

Mikroprocesory

9. How to build own RTOS

Stanislav Vítek

Katedra radioelektroniky

České vysoké učení technické v Praze

What is a pre-emptive scheduler?

- Assume that we intend to toggle two LEDs at 1s and 2s intervals respectively.
- Below is a bare-metal approach(without timers) of doing it:

```
int main() {
    while(1) {
        LED1_TURN_ON(); LED2_TURN_ON();
        delay_seconds(1);
        LED1_TURN_OFF();
        delay_seconds(1);
        LED1_TURN_ON(); LED2_TURN_OFF();
        delay_seconds(1);
        LED1_TURN_OFF();
        delay_seconds(1);
    }
    return 0;
}
```

Viability of the solution

- A decision about LED states needs to be taken at an interval of the highest common factor of the delays(in the above example it is 1 second).
- It is cumbersome to design with this approach if the number of LEDs is large.
- Also, adding a newer LED(with a different blink rate) needs considerable re-work of the older code.

Hence, this approach is not scalable.

-
- This solution could have been much simpler if there was a possibility of having multiple "main()" functions with each main function dedicated to a LED.

Function as a task

```
/// Main 1
void task1()
{
    LED1_TURN_ON();
    delay_seconds(1);
    LED_1_TURN_OFF();
    delay_seconds(1);
}

/// Main 2
void task2()
{
    LED2_TURN_ON();
    delay_seconds(2);
    LED_2_TURN_OFF();
    delay_seconds(2);
}
```

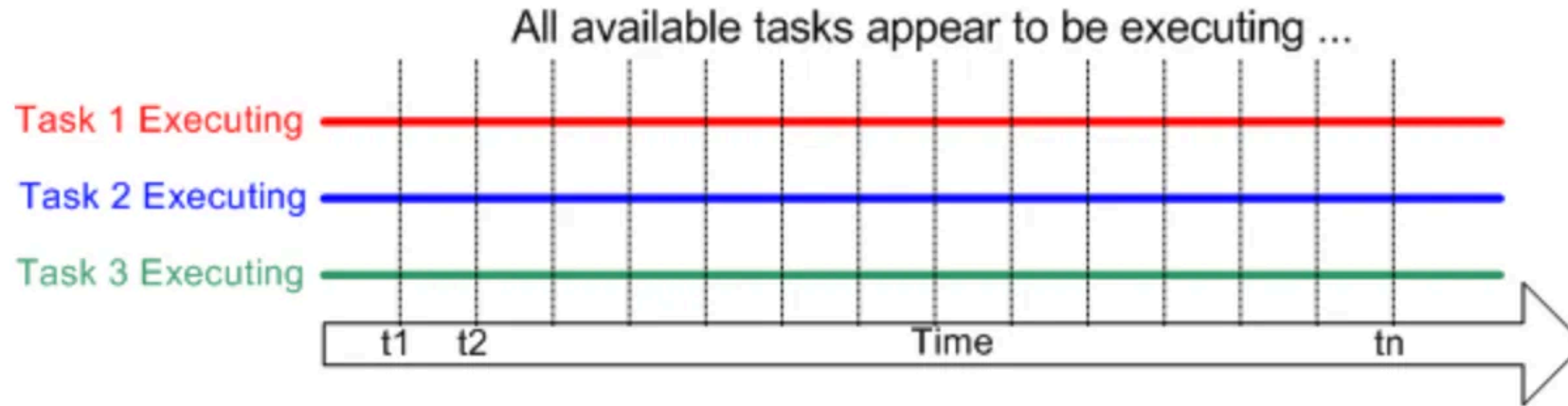
Pre-emptive scheduler

To add a new LED we add a new task. This opens up the possibility for multi-tasking. The use of a pre-emptive scheduler can simplify the design of what would otherwise be a complex software application:

- Multitasking allows the complex application to be partitioned into a set of smaller and more manageable tasks.
- The partitioning can result in easier software testing, work breakdown within teams, and code reuse.
- Complex timing and sequencing details can be removed from the application code and become the responsibility of the operating system.

A pre-emptive scheduler forms the heart of an RTOS.

Illusion of Multitasking



Intuition for building a pre-emptive scheduler

- As a task executes it utilizes the processor/microcontroller registers and accesses RAM and ROM just as any other program.
 - These resources together (the processor registers, stack, etc.) comprise the task execution context.
- A task is a sequential piece of code
 - it does not know when it is going to get suspended or resumed by the scheduler and does not even know when this has happened.
- While the task is suspended other tasks will execute and may modify the processor register values.
- Upon resumption the task will not know that the processor registers have been altered
 - if it used the modified values the summation would result in an incorrect value.

Context switching

- To prevent this type of error it is essential that upon resumption a task has a context identical to that immediately prior to its suspension.
- The scheduler is responsible for ensuring this is the case — and does so by saving the context of a task before it is taken out of execution.
- When the task is taken for execution, its saved context is restored by the scheduler prior to its execution.
- The process of saving the context of a task being suspended and restoring the context of a task being resumed is called context switching.

Context switching strategy

- A task context comprises of the data it has stored in the stack and the CPU registers(including the program counter) values.
- The strategy is to maintain a separate stack for each task and save all CPU registers onto the stack before taking the next task for execution.
- This way the only parameter that needs to be remembered to restore a task back to execution is its stack pointer.
- We will use a periodic interrupt to save the current task context and restore a suspended task context.

Context switching details

1. Create a stack for each task (equivalent to reserving some memory).
2. Initialize the CPU to execute the first task.
 - Set the CPU to use the first task's stack.
 - Though obvious, the only information that CPU has about stack is the stack pointer(SP).
 - Hence, this operation can be accomplished by copying the stack address of the first task to SP.
3. Initialize a periodic interrupt.

Periodic ISR

1. Save/Push the CPU registers onto the stack. Note that the SP is currently pointing to the task stack that is currently in execution. Let us call this task X.
2. Change the SP value to point to the stack of the new task (call this Y) to be executed.
3. Restore/Pop the CPU register values stored in the Y's stack to the actual CPU registers.
4. Exit out of interrupt. Upon exit, the control will go to the Y task because all CPU registers have been loaded with the Y's context (also remember the task context contains the program counter).

ARM Cortex-M implementation

Cortex-M Operation Modes

- When a Cortex-M based MCU is running from an exception handler such as an Interrupt Service Routine (ISR), it is known as running in Handler Mode. The rest of the time the MCU runs in Thread Mode.
- The core can operate at either a privileged or unprivileged level. Certain instructions and operations are only allowed when the software is executing as privileged. For example, unprivileged code may not access NVIC registers. In Handler Mode, the core is always privileged. In Thread Mode, the software can execute at either level.
- Switching Thread Mode from the unprivileged to privileged level can only happen when running from Handler Mode.

Registers

- r0 - r3 - argument registers
- r4 - r8 - variable registers
- r9 - platform register
- r10 - variable register
- r11 - variable register
- r12 - intra-procedure scratch register
- r13 - stack pointer
- r14 - link register
- r15 - program counter

r12 - Intra-Procedure-call Scratch Register

- The address space for ARM Cortex-M devices is 32 bits.
- However, it's not possible for a branch and link (bl) instruction to jump across the entire address region (because some bits encode the instruction itself).
- In this situation, a jump to a function that is far away in the address space may require passing through a shim function generated by the linker known as a veneer.
- r12 is the only register that may be used within the veneer without needing to preserve the original state.

<https://developer.arm.com/documentation/dui0803/c/Image-Structure-and-Generation/Linker-generated-veneers/What-is-a-veneer->

r9 as Platform Register

- One application is to use r9 as a static base (SB).
- Normally when code is compiled, the code is dependent on the position it runs from. That is, functions are linked together based on the fact that the code and data will always be located at a specific location.
- However, for some applications you may want the ability to run code from arbitrary locations.

- For example, maybe you want to load a function from flash into RAM for faster execution.
- In these situations you will need to generate Position Independent Code (PIC). When executing PIC, the address of global & static data needs to be looked up.
- These addresses are stored in a table known as the Global Offset Table. The base of this table can be stored in r9 and then functions will reference this register to look it up. For example, this behavior will be triggered for ARM Cortex-M devices when compiling with the `-fpic` and `-msingle-pic-base` compiler options.

-
- Another application is to use r9 as the thread register (TR). In this situation, the register holds a pointer to the current thread-local storage context

Special registers

- IPSR - The Interrupt status register.
- EPSR - The execution status register. This is RAZ/WI.
- TEPSR - A composite of IPSR and EPSR.
- MSP - The Main stack pointer.
- PSP - The Process stack pointer.
- PRIMASK - Register to mask out configurable exceptions.
- BASEPRI - The base priority register.
- FAULTMASK - Register to raise priority to the HardFault level.
- CONTROL -The special-purpose control register.

Privilege needed to work with special registers

Read

- If unprivileged code attempts to read any stack pointer, the priority masks, or the IPSR, the read returns zero.

Write

- The processor ignores writes from unprivileged Thread mode to any stack pointer, the EPSR, the IPSR, the masks, or CONTROL.
- If privileged Thread mode software writes a 1 to the CONTROL.nPRIV bit, the processor switches to unprivileged Thread mode execution, and ignores any further writes to special-purpose registers.
- After any Thread mode transition from privileged to unprivileged execution, software must issue an ISB instruction to ensure instruction fetch correctness.

Stack Pointers

The Cortex-M architecture implements two stacks known as the Main Stack (tracked in the `msp` register) and the Process Stack (tracked in the `psp` register).

- On reset, the MSP is always active and its initial value is derived from the first word in the vector table. When a stack pointer is “active”, its current value will be returned when the `sp` register is accessed.

In Handler Mode, the `msp` is always the stack which is used. In Thread Mode, the stack pointer which is used can be controlled in two ways:

- Writes of 1 to the `SPSEL` bit in the `CONTROL` register will switch from using the `msp` to the `psp`
- The value placed in `EXC_RETURN` upon exception return

Task Control Block

- A task control block (TCB) in an RTOS is used to store parameters related to a task:
 - stack pointer,
 - stack size,
 - inter-process parameters, etc.
- We keep it simple by only storing the stack pointer of the task in the TCB.
- Also, the TCB is defined as a linked list to enable easy hopping to the next task.
- We also define a pointer to the TCB that points to the currently active task.

```
// For simplicity we will assume only two tasks
#define NUM_OF_THREADS 2

// Task control block, implemented as a linked list to point to the
// TCB of the next task.
struct tcb{
    int32_t    *stackPt;
    struct tcb *nextPt;
};

// Define tcb_t datatype
typedef struct tcb tcb_t;

// Define an array to store TCB's for the tasks.
tcb_t tcbs[NUM_OF_THREADS];

// Points to the TCB of the currently active task
tcb_t *pCurntTcb;
```

Stack reservation

- Reserve stack for each task. For simplicity, we assume only two tasks.

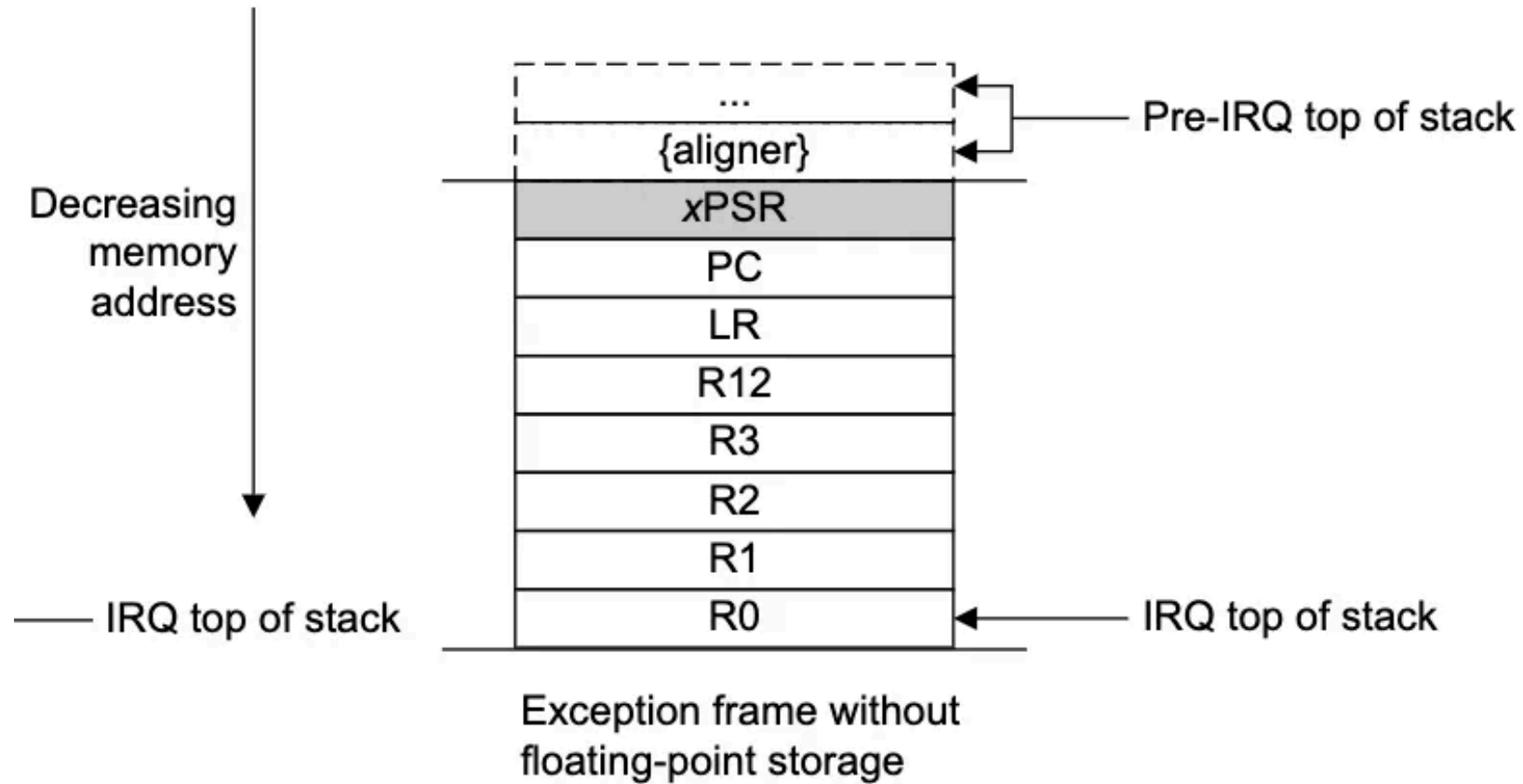
```
#define STACKSIZE      100

// Define stack for each task. Note that the processor expects the stacks
// to be ended on word boundaries.
int32_t TCB_STACK[NUM_OF_THREADS][STACKSIZE];
```

- The task context comprises of CPU registers R0-R12, SP, LR, PC, and PSR (program status register).
- They need to be saved and restored when the task is taken into or out of execution.

Swapping tasks with SysTick

- We intend to use the ARM Cortex M's systick interrupt as the periodic interrupt for swapping tasks.
- The processor is designed to save and restore some CPU registers as part of the interrupt entry and exit mechanism.
 - The processor pushes 8 registers PSR, PC, LR, R12, R3, R2, R1, and R0 onto the stack on an exception.
 - Then the exception routine is executed.
 - On exit from the exception, the processor pops 8 words (the same ones those where pushed) from the stack and loads them onto the respective CPU registers (in the same order as they where saved).



SysTick ISR

- We have to only save and restore the rest of the registers (R4, R5, R6, R7, R8, R9, R10 & R11) within the interrupt.
- Since SP is directly stored in the TCB, we don't have to push it to the stack.

```
__attribute__((naked)) void SysTick_Handler(void)
{
    /*
     STEP 1 – SAVE THE CURRENT TASK CONTEXT

     At this point the processor has already pushed
     PSR, PC, LR, R12, R3, R2, R1 and R0 onto the stack.
     We need to push the rest(i.e R4, R5, R6, R7, R8,
     R9, R10 & R11) to save the context of the current task.
    */

    // Disable interrupts
    __asm("CPSID    I");
}
```

```
// Push registers R4 to R7
__asm("PUSH    {R4-R7}");

// Push registers R8-R11
__asm("MOV     R4, R8");
__asm("MOV     R5, R9");
__asm("MOV     R6, R10");
__asm("MOV     R7, R11");
__asm("PUSH    {R4-R7}");

// Load R0 with the address of pCurntTcb
__asm("LDR     R0, =pCurntTcb");

// Load R1 with the content of pCurntTcb
// (i.e post this, R1 will contain the address of current TCB).
__asm("LDR     R1, [R0]");

// Move the SP value to R4
__asm("MOV     R4, SP");
```

```
// Store the value of the stack pointer (copied in R4)
// to the current tasks "stackPt" element in its TCB.
// This marks an end to saving the context of the current task.
__asm("STR    R4, [R1]");

// STEP 2: LOAD THE NEW TASK CONTEXT FROM ITS STACK
// TO THE CPU REGISTERS, UPDATE pCurntTcb.

/// Load the address of the next task TCB onto the R1.
__asm("LDR    R1, [R1,#4]");

/// Load the contents of the next tasks stack pointer to pCurntTcb,
// equivalent to pointing pCurntTcb to the newer tasks TCB.
// Remember R1 contains the address of pCurntTcb.
__asm("STR    R1, [R0]");
```

```
// Load the newer tasks TCB to the SP using R4.
```

```
__asm("LDR    R4, [R1]");
```

```
__asm("MOV    SP, R4");
```

```
// Pop registers R8-R11
```

```
__asm("POP    {R4-R7}");
```

```
__asm("MOV    R8, R4");
```

```
__asm("MOV    R9, R5");
```

```
__asm("MOV    R10, R6");
```

```
__asm("MOV    R11, R7");
```

```
// Pop registers R4-R7
```

```
__asm("POP    {R4-R7}");
```

```
__asm("CPSIE  I ");
```

```
__asm("BX    LR");
```

```
}
```

GCC attributes explained

- Adding `attribute((naked))` suppresses the generation of assembly code that allocates/deallocates space for auto variables in the program stack during the entry/exit of the function.
- We add this attribute to `SysTick_Handler` to ensure that the stack is not altered between the occurrence of the interrupt and execution of the interrupt routine instructions.

SysTick_Handler

SysTick_Handler assumes that there is always a task context stored in the suspended task's stack. Hence, we need to initialize the task stacks such that they contain a task context even during the first task context switch.

We need to do the following to initialize the task stack:

1. Set the tasks stack pointer such that it points to the top of the task context (equivalent to have 16 words stored in the task stack).
2. Copy 0x01000000 to xPSR in the stacked context. This makes sure that the processor knows that it needs to run on thumb mode when the value is copied from to xPSR register.
3. Copy the address of the function that implements the task to the stacked PC.

```

void OsInitThreadStack()
{
    /// Enter critical section
    /// Disable interrupts
    __asm("CPSID   I");
    /// Make the TCB linked list circular
    tcbs[0].nextPt = &tcbs[1];
    tcbs[1].nextPt = &tcbs[0];

    /// Setup stack for task0

    /// Setup the stack such that it is holding one task context.
    /// Remember it is a descending stack and a context consists of 16 registers.
    tcbs[0].stackPt = &TCB_STACK[0][STACKSIZE-16];
    /// Set the 'T' bit in stacked xPSR to '1' to notify processor
    /// on exception return about the thumb state. V6-m and V7-m cores
    /// can only support thumb state hence this should be always set
    /// to '1'.
    TCB_STACK[0][STACKSIZE-1] = 0x01000000;
    /// Set the stacked PC to point to the task
    TCB_STACK[0][STACKSIZE-2] = (int32_t)(Task0);
}

```

```

// Setup stack for task1

// Setup the stack such that it is holding one task context.
// It is a descending stack and a context consists of 16 registers.
tcbs[1].stackPt = &TCB_STACK[1][STACKSIZE-16];

// Set the 'T' bit in stacked xPSR to '1' to notify processor
// on exception return about the thumb state. V6-m and V7-m cores
// can only support thumb state hence this should be always set to '1'.
TCB_STACK[1][STACKSIZE-1] = 0x01000000;

// Set the stacked PC to point to the task
TCB_STACK[1][STACKSIZE-2] = (int32_t)(Task1);

// Make current tcb pointer point to task0
pCurntTcb = &tcbs[0];

// Enable interrupts
__asm("CPSIE    I ");
}

```

Start of scheduler

- After executing `OsInitThreadStack`, `pCurntTcb` will be pointing to TCB of task0.
- `LaunchScheduler` starts the scheduler by restoring the context saved in TCB pointed by `pCurntTcb` to the CPU registers.

```
__attribute__((naked)) void LaunchScheduler(void)
{
    // R0 contains the address of currentPt
    __asm("LDR    R0, =pCurntTcb");

    // R2 contains the address in currentPt(value of currentPt)
    __asm("LDR    R2, [R0]");

    // Load the SP reg with the stacked SP value
    __asm("LDR    R4, [R2]");
    __asm("MOV    SP, R4");
}
```

```
// Pop registers R8–R11(user saved context)
__asm("POP      {R4–R7}");
__asm("MOV      R8, R4");
__asm("MOV      R9, R5");
__asm("MOV      R10, R6");
__asm("MOV      R11, R7");

// Pop registers R4–R7(user saved context)
__asm("POP      {R4–R7}");

// Start popping the stacked exception frame.
__asm("POP      {R0–R3}");
__asm("POP      {R4}");
__asm("MOV      R12, R4");

// Skip the saved LR
__asm("ADD      SP, SP, #4");
```

```
// POP the saved PC into LR via R4,  
// We do this to jump into the  
// first task when we execute  
// the branch instruction to exit this routine.  
__asm("POP      {R4}");  
__asm("MOV      LR, R4");  
__asm("ADD      SP, SP, #4");  
  
// Enable interrupts  
__asm("CPSIE   I ");  
__asm("BX      LR");  
}
```

Full project: https://github.com/dheeptuck/rtos_groundup

Task creation

```
volatile void Task0()  
{  
    while(1)  
    {  
        // Toggle @ 50 ms  
        portable_delay_cycles(100000);  
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_2);  
    }  
}  
  
volatile void Task1()  
{  
    while(1)  
    {  
        // Toggle @ 100 ms  
        portable_delay_cycles(200000);  
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_3);  
    }  
}
```