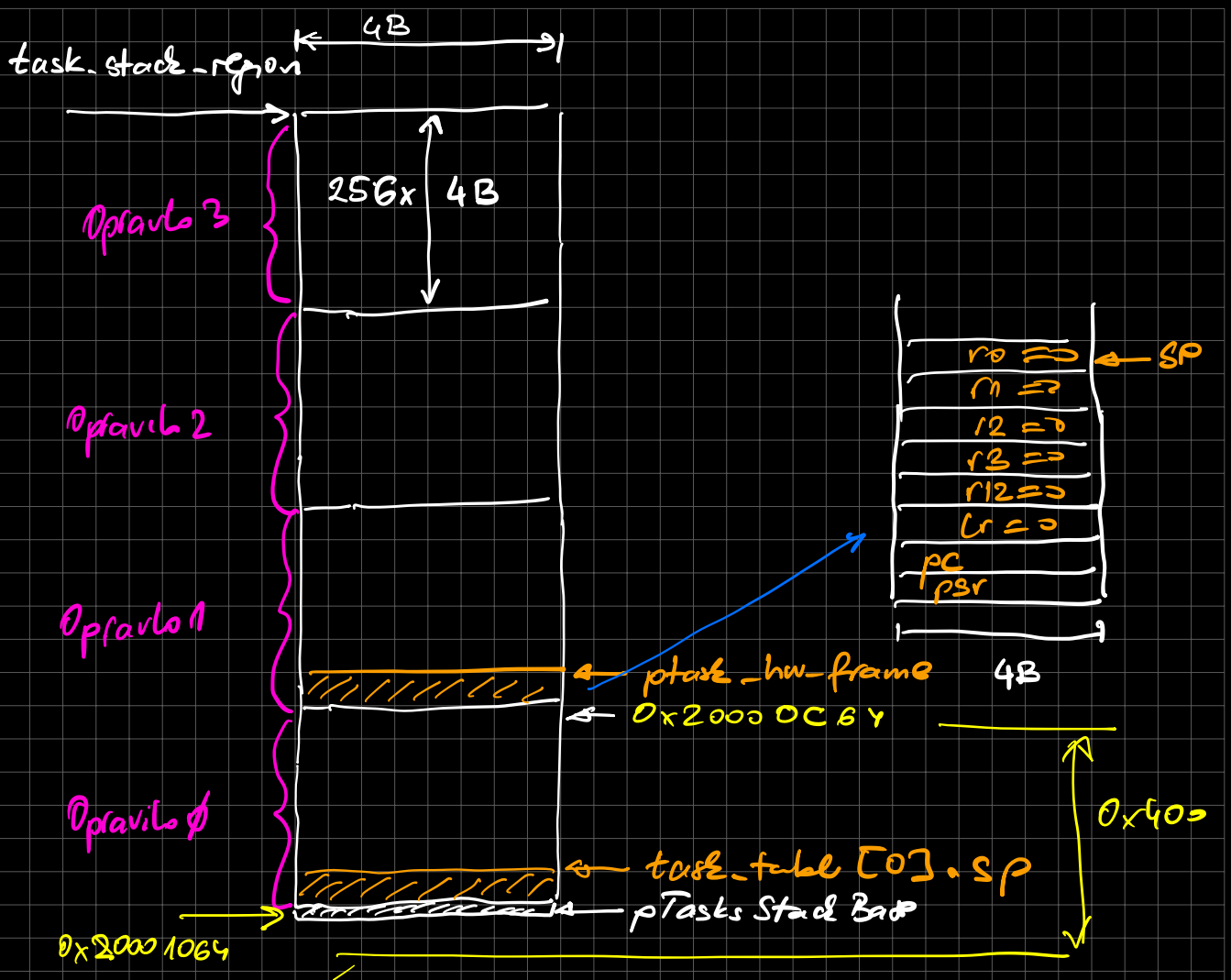


ko se bo zgodil 1. prekinitev po restu:

→ na sklad (kamor kaže SP se topoče MSP od maina

→ LR se vpiše 0xFFFFFFF9, kar se pri shranjevanju na sklad vpodbi MSP

V PSP moramo vpisat vrednost skladovnega kazalca opravila, ki ga želimo prvega zagnati

↙ izbrano opravilo 0

PSP ← task_table[0].sp

⇓ uporabimo assembler

GCC INLINE ASSEMBLER

The GCC Inline Assembly full syntax is:

```
__asm volatile (
    AssemblerTemplate
    : OutputOperands
    [: InputOperands
    [: Clobbers ]])
```

- AssemblerTemplate → ASS. UKAZ, KI GA ŽELIMO UPORABITI
- OutputOperands → izhodne spremenlj.
- InputOperands → vhodne sprem.

- **AssemblerTemplate**: This is a literal string that is the template for the assembler code. It is a combination of fixed text and tokens that refer to the input, output, and goto parameters.
- **OutputOperands**: A comma-separated list of the C variables modified by the instructions in the AssemblerTemplate. An empty list is permitted.
- **InputOperands**: A comma-separated list of C expressions read by the instructions in the AssemblerTemplate. An empty list is permitted.
- **Clobbers**: A comma-separated list of registers or other values changed by the AssemblerTemplate, beyond those listed as outputs. An empty list is permitted.

```
__asm ("ADD %[result], %[input_i], %[input_j]"
  : [result] "=r" (res)
  : [input_i] "r" (i), [input_j] "r" (j)
);
```

[result], [input_i], [input_j] → Simbolna imena, namesto reg

tole pove, da je simbol [result] spremenljivka res

Je en zpet:

```
int var1=10;
int var2;
__asm volatile("MOV %0, %1": "=r"(var2): "r"(var1)); // var2 = var1
```

```
ldr     r3, [r7, #4]    ; load R3 from var1 at R7+4
str     r3, [r7, #0]    ; store R3 to var2 at R7+0
```

If you already looked to some working inline assembler statements written by other authors, you may have noticed a significant difference. In fact, the GCC compiler supports symbolic names since version 3.1. For earlier releases the rotating bit example must be written as

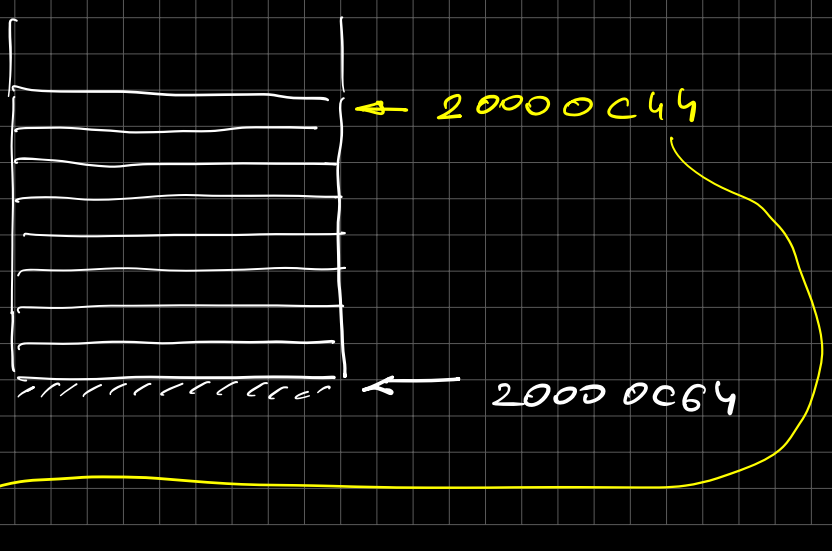
```
asm("mov %0, %1, ror #1" : "=r" (result) : "r" (value));
```

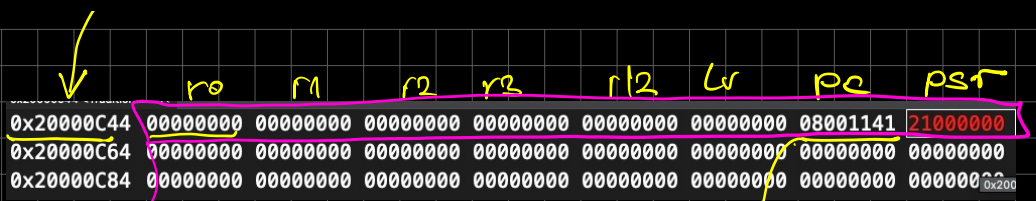
Operands are referenced by a percent sign followed by a single digit, where *%0* refers to the first *%1* to the second operand and so forth. This format is still supported by the latest GCC releases, but quite error-prone and difficult to maintain. Imagine, that you have written a large number of assembler instructions, where operands have to be renumbered manually after inserting a new output operand.

Naša kode za pisanje v PSP:

```
static __attribute__((always_inline)) void write_process_stack_pointer(void* ptr) {
    asm volatile ( "MSR PSP, %0\n\t"
                   :
                   : "r" (ptr) );
}
```

Tok 1:





naslov funkcije Opravilo1

Hardware Stack Frame, ki ga ustvarja funkcija CreateTask()

Sedaj v PSP:

preklicano_opravilo.SP ← PSP

PSP ← naslednje_opravilo.SP

beri PSP v inline assemblyju

piši PSP v inline assemblyju

context_switch();

Ko se prvič požene main:

r0 r1 r2 r3

0x2007ffd0 <Traditional>

```
0x2007FFD0 20001048 FFFFFFF9 00000003 20000468 08001145 20001048 FFFFFFF0
0x2007FFEC 0800139B 080013F4 21000000 0800021D 08001123 ???????? ????????
0x20080008 ???????? ???????? ???????? ???????? ???????? ???????? ????????
```

Hardwar Stack Frame od prekinjene funkcije main.

LR

prva ukaza v PSP:

push LR;
push r3;

ob izstopu iz PSP se ta vrednost zapiše v PC: začne se izvajati z uporabo MSP (0xFFFFFFF9)

mi izklopimo izvajanje z uporabo PSP

na tem mestu izklopimo vrednost FFFFFFFD

$$1064 - 0C64 = 0x0400 = 1024_{10}$$

```
static inline void* read_process_stack_pointer(void) {
    void* result;
    /* read special register PSP into a general-purpose register (picked by compiler)
      and write it to result: */
    asm volatile ( "MRS %0, PSP\n\t"
                   : "=r" (result) );

    return (result);
}
```

Qualifiers

`volatile`

The typical use of extended `asm` statements is to manipulate input values to produce output values. However, your `asm` statements may also produce side effects. If so, you may need to use the `volatile` qualifier to disable certain optimizations. See Volatile.

```
void EXTI0_IRQHandler(void)
{

  HAL_GPIO_EXTI_IRQHandler(USER_BUTTON_PIN);

  if (current_task == -1){
      main_stack_pointer = read_main_stack_pointer();
      *((uint32_t*)main_stack_pointer + 1) = 0xFFFFFFFD;
  }
  switch_context();
}
```

A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change:

1. Memory-mapped peripheral registers

2. Global variables modified by an interrupt service routine

3. Global variables accessed by multiple tasks within a multi-threaded application

We'll talk about each of these cases in the sections that follow.

`"cc"`

The `"cc"` clobber indicates that the assembler code modifies the flags register. On some machines, GCC represents the condition codes as a specific hardware register; `"cc"` serves to name this register. On other machines, condition code handling is different, and specifying `"cc"` has no effect. But it is valid no matter what the target.

`"memory"`

The `"memory"` clobber tells the compiler that the assembly code performs memory reads or writes to items other than those listed in the input and output operands (for example, accessing the memory pointed to by one of the input parameters). To ensure memory contains

```
void switch_context (void) {
    // save current stack pointer to current task in task_table:
    if (current_task != -1) { // -1: the scheduler has interrupted the main() - de
                              // we should never return to main
        task_table[current_task].sp = read_process_stack_pointer();
    }

    // select a new task in a round-robin fashion:
    current_task++;
    if (current_task == MAX_TASKS) {
        current_task = 0;
    }

    // select a new psp:
    write_process_stack_pointer( task_table[current_task].sp );
}
```