RTOS from scratch

Authored by

Hazem Anwer
Sara Mounier
Samir Rizk
Islam Ahmed

Contents

Preface

This report deep-dives into the implementation of a minimal pre-emptive RTOS kernel, from scratch. The implementation is based on the ARM Cortex-M4 processor, using a GNU C toolchain, and utilizes all OS-specific features built into the hardware. These include,

- The `PendSV` exception handler, usually employed for context switching.
- The `SVC` (i.e, Supervisor Call) assembly instruction, and its associated exception handler.
- The `SysTick` timer peripheral, and its associated exception handler.
- Support for privileged and unprivileged operation modes.
- Support for two stack pointers, `MSP` (main) and `PSP` (process), shadowed behind `SP`, utilized by the OS and the application, respectively.

As prerequisites to this text, familiarty with the C programming language and Real-Time Operating System Concepts (RTOS) is a must, and previous exposure to the ARM architecture is recommended.

Also, check out the author's implementation of a similarly minimalistic pre-emptive RTOS kernel, with support for many OS resources (e.g: mutexes, semaphores and queues).

> `https://github.com/hazemanwer2000/BabyRTOS`

Chapter 1

Context Switching

This chapter elaborates on how context switching may be implemented, utilizing built-in hardware features such as the `PendSV` exception, and the `PSP` stack pointer.

1.1 The `PendSV` exception

According to the Application Binary Interface (`ABI`) of the Cortex-M4 architecture, a function must save any values residing in the `R0-R3, R12, LR` range of registers, called *caller-saved registers*, onto the stack, before making a function call, if any of the values is required unmodified after the function call. The function restores the values into their respective registers, after the function call. `LR` is always saved, however, since it contains the return address a function shall return to upon termination. After saving `LR`, the return address is loaded into the register before branching. At the end of every function call, the following assembly instruction must be present,

```
    /* Branch to address contained in a register. */
__asm("BX LR");
```

When handling an interrupt, the Cortex-M4 hardware saves all caller-saved registers onto the stack, automatically, in addition to the status register, `xPSR`, and the return address (i.e, `PC`). A special address, reserved in the Cortex-M4 memory map, is placed in `LR` to indicate to the hardware when a return from interrupt is due, for it to perform unstacking of all saved registers automatically.

To perform context switching, the interrupt stacking and unstacking

hardware mechanism is hijacked. The PendSV handler is triggered, and the hardware automatically performs stacking. Within the handler, the remaining processor registers, called *callee-saved registers*, are pushed onto the stack. Having saved the full context of the processor, prior to the interrupt, the stack pointer of the previous task is saved, and the stack pointer of the next task is loaded. Callee-saved registers are popped from the stack of the next task and upon return from the handler, the hardware restores all saved registers, successfully restoring the full context of the next task.

```
    /* Save registers (Store Multiple Instruction). */
__asm("STMIA R0, {R4-R11}");

    /* Switch stack pointer. */
// ...

    /* Restore registers (Load Multiple Instruction). */
__asm("LDMIA R0, {R4-R11}");
```

The writing above implies that the PendSV handler must be written in assembly. In GNU C, use the naked compiler directive to tell the compiler not to inject any assembly instructions in the function. Accordingly, a naked function should be composed of assembly directives only. It is unsafe to write C code in naked functions.

```
void __attribute__ ((naked))
PendSV_Handler(void) {
    // ...
}
```

Initially, when restoring context of the stack of a task, there must be a set of *counterfeit* register values, placed there explicitly, with the PC value being the address of the task handler.

Note that if interrupt pre-emption is enabled, the PendSV exception should never pre-empt another interrupt, to perform context switching successfully. Hence, it must be the least priority interrupt in the system.

1.2 MSP versus PSP

In many processors, there is a single stack pointer register. If an interrupt occurs, the stacking of caller-saved registers and operation within the interrupt handler itself all employ the stack of the task. This implies,

that all tasks must accomodate extra space for the maximum depth an interrupt may require. If interrupt pre-emption is supported and enabled, more space is required. This, obviously, wastes much memory.

The Cortex-M4 provides two stack pointers. `MSP` is always used in *handler mode* (i.e, when handling interrupts), while `MSP` or `PSP` may be used in *thread mode*. Before triggering `PendSV` for the first time, `PSP` is switched to, by setting a specific bit in the `CONTROL` special register,

```
1     /* MRS and MSR are used to read from, and write to */
2     /*      special registers, such as CONTROL.        */
3
4 __asm("MRS R0, CONTROL");
5 __asm("ORR R0, R0, #0x2");
6 __asm("MSR CONTROL, R0");
```

Thereafter, when interrupting an executing task, hardware stacking and unstacking employs the `PSP`. Within the handler itself, and any pre-empting interrupt, `MSP` is used. This saves memory, because only a single stack, the stack the OS employs, needs to handle the extra space required. Additionally, a level of security is added, by making it difficult for a task to sniff the value of a previously local variable in a handler, on the stack of the OS, upon return from, for example, a system call.

Switching to `PSP` must occur in thread mode, before triggering `PendSV`. The specific bit in the `CONTROL` special register may not be modified in handler mode. If it was permissible, then an alteration to the `LR` register before returning from `PendSV` would be due, since the hardware sets a specific bit in `LR` before interrupt entry, to indicate whether it should return (i.e, unstack saved registers) using `MSP` or `PSP`.

Usually, the initial value of `PSP` is that of the stack of the *idle task* of the OS.

1.3 Special Considerations

The Cortex-M4 architecture requires that the stack pointer is 4-byte aligned at any time. Additionally, there is a memory-mapped register that determines whether `SP` should be 8-byte aligned at interrupt handler entry and exit. This is usually enabled by default by most vendors, due to compatibility with the `ABI` that most toolchains conform to.

If when handling an interrupt the hardware determines that the stack

pointer is not 8-byte aligned, a padding word is appended to the stack, and a specific bit is set in the saved `xPSR` register, to be used when unstacking to determine whether padding occured. Accordingly, ensure that all saved task stack pointers are initially 8-byte aligned, and the counterfeit `xPSR` value matches that requirement. Additionally, before switching to `PSP`, ensure `MSP` is 8-byte aligned,

```
    /* Ensure MSP is double-word aligned. */

__asm("MRS R0, MSP");
__asm("AND R0, R0, #0xFFFFFFFC");
__asm("MSR MSP, R0");
```

Chapter 2

System Calls

This chapter elaborates on how system calls may be implemented, utilizing built-in hardware features such as unprivileged mode, and the `SVC` assembly instruction and its associated exception handler.

2.1 Unprivileged mode and `SVC` calls

The Cortex-M4 has privileged and unprivileged operations modes in thread mode. Usually, for security purposes, tasks run in unprivileged mode. For example, a task in unprivileged mode may not switch to using `MSP`. Additionally, in unprivileged mode, the `PendSV` exception may not be triggered. Hence, context switching itself is a privileged operation, which is logical, and may be performed through the OS, using OS requests, usually called *system calls*.

To request a privileged operation from the OS, a special assembly instruction must be executed, called `SVC`, which triggers the execution of the `SVC` exception handler. A special number may be passed along with the instruction, to indicate the type of system call to be executed. Alternatively, a pointer to arguments may be passed in `R0`, as a first argument to the handler.

```
1 OS_REQ_status_t OS_wait(OS_task *task) {
2     OS_REQ_wait_t req = {
3         .base.id = OS_REQ_id_WAIT,
4         .task = task
5     };
6 
7         /* Load 'req' address into R0 */
8         /*      then make 'SVC' call   */
```

```
9       __asm("MOV R0, SP");
10      __asm("SVC #0");
11
12          /* Memory-Barrier Synchronization Instructions. */
13          /*     to force the execution of SVC before any */
14          /*     subsequent instruction. */
15      __asm("DSB");
16      __asm("ISB")
17
18      return req.base.status;
19  }
```

Note that the SVC exception may not be pending, otherwise a *hard fault* exception is triggered. This means that SVC should usually be the highest priority interrupt in the system.

To gurantee that the system call is atomic, several actions might be taken. The easiest choice is to disable interrupt pre-emption system-wide. A harder action would be to only disable *OS-aware interrupts*, those interrupts that make calls to the OS, and re-enable them at interrupt exit.

Once the atomicity of all system calls is guranteed, it is easy to envision an implementation of all common system calls and resources, some of which are shown in Table 2.1.

System Call	Operation
OS_wait	Make OS_task wait.
OS_ready	Make OS_task ready.
OS_delay	Delay an OS_task.
OS_lock	Lock an OS_mutex.
OS_unlock	Unlock an OS_mutex.
OS_give	Increase an OS_semaphore.
OS_take	Decrease an OS_semaphore.
OS_enqueue	Enqueue to an OS_queue.
OS_dequeue	Dequeue from an OS_queue.
...	...

Table 2.1: Common system calls.

2.2 The `SysTick` timer

All Cortex-M processors contain a hardware timer peripheral, called the `SysTick` timer. It is usually used by an OS for time-keeping, and to implement various timeout mechanisms (e.g: the `OS_delay` system call). The benefit of having a timer peripheral, memory-mapped to the same region for a family of processors, is easy porting of the OS to multiple processors, since the dependency is on the architecture, as opposed to using a vendor-specific hardware timer peripheral, which would imply dependency on a family of micro-controllers instead (i.e, lower portability).

References

[1] Yiu, J. (2014) The Definitive Guide to ARM Cortex-M3 and Cortex-M4 processors. Oxford, UK.