Porting µC/OS-II

Part - I

Prateek Rajgarhia
CDAC, Hyderabad

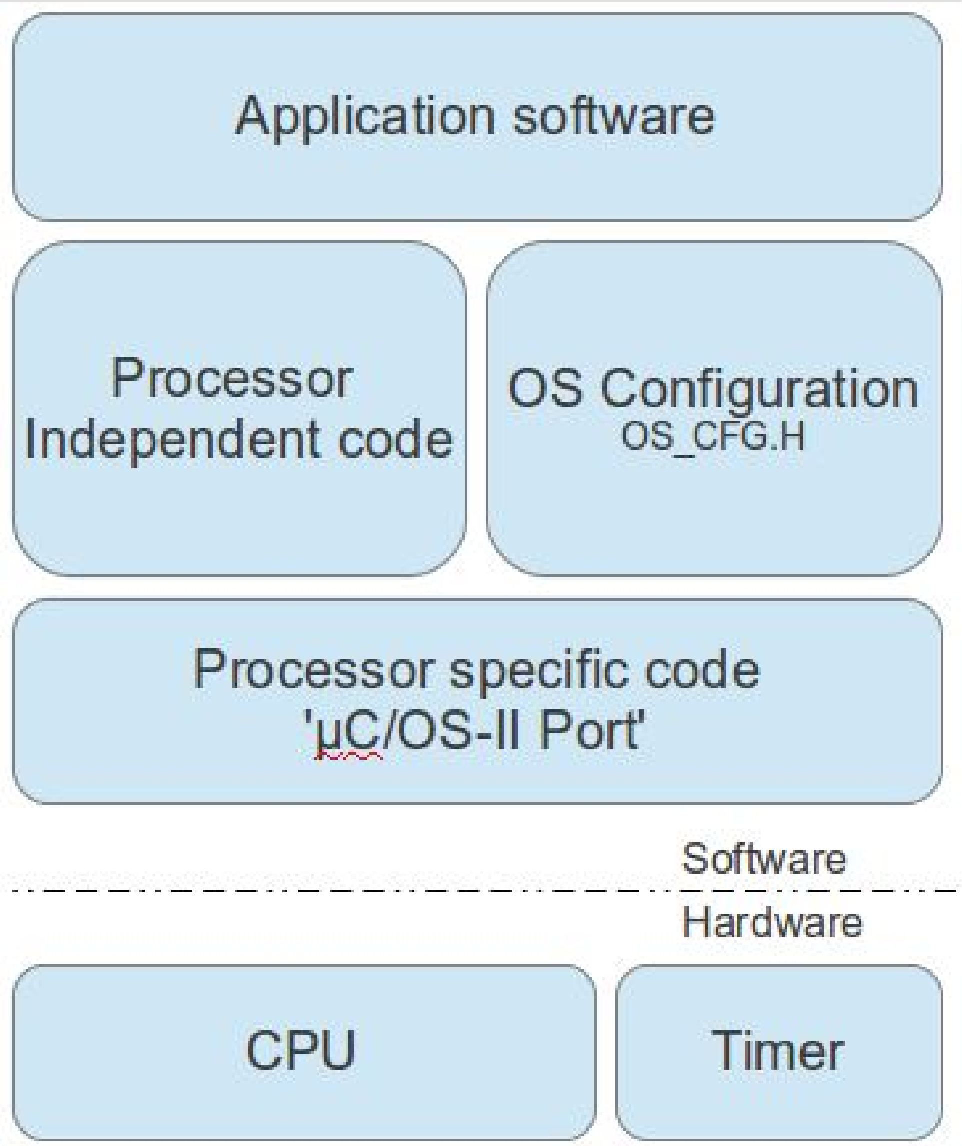
What is Porting ?

- Adapting a software to a different hardware or platform is called 'porting'.
- A software may not work on another system because of different processor, OS or libraries used.
- A software is said to be portable, if the effort of writing port code is less than writing it completely from scratch.
  - This is achieved by using portable languages like C/C++, Java, etc. for which compilers are available for different platforms.
  - Also, by segregating machine-independent code & machine-dependent code.

Requirements to port µC/OS-II

- C compiler must produce re-entrant code.
- Must be able to control interrupts from C code.
- Support for timer interrupts.
- Hardware stack.
- Instructions to load/store stack pointer & other CPU registers.

Hardware/Software Architecture



Steps for Porting µC/OS-II

- Setting the value of a #define constant (OS_CPU.H)
- Declaring 10 data types (OS_CPU.H)
- Declaring 3 #define macros (OS_CPU.H)
- Writing 6 simple functions in C (OS_CPU_C.C)
- Writing 4 assembly language functions (OS_CPU_A.ASM)

Step 1 : OS_STK_GROWTH

- The Stack on some processors/controllers grows from high-memory to low-memory, while low to high-memory on others.

OS_STK_GROWTH	Memory stack growth
0	Low to High
1	High to Low

- In AVR, stack grows from HIGH to LOW memory address (Full Decrementing).

```
#define   OS_STK_GROWTH       1
```

Steps for Porting µC/OS-II

✔ Setting the value of a #define constant	(OS_CPU.H)
▪ Declaring 10 data types	(OS_CPU.H)
▪ Declaring 3 #define macros	(OS_CPU.H)
▪ Writing 6 simple functions in C	(OS_CPU_C.C)
▪ Writing 4 assembly language functions	(OS_CPU_A.ASM)

Step 2 : Data types

- Because different microprocessors have different word length, the port of μC/OS-II includes a series of type definitions that ensures portability.
- For 8-bit AVR, definitions are :

```
typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;         /* Unsigned  8 bit quantity         */
typedef signed   char  INT8S;         /* Signed    8 bit quantity         */
typedef unsigned int   INT16U;        /* Unsigned 16 bit quantity         */
typedef signed   int   INT16S;        /* Signed   16 bit quantity         */
typedef unsigned long  INT32U;        /* Unsigned 32 bit quantity         */
typedef signed   long  INT32S;        /* Signed   32 bit quantity         */
typedef float          FP32;          /* Single precision floating point */
typedef unsigned char  OS_STK;        /* Each stack entry is 8-bit wide  */
```

Steps for Porting µC/OS-II

✔ Setting the value of a #define constant	(OS_CPU.H)
✔ Declaring 10 data types	(OS_CPU.H)
▪ Declaring 3 #define macros	(OS_CPU.H)
▪ Writing 6 simple functions in C	(OS_CPU_C.C)
▪ Writing 4 assembly language functions	(OS_CPU_A.ASM)

Step 3 : Macros

OS_ENTER_CRITICAL() & OS_EXIT_CRITICAL()

- µC/OS-II defines two macros to disable and enable interrupts for protecting critical sections of code.
- Interrupt Disable time largely depends on the method chosen.
- Method 1 :
  - The simplest way to implement is to invoke the processor instruction to disable and enable interrupts.
  - For AVR :

```
#if      OS_CRITICAL_METHOD == 1
#define  OS_ENTER_CRITICAL()    asm volatile ("cli")    /* Disable interrupts   */
#define  OS_EXIT_CRITICAL()     asm volatile ("sei")    /* Enable  interrupts   */
#endif
```

Step 3 : Macros

OS_ENTER_CRITICAL() & OS_EXIT_CRITICAL()

  - Both "CLI" & "SEI" instructions take 1 clock cycle each to execute, therefore adds 2 clock cycles to the Interrupt Disable time.
  - But there is a problem: if a µC/OS-II function is called with interrupts disabled then, upon return, interrupts would be enabled.
- Method 2 :
  - In OS_ENTER_CRITICAL(), first save the interrupt disable status onto the stack and then, disable interrupts.
  - OS_EXIT_CRITICAL() would simply be implemented by restoring the interrupt status from the stack.

Step 3 : Macros

OS_ENTER_CRITICAL() & OS_EXIT_CRITICAL()

```
#if      OS_CRITICAL_METHOD == 2
#define  OS_ENTER_CRITICAL()    {      asm volatile (  \
                                                       "in %0,63" "\n\t"        \
                                                       "cli" "\n\t"             \
                                                       "push %0" "\n\t"         \
                                                       : /*no outputs*/         \
                                                       : "r" (0) );             \
                                                        }

#define  OS_EXIT_CRITICAL()     {      asm volatile (  \
                                                       "pop %0"        "\n\t"   \
                                                       "out 63,%0" "\n\t"       \
                                                       : /*no outputs*/         \
                                                       : "r" (0) );             \
                                                       }

#endif
```

- This method takes 4+3 clock cycles to execute, therefore adds 7 clock cycles to Interrupt Disable time.

- Selected method has to be defined in OS_CPU.H

```
#define  OS_CRITICAL_METHOD    1
```

Step 3 : Macros

OS_TASK_SW()

- This macro is invoked when µC/OS-II switches from a low-priority task to the highest-priority task.
- It is always called from task level code.
- In µC/OS-II, the stack frame for a ready task always looks as if an interrupt has just occurred and all processor registers were saved onto it.
- To switch context, OS_TASK_SW() has to be implemented so as to simulate an interrupt.
  - Most processors provide either software interrupt (SWI) or TRAP instructions to accomplish this.
- But AVR does not provide any such instructions.

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

Steps for Porting µC/OS-II

- ✔ Setting the value of a #define constant (OS_CPU.H)
- ✔ Declaring 10 data types (OS_CPU.H)
- ✔ Declaring 3 #define macros (OS_CPU.H)
- Writing 6 simple functions in C (OS_CPU_C.C)
- Writing 4 assembly language functions (OS_CPU_A.ASM)

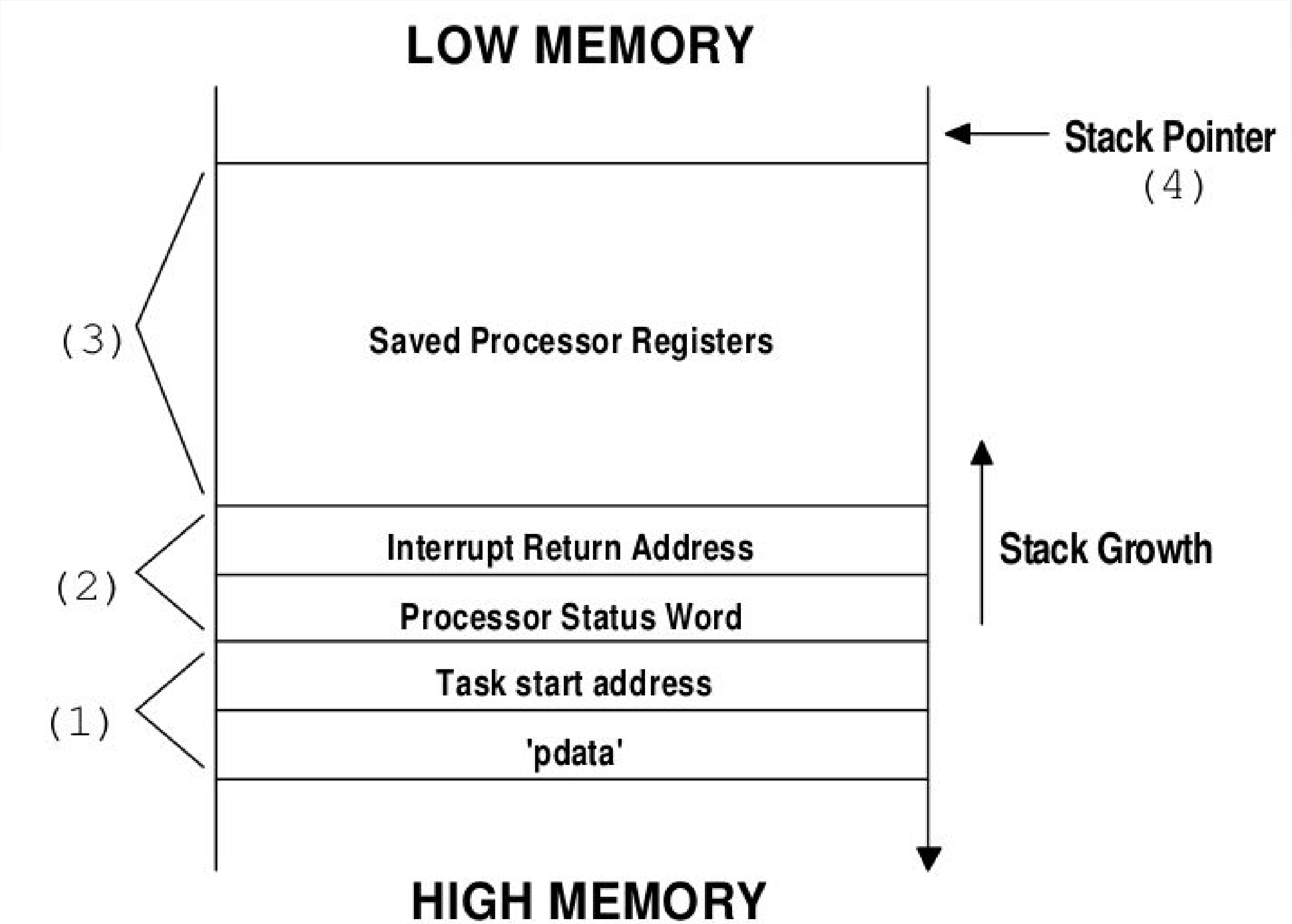
Step 4 : C functions

- µC/OS-II port requires six C functions to be written:
    - OSTaskStkInit()
    - OSTaskCreateHook()
    - OSTaskDelHook()
    - OSTaskSwHook()
    - OSTaskStatHook()
    - OSTimeTickHook()
- Hook functions are used when user wants to extend the functionality of µC/OS-II.
- Only OSTaskStkInit() is necessary; rest all can be just declared.

Step 4 : C functions

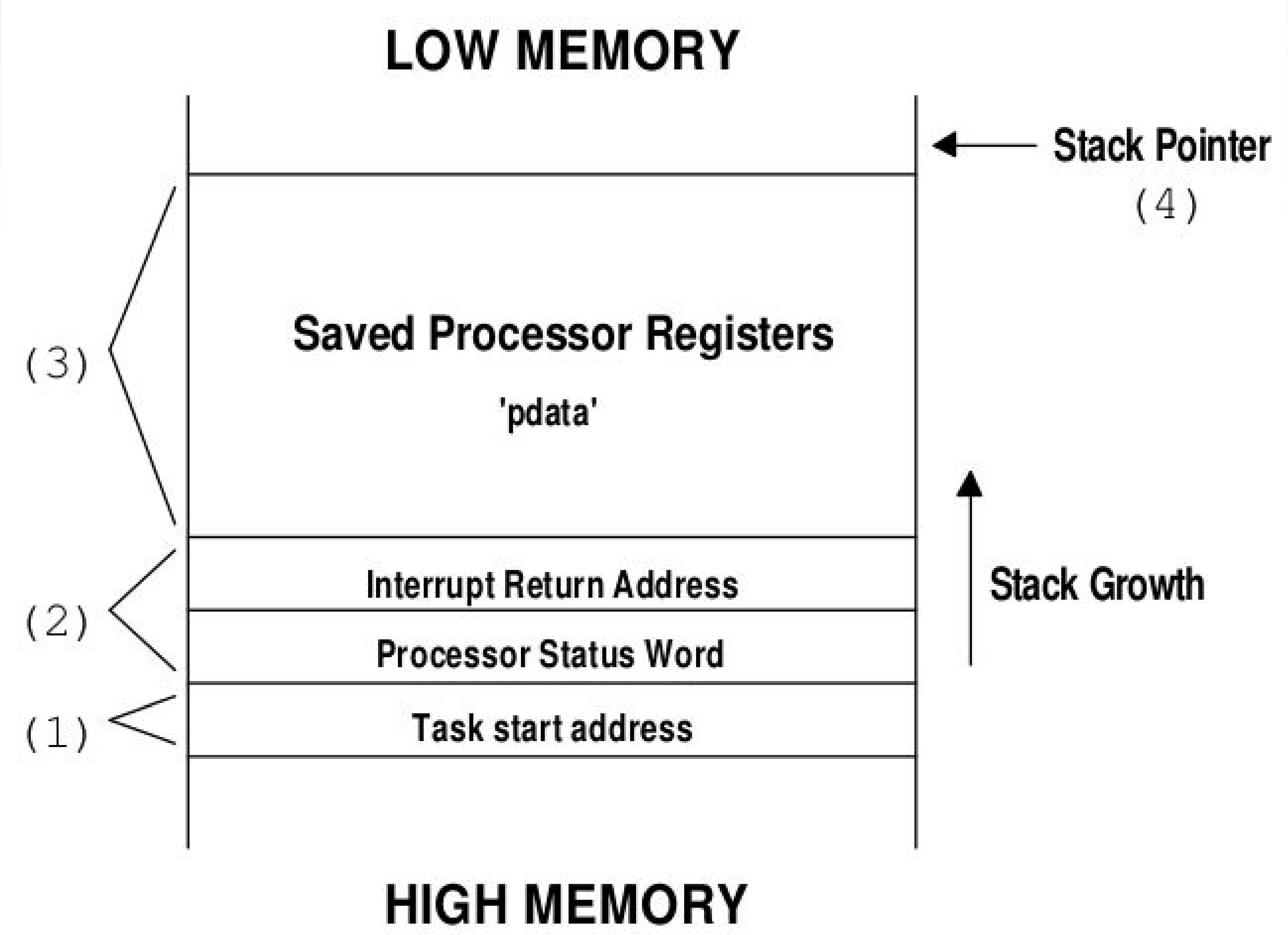
OSTaskStkInit()

- This function is called by OSTaskCreate() and OSTaskCreateExt() to initialize the stack frame of a task.



Step 4 : C functions
OSTaskStkInit()

- Some C compilers pass 'pdata' argument in registers instead of on the stack.



Steps for Porting µC/OS-II

- ✔ Setting the value of a #define constant (OS_CPU.H)
- ✔ Declaring 10 data types (OS_CPU.H)
- ✔ Declaring 3 #define macros (OS_CPU.H)
- ✔ Writing 6 simple functions in C (OS_CPU_C.C)
- Writing 4 assembly language functions (OS_CPU_A.ASM)

Step 5 : Assembly functions

- µC/OS-II port requires 4 assembly language functions to be written :
    - OSStartHighRdy()
    - OSTickISR()
    - OSCtxSw()
    - OSIntCtxSw()

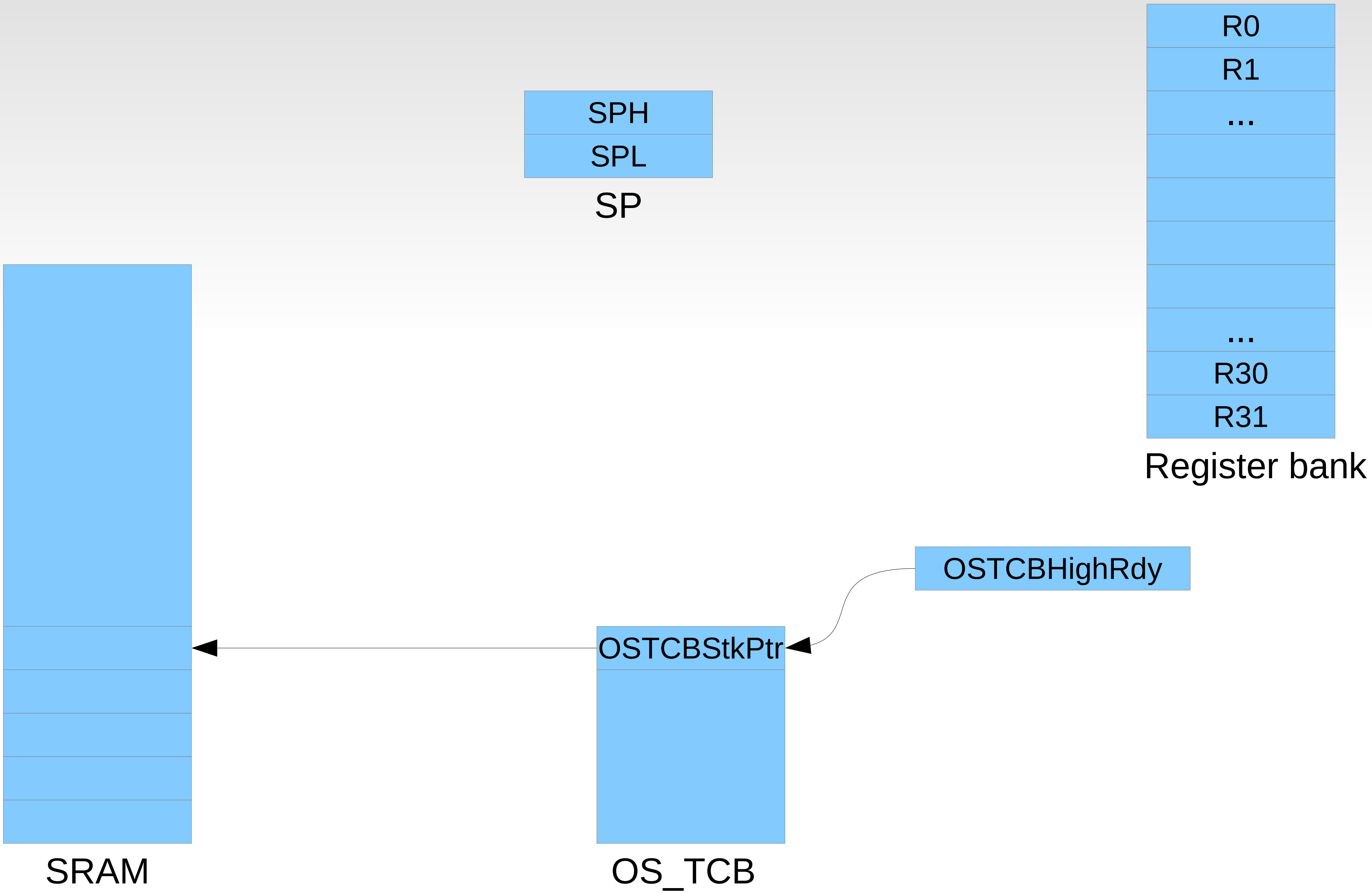
- All these functions need to manipulate CPU registers directly, therefore, they are written in assembly.
    - If C compiler supports in-line assembly code, then they can be written in C also.

Step 5 : Assembly functions
OSStartHighRdy()

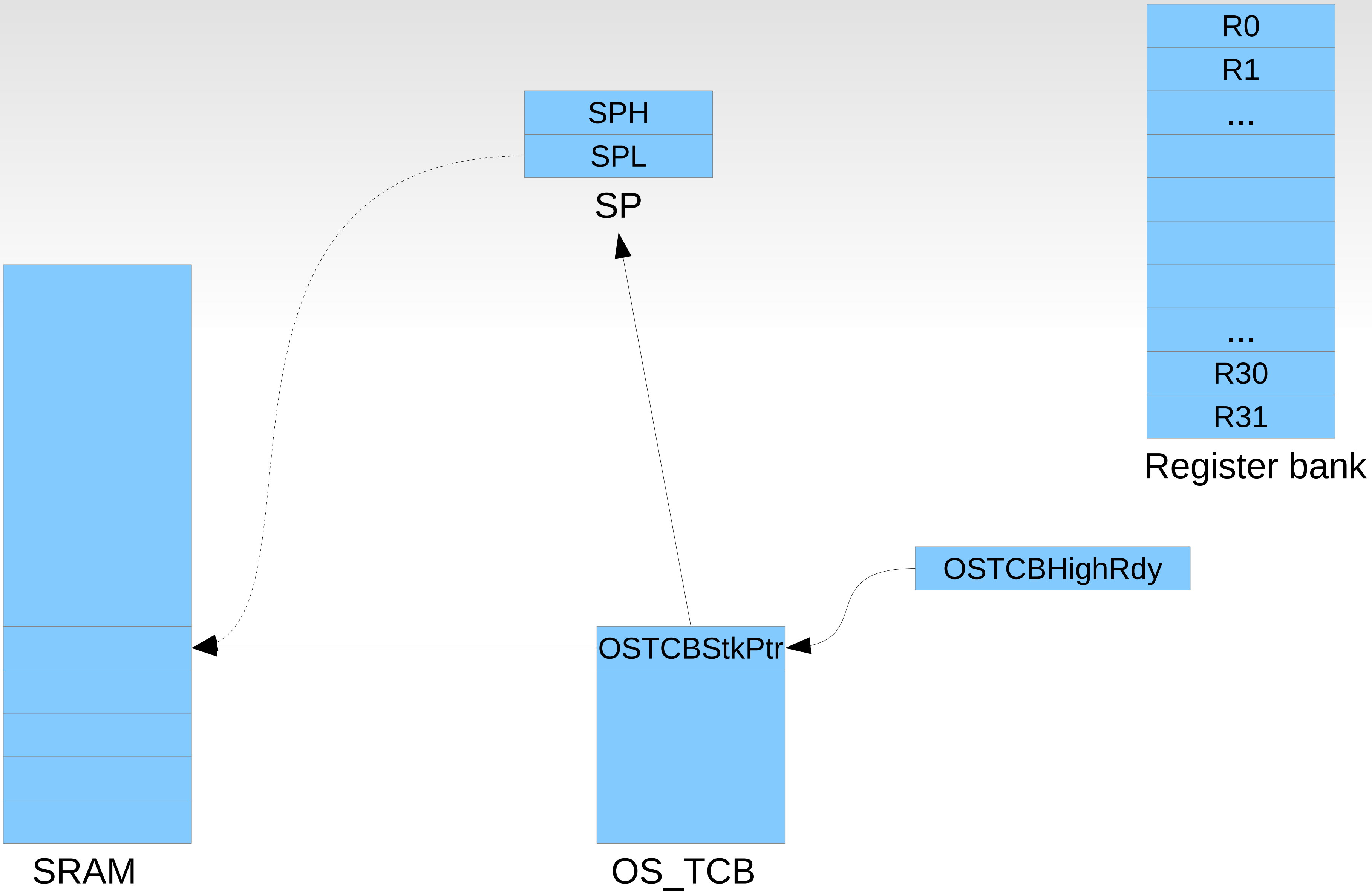
- This function is called by OSStart() to start the highest priority task ready-to-run.

- It assumes OSTCBHighRdy points to TCB of highest priority task.
  - **Therefore, at least one task should be created before calling OSStart() {starting multitasking}.

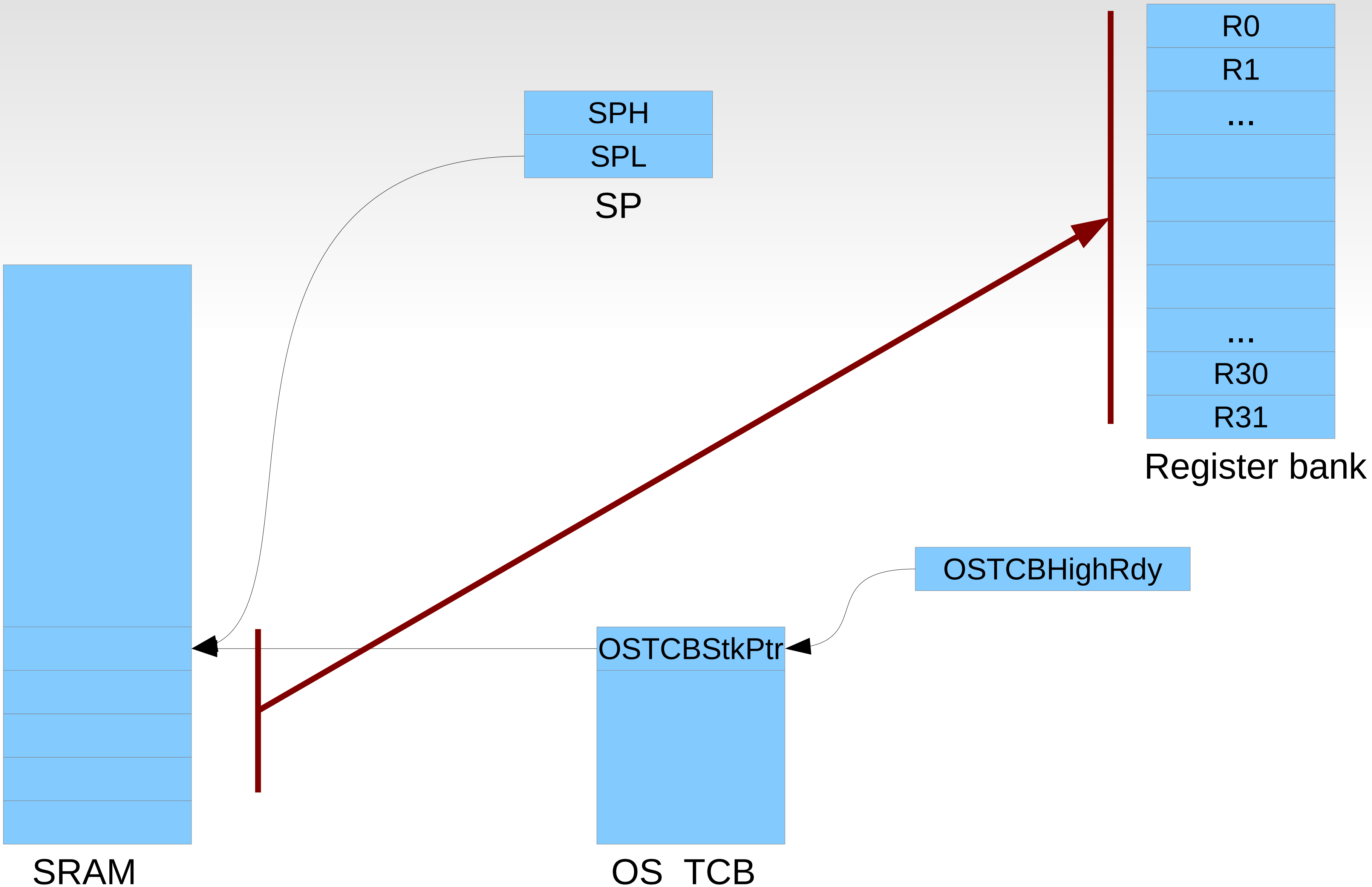
OSStartHighRdy() : Logic flow



OSStartHighRdy() : Logic flow



OSStartHighRdy() : Logic flow



OSStartHighRdy() : Pseudo code

- Call OSTaskSwHook()
- Set OSRunning to TRUE
- Get the StackPointer of the highest priority task
  - `SP = OSTCBHighRdy->OSTCBStkPtr`
- Restore all registers from the stack
- Execute Return

OSStartHighRdy() : C to Assembly

```
void OSStartHighRdy()
{
    OSTaskSwHook();
    OSRunning = TRUE;
    SP = OSTCBHighRdy->OSTCBStkPtr;
    R31 = pop();
    R30 = pop();
    …
    R0 = pop();
    return;
}
```

OSStartHighRdy:

OSStartHighRdy() : C to Assembly

```
void OSStartHighRdy()
{
    OSTaskSwHook();
    OSRunning = TRUE;
    SP = OSTCBHighRdy->OSTCBStkPtr;
    R31 = pop();
    R30 = pop();
    …
    R0 = pop();
    return;
}
```

```
OSStartHighRdy:
    RCALL OSTaskSwHook
```

OSStartHighRdy() : C to Assembly

```
void OSStartHighRdy()
{
   OSTaskSwHook();
   OSRunning = TRUE;
   SP = OSTCBHighRdy->OSTCBStkPtr;
   R31 = pop();
   R30 = pop();
   …
   R0 = pop();
   return;
}
```

```
OSStartHighRdy:
   RCALL OSTaskSwHook
   LDS    R16, OSRunning
   INC     R16
   STS     OSRunning, R16
```

OSStartHighRdy() : C to Assembly

```c
void OSStartHighRdy()
{
   OSTaskSwHook();
   OSRunning = TRUE;
   SP = OSTCBHighRdy->OSTCBStkPtr;
   R31 = pop();
   R30 = pop();
   …
   R0 = pop();
   return;
}
```

```asm
OSStartHighRdy:
   RCALL OSTaskSwHook
   LDS    R16, OSRunning
   INC    R16
   STS    OSRunning, R16
   LDS    R30, OSTCBHighRdy
   LDS    R31,OSTCBHighRdy+1
   LD     R28,Z+
   OUT  _SFR_IO_ADDR(SPL), R28
   LD     R29,Z+
   OUT  _SFR_IO_ADDR(SPH), R29
```

OSStartHighRdy() : C to Assembly

```c
void OSStartHighRdy()
{
   OSTaskSwHook();
   OSRunning = TRUE;
   SP = OSTCBHighRdy->OSTCBStkPtr;
   R31 = pop();
   R30 = pop();
   …
   R0 = pop();
   return;
}
```

```
OSStartHighRdy:
   RCALL OSTaskSwHook
   LDS    R16, OSRunning
   INC    R16
   STS    OSRunning, R16
   LDS    R30, OSTCBHighRdy
   LDS    R31,OSTCBHighRdy+1
   LD     R28,Z+
   OUT _SFR_IO_ADDR(SPL), R28
   LD     R29,Z+
   OUT _SFR_IO_ADDR(SPH), R29
   POP    R31
   POP    R30
   …
   POP    R0
   RET
```

Step 5 : Assembly functions
OSTickISR()

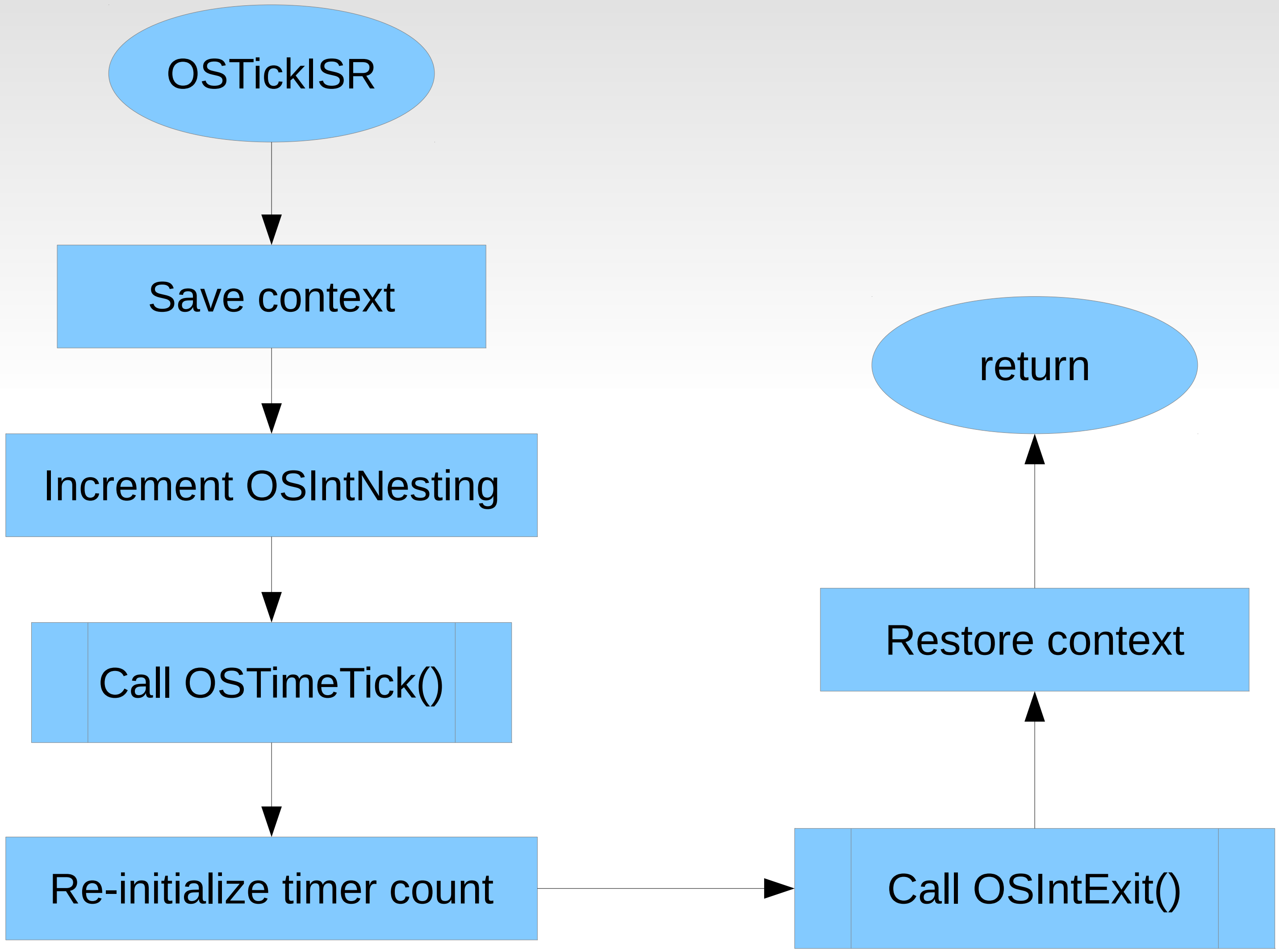
- In µC/OS-II, system 'tick' should occur between 10 to 100 times/second (Hz).
- OSTickISR() is the ISR used to notify µC/OS-II that a system tick has occurred.
- Pseudo code:

```
void OSTickISR(void)
{
    Save processor registers;
    Call OSIntEnter() or increment OSIntNesting;

    Call OSTimeTick();

    Call OSIntExit();
    Restore processor registers;
    Execute a return from interrupt instruction;
}
```

OSTickISR() : Flow chart



Step 5 : Assembly functions
OSTickISR()

- If using Timer0, the timer has to be reloaded before calling OSInitExit() since there is no auto-reload feature.
- **Tick interrupts should NOT be enabled before starting multitasking (calling OSStart()).
  - Or else tick interrupt may be serviced before multitasking starts and application may crash.
  - It should be enabled in the beginning of the highest priority task.

Thank You...