Porting µC/OS-II

Part - II

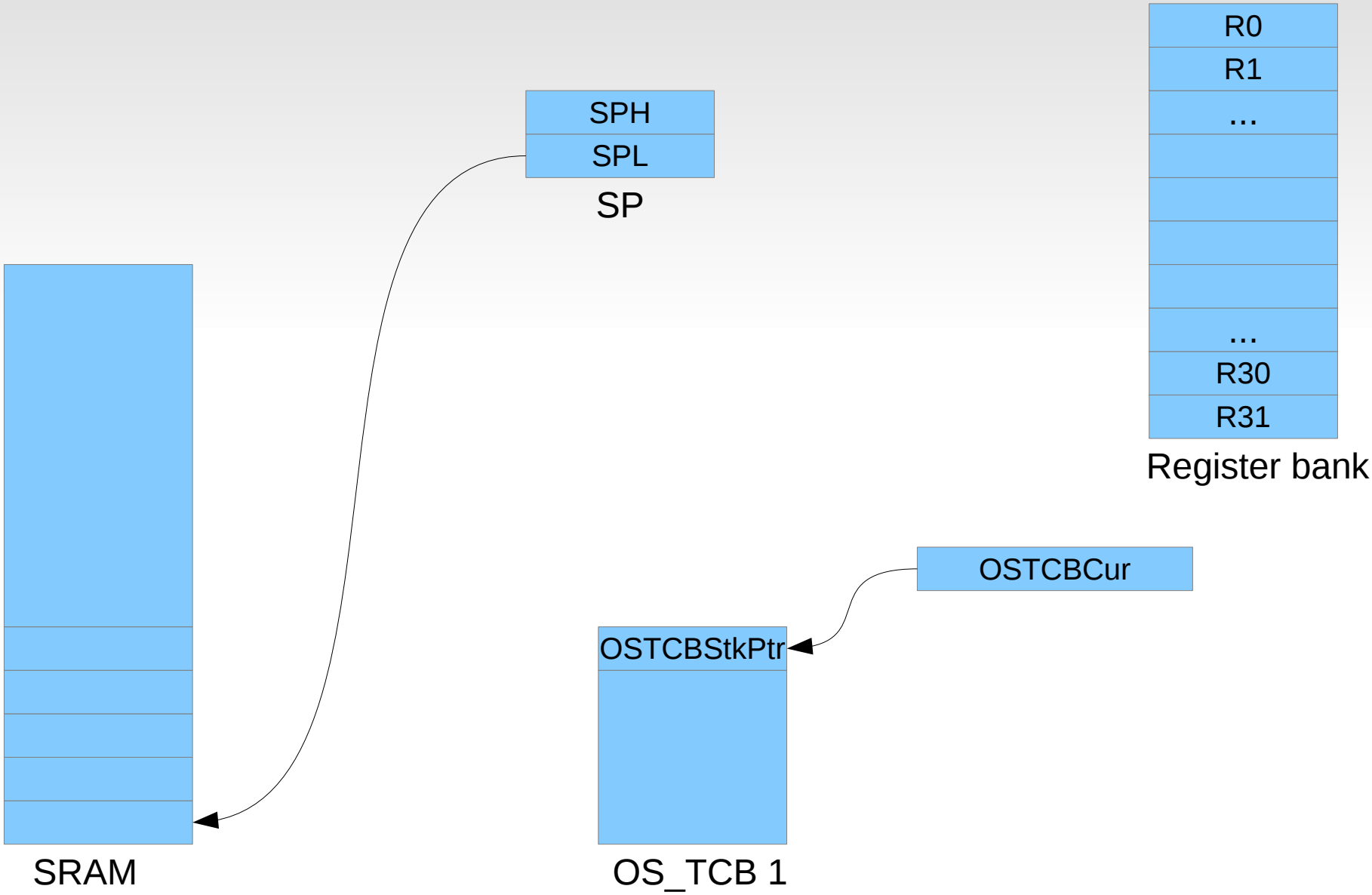
Prateek Rajgarhia
CDAC, Hyderabad

Assembly functions
OSCtxSw()

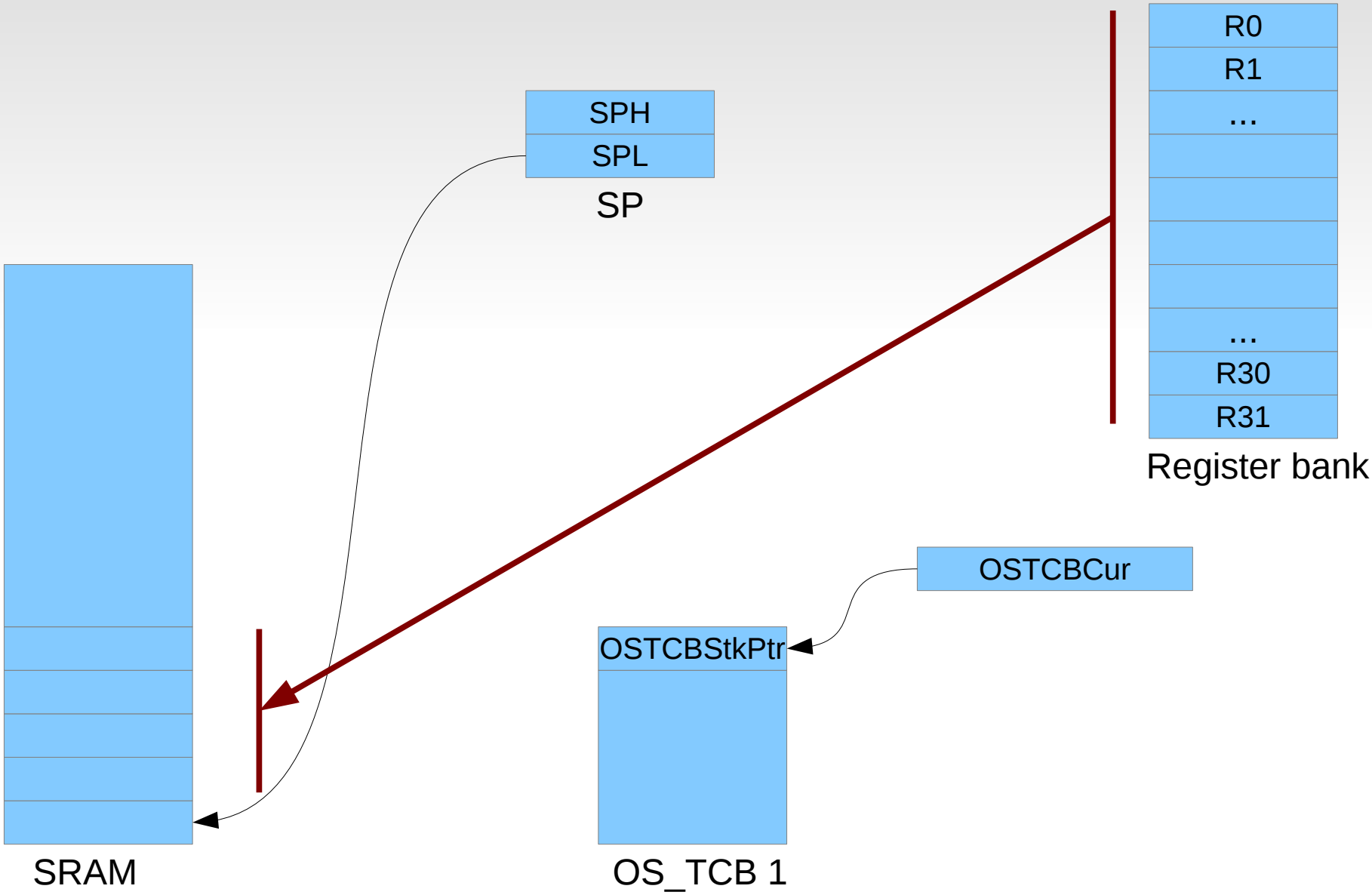
- OSCtxSw() function performs the task-level context switch.
- Pseudo code:

```
void OSCtxSw(void)
{
    Save processor registers;
    Save the current task's stack pointer into the current task's OS_TCB:
      OSTCBCur->OSTCBStkPtr = Stack pointer;
    Call user definable OSTaskSwHook();
    OSTCBCur  = OSTCBHighRdy;
    OSPrioCur = OSPrioHighRdy;
    Get the stack pointer of the task to resume:
      Stack pointer = OSTCBHighRdy->OSTCBStkPtr;
    Restore all processor registers from the new task's stack;
    Execute a return from interrupt instruction;
}
```

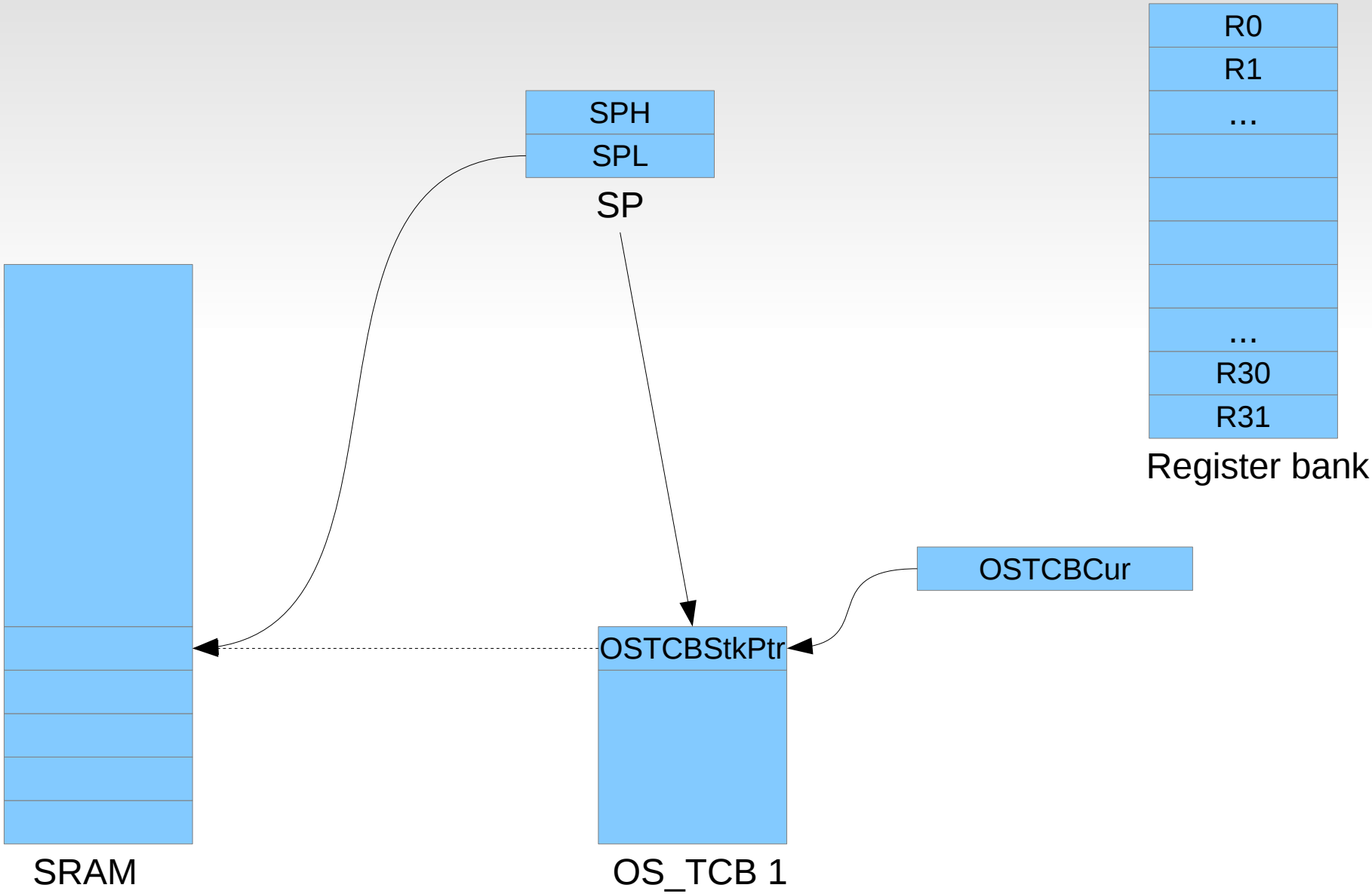
OSCtxSw() : Logic flow



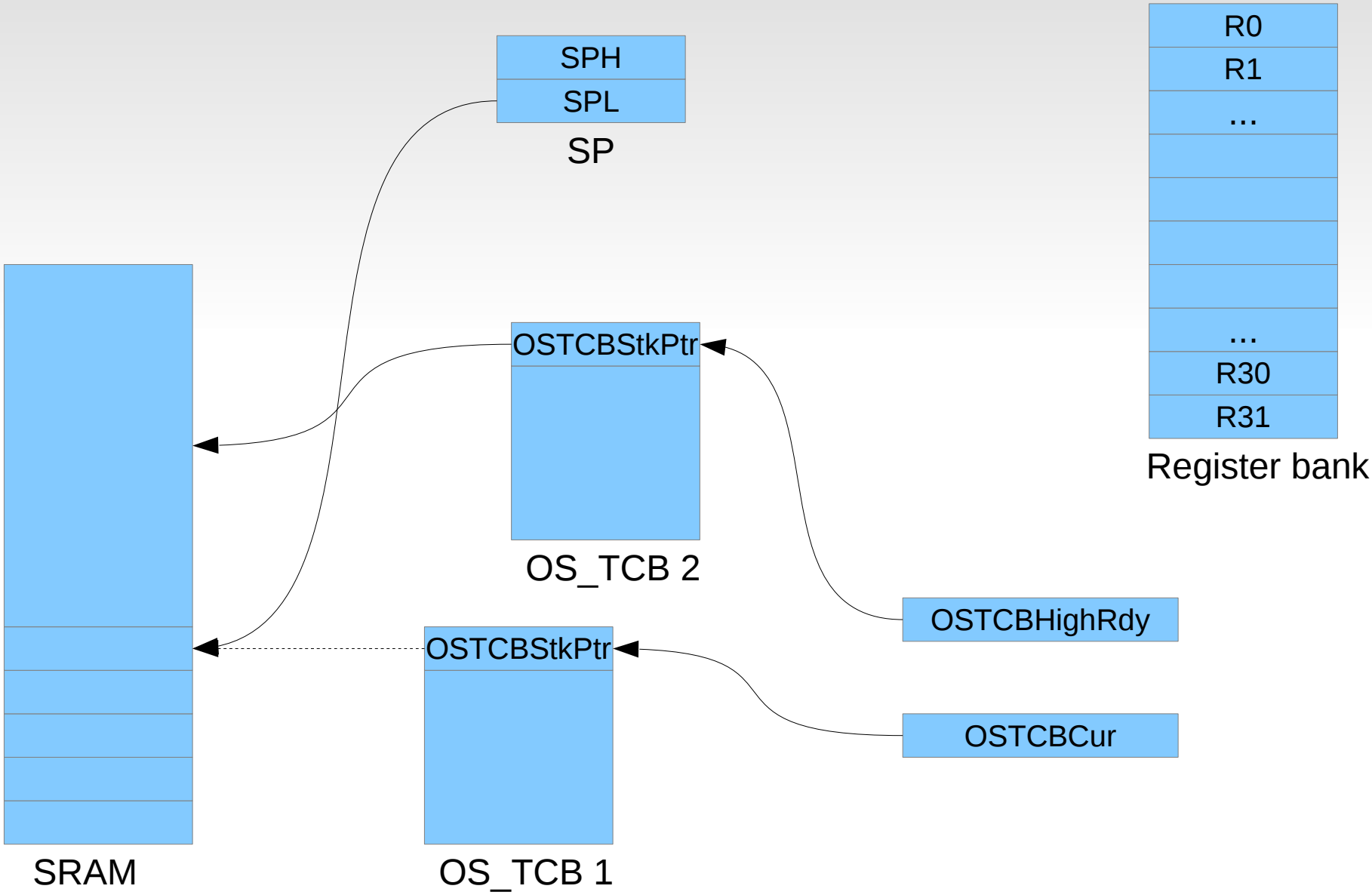
OSCtxSw() : Logic flow



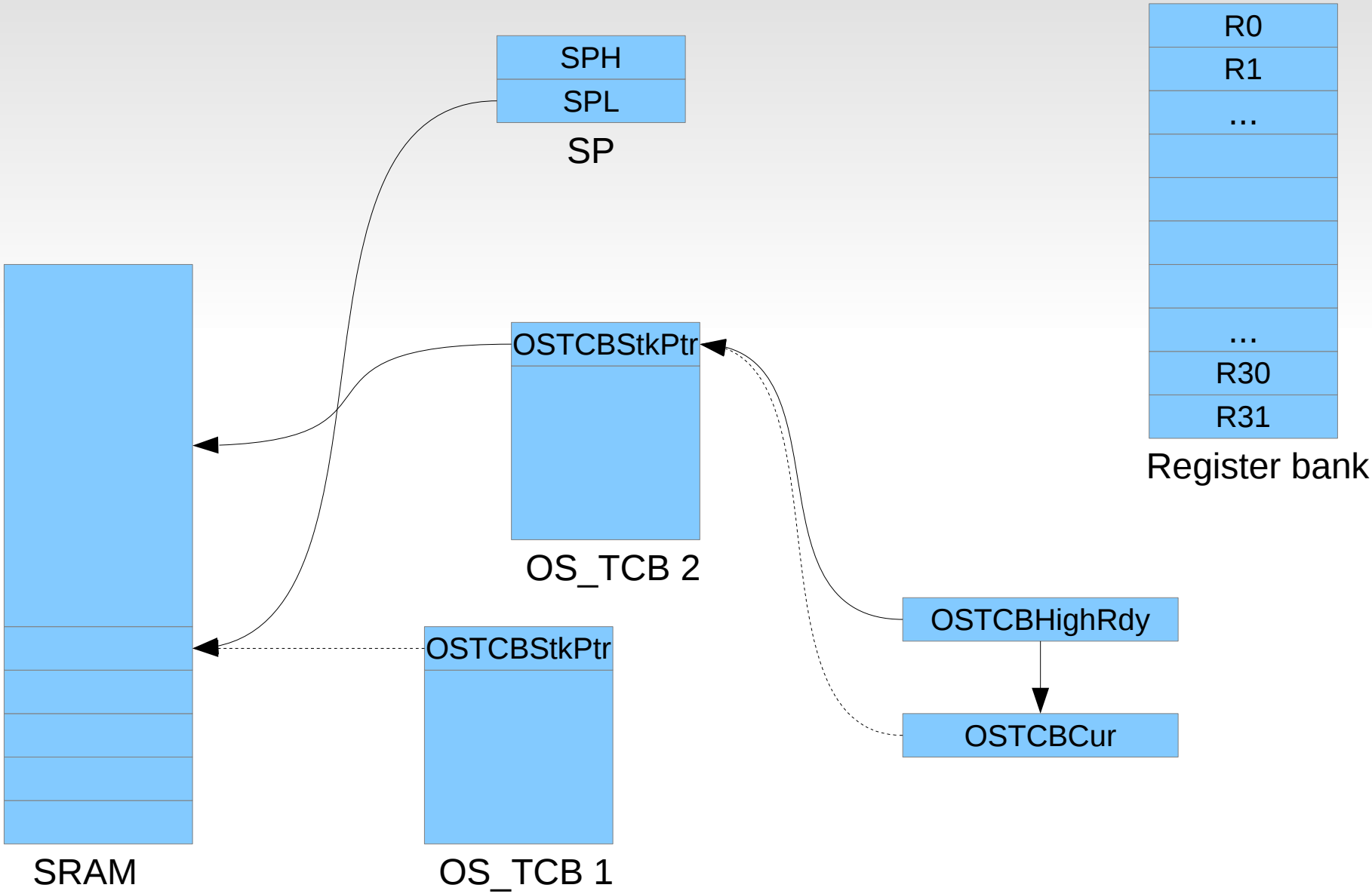
OSCtxSw() : Logic flow



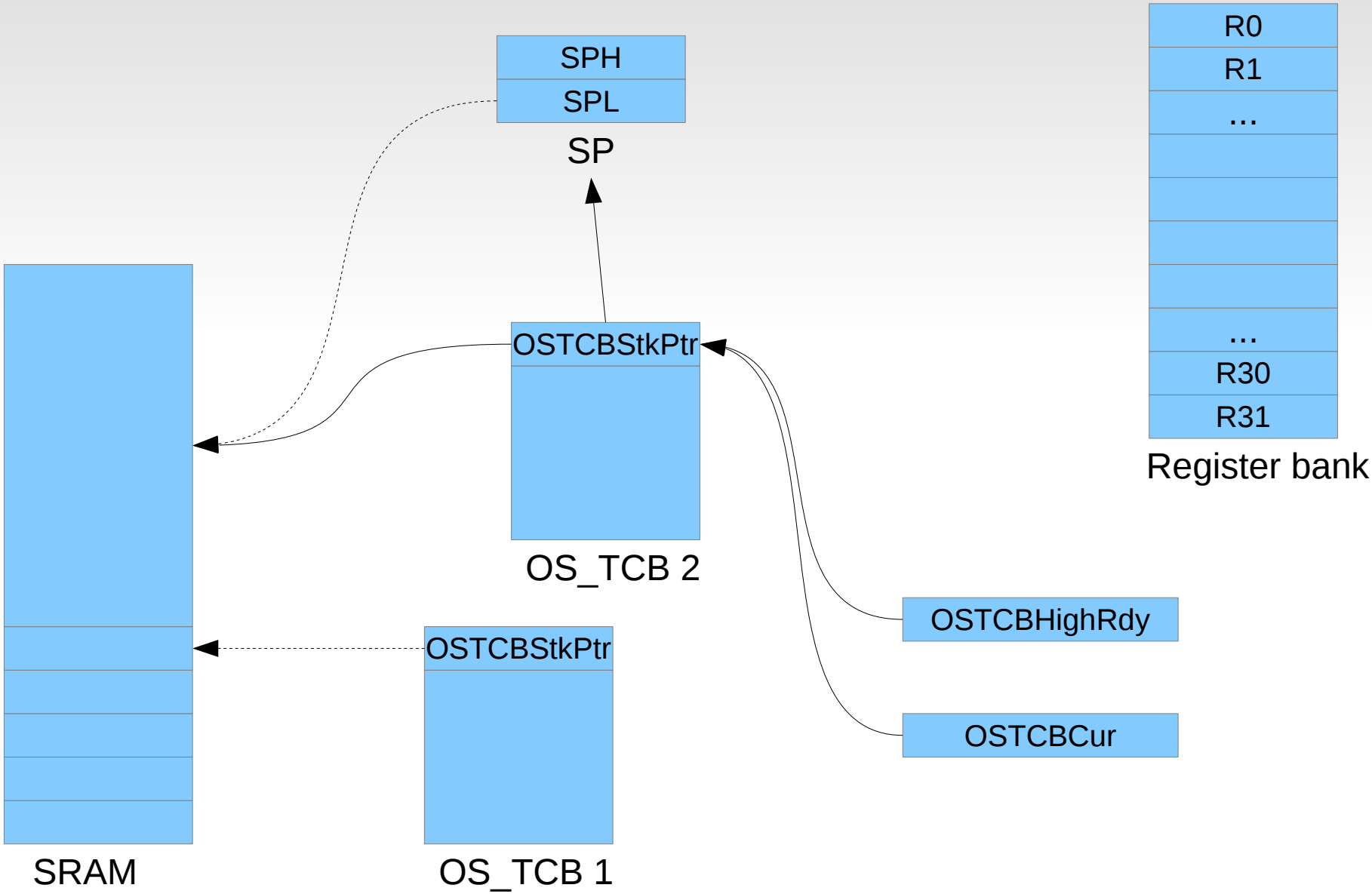
OSCtxSw() : Logic flow



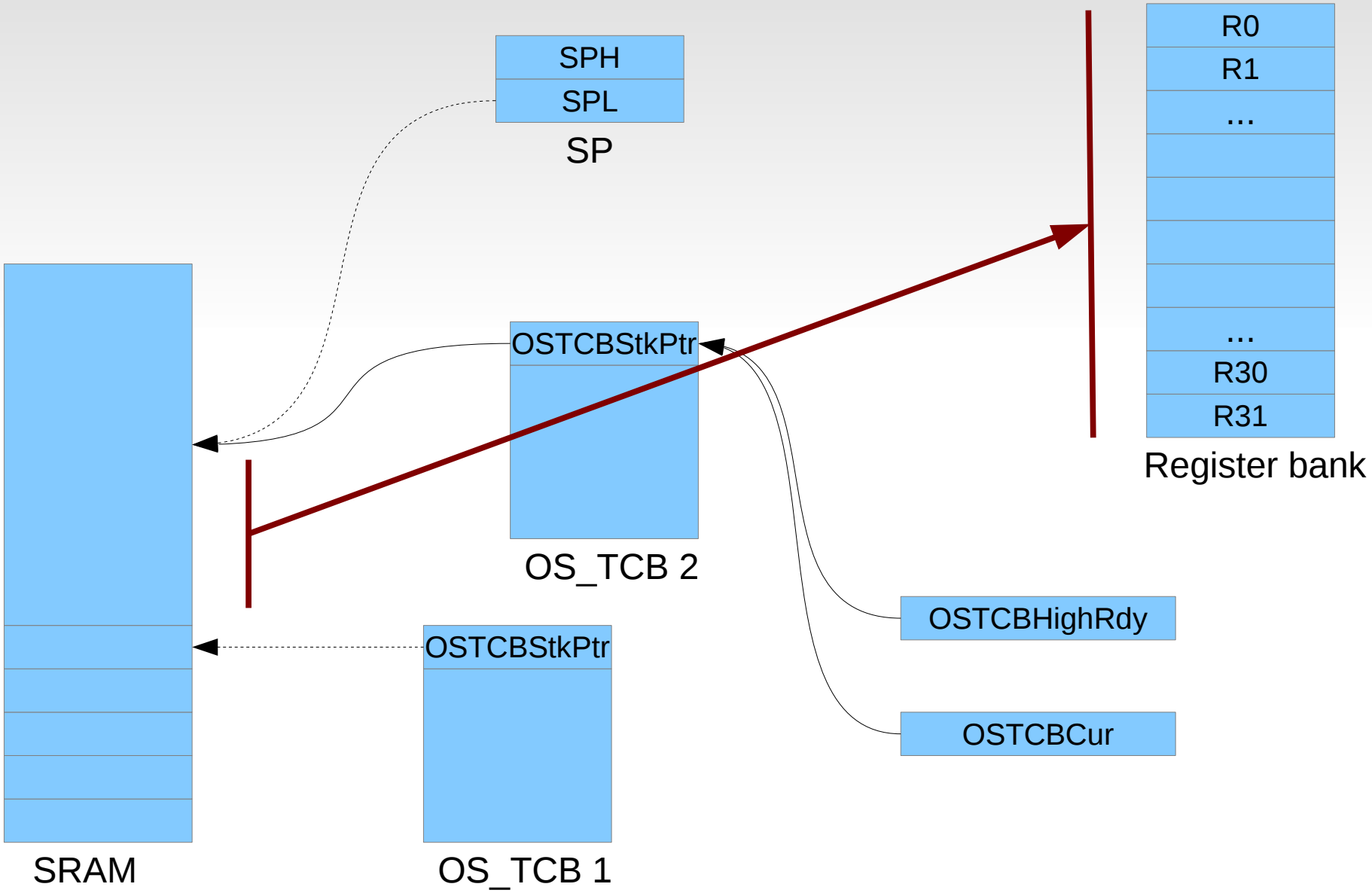
OSCtxSw() : Logic flow



OSCtxSw() : Logic flow

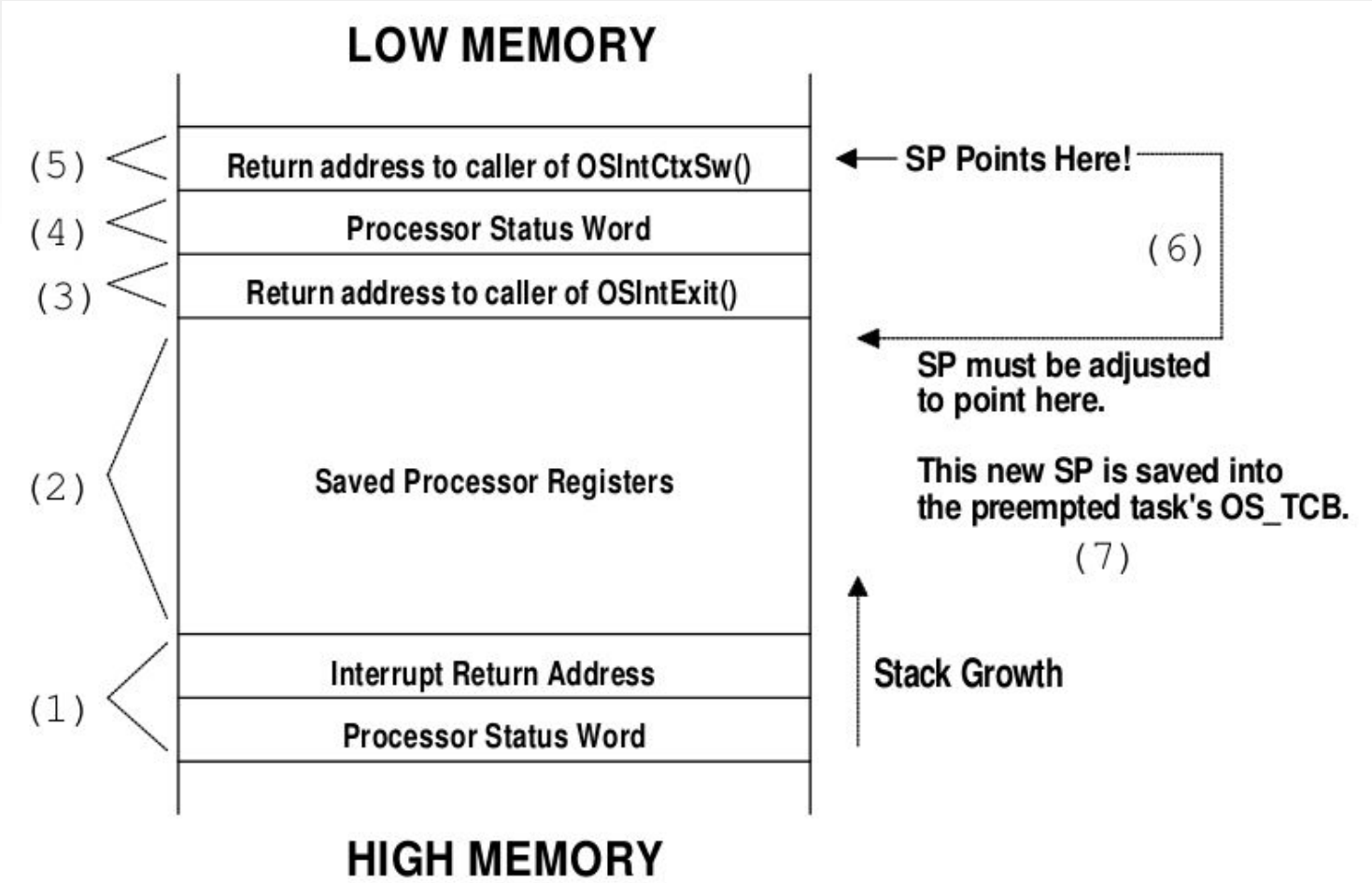


OSCtxSw() : Logic flow

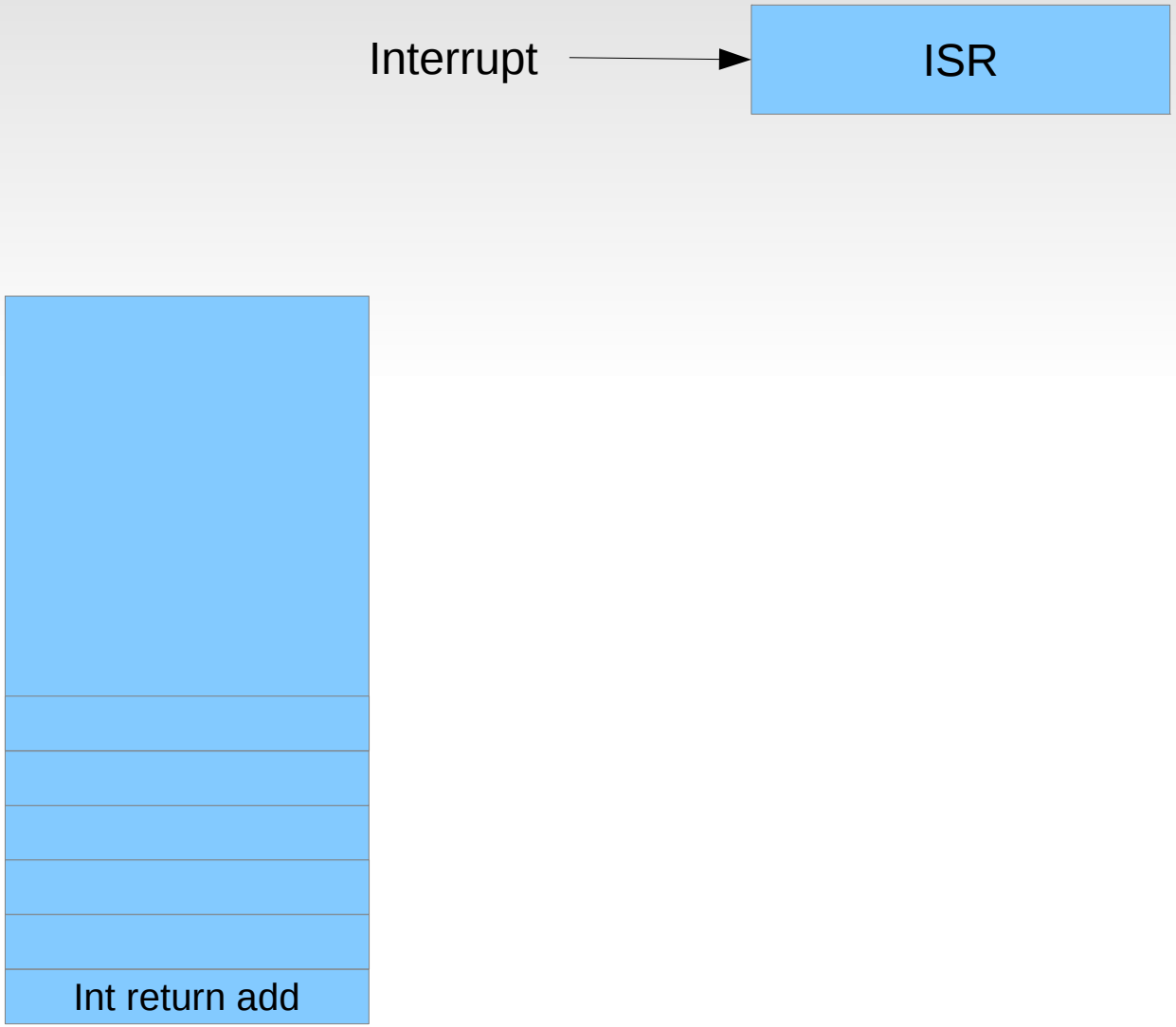


Assembly functions
OSIntCtxSw()

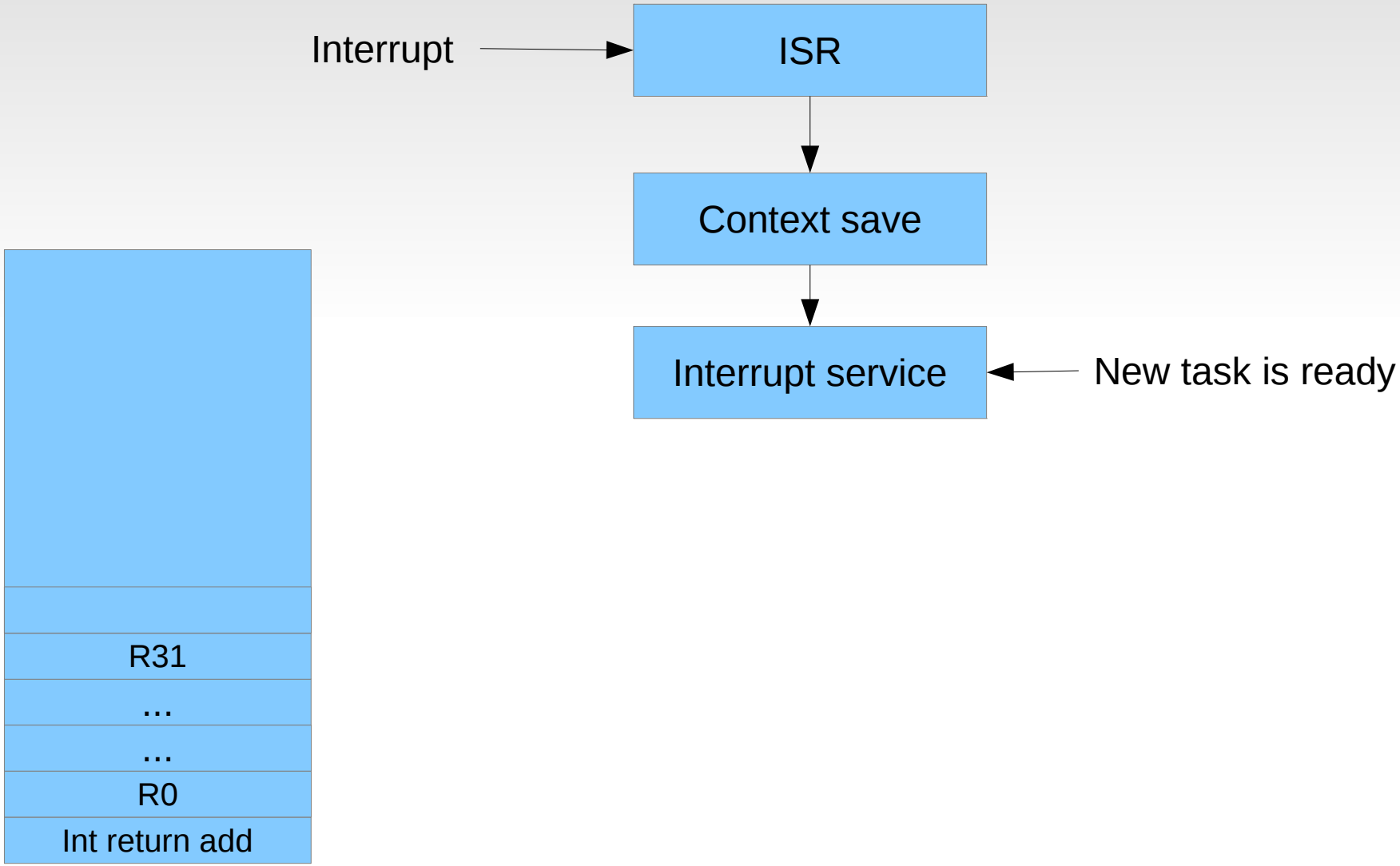
- OSIntCtxSw() function is called by OSIntExit() to perform a context switch from an ISR.



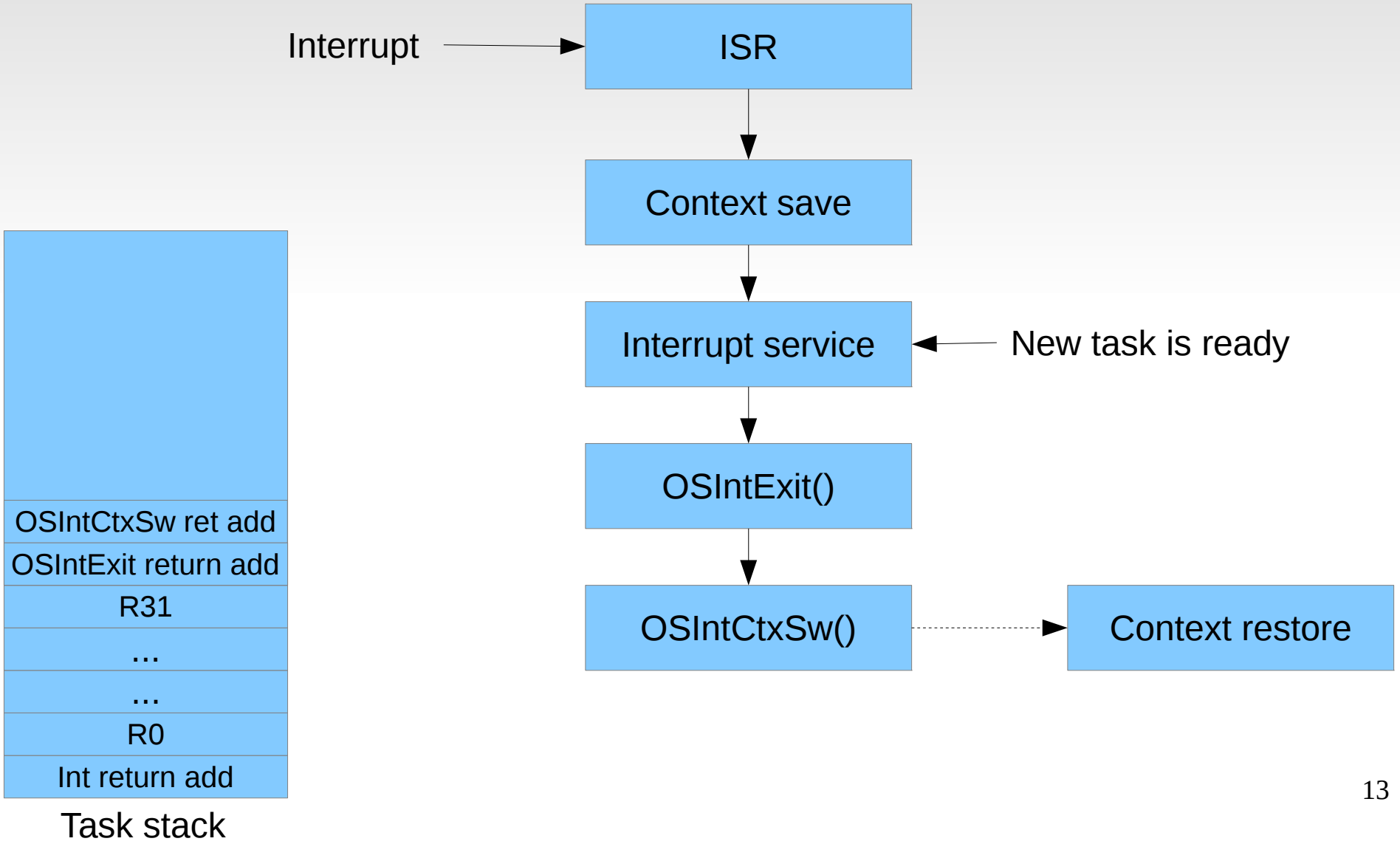
OSIntCtxSw() : Logic flow



OSIntCtxSw() : Logic flow



OSIntCtxSw() : Logic flow



Assembly functions
OSIntCtxSw()

- Since OSIntCtxSw() is called from ISR, no need to save processor context registers again.
- Pseudo code:

```
void OSIntCtxSw(void)
{
    Adjust the stack pointer to remove calls to:
        OSIntExit(),
        OSIntCtxSw() and possibly the push of the processor status word;
    Save the current task's stack pointer into the current task's OS_TCB:
        OSTCBCur->OSTCBStkPtr = Stack pointer;
    Call user definable OSTaskSwHook();
    OSTCBCur  = OSTCBHighRdy;
    OSPrioCur = OSPrioHighRdy;
    Get the stack pointer of the task to resume:
        Stack pointer = OSTCBHighRdy->OSTCBStkPtr;
    Restore all processor registers from the new task's stack;
    Execute a return from interrupt instruction;
}
```

Configuring the kernel

- The kernel can be configured/customized using OS_CFG.H file
- Disable kernel services not to be used, in order to save RAM & ROM memory.
- Some important configurations are :
  - CPU_CLOCK_HZ
  - OS_TICKS_PER_SEC
  - OS_TASK_DEF_STK_SIZE
  - OS_LOWEST_PRIO
  - OS_MAX_TASKS
- Services disabled in os_cfg.h, should not be used in application code, or would result in failure.

Writing Application code

- Writing application code is fairly simple, similar to a C program with a main() and tasks defined as functions, all calling kernel services (which are again functions).
- Few important things to be done :
  - Include master header file - “includes.h”
  - Define stacks for all tasks.
  - Initialize board-specific things like UARTs, GPIOs, etc.
  - Create at least one task before calling OSStart()
  - Do not enable timer Tick interrupt before calling OSStart()
    - It should be done in first task to be scheduled.
  - Define all ISRs.
  - Define all tasks.

Final points..

- **In addition to the target processor, writing port code is also highly-dependent on the C compiler used.
- Compiler subtleties should be known, like function epilogue/prologue, CPU registers it uses internally, etc.
- Also, assembler directives, how it manages interrupt vector table should be known.
- In case of avr-gcc compiler :
  - Include header files “avr/io.h”, “avr_isr.h”, “avr/interrupt.h” in the application code.
  - Include “avr/io.h” in os_cpu_a.S assembly file.
    - In VMLAB, assembly files should have “.S” extension.

Testing the Port

- Port should be tested by a simple application using a kernel operation, like timer Tick interrupt service.
    - A simple Led Blink application with one task calling OSTimeDly() service.
- It is preferable to first test the port on simulator than on controller board, as debugging is easier in simulator IDE.

- **If the port crashes after a few context switches then, you should suspect that the stack is not being properly adjusted in OSIntCtxSw()

Compile, build, simulate & burn...