Vsebine registrov pred vstopom v prekinitveno-servisni podprogram za TIM6:

r0 = 40021C00	r0 = 40021C00
r1 = 2007FFDC	r1 = 2007FFDC
r2 = 0	r2 = 0
r3 = 00000010	r3 = 00000010
r4 = 0	r4 = 0
r5 = 0	r5 = 0
r6 = 0	r6 = 0
r7 = 2007FFF8	r7 = 2007FFD0
r8 = r9 = .. = r12 = 0	r8 = .. r12 = 0
! sp = 2007FFF8	sp = 2007FFD0
lr = 08003FCD	lr = FFFFFFF9
pc = 08003B88	pc = 08004210
psr = 01000000	psr = 01000046

Vsebine registrov po vstopu v TIM6 PSP

skladovni kazalec se je zmanjšal ⇒ sklad narašča v smeri padajočih naslovov

! očitno smo nekaj "ponvali" na sklad

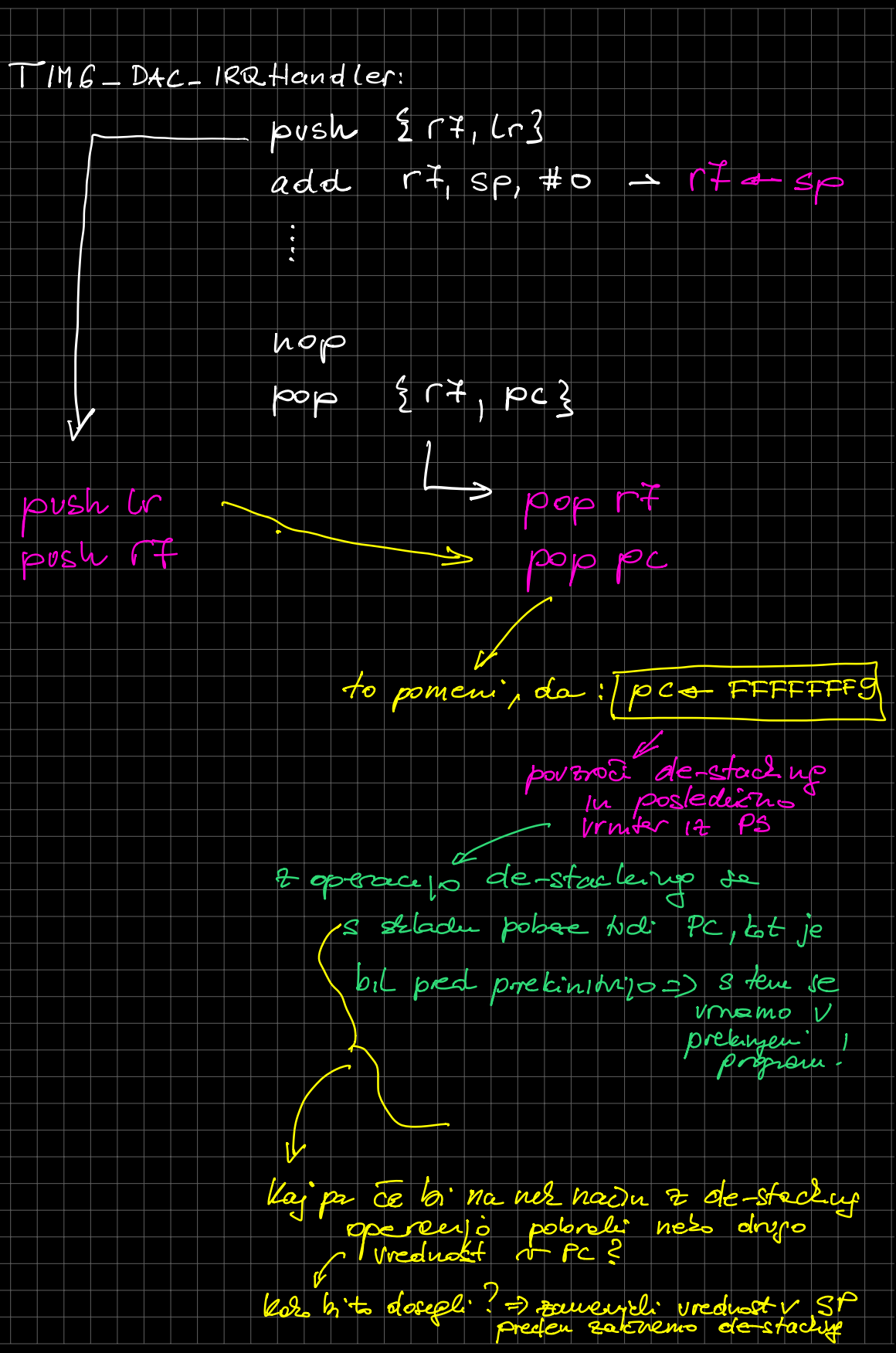
Poglejmo, kaj je na skladu med tema dvema naslovoma:

2007FFD0 :	2007FFF8	r7
2007FFD4 :	FFFFFFF9	lr
2007FFD8 :	40021C00	r0
2007FFDC :	2007FFDC	r1
2007FFE0 :	0	r2
2007FFE4 :	00000010	r3
2007FFE8 :	0	r12
2007FFEC :	08003FCD	lr
2007FFF0 :	08003B88	pc
2007FFF4 :	01000000	psr
2007FFF8 :	////////	

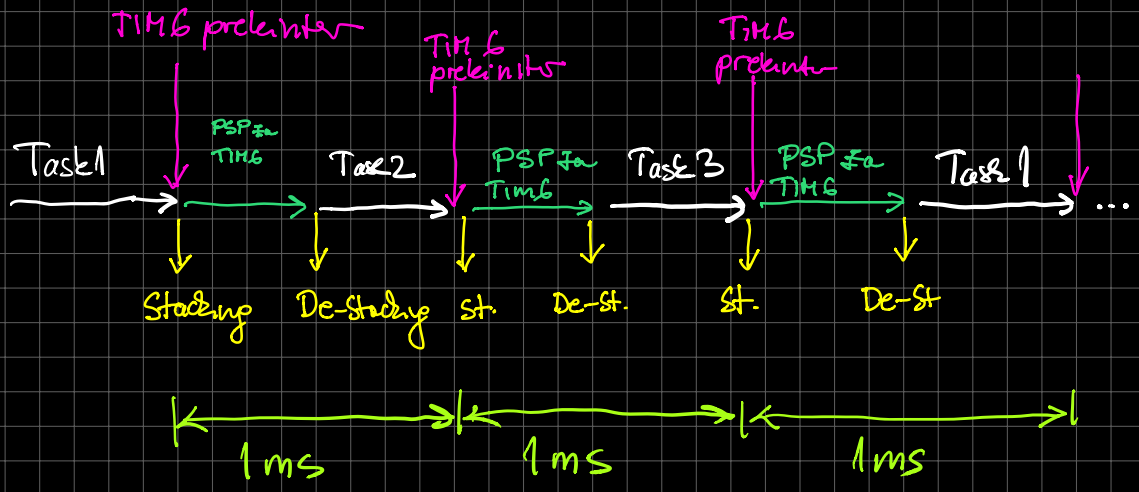
to dodala prevajalnik

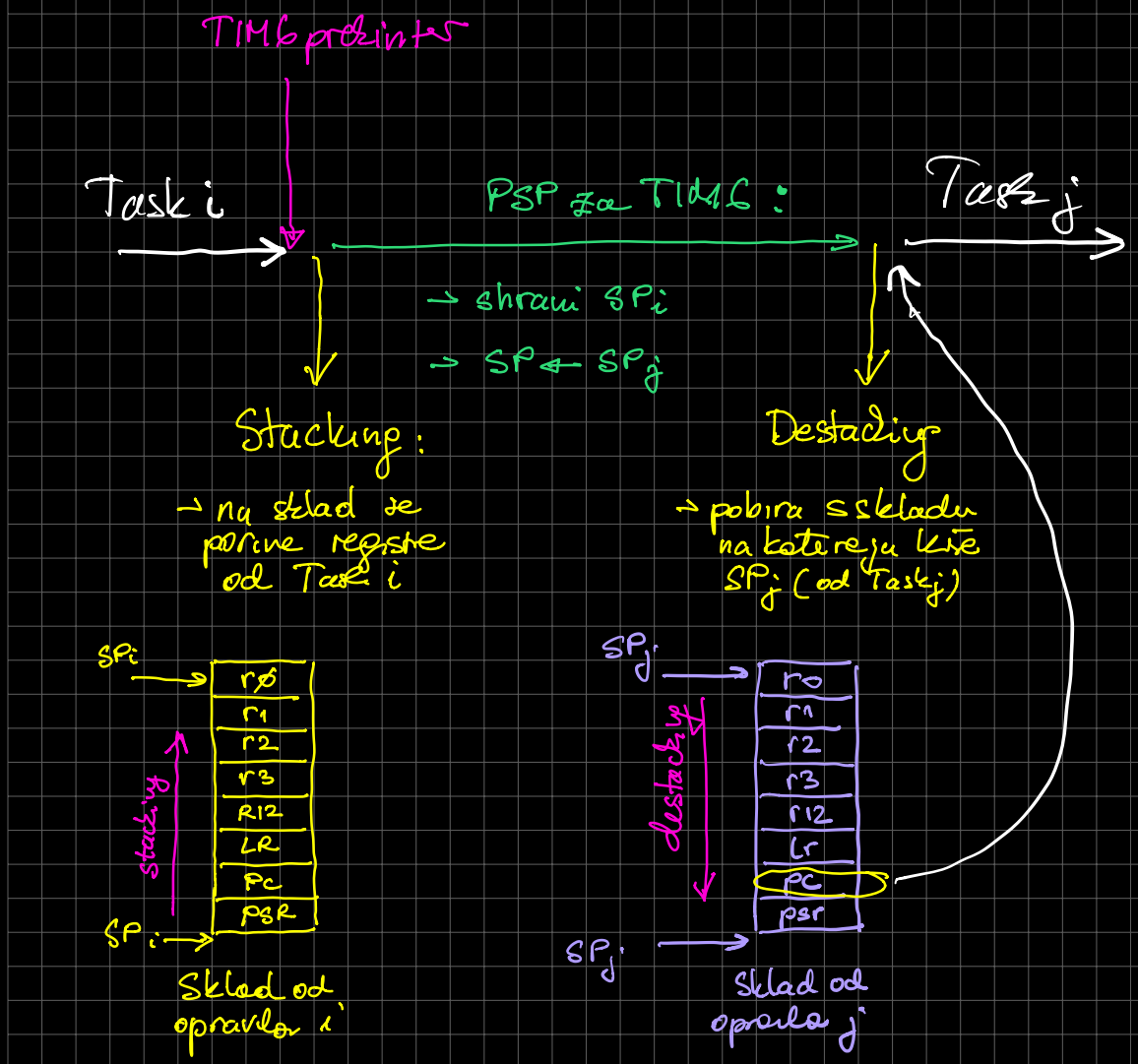
HW STACK FRAME

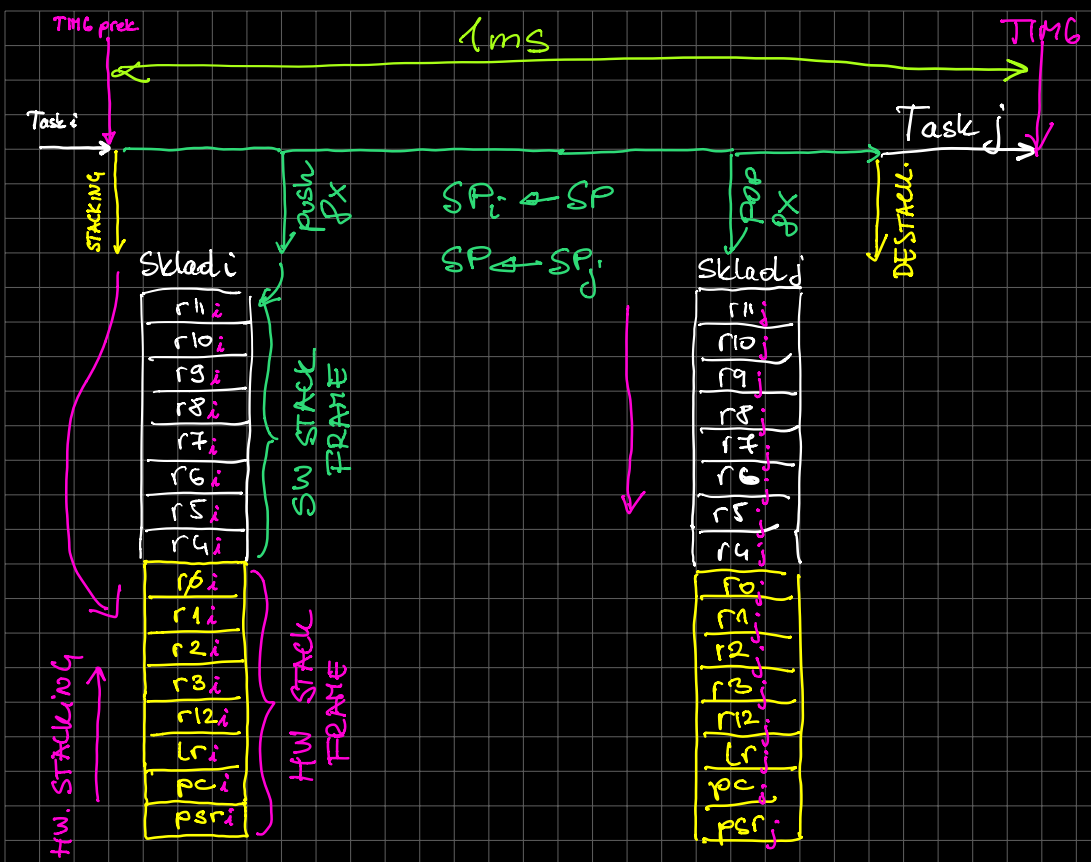
← SP pred prekinitvijo


TIM6_DAC_IRQHandler:
push {r7, lr}
add r7, sp, #0 → r7 ← sp
nop
pop {r7, pc}
push lr
push r7
pop r7
pop pc
to pomeni, da: pc ← FFFFFFF9
povzroči de-stacking in posledično vrnitev iz PS
z operacijo de-stackinga se s sklada pobere ndi PC, kot je bil pred prekinitvijo => s tem se vrnemo v prekinjeni program!
Kaj pa če bi na nek način z de-stackingom operacijo pobrali neko drugo vrednost v PC?
kdo bi to dosegel? => zamenjali vrednost v SP preden zaključimo de-stacking

Razvrščanje opravil (Prellapljanje)
→ Task/Context switching




TIM6 prekinitev
Task i
PSP za TIM6:
Task j
→ shrani SPi
→ SP ← SPj
Stacking:
→ na sklad se porine registre od Task i
Destacking:
→ pobira s sklada na katerega kaže SPj (od Taskj)
SPi
r0
r1
r2
r3
R12
LR
Pc
PSR
stacking
SPi
Sklad od opravila i
SPj
r0
r1
r2
r3
r12
Lr
Pc
psr
destacking
SPj
Sklad od opravila j



Koda in pod. strukture za impl. preklapljanja opravil

① Za vsako opravilo bomo hranili:

kazalec
- → skladovni kazalec
- → naslov funkcije s katero implementiramo opravilo

```
typedef struct {
        void *  sp;
        void *  pTask(void);
} task_entry_t;
```

Če imam N opravil:

```
task_entry_t task_table[N];
```

② Kako bomo pisali funkcije opravil:

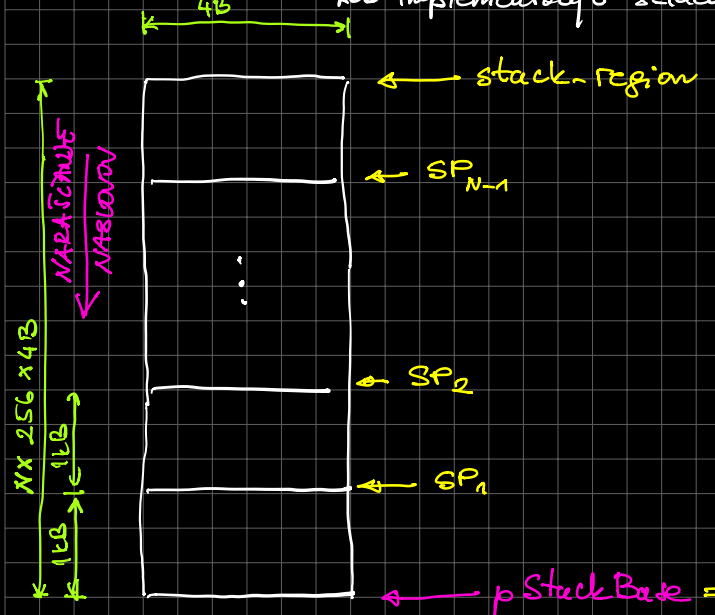
- nikoli ne bodo klicane ⇒ nimajo argumentov
- nikoli se ne smejo zaključit ⇒ nič sam ne vračajo

```
void Task1 (void){
    while (1){
    }
}
```

tu bo naša koda

③ Vsako opravilo ima svoj sklad

→ v pomnilniku moramo rezervirat prostor za implementacijo skladu vsakega opravila



deklaracija:

```
uint32_t stack_region [N * 256]
```

```
pStackBase = stack_region + (N * 256)
```

SP_0 = pStackBase, $SP_1 = SP_0 - 256$

$SP_2 = SP_0 - 2 * 256$

$SP_i = SP_0 - i * 256$

④ TIM6_DAC_Handler:

save_context(): 8x PUSH

switch_context(); { SP_i ← SP ; SP ← SP_j }

restore_context(); 8x POP

⑤ Inicializacija opravil:

→ za vsako opravilo bomo ustvarili SW in HW stack frame na pripravem skladu

r0 – r3 = ∅
r2 = ∅
lr = ∅
pc ← pTask
psR ← 0x21000000

reset vrednost psr-ja