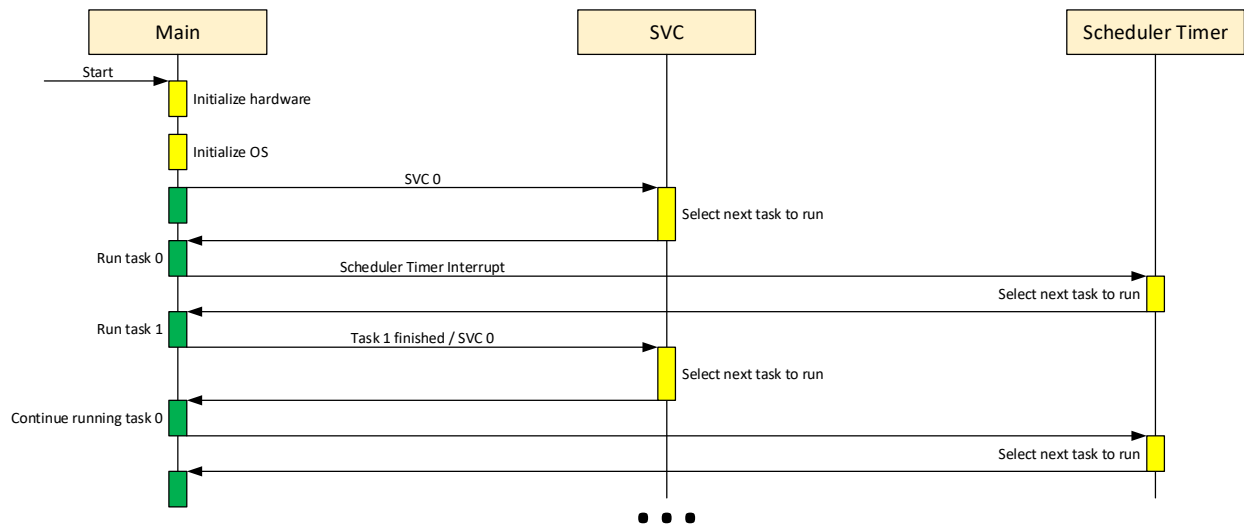


High level design

Design is for single core MCU (at least for now).



Arm implementation details

Arm supports context switching on hardware level. There are 2 features which allow this:

- Supervisor call (SVCall) and deferred supervisor call (PendSV) interrupts
- SP_main and SP_process stacks

SVCall and PendSV allow us to avoid critical sections when doing context switching. If you have the same priority for SVCall, PendSV, and scheduling timer interrupts, then all system calls do not need any synchronization.

OS will have to issue SVCall:

- to start running task
- after task finished execution
- when we want to switch tasks on regular basis using scheduler timer

To understand how to achieve this we need to talk about ARM interrupts, stacks and registers.

Registers

32-bit ARM processors have 16 registers:

R0		General / C function arguments / result values	Caller saved
R1		General / C function arguments / result values	Caller saved
R2		General / C function arguments / result values	Caller saved
R3		General / C function arguments / result values	Caller saved
R4		General	Callee saved
R5		General	Callee saved
R6		General	Callee saved
R7		General	Callee saved
R8		General	Callee saved
R9		General	Callee saved
R10		General	Callee saved
R11		General	Callee saved
R12		General	Caller saved
R13	SP	Stack pointer	Callee saved
R14	LR	Link register (return address) / General	Caller saved
R15	PC	Program counter	N/A

From context point of view there are 2 important groups of registers:

- caller saved – if you are going to call a function and you use any of the “caller saved” registers it is your responsibility to save them. If you do not the function you are calling is free to change their values.
- callee saved – before modifying these registers in your function you have to store their original values and restore them before you return from your function. Failure to do so can result in an undefined behavior.

Stacks

ARM has 2 stacks MSP (SP_main) and PSP (SP_process). By default, when processor resets it uses MSP stack. When interrupt is generated interrupt handler code will also use MSP stack (even if code which was running before interrupt was using PSP stack).

You can choose which stack your code is using by changing CONTROL.SPSEL bit. Here is a sample code which sets SPSEL to 1.

```
# load CONTROL register to R3
MRS      R3, CONTROL
# set bit 1 (SPSEL = 1, this will make sure we use process stack instead of main stack)
ORR      R3, 0x02
# write R3 register value into CONTROL register
MSR      CONTROL, R3
```

To change PSP or MSP stack address you can simply modify those registers. Here is a sample code which sets PSP address to address store in R0 register

```
# write R0 which has process stack start address to SP (stack register)
MSR      PSP, R0
```

If you want to change current stack address you can simply modify SP register as you do with any other register. Here is a sample code which allocates 12 bytes on a current stack

```
# allocate 12 bytes on current stack
SUB      SP, 12
```

Interrupts

When interrupt is generated ARM stores all caller saved registers on a stack which was active in the code which was running prior to interrupt. After that ARM sets MSP as a main stack. So, by default, all interrupt handlers will execute with MSP as their stack regardless of which stack was used prior interrupt request.

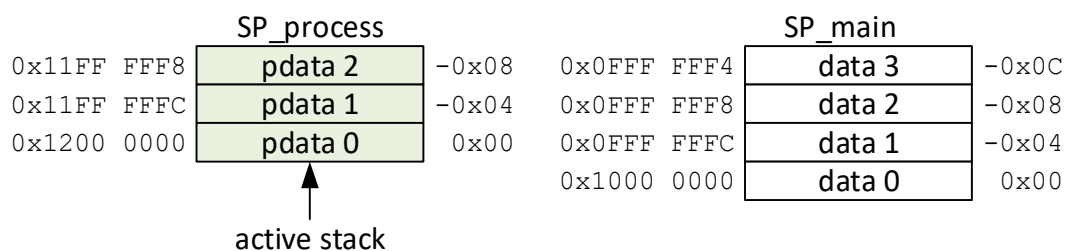


Figure 1 Main and Process stacks before interrupt

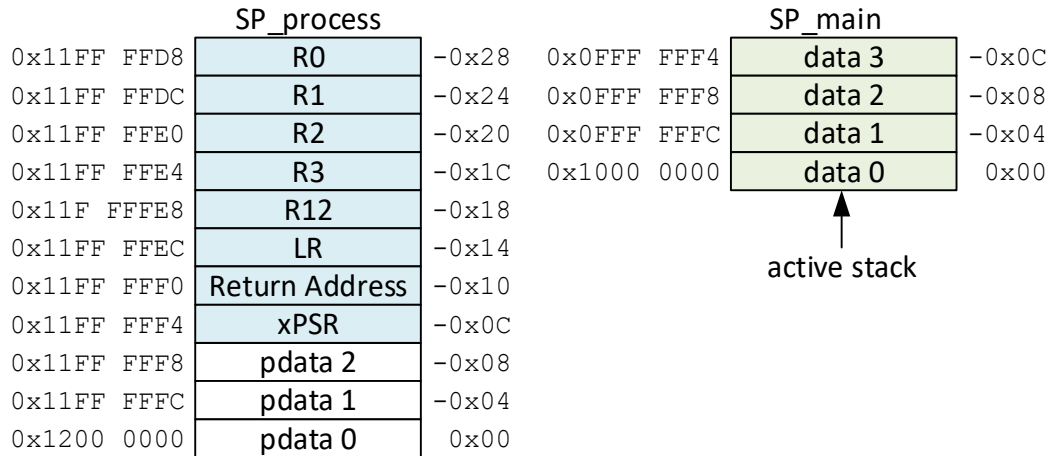


Figure 2 Main and Process stacks after interrupt occurred but prior entering interrupt handler

If code which was executing prior exception had stack which wasn't 8-byte aligned interrupt handler will insert dummy value before storing any registers on the stack.

Context Switching

Given all the information about registers, stacks, and interrupts we can now describe how context switching works.

When we generate SVCcall 0 at the start of OS or when current task finished executing or when scheduler timer interrupt handler is called, we have to do the following:

- 1) select next task which will be running (if no task found we can run dummy idle task)
- 2) If there was a task running before we selected next task, we need to save old task's "callee saved" registers (R4, R5, R6, R7, R8, R9, R10, R11, PSP).
- 3) next task has never executed before

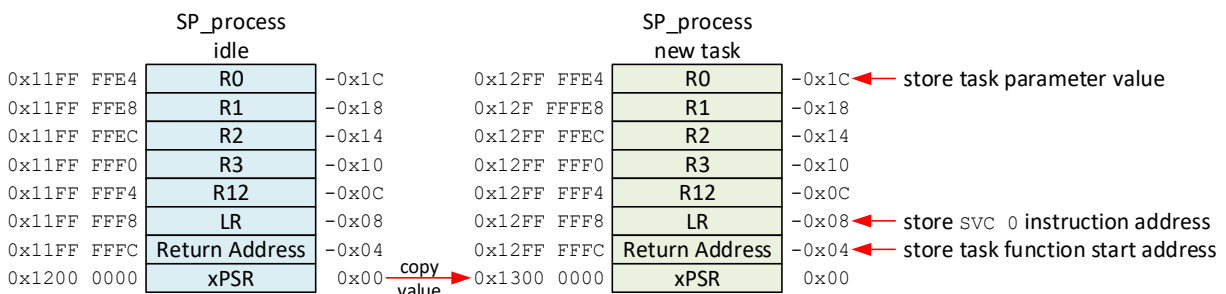


Figure 3 Fixing process stack for the new task (in this case old stack is from idle task but it will be similar if other task was executing before)

- a. allocate space on task specific process stack for caller saved registers (8 x 4 bytes = 32 bytes)
- b. set PSP to point to new task specific process stack (in Figure 3 it's 0x12FF FFE4, processor will unwind stack and restore respective registers during interrupt exit procedure).
- c. copy xPSR register value from idle task stack

- d. save task parameter to the top of the stack (this is where register R0 is stored)
 - e. save address of `SVC 0` instruction on the stack where LR register is located (in our case `0x12FF FFF8`). This will ensure that after task function completes it will request OS to schedule next task
 - f. save task function start address on the stack where Return Address is located (in our case `0x12FF FFFC`). This will ensure that task function will start executing immediately after we exit from interrupt handler
- 4) next task was previously suspended
- a. restore next task's "callee saved registers (R4, R5, R6, R7, R8, R9, R10, R11, PSP)
- 5) return from interrupt handler