

HW6-1

Meaning of the line:

```
KernelTaskContext_t* ctx = (KernelTaskContext_t*)new_tcb->sp;
```

This line **casts the task's stack pointer (sp) to a task-context structure pointer**.

Each task stores its saved CPU register values (PC, SPSR, R0-R12) in memory using the structure:

```
typedef struct KernelTaskContext_t {
    uint32_t spsr;
    uint32_t r0_r12[13];
    uint32_t pc;
} KernelTaskContext_t;
```

Lab06_task_scheduler

During task creation, the RTOS stores this context **in the task's stack**, so by converting the stack pointer (sp) into a KernelTaskContext_t*, the kernel can **write values into the saved register area**.

So in one sentence:

It means “interpret the memory at the task's stack pointer as a stored CPU context so we can initialize the task's registers.”

HW6-2

Calculate the address where each task function is stored

During task creation (Kernel_task_create()), only the PC field of the saved context is written:

```
KernelTaskContext_t* ctx = (KernelTaskContext_t*)new_tcb->sp;
```

```
ctx->pc = (uint32_t)startFunc;
```

Lab06_task_scheduler

Each TCB is allocated sequentially:

TCB index: 0 1 2

stack_base: TASK_STACK_START + (i * USR_TASK_STACK_SIZE)

stack_top: stack_base + USR_TASK_STACK_SIZE - 4

context_addr: stack_top - sizeof(KernelTaskContext_t)

Given:

USR_TASK_STACK_SIZE = 0x100000

TASK_STACK_START = 0x80000000 (from memory map in project)

sizeof(KernelTaskContext_t) = 4 (spsr) + (13 × 4) + 4 = 4 + 52 + 4 = 60 bytes = 0x3C

Stack addresses used to store the function pointer:

Task	stack_base	SP after context allocation	Stored PC address
Task0	0x80000000	0x800FFFFC - 0x3C = 0x800FFF C0	ctx->pc = &User_task0
Task1	0x80100000	0x801FFFFC - 0x3C = 0x801FFF C0	ctx->pc = &User_task1
Task2	0x80200000	0x802FFFFC - 0x3C = 0x802FFF C0	ctx->pc = &User_task2

So the function PC values are stored at:

Task 0 PC stored at: 0x800FFFC0

Task 1 PC stored at: 0x801FFFC0

Task 2 PC stored at: 0x802FFFC0

(Your numbers may vary if the memory map uses a different base.)

HW6-3

Modify Makefile to build a binary file

Add this conversion rule:

```
rtos.bin: rtos.elf
    $(OBJCOPY) -O binary rtos.elf rtos.bin
```

And modify the final build target:

```
all: rtos.elf rtos.bin
```

What changed?

- We added an objcopy step so the compiler produces both:
rtos.elf → used for debugging
rtos.bin → raw binary required by the bootloader/QEMU
- No source code changed, only build output format.

HW6-4

Complete Round Robin Scheduler

Template given:

```
static uint32_t sCurrent_tcb_index;
static KernelTcb_t* Scheduler_round_robin_algorithm(void);
```

✓ Final working version:

```
static KernelTcb_t* Scheduler_round_robin_algorithm(void)
{
    // Move to next task
    sCurrent_tcb_index++;

    // Wrap around when reaching last task
    if (sCurrent_tcb_index >= sAllocated_tcb_index)
    {
        sCurrent_tcb_index = 0;
    }

    // Return next TCB pointer
    return &sTask_list[sCurrent_tcb_index];
}
```

This scheduling logic simply selects the next task in order and loops back when reaching the end — the definition of **Round Robin scheduling**.