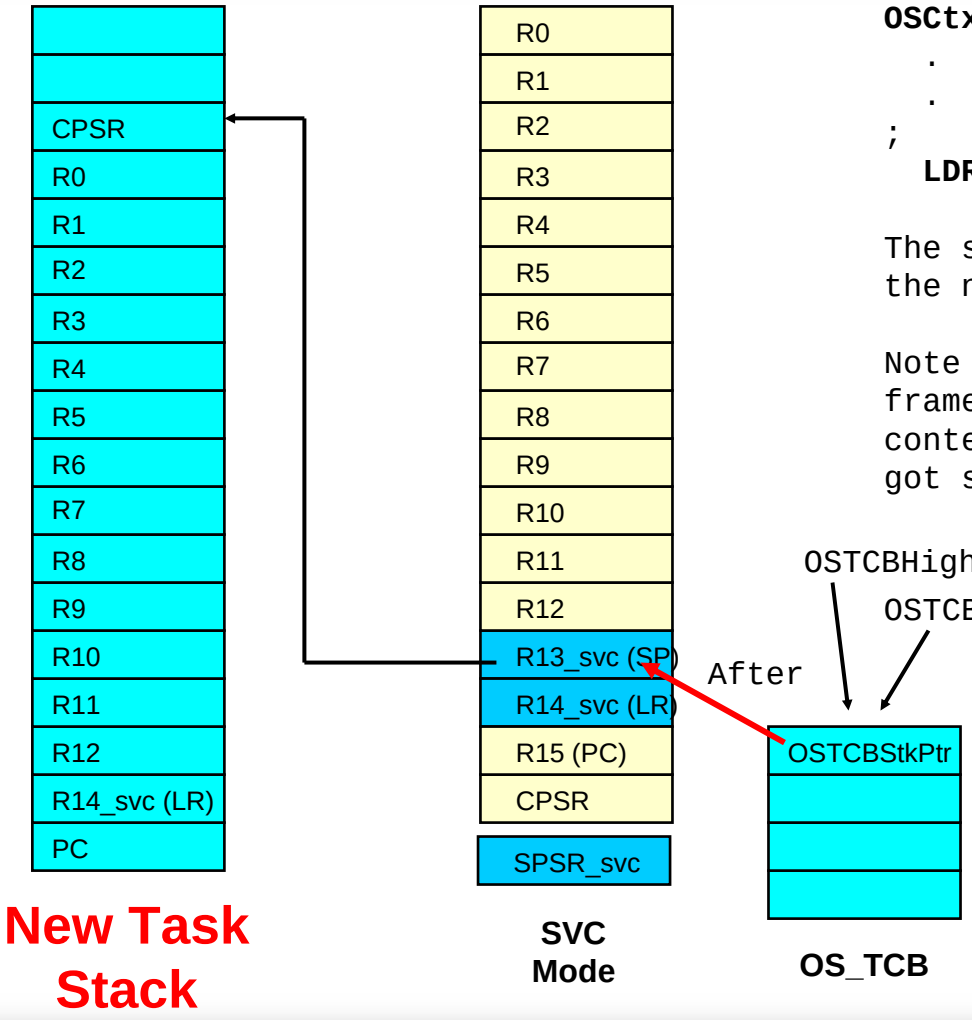

```
OSCtxSw:
   .
   .
;          OSTCBCur = OSTCBHighRdy
  LDR      R6,??OS_TCBHighRdy
  LDR      R4,??OS_TCBCur
  LDR      R6,[R6]
  STR      R6,[R4]
```

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	0x13

Task Level Context Switch

OSCtxSw() (ARM mode)




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

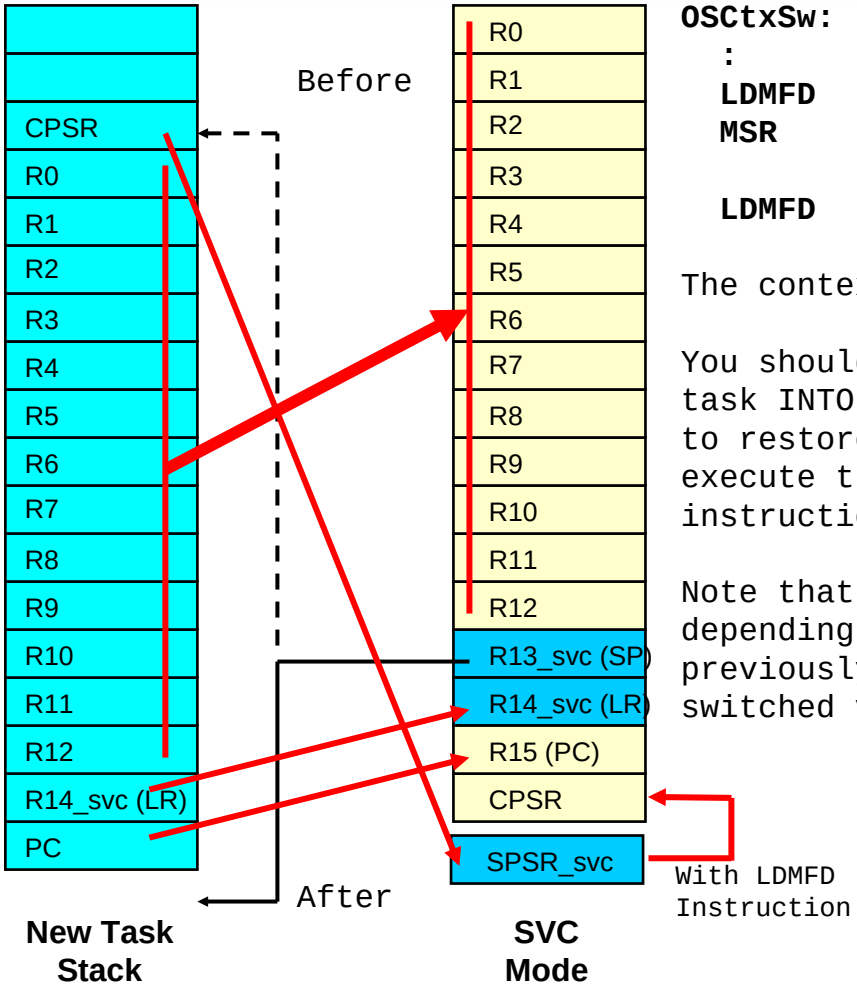
The stack pointer is loaded from the **OS_TCB** of the new task.

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.

	I	F	T	MODE
CPSR:	**1**	**1**	**0**	**0x13**

Task Level Context Switch

OSCtxSw() (ARM mode)




```
OSCtxSw:
  :
  LDMFD   SP!, {R4}             ; Pop new task's CPSR
  MSR     SPSR_cxsf, R4

  LDMFD   SP!, {R0-R12,LR,PC}^ ; Pop new task's context
```

The context of the new task is pulled off the stack.

You should notice that we restore the CPSR of the new task INTO the SPSR register. The reason we do this is to restore both the CPSR and PC at the same time when we execute the LDMFD instruction. After the last instruction, the CPU resumes the new task.

Note that the interrupts are either enabled or disabled depending on whether we return to a task that was previously interrupted or, a task that was context switched via OSCtxSw().

	I	F	T	MODE
CPSR:	1	1	0	0x13

Task Level Context Switch – OSCtxSw()

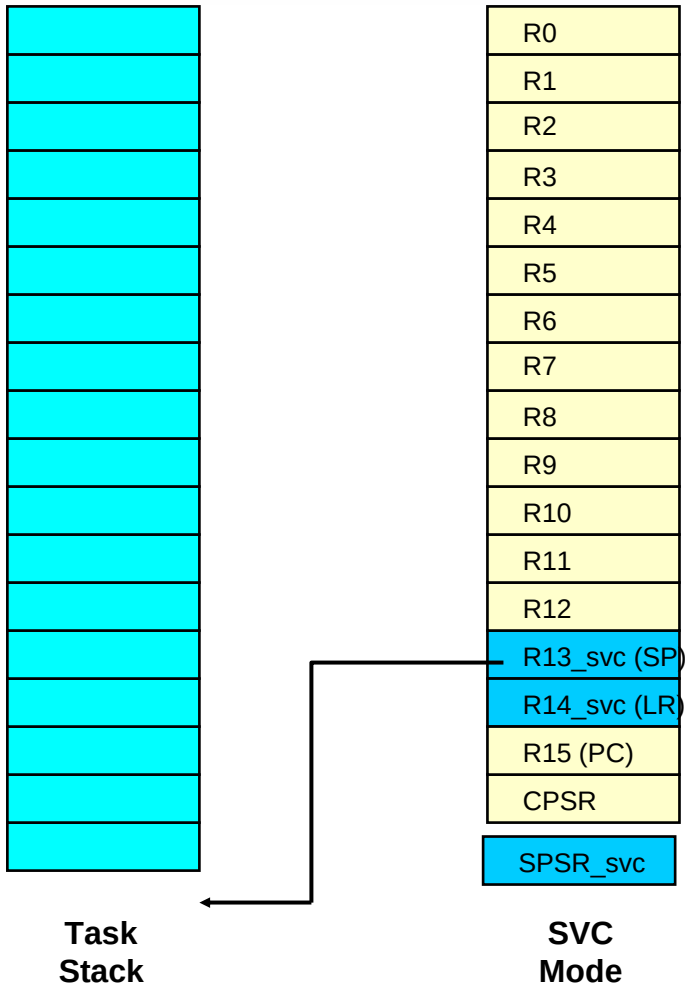
Servicing Interrupts – IRQ

Servicing Interrupts - FIQ

Interrupt Level Context Switch – OS_IntCtxSw()

Servicing Interrupts - IRQ

Task running in SVC mode (ARM or Thumb)



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

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

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

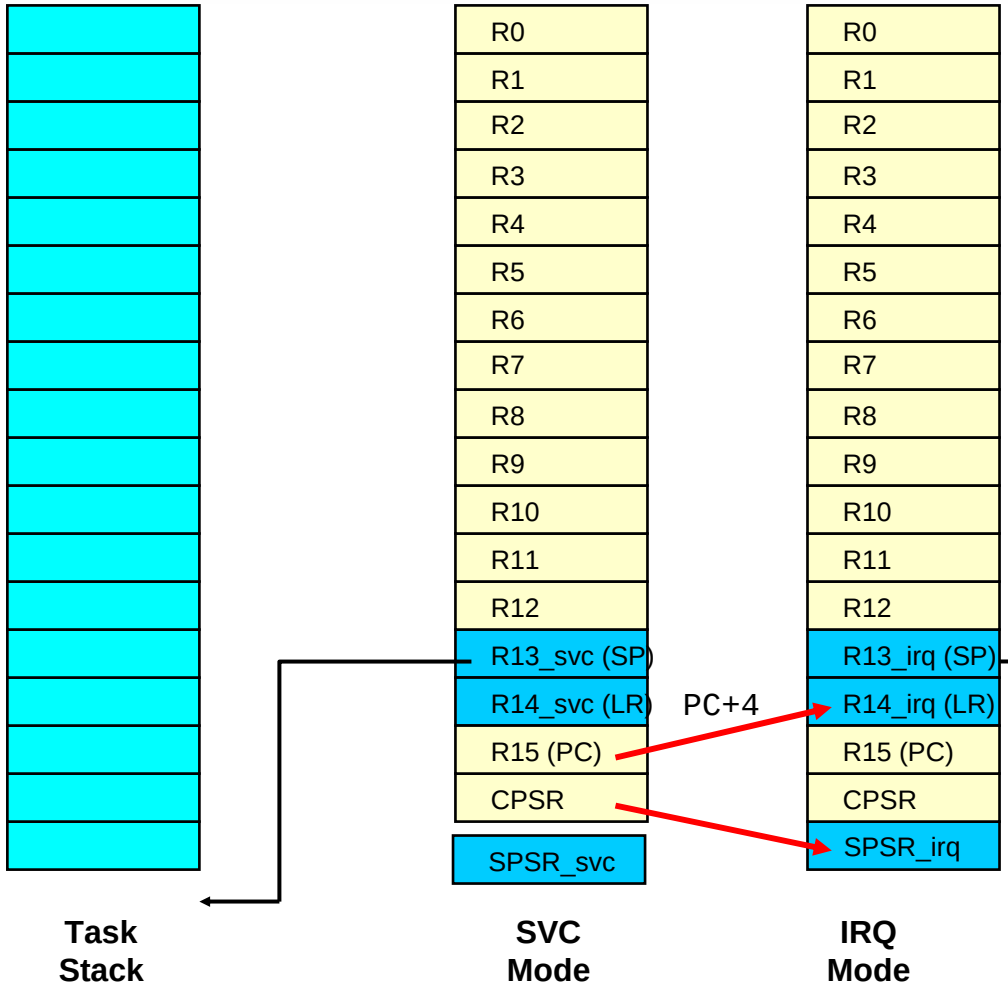
Specifically, your ISRs need to be written as follows:

```
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	0/1	0x13

Interrupt Context Switch

IRQ occurs



The processor recognizes the IRQ:

- **PC+4** is saved in **LR** (R14_irq)
- **CPSR_svc** 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,#0x18]
```

At 0x0038:

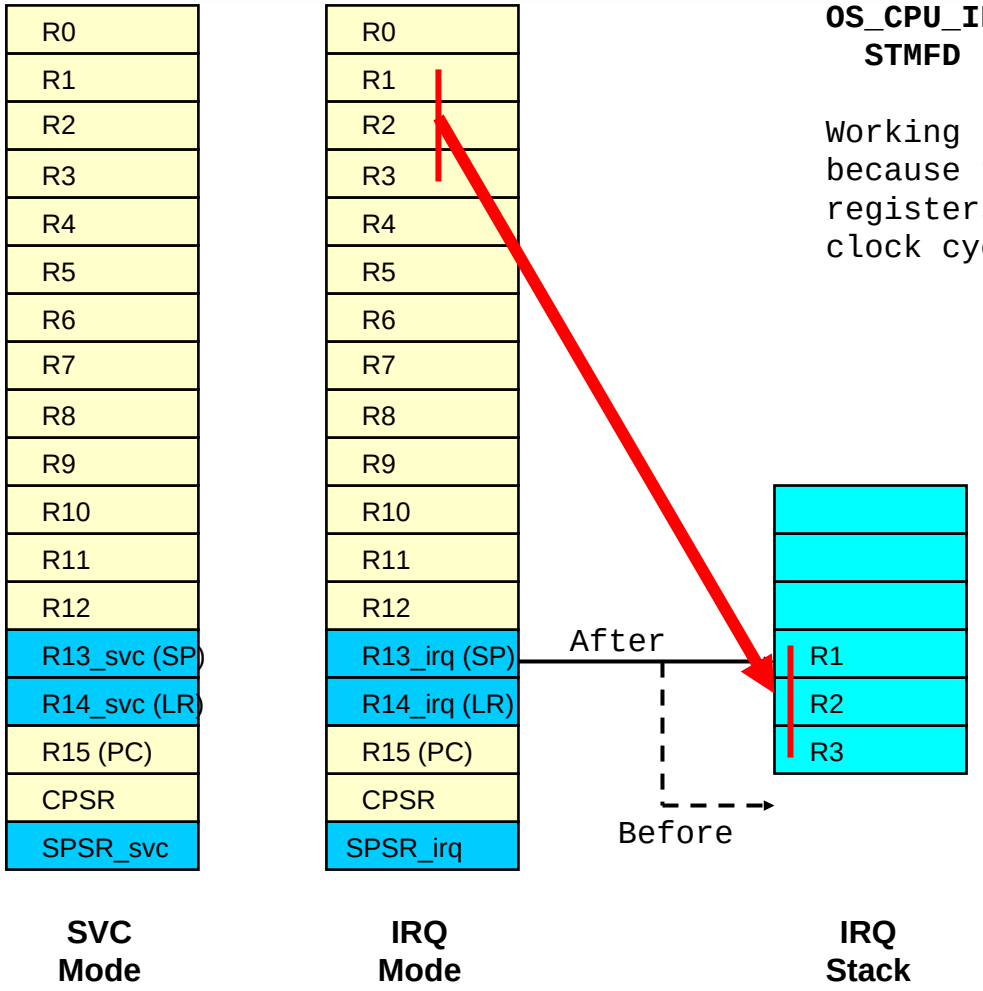
We store the address of **OS_CPU_IRQ_ISR()**

Ptr to IRQ stack

	I	F	T	MODE
CPSR:	1	0	0	0x12

	I	F	T	MODE
SPSR:	0	0	0/1	0x13

Interrupt Context Switch
OS_CPU_IRQ_ISR()



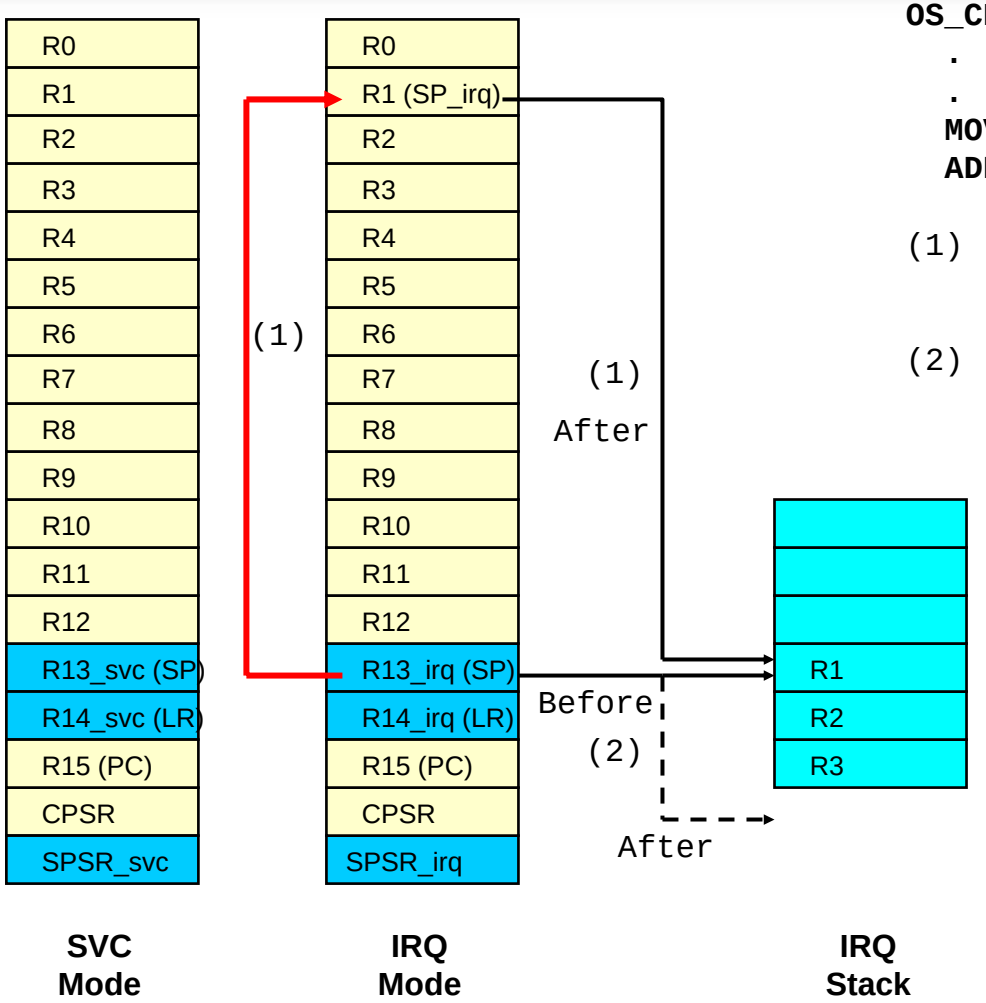

```
OS_CPU_IRQ_ISR
  STMFD  SP!, {R1-R3}
```

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	0/1	0x13

Interrupt Context Switch
OS_CPU_IRQ_ISR()




```
OS_CPU_IRQ_ISR
  .
  .
  MOV  R1,SP          ; (1)
  ADD  SP,SP,#(3*4)   ; (2)
```

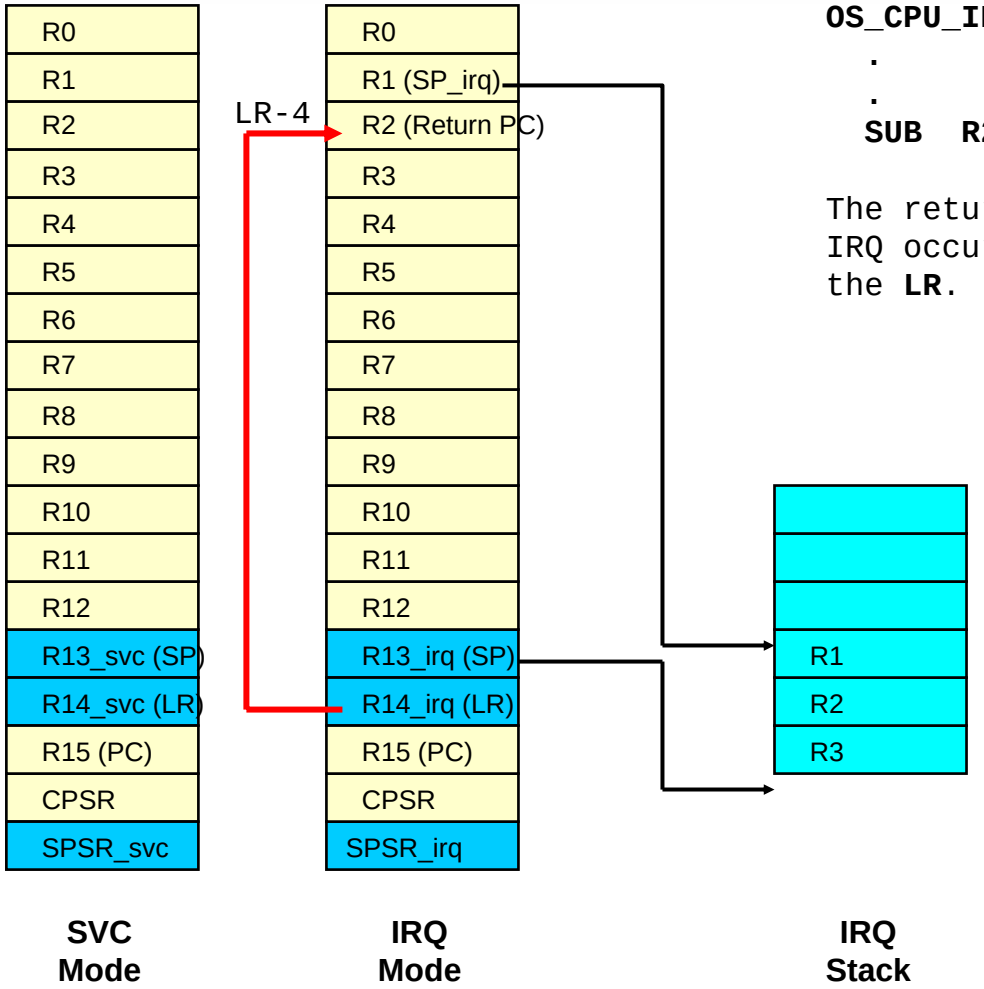
(1) The IRQ stack pointer is copied to the **R1** for future use.

(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.

	I	F	T	MODE
CPSR:	1	0	0	0x12

	I	F	T	MODE
SPSR:	0	0	0/1	0x13

Interrupt Context Switch
OS_CPU_IRQ_ISR()



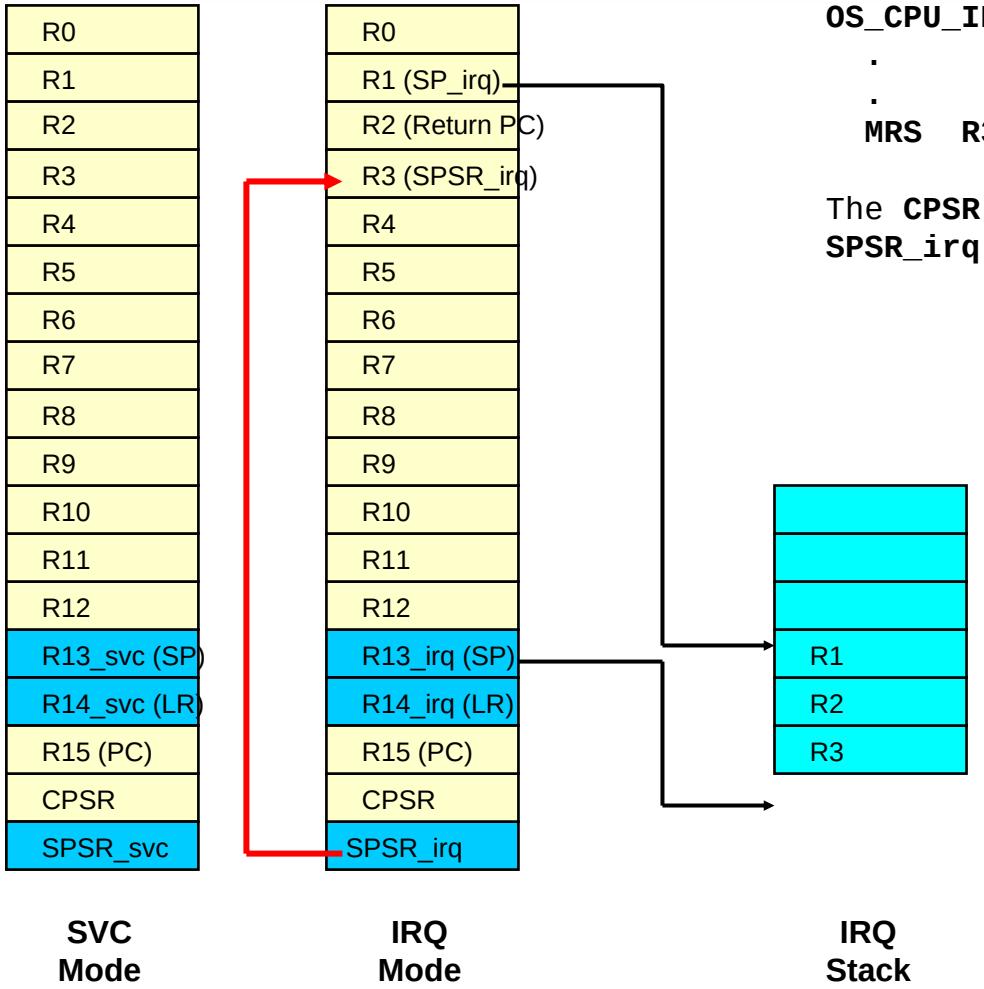

```
OS_CPU_IRQ_ISR
   .
   .
   SUB   R2,LR,#4
```

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	0/1	0x13

Interrupt Context Switch
OS_CPU_IRQ_ISR()

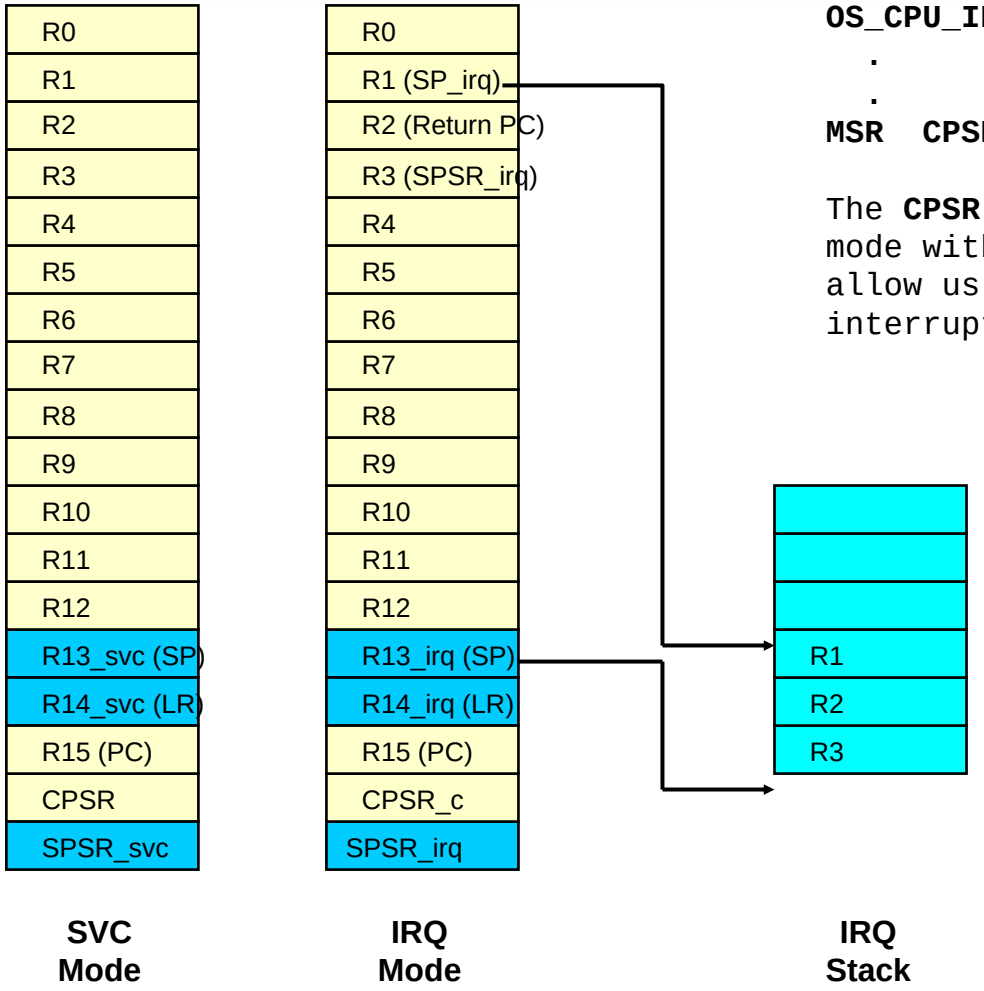



```
OS_CPU_IRQ_ISR
   .
   .
  MRS  R3,SPSR
```

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	0/1	0x13

Interrupt Context Switch
OS_CPU_IRQ_ISR()

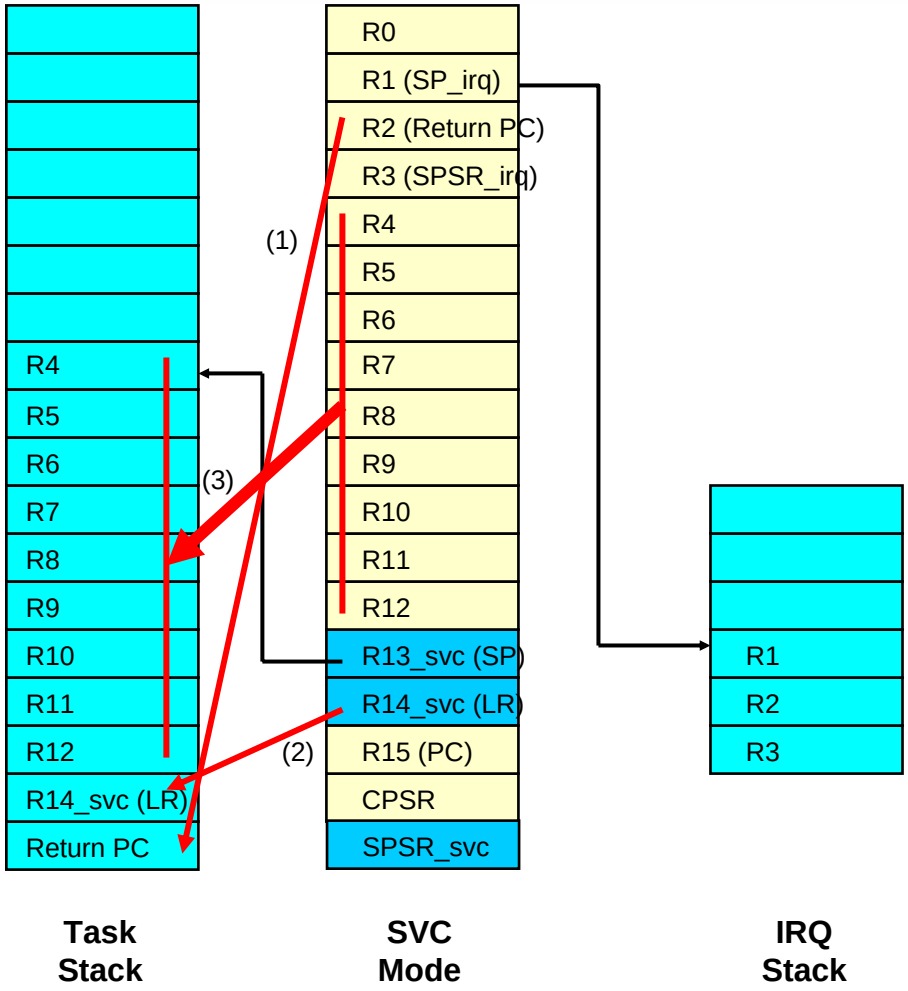



```
OS_CPU_IRQ_ISR
   .
   .
MSR  CPSR_c,#(NO_INT | SVC32_MODE)
```

The **CPSR** is changed to mode change back to SVC mode with ALL interrupts disabled. This will allow us to save the CPU registers of the interrupted task's onto its stack.

	I	F	T	MODE
CPSR:	1	1	0	0x13
SPSR:	1	1	0/1	0x13

Interrupt Context Switch
OS_CPU_IRQ_ISR()

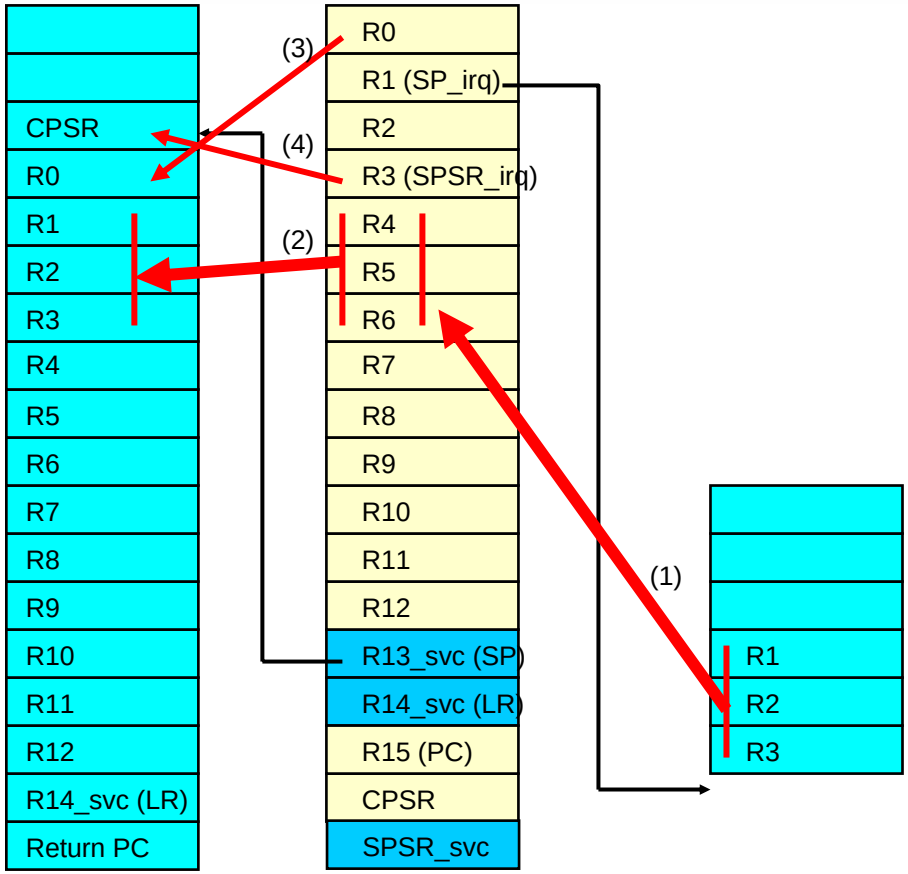



```
OS_CPU_IRQ_ISR
  :
  STMFD   SP!, {R2}      ; (1)
  STMFD   SP!, {LR}      ; (2)
  STMFD   SP!, {R4-R12}  ; (3)
```

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

	I	F	T	MODE
CPSR:	1	1	0	0x13

Interrupt Context Switch
OS_CPU_IRQ_ISR()



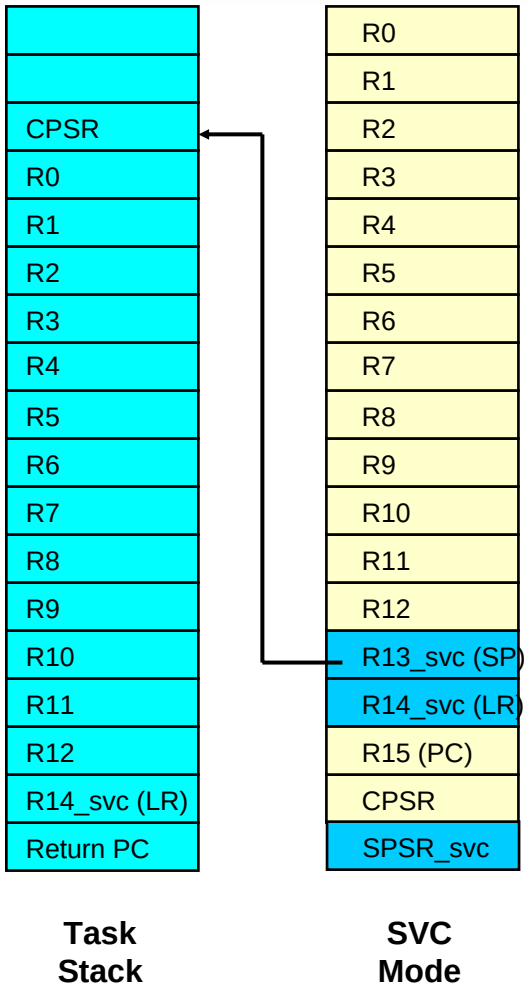

```
OS_CPU_IRQ_ISR
  :
  LDMFD   R1!, {R4-R6}  ; (1)
  STMFD   SP!, {R4-R6}  ; (2)

  STMFD   SP!, {R0}     ; (3)
  STMFD   SP!, {R3}     ; (4)
```

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

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

	I	F	T	MODE
CPSR:	1	1	0	0x13

Interrupt Context Switch
OS_CPU_IRQ_ISR()

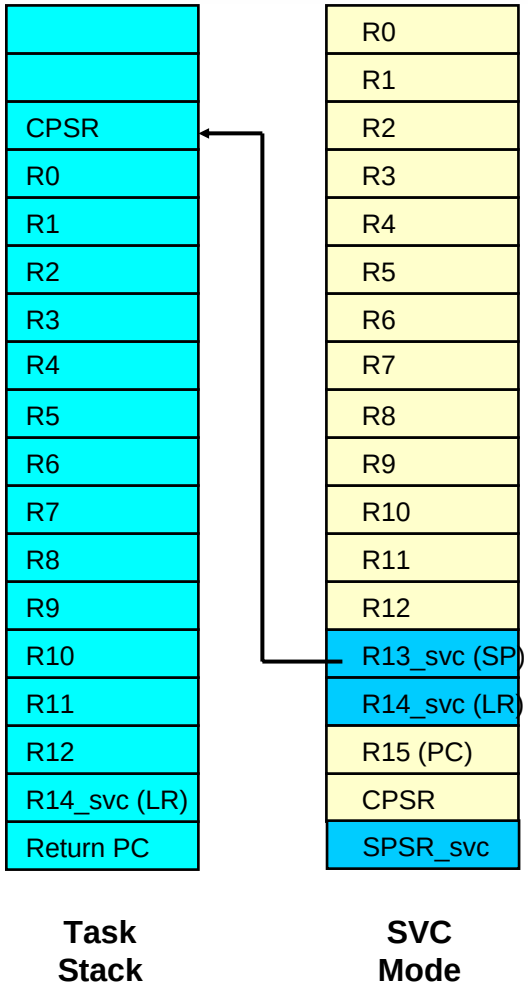



```
OS_CPU_IRQ_ISR
  :
  LDR   R0,??OSIntNesting  ; OSIntNesting++
  LDRB  R1,[R0]
  ADD   R1,R1,#1
  STRB  R1,[R0]
```

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

	I	F	T	MODE
CPSR:	1	1	0	0x13

Interrupt Context Switch
OS_CPU_IRQ_ISR()

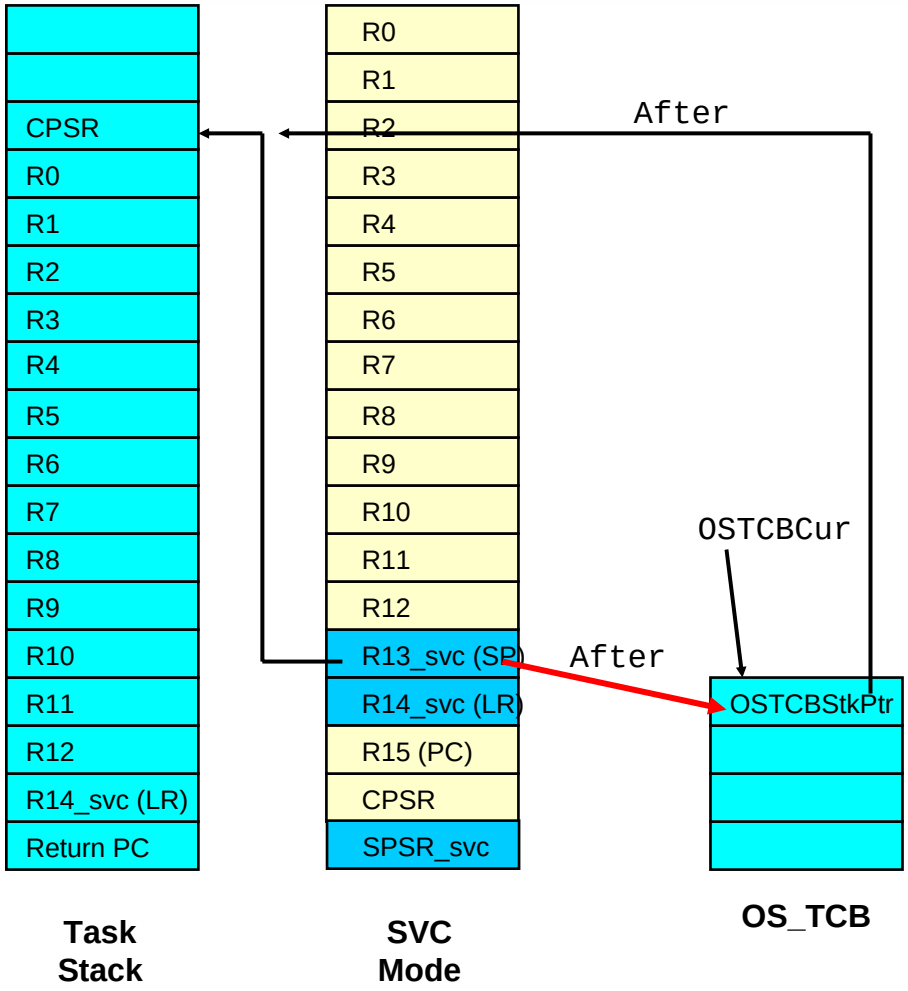



```
OS_CPU_IRQ_ISR
  :
  CMP  R1,#1          ; if (OSIntNesting == 1) {
  BNE  OS_CPU_IRQ_ISR_1
```

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	0x13

Interrupt Context Switch
OS_CPU_IRQ_ISR()

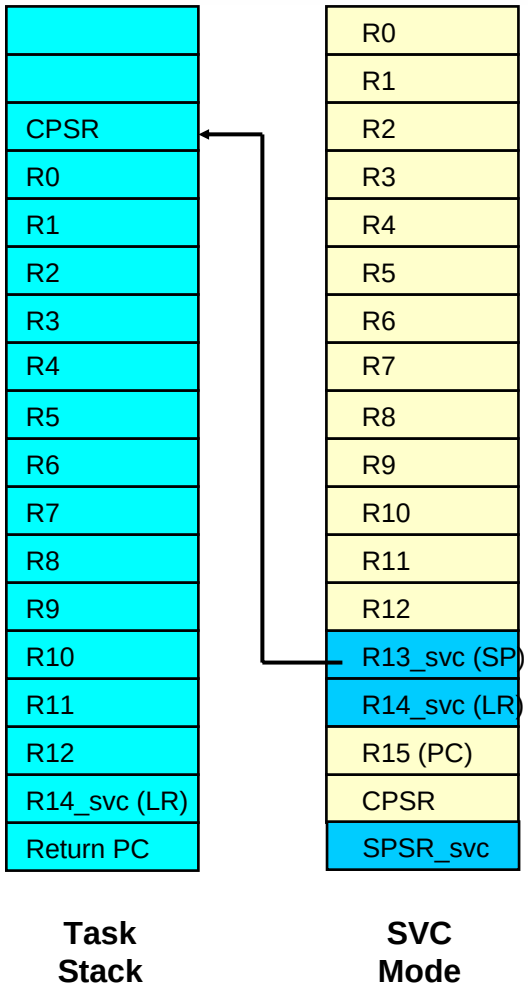



```
OS_CPU_IRQ_ISR
   :
;       OSTCBCur->OSTCBStkPtr = SP
  LDR   R4,??OSTCBCur
  LDR   R5,[R4]
  STR   SP,[R5]
```

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	0x13

Interrupt Context Switch
OS_CPU_IRQ_ISR()




```
OS_CPU_IRQ_ISR
  :
OS_CPU_IRQ_ISR_1
  MSR   CPSR_c,#(NO_INT | IRQ32_MODE)  (1)
;
  LDR   R0,??OS_CPU_IRQ_ISR_Handler    (2)
  MOV   LR,PC
  BX    R0
```

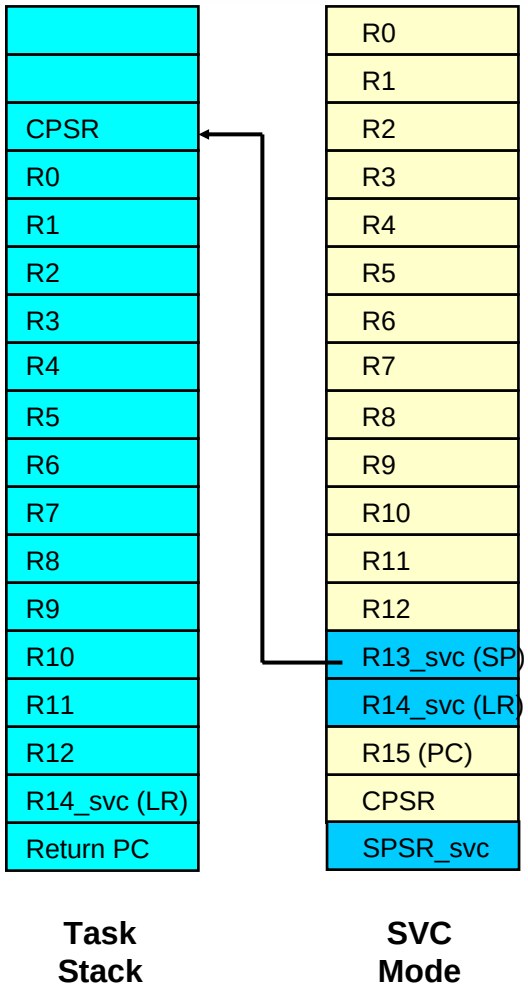
We now switch back to IRQ mode in order to process the ISR using the IRQ stack. This allows to reduce the RAM requirements on the task stack because all ISRs are processed on the IRQ stack.

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
CPSR:	1	1	0	0x12

Interrupt Context Switch
OS_CPU_IRQ_ISR()



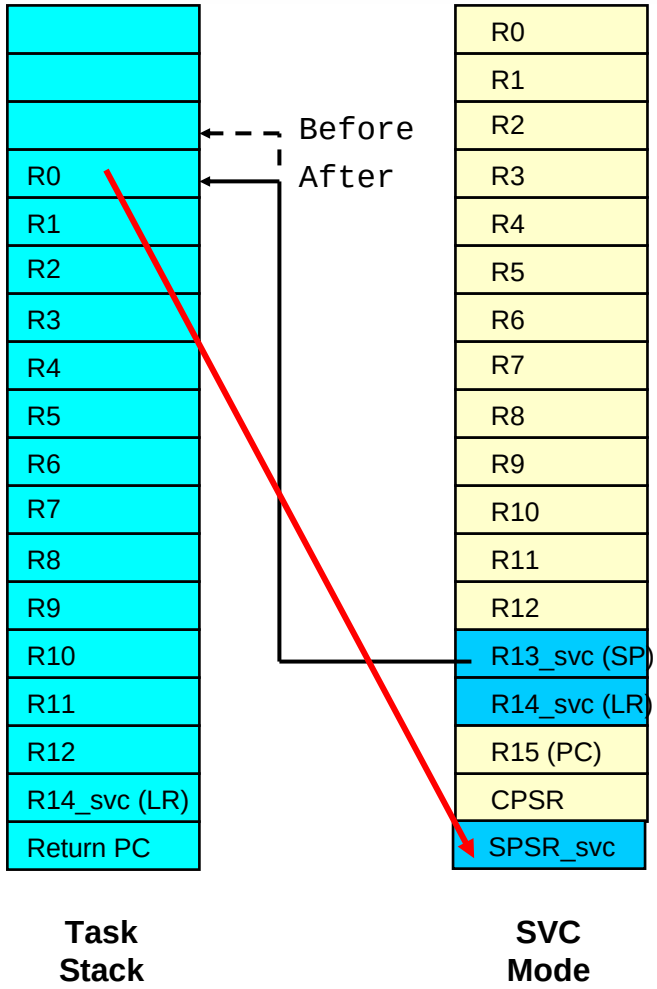

```
OS_CPU_IRQ_ISR:
  :
  MSR   CPSR_c,#(NO_INT | SVC32_MODE)  (1)

  LDR   R0,??OS_IntExit                (2)
  MOV   LR,PC
  BX    R0
```

We now switch back to SVC 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**	**0x13**

Interrupt Context Switch
OS_CPU_IRQ_ISR()



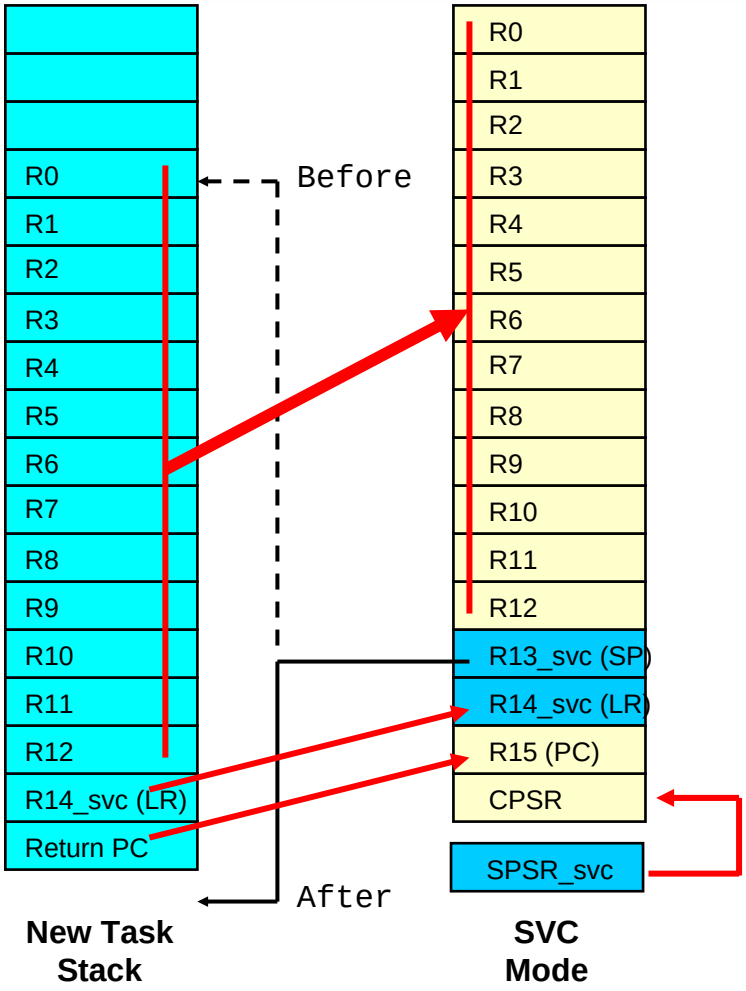

```
OS_CPU_IRQ_ISR:
   :
   LDMFD   SP!, {R4}          ; pop new task's CPSR
   MSR     SPSR_cxsf, R4
```

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 **SPSR** register (and NOT the **CPSR**).

	I	F	T	MODE
CPSR:	1	1	0	0x13

Interrupt Context Switch
OS_CPU_IRQ_ISR()




```
OS_CPU_IRQ_ISR:
  :
  LDMFD  SP!, {R0-R12,LR,PC}^
```

This single instruction retrieves the task's registers from the new task's stack and copies the SPSR into the CPSR.

If the task was executing in Thumb mode, it will resume in Thumb mode. If the task was executing in ARM mode, it will resume in ARM mode.

	I	F	T	MODE
CPSR:	1	1	0	0x13

Task Level Context Switch – OSCtxSw()

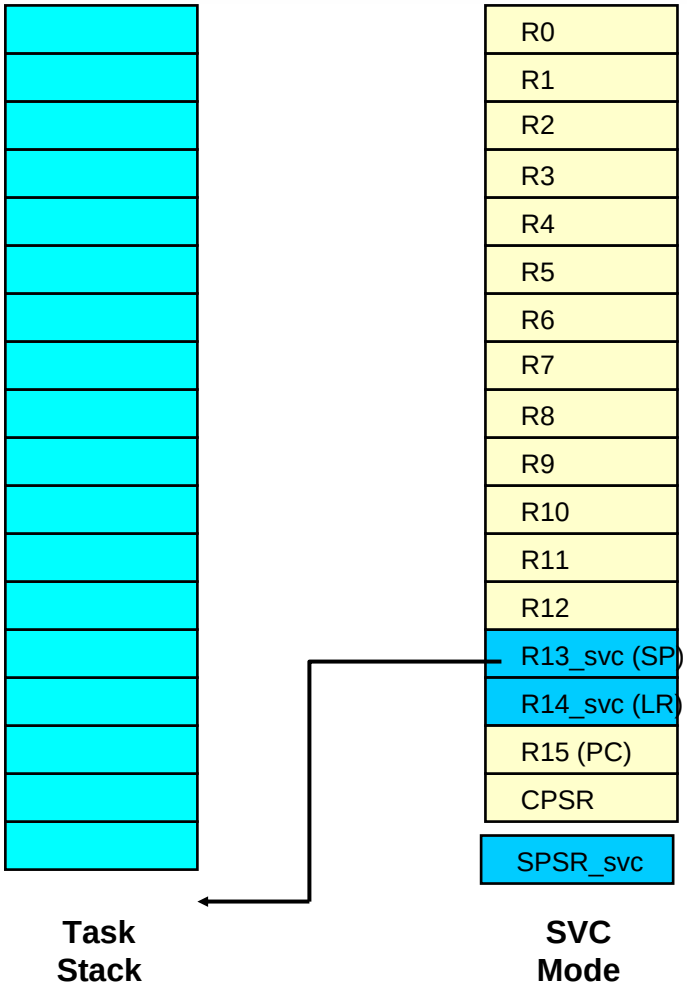
Servicing Interrupts – IRQ

Servicing Interrupts - FIQ

Interrupt Level Context Switch – OS_IntCtxSw()

Servicing Interrupts - FIQ

Task running in SVC mode (ARM or Thumb)



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

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

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

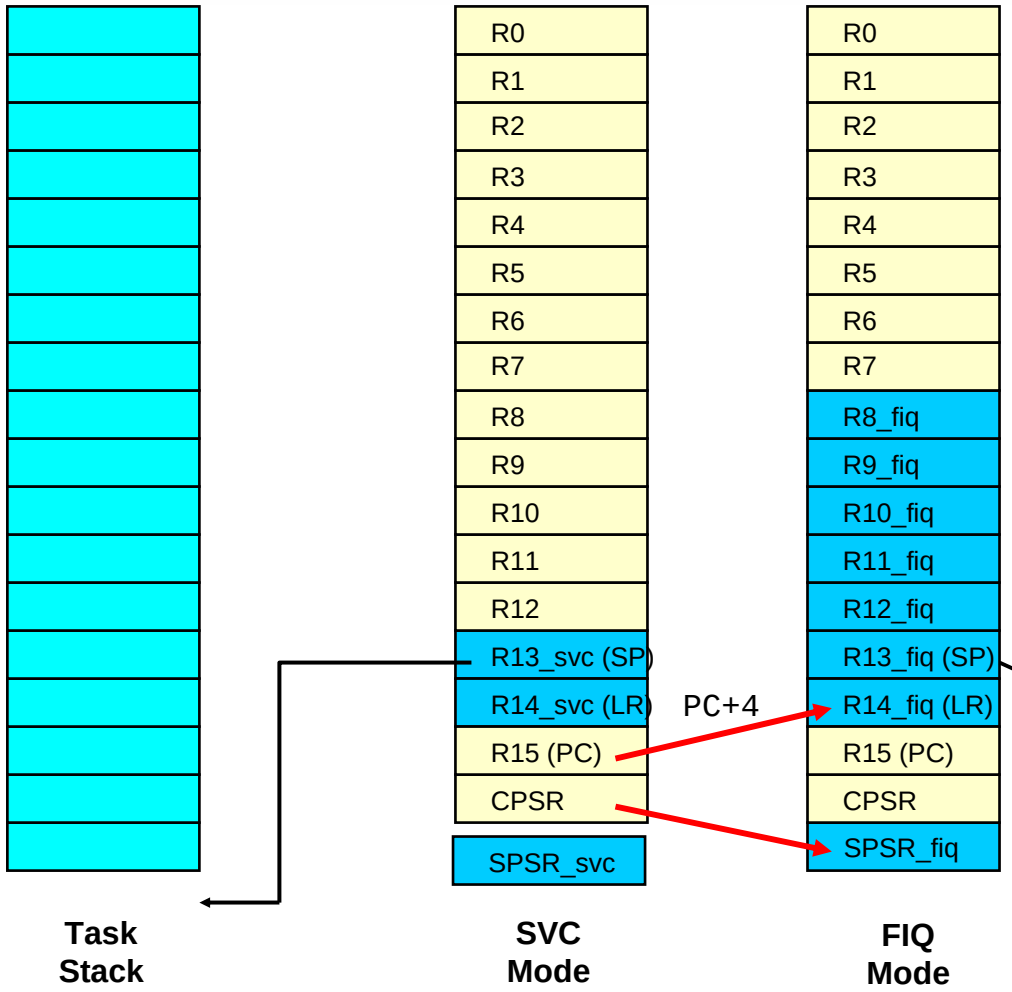
Specifically, your ISRs need to be written as follows:

```
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	0/1	0x13

Interrupt Context Switch

IRQ occurs



The processor recognizes the IRQ:

- **PC+4** is saved in **LR** (R14_irq)
- **CPSR_svc** is saved in **SPSR_fiq**
- The CPU switches to **FIQ mode**
- IRQs and FIQs are disabled
  (**CPSR**, bit 7 = 1, bit 6 = 1)
- **R13_fiq** points to the ISR stack
- The CPU vectors to **0x0018**

The code at 0x0018:

```
LDR  PC,[PC,#0x18]
```

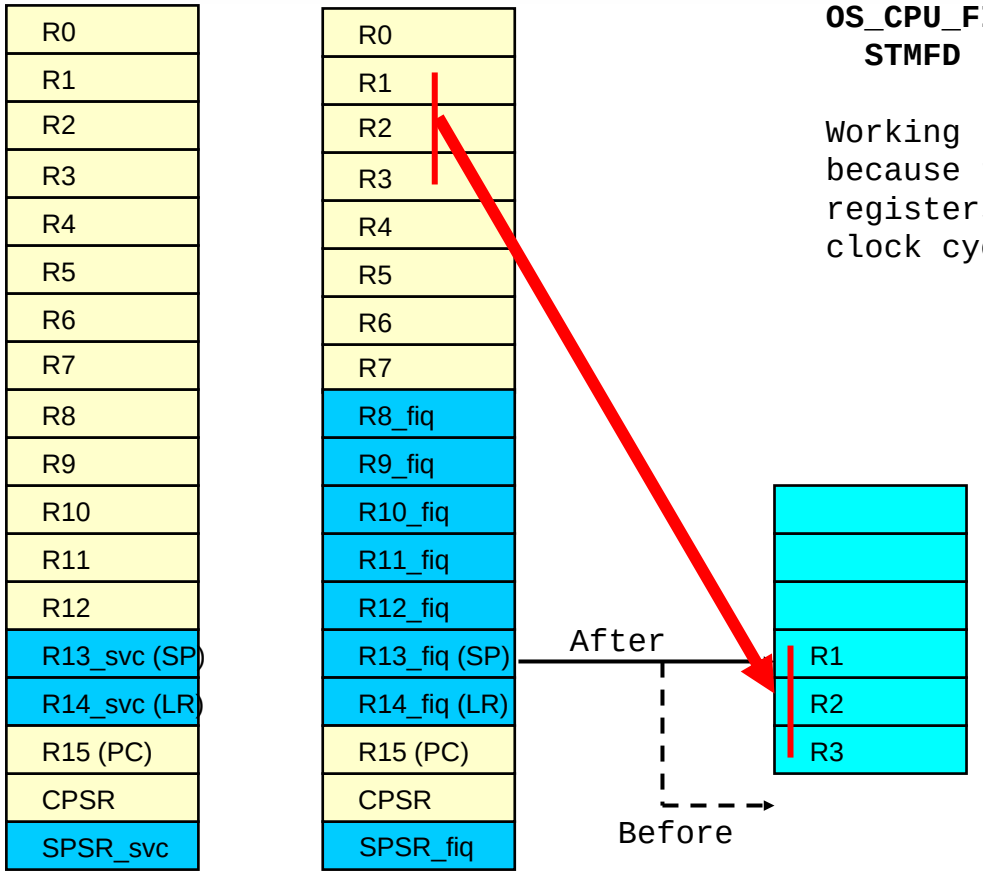
At 0x0038:
We store the address of
OS_CPU_FIQ_ISR()

Ptr to FIQ stack

	I	F	T	MODE
CPSR:	1	1	0	0x11

	I	F	T	MODE
SPSR:	0	0	0/1	0x13

Interrupt Context Switch
OS_CPU_FIQ_ISR()



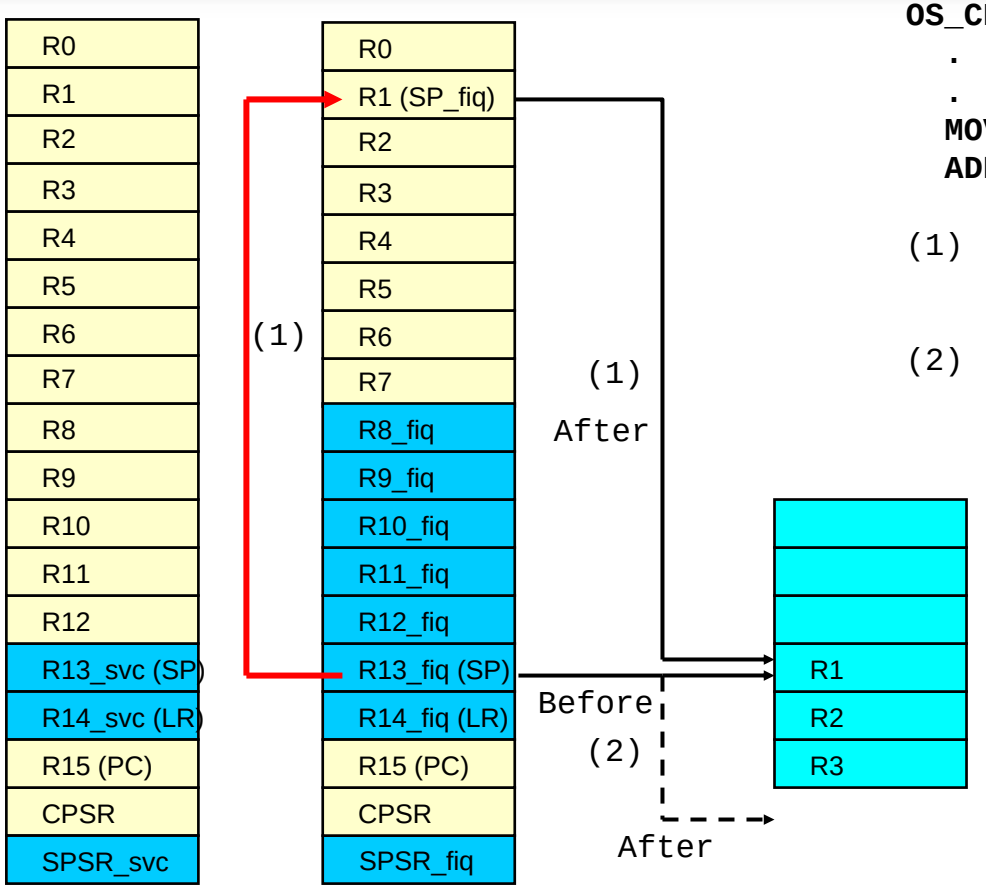

```
OS_CPU_FIQ_ISR
  STMFD  SP!, {R1-R3}
```

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	1	0	0x11

	I	F	T	MODE
SPSR:	0	0	0/1	0x13

Interrupt Context Switch
OS_CPU_FIQ_ISR()




```
OS_CPU_FIQ_ISR
  .
  .
  MOV  R1,SP          ; (1)
  ADD  SP,SP,#(3*4)   ; (2)
```

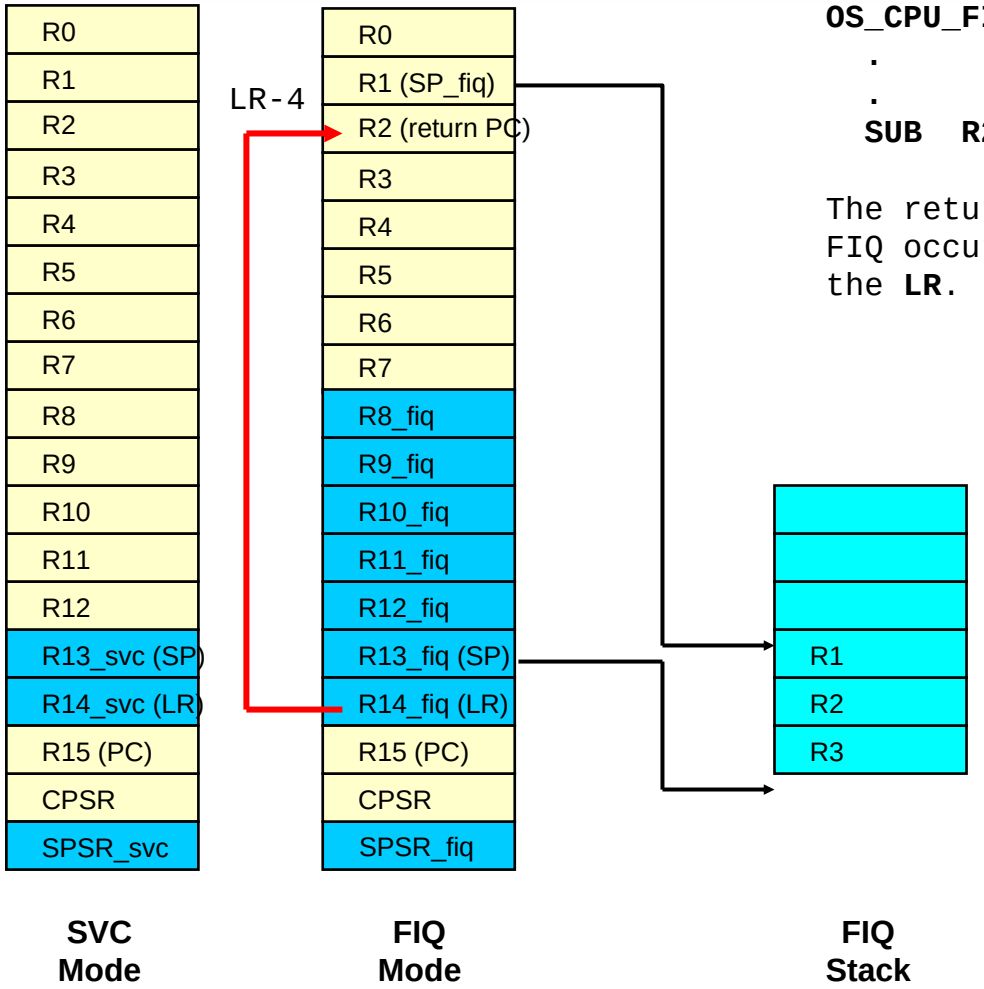
(1) The FIQ stack pointer is copied to the **R1** for future use.

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

	I	F	T	MODE
CPSR:	1	1	0	0x11

	I	F	T	MODE
SPSR:	0	0	0/1	0x13

Interrupt Context Switch
OS_CPU_FIQ_ISR()

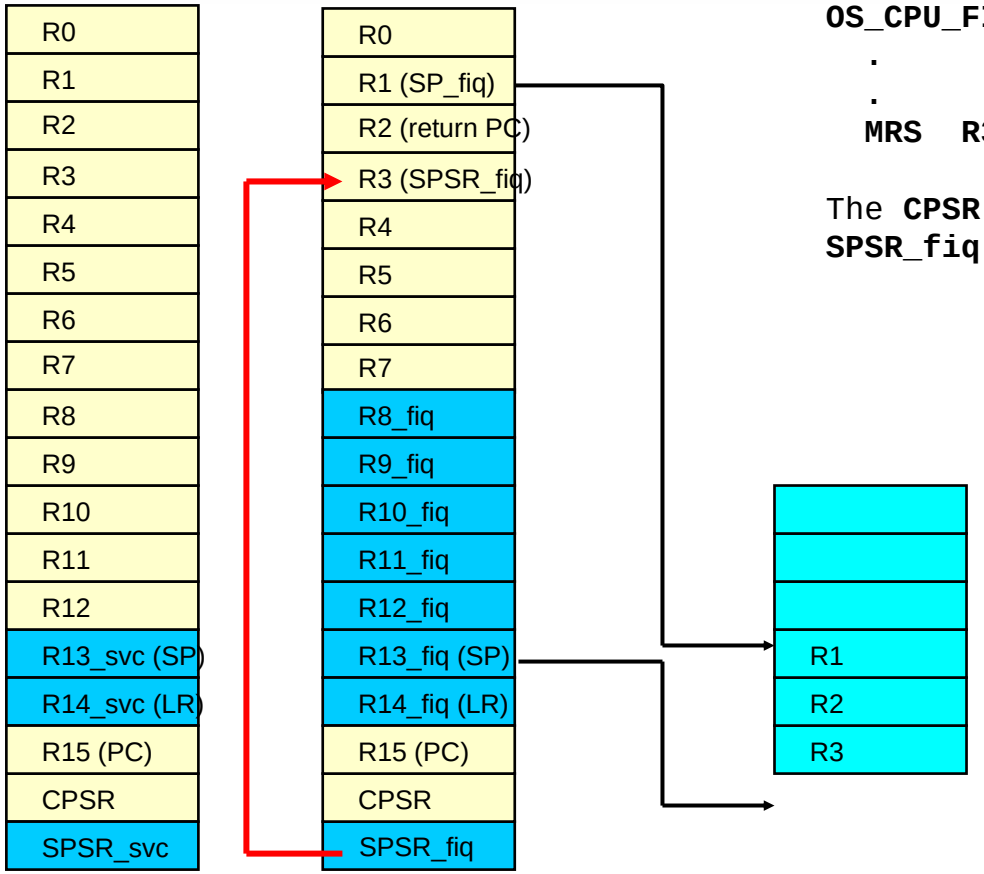



```
OS_CPU_FIQ_ISR
   .
   .
   SUB  R2,LR,#4
```

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

	I	F	T	MODE
CPSR:	1	1	0	0x11
SPSR:	0	0	0/1	0x13

Interrupt Context Switch
OS_CPU_FIQ_ISR()

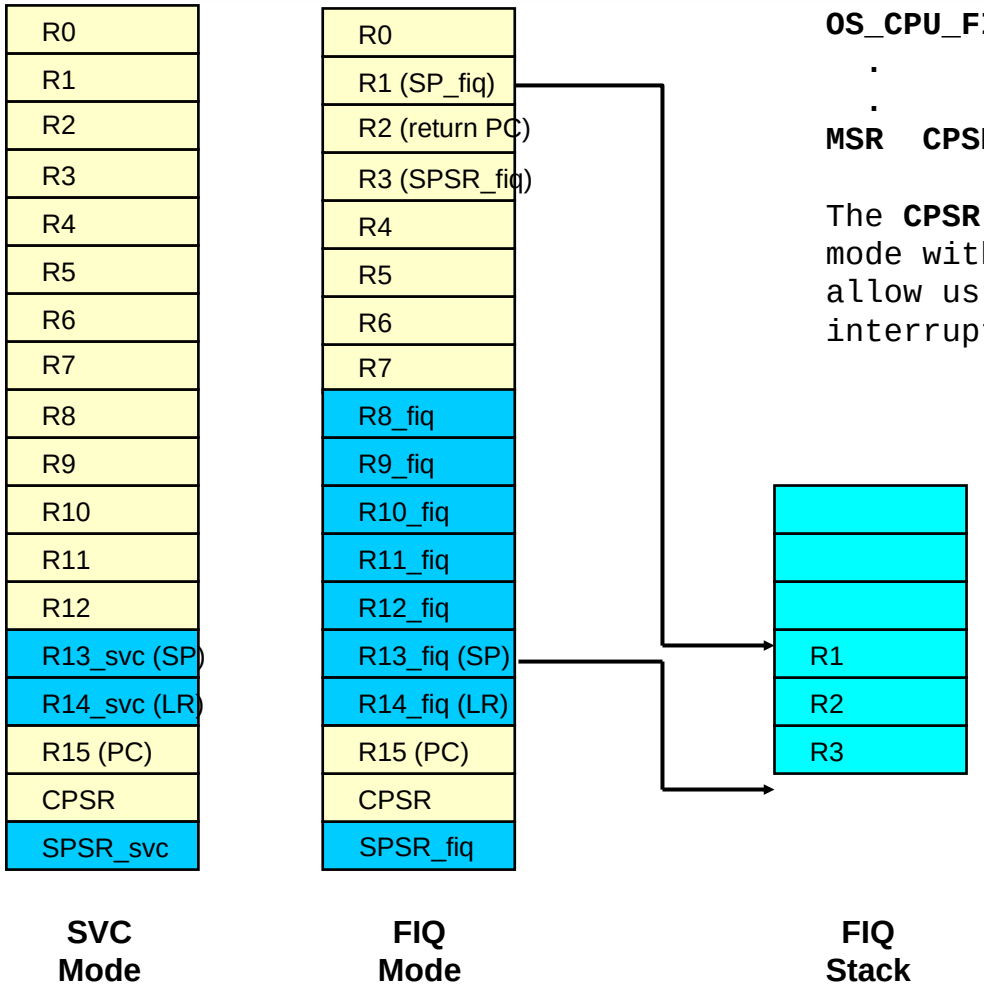



```
OS_CPU_FIQ_ISR
   .
   .
  MRS   R3,SPSR
```

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

	I	F	T	MODE
CPSR:	1	1	0	0x11
SPSR:	0	0	0/1	0x13

Interrupt Context Switch
OS_CPU_FIQ_ISR()



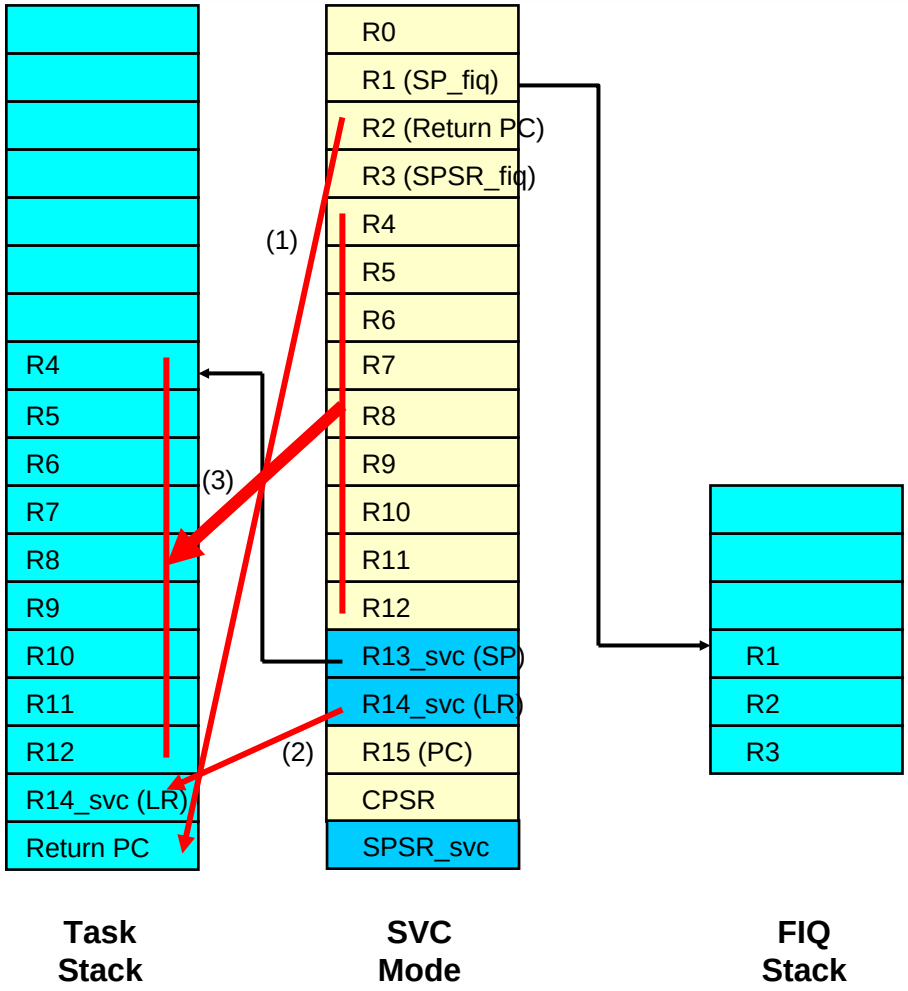

```
OS_CPU_FIQ_ISR
   .
   .
MSR  CPSR_c,#(NO_INT | SVC32_MODE)
```

The **CPSR** is changed to mode change back to SVC mode with ALL interrupts disabled. This will allow us to save the CPU registers of the interrupted task's onto its stack.

	I	F	T	MODE
CPSR:	1	1	0	0x13

	I	F	T	MODE
SPSR:	1	1	0/1	0x13

Interrupt Context Switch
OS_CPU_FIQ_ISR()

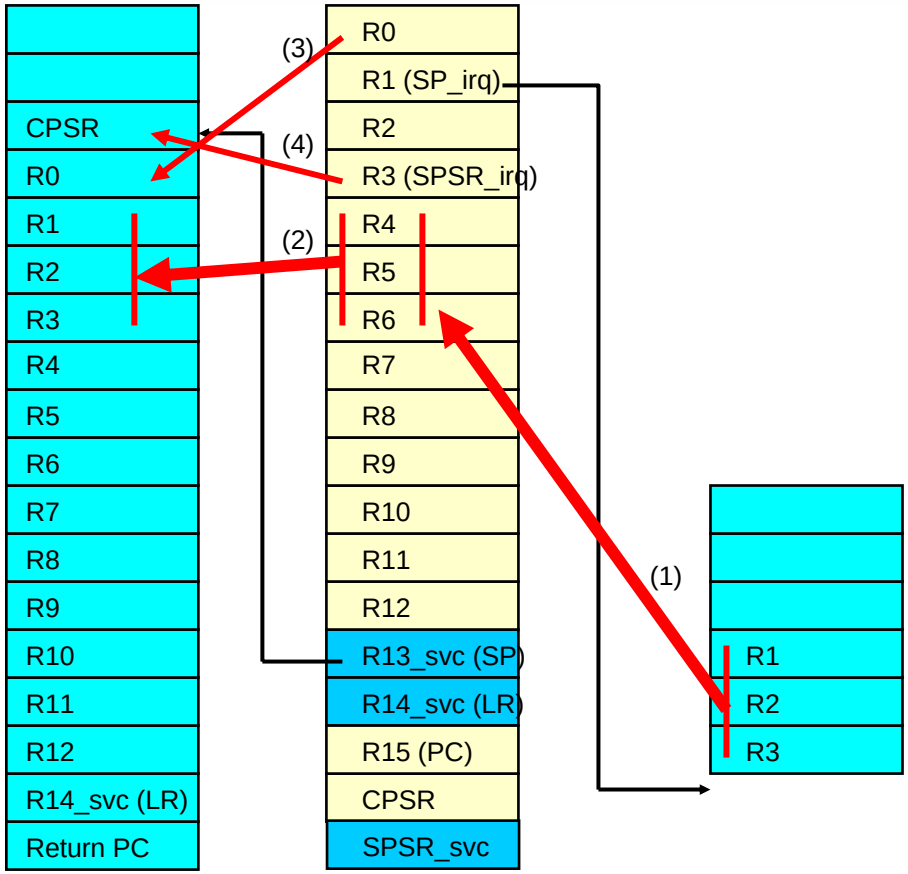



```
OS_CPU_FIQ_ISR
  :
  STMFD   SP!, {R2}       ; (1)
  STMFD   SP!, {LR}       ; (2)
  STMFD   SP!, {R4-R12}   ; (3)
```

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

	I	F	T	MODE
CPSR:	1	1	0	0x13

Interrupt Context Switch
OS_CPU_FIQ_ISR()



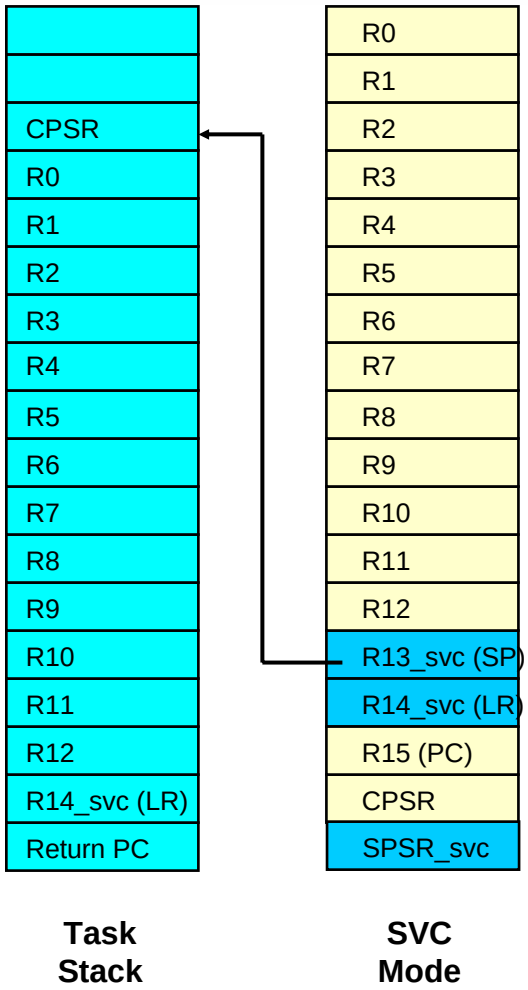

```
OS_CPU_FIQ_ISR
  :
  LDMFD   R1!, {R4-R6}  ; (1)
  STMFD   SP!, {R4-R6}  ; (2)

  STMFD   SP!, {R0}     ; (3)
  STMFD   SP!, {R3}     ; (4)
```

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

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

	I	F	T	MODE
CPSR:	1	1	0	0x13

Interrupt Context Switch
OS_CPU_IRQ_ISR()

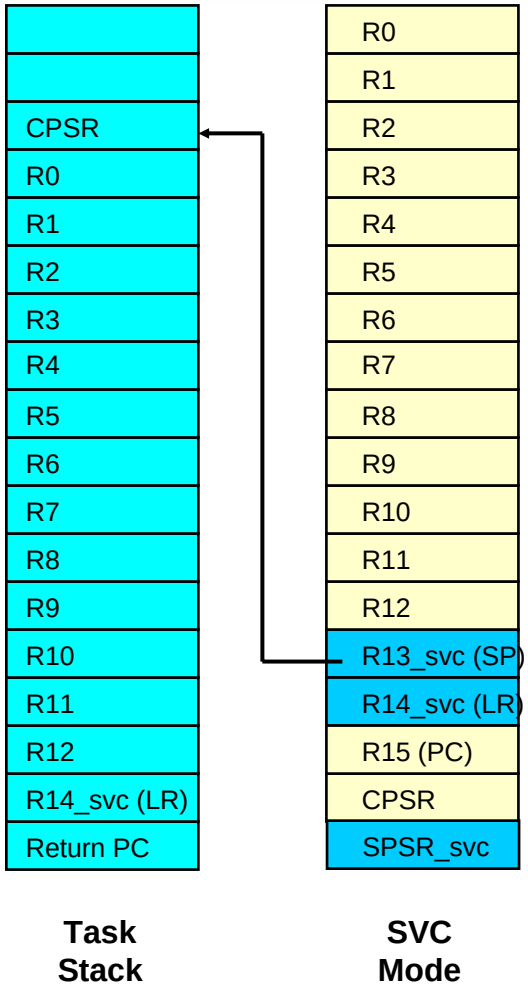



```
OS_CPU_FIQ_ISR
  :
  LDR   R0,??OSIntNesting  ; OSIntNesting++
  LDRB  R1,[R0]
  ADD   R1,R1,#1
  STRB  R1,[R0]
```

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

	I	F	T	MODE
CPSR:	1	1	0	0x13

Interrupt Context Switch
OS_CPU_FIQ_ISR()

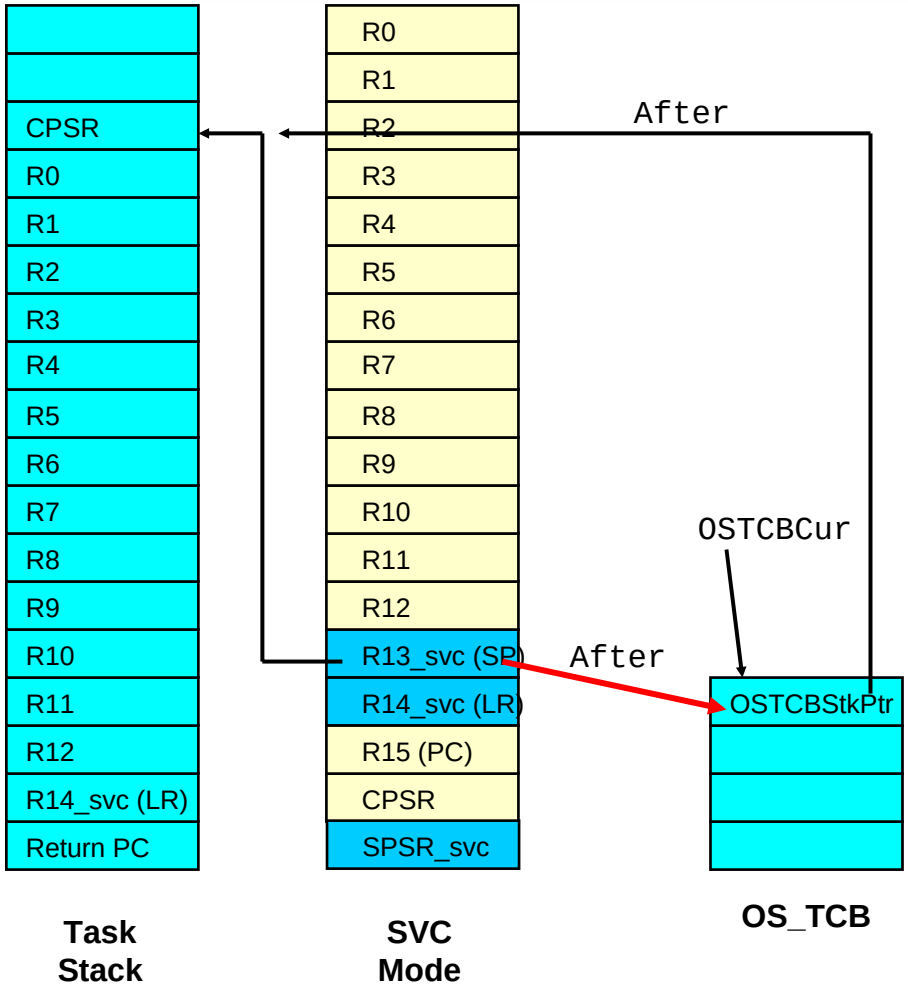



```
OS_CPU_FIQ_ISR
  :
  CMP  R1,#1          ; if (OSIntNesting == 1) {
  BNE  OS_CPU_IRQ_ISR_1
```

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	0x13

Interrupt Context Switch
OS_CPU_FIQ_ISR()

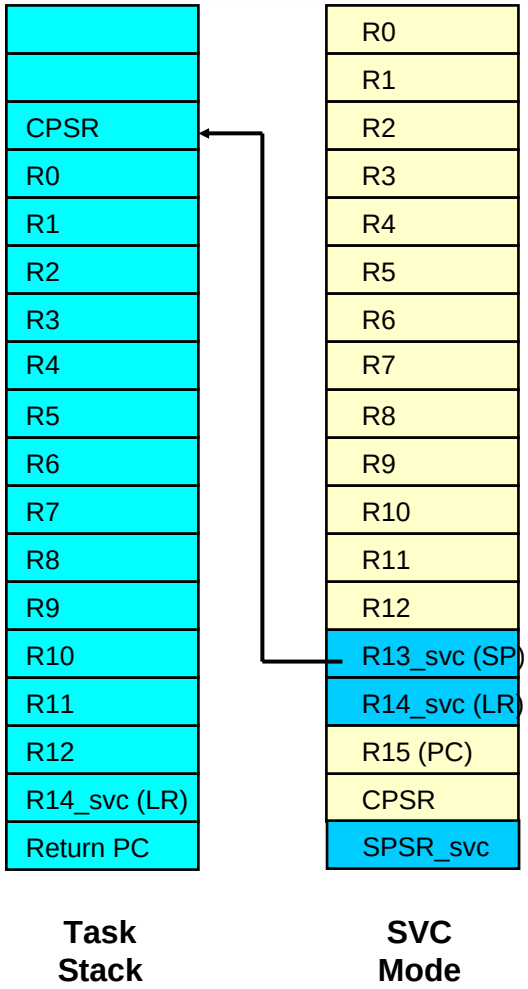



```
OS_CPU_FIQ_ISR
  :
;       OSTCBCur->OSTCBStkPtr = SP
  LDR   R4,??OSTCBCur
  LDR   R5,[R4]
  STR   SP,[R5]
```

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	0x13

Interrupt Context Switch
OS_CPU_FIQ_ISR()




```
OS_CPU_FIQ_ISR
  :
OS_CPU_FIQ_ISR_1
  MSR   CPSR_c,#(NO_INT | FIQ32_MODE)  (1)
;
  LDR   R0,??OS_CPU_FIQ_ISR_Handler    (2)
  MOV   LR,PC
  BX    R0
```

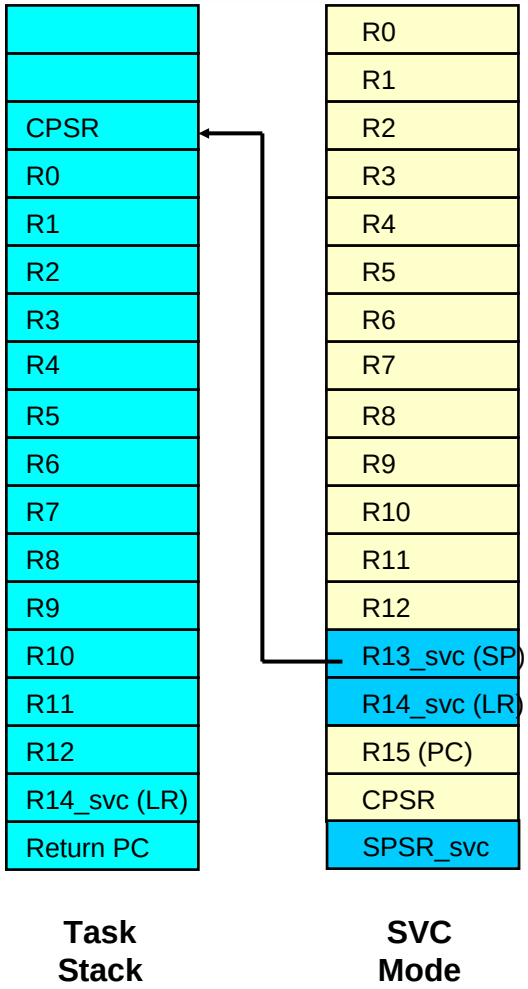
We now switch back to FIQ mode in order to process the ISR using the FIQ stack. This allows to reduce the RAM requirements on the task stack because all ISRs are processed on the FIQ stack.

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
CPSR:	1	1	0	0x11

Interrupt Context Switch
OS_CPU_FIQ_ISR()



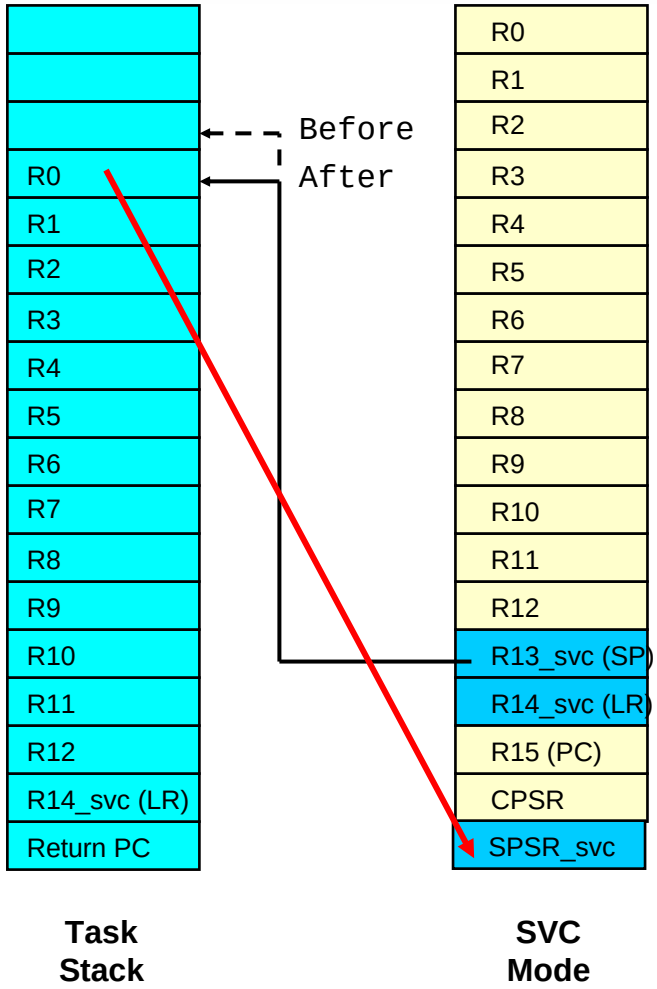

```
OS_CPU_FIQ_ISR:
  :
  MSR   CPSR_c,#(NO_INT | SVC32_MODE)  (1)

  LDR   R0,??OS_IntExit                (2)
  MOV   LR,PC
  BX    R0
```

We now switch back to SVC 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_FIQ_ISR()** but instead, will context switch to the new, more important task (via **OSIntCtxSw()** (described later).

	I	F	T	MODE
CPSR:	**1**	**1**	**0**	**0x13**

Interrupt Context Switch
OS_CPU_FIQ_ISR()



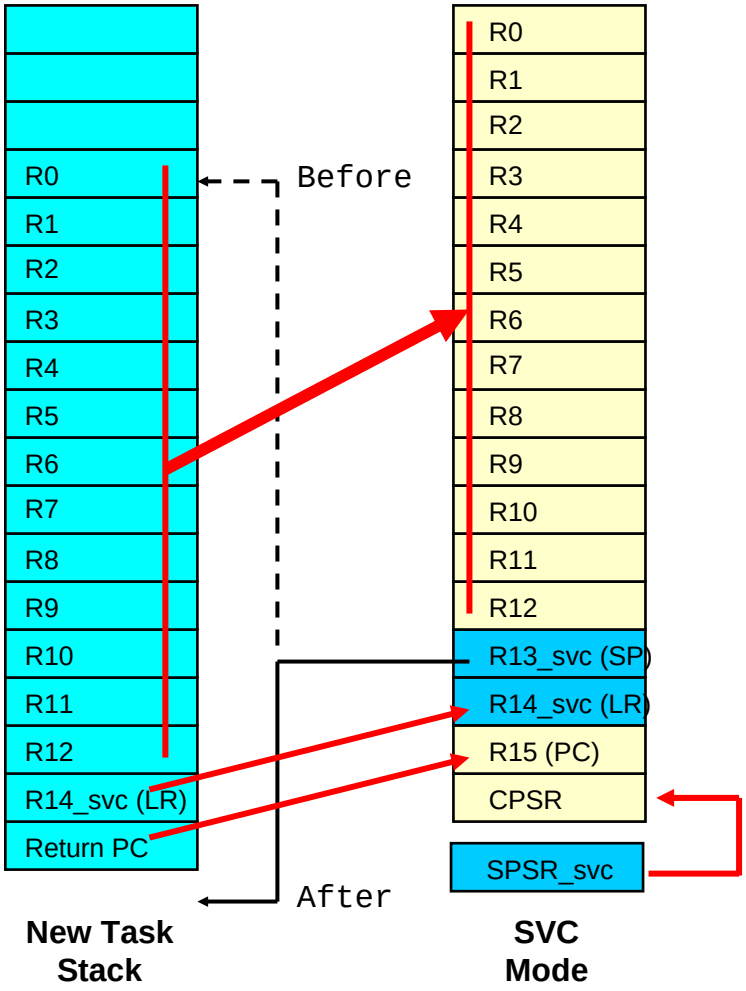

```
OS_CPU_FIQ_ISR:
  :
  LDMFD   SP!, {R4}          ; pop new task's CPSR
  MSR     SPSR_cxsf, R4
```

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 **SPSR** register (and NOT the **CPSR**).

	I	F	T	MODE
CPSR:	1	1	0	0x13

Interrupt Context Switch
OS_CPU_FIQ_ISR()




```
OS_CPU_FIQ_ISR:
  :
  LDMFD  SP!, {R0-R12,LR,PC}^
```

This single instruction retrieves the task's registers from the new task's stack and copies the SPSR into the CPSR.

If the task was executing in Thumb mode, it will resume in Thumb mode. If the task was executing in ARM mode, it will resume in ARM mode.

	I	F	T	MODE
CPSR:	1	1	0	0x13

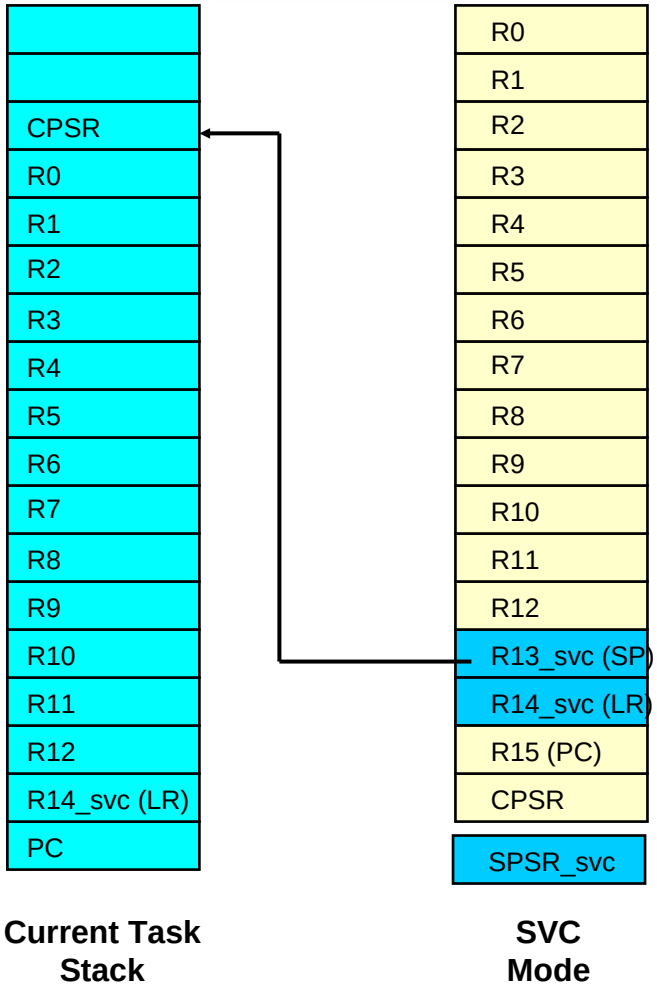
Task Level Context Switch – OSCtxSw()

Servicing Interrupts – IRQ

Servicing Interrupts - FIQ

Interrupt Level Context Switch - OSIntCtxSw()

Interrupt Context Switch
OSIntCtxSw()

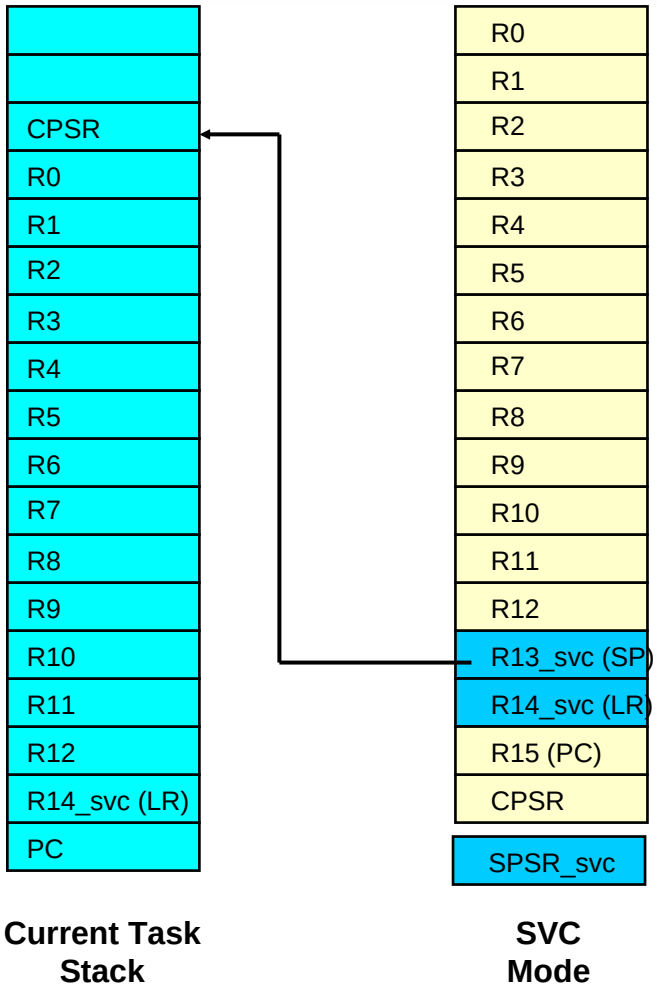


OSIntCtxSw() is called by **OSIntExit()** if µC/OS-II determines that there is a more important task to run than the interrupted task. In this case, the CPU is in SVC mode with interrupts disabled and the SP is pointing to the interrupted task's stack.

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

	I	F	T	MODE
CPSR:	1	1	0	0x13

Interrupt Level Context Switch
OSIntCtxSw()

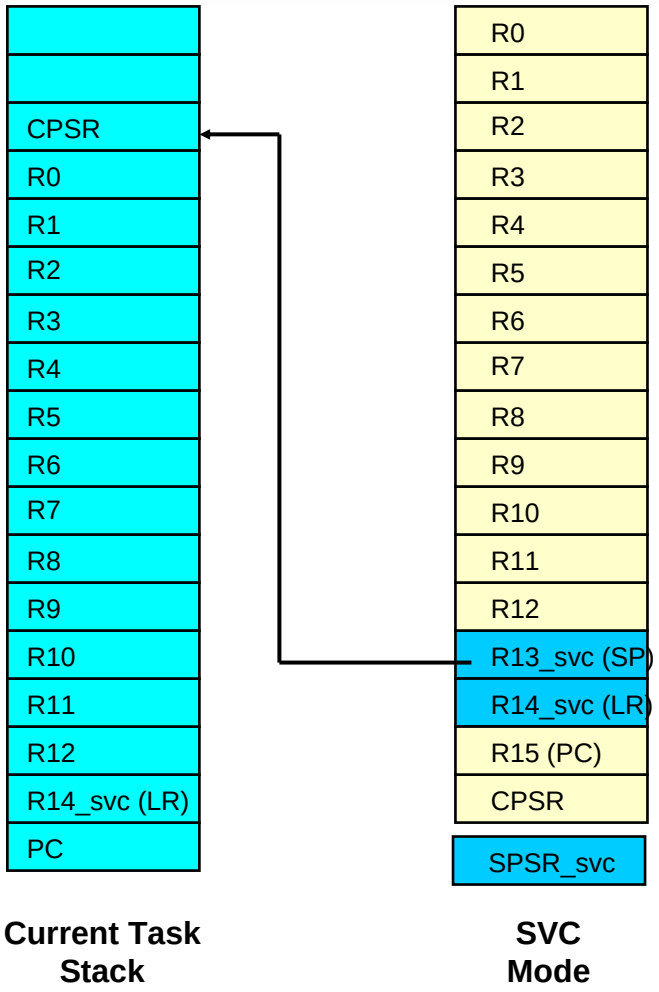



```
OSIntCtxSw:
  LDR  R0, ??OS_TaskSwHook
  MOV  LR,PC
  BX   R0
```

The task switch hook is called.

	I	F	T	MODE
CPSR:	1	1	0	0x13

Interrupt Level Context Switch
OSIntCtxSw()

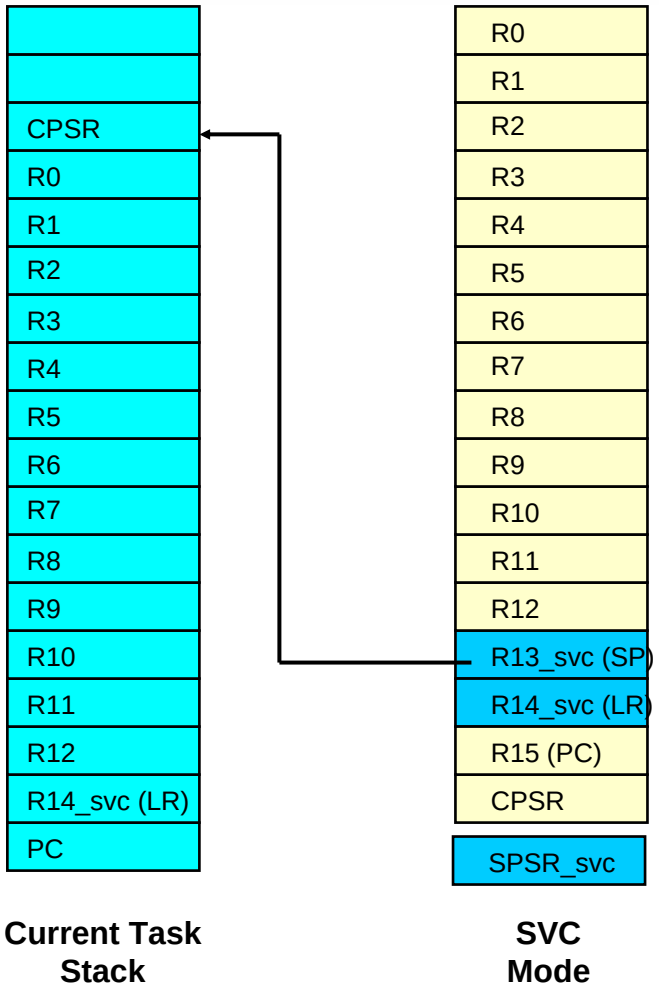



```
OSIntCtxSw:
   .
   .
;          OSPrioCur = OSPrioHighRdy
   LDR     R4,??OS_PrioCur
   LDR     R5,??OS_PrioHighRdy
   LDRB    R5,[r5]
   STRB    R5,[r4]
```

The new high priority is copied to the current priority.

	I	F	T	MODE
CPSR:	1	1	0	0x13

Interrupt Level Context Switch
OSIntCtxSw()



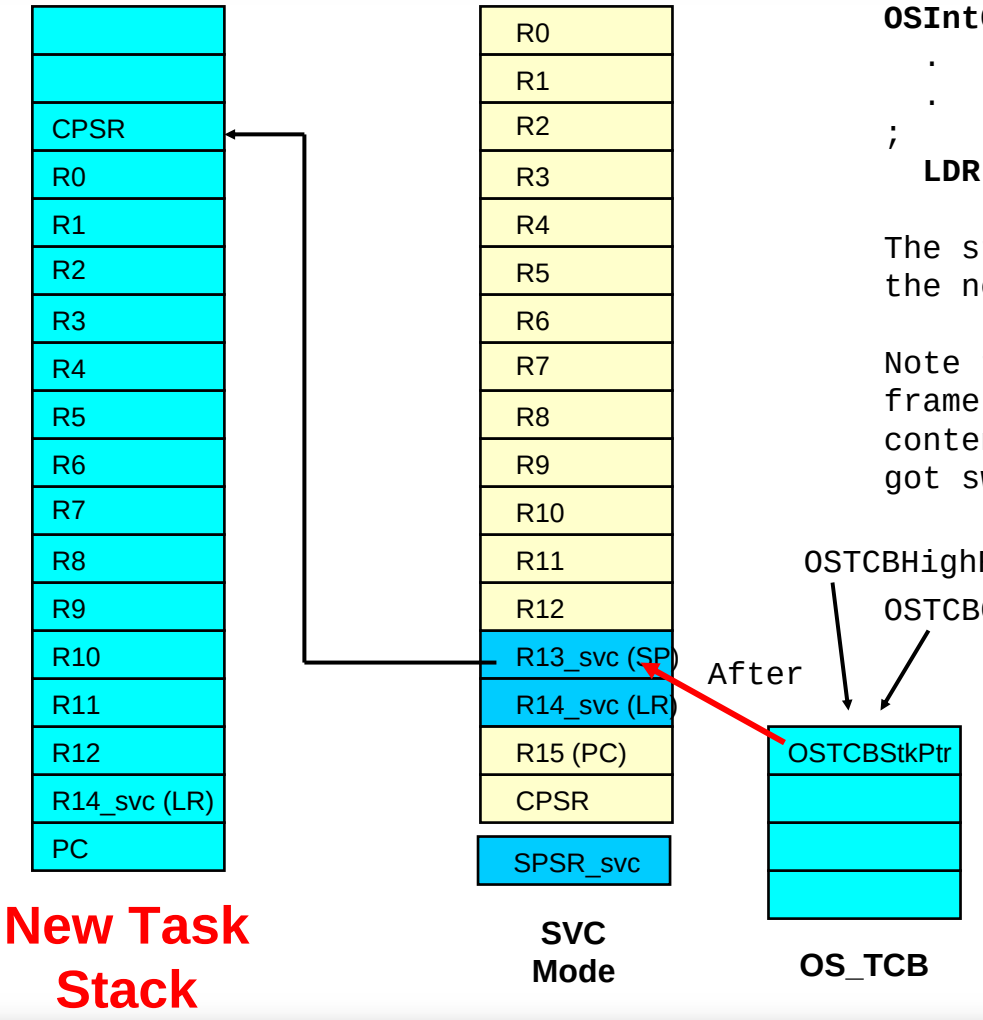

```
OSIntCtxSw:
   .
   .
;          OSTCBCur = OSTCBHighRdy
  LDR      R6,??OS_TCBHighRdy
  LDR      R4,??OS_TCBCur
  LDR      R6,[R6]
  STR      R6,[R4]
```

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

	I	F	T	MODE
CPSR:	**1**	**1**	**0**	**0x13**

Interrupt Level Context Switch

OSIntCtxSw()




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

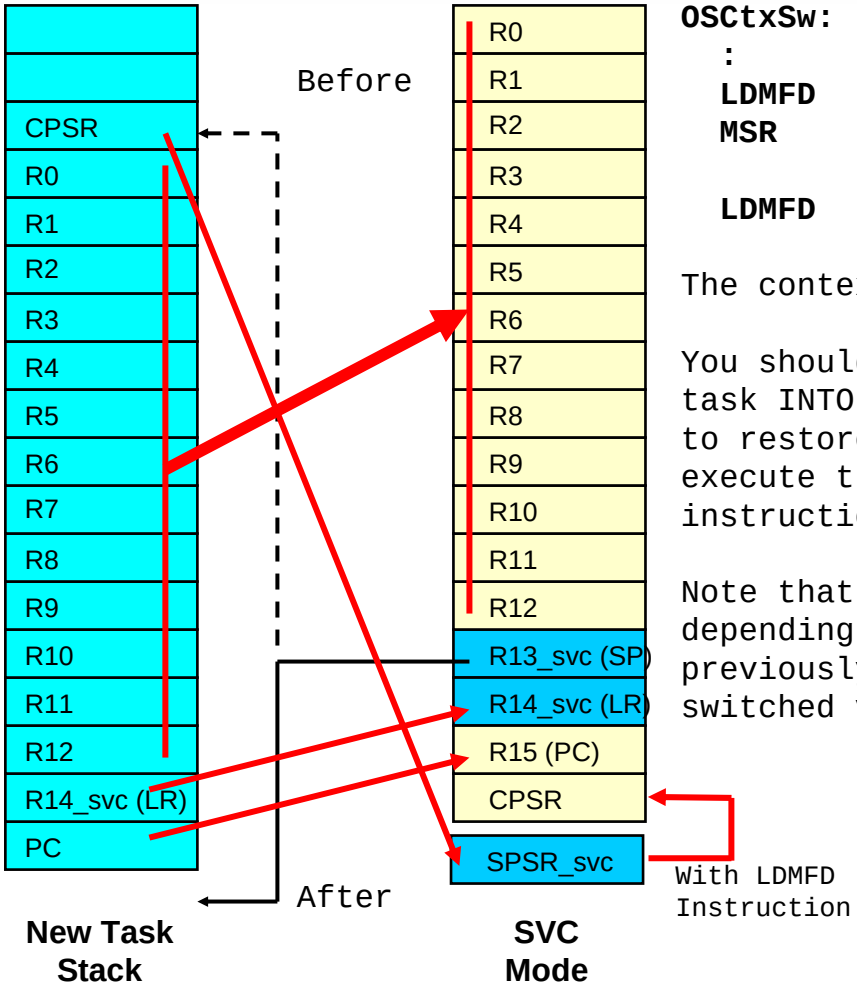
The stack pointer is loaded from the **OS_TCB** of the new task.

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:	**1**	**1**	**0**	**0x13**

Task Level Context Switch

OSIntCtxSw()




```
OSCtxSw:
  :
  LDMFD   SP!, {R4}             ; Pop new task's CPSR
  MSR     SPSR_cxsf, R4

  LDMFD   SP!, {R0-R12,LR,PC}^ ; Pop new task's context
```

The context of the new task is pulled off the stack.

You should notice that we restore the CPSR of the new task INTO the SPSR register. The reason we do this is to restore both the CPSR and PC at the same time when we execute the LDMFD instruction. After the last instruction, the CPU resumes the new task.

Note that the interrupts are either enabled or disabled depending on whether we return to a task that was previously interrupted or, a task that was context switched via OSCtxSw().

	I	F	T	MODE
CPSR:	1	1	0	0x13

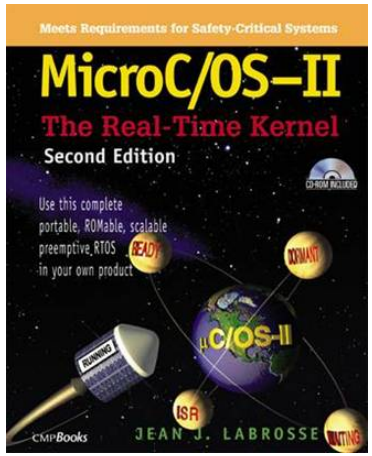
References

"µC/OS-II, The Real-Time Kernel,

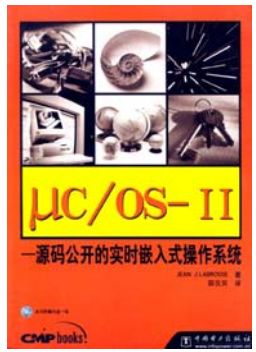
2nd Edition"

Jean J. Labrosse, CMP Books

ISBN 1-57820-103-9



Chinese



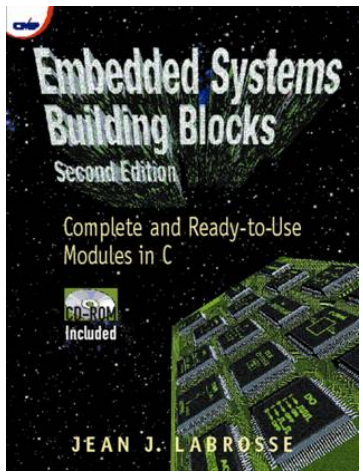
Korean

"Embedded Systems Building Blocks,

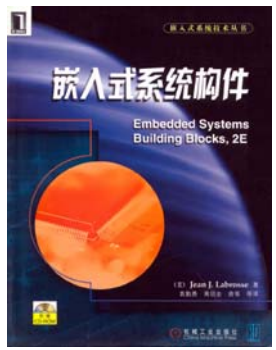
Complete and Ready-to-Use Modules in C"

Jean J. Labrosse, CMP Books

ISBN 1-97930-604-1



Chinese



Korean