PART 0 – A SIMPLE PREEMPTIVE-COOPERATIVE ROUND-ROBIN SCHEDULER

Outline

This part provides a more hands-on approach, so the reader can quickly get a minimal cooperative-preemptive kernel up and running - which for those who never made it, the satisfaction can be considerable and will foster the interest on the upcoming topics.

It does not refrain from addressing important concepts though, such as making a clear distinction between process and task/thread, explaining what an execution image is and how it relates to a process, highlight some important architectural features of the ARMv7m family, and details the mechanism of context-switching to achieve multitasking.

0.0. INTRODUCTION

Not long ago, small embedded system software needed to be very economical. Reuse, scalability, maintainability was not a concern for embedded software engineers – making it to fit in the small computer was. Low-end system would normally have the application highly coupled to the hardware. Super-loops polling devices or being interrupted by them, fully interrupt-driven systems, and also time-triggered run-to-completion schedulers are common execution models on these systems. These systems are sometimes referred as *pseudo-kernels [7].*

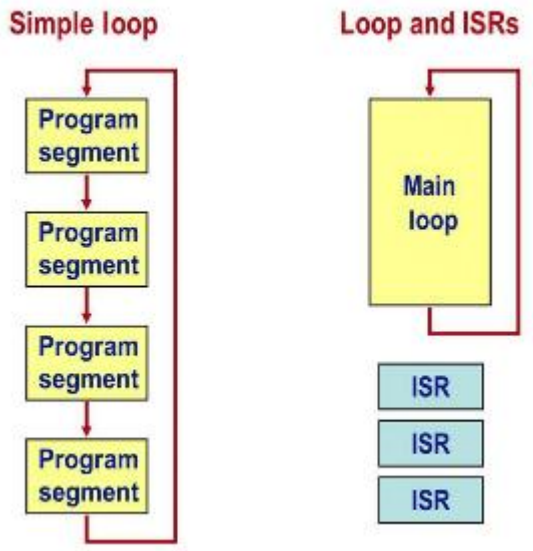

Figure 1 Common bare-metal embedded system software models [6]

Employing an Embedded Operating System allows to leverage concurrency, a core programming technique to improve the responsiveness and predictability of a software system. These characteristics are paramount on a real-time system, and as it grows, it gets harder to guarantee correctness of run-time behaviour without a kernel. Also, the leverage of a consistent kernel makes it is easier to create more modular, scalable, and readable system software - quality software.

0.1. EMBEDDED AND REAL-TIME OPERATING SYSTEMS

An operating system [10] acts as a resource manager and extends the machine for the user. It is a resource manager because it orchestrates the allocation of resources to - potentially conflicting – application requests.

It is an extended machine in the sense that, it makes it easier for the application programmer to achieve the application requirements, by providing a set of abstractions and mechanisms, that combined can be seen as services. For instance, an essential abstraction is a key word of this part: a task or thread. An essential mechanism: its management – dispatching, suspending, and resuming following a criterion.

While embedded operating systems can be of general purpose, often they are not. Real-time means that the distinction between success and failure of a computation depends on the result and the moment it is made available. It does not mean "fast", it means "on time".

0.2. HARDWARE-DEPENDENCY

"Yes, Minix is portable, but you can rewrite that as
'doesn't use any (hardware) features', and still be right."
(Linus Torvalds in response to Prof. Tanenbaum, on Usenet, 1992)

This series is about Embedded Real-Time Operating Systems design and implementation. As operating systems are tightly coupled to the hardware they run, and this series will provide concrete implementations, a hardware architecture had to be chosen. ARMv7m was the choice given its ubiquity on low to middle-end application domains – another focus of these articles.

The lowest-level kernel parts are shaped by the CPU architecture, and understanding its characteristics and programmer's model is paramount. But, again, this is not a series about ARM, neither I want it to be. That said, I chose an approach of addressing ARMv7m characteristics as needed throughout the text, relating them to the operating system concepts to be implemented, instead of dedicating a full section to the subject on the beginning. If the reader feels like having a more thorough understanding of the architecture before starting – I encourage consulting [1], [5], [9] and ARM Technical Reference Manuals.

Therefore, consider that all the low-level code presented in this text can be applied "as is" on ARM Cortex M3, M4 and M7 - unless you are using extended features which requires an extended core frame, e.g., a hardware floating-point unit. Besides the compiler is ARM GCC.

0.3. A GLANCE ON THE ARMv7M ARCHITECTURE

To start off I will address the features we will be using right now – and just a bit more - taking the ARM Cortex-M3 as the chosen model of this family. The other models such as M4 and M7 have extended features but same ISA. It does not apply for an ARM Cortex-M0, for instance, that is ARMv6m.

Every CPU has a Program Counter (PC), sometimes referred as the Instruction Pointer, a Status Register (SR), a Stack Pointer (SP) and several general data registers. PC is always

pointing to the next instruction to be fetched. SR contains different flags that are related to a status of the CPU, such as operating mode, active or pending interrupts, execution state, etc.

In the *ARM* Cortex-M3, the 32-bit core registers are: R0-R12 for general use and R13-R15 registers for special use, in addition to the *Program Status Register* (x**PSR**). The latter is not actually a single physical register, but a composition of three: *Application Status Register, Execution Status Register and Interrupt Status Register*. R14 is the Link Register (LR), which holds the return address when a subroutine (function) is called. R15 is the Program Counter (PC). R13 is the *Stack Pointer* – a special register that keeps track of the current position in stack (more about this later). It is a banked register meaning it is duplicated on hardware and can assume the role of *Main Stack Pointer* (MSP) or *Process Stack Pointer (PSP)*. On simple systems, the MSP is used all the way.

If the system programmer wishes to split user tasks from kernel tasks, both will be used. A special register, *CONTROL*, dictates which mode of operation is the current - roughly, *privileged*, and *unprivileged*. *Privileged mode* can either be a privileged thread or a privileged handler. There is no such a thing as an *unprivileged handler*. To begin with, we shall use only MSP on privileged mode. We will change this approach and discuss operation modes in more depth when it comes the time to split user stacks from kernel stack.

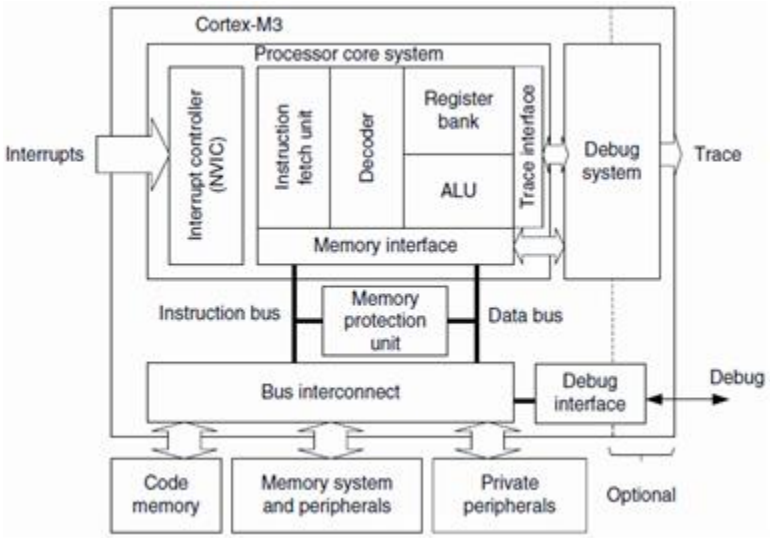

Figure 2 ARM Cortex-M3 block diagram (possible implementation)

Figure 2 shows a possible ARM Cortex-M3 implementation. Note that the Instruction Bus separated from the Data Bus is a characteristic of a *Harvard Architecture*. This separation allows simultaneous fetching of instruction and data, what yields better performance and some headaches to those programming – given the somewhat unexpected behaviours, here and there. Also, ARM architectures are all classified as *RISC – Reduced Instruction Set Computing. RISC* architectures aims to provide a simplified set of instructions. Allegedly its focus is on simplicity as a way for efficiency.

Common characteristics of RISC processors are heavily relying on hardware registers for storing intermediate values - following a *Load-Store* approach (not strictly, some fancy instructions blur this line), careful pipelining design and low code density. Another

examples of RISC architectures are MIPS, PowerPC, SPARC and more recently RISC-V - to name a few. Saying that nowadays, embedded programming is almost a synonymous of ARM programming is not much far from true and it does not seem to be changing too soon.

0.3.1. Load-Store Architecture

A *load-store* architecture is a processor architecture in which data from memory needs to be loaded to the CPU registers before being processed. Also, the result of this processing before being stored in memory must be in a register. That said, the two basic memory access operations on ARMv7m are:

```
// reads the data contained in the address indicated by Rn + offset and places it
in Rd.
LDR Rd, [Rn, #offset]
 // stores data contained in Rn at the address pointed by Rd + offset
STR Rn, [Rd, #offset]
```

Listing 1 Load and Store instructions

So, when you write a C statement such as incrementing an integer, `++index`, bear in mind the processor will be doing something like this, assuming `index` address is already on R0, (indeed, it will be if you have received it as a first function parameter):

```
LDR R1, [R0] //Load the value of 'index' into register R1
ADD R1, R1, #1 // Add 1 to the current value
STR R1, [R0] // Finally store the updated value in register R1 back to the memory
address stored in register R0
```

Listing 2 Incrementing a variable in ARMv7m assembly

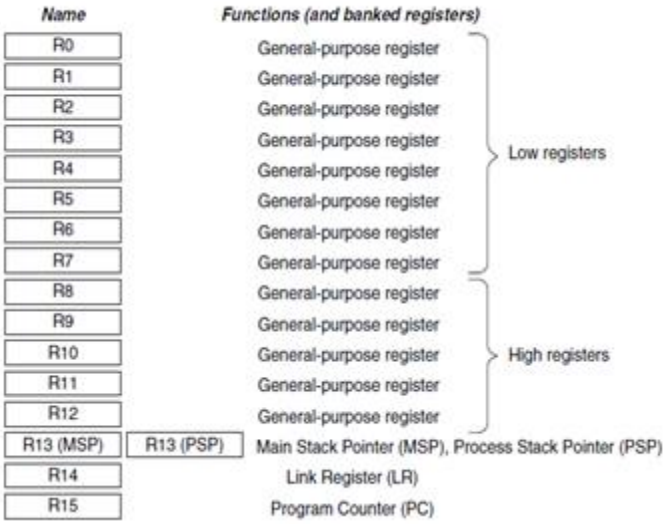

Figure 3 Figure 3 ARM Cortex-M3 register file[1]

0.3.2. Thumb2 Mode

In the past, ARM had only one instruction set, also called ARM, that was backwards compatible. There was a demand to decrease the memory used by code, which led to the birth of the Thumb mode [5]. In this mode, instructions could be 16-bit wide, and

programmers could go back and forth from the Thumb mode to the ARM mode on architectures such ARM7 and ARM9. Around 2005, the Cortex cores were introduced [8]. The Cortex architecture brought a level of commonality on the hardware//software interface by standardizing interrupt handling, power management and bus structures, plus a standard core across profiles [9]. This commonality benefited both chip designers and software designers and has contributed to the widespread adoption of ARM-based processors in diverse applications.

– *The Cortex-M series*: These are microcontroller-oriented processors intended for microcontroller unit (MCU) and system-on-chip (SoC) applications.

– *Cortex-R series*: These are embedded processors intended for real-time signal processing and control applications.

– *The Cortex-A series*: These are application processors intended for general-purpose applications, such as embedded systems with full featured operating systems.

Arm Cortex R and A still has the ability to go back and forth between the ARM and Thumb instruction sets. However, to keep the complexity and price down on Cortex-M, ARM created the Thumb2 instruction set as the only one for Cortex-M series [5].

0.3.3. Stacks and Stack Pointer

A *stack* is a memory usage model. It works on Last-In First-Out fashion, that is, the last element *pushed* to the stack will be first to be *popped*. The common use of a stack is that when an operation is about to modify some register value – like when we call a function (if you are writing assembly code) - we save the registers content to the stack, to restore them afterwards, as illustrated on Figure 4.

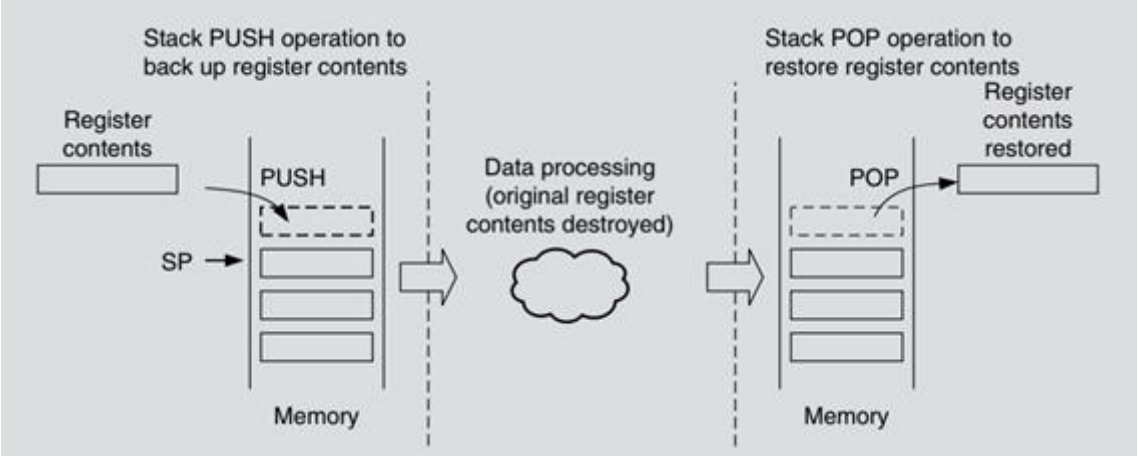

Figure 4 Stack usage model [1]

When doing PUSH and POP operations, the stack pointer is adjusted automatically to prevent next stack operations from corrupting previous stacked data. [1]

It is worth noting that when you're operating on a stackframe, the number of push and pops must be the same, on the reverse order, otherwise you will corrupt the stackframe, probably by accessing part of a neighbour's one.

The figure below shows the stack region within the physical memory layout of an ARM Cortex-M3 MCU:

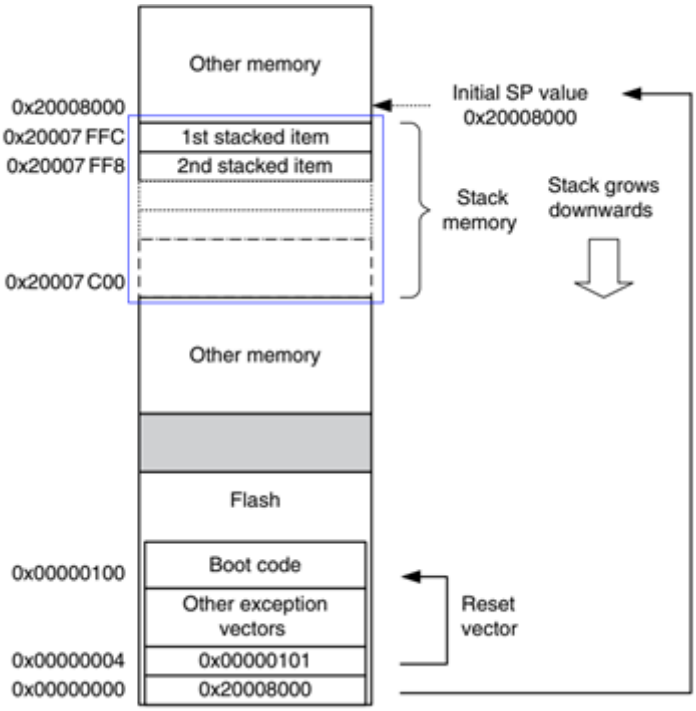

Figure 5 The stack within other memory regions of an ARM Cortex-M3 [1]

The initial stack pointer is the highest address of the stack. When *pushing* an item to the stack, the stack pointer will be decremented in 32 bits. That's why we say the stack "grows downwards". When popping, the stack pointer will be incremented in 32 bits.

If there is no memory virtualization, this is the absolute address space you are working on. On the highest privileged execution mode, you can access any address and write or read to it. This is bad and this is good. Bad because the program flow is free to read and modify data that can make the system crash. You read over 1 byte on an array and pretty much anything can happen; increased shared memory means more race conditions. Good because there is significant less overall latency when there's no need to translate virtual addresses. Plus, memory accesses are more predictable. The last two characteristics benefit real-time goals.

0.4. PROCESSES, TASKS AND THREADS

We need to make a clear distinction between process and thread. The latter is the fundamental entity we will be managing.

Formally, a *process* is the execution of an image. An execution image is a memory area containing the execution's code, data, and stack [8].

As explained on [4]:

The terms "process" and "program" are often used synonymously, but technically a process is more than a program: it includes the execution environment for the program and handles program bookkeeping details for the operating system. A process can be

launched as a separately loadable program, or it can be a memory-resident program that is launched by another process. Operating systems are often capable of running many processes concurrently. Typically, when an operating system executes a program, it creates a new process for it and maintains within that process all the bookkeeping information needed. This implies that there is a one-to-one relationship between the program and the process, i.e., one program, one process. When a program is divided into several segments that can execute concurrently, we refer to these segments as threads. A thread is a semi-independent program segment; threads share the same memory space within a program. The terms "task" and "thread" are frequently used interchangeably.

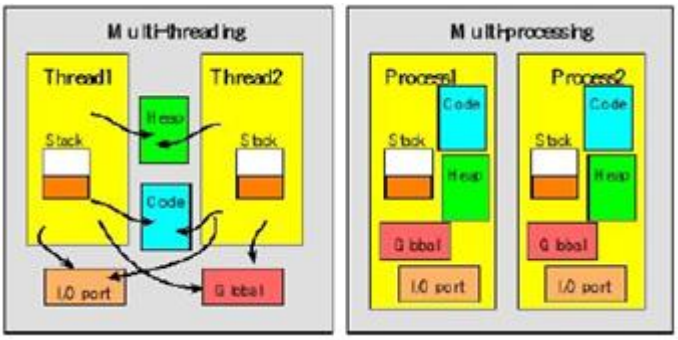

Figure 6. Multi-threading and Multiprocessing [4]

Many embedded software systems are deployed as a single process running multiple threads. On the other side, multiprocessing needs reasonable memory virtualization hardware and is employed mostly on general-purpose embedded operating systems running on bigger machines. Figure 6, taken from [4] provides a good distinction between multi-processing and multi-threading. Note that multi-processing does not prevent multi-threading, but multi-threading does not mean multi-processing.

In the in-between, several thread-based RTOSes support the use of an MMU to simply protect memory from unauthorized access. So, while a task is in context, only its code/data and necessary parts of the RTOS are "visible"; all the other memory is disabled and an attempted access would cause an exception. This makes the context switch just a little more complex but renders the application more secure. This may be called "*Thread Protected Mode*" or "*Lightweight Process Model*" (Figure 7).

I prefer the term *concurrent unit* to describe a process or a task/thread. Units of execution waiting/holding CPU time and resources.

0.4.1. Stackframes

A stack of a thread may contain several stackframes. Each time a function is called, a new stackframe is added on top of the stack to hold the information specific to that function call:

(1) Function parameters.
(2) Return address.
(3) Context information required to resume the function's execution after it finishes.

Sometimes we loosely use the term "stack" when talking about a stackframe.

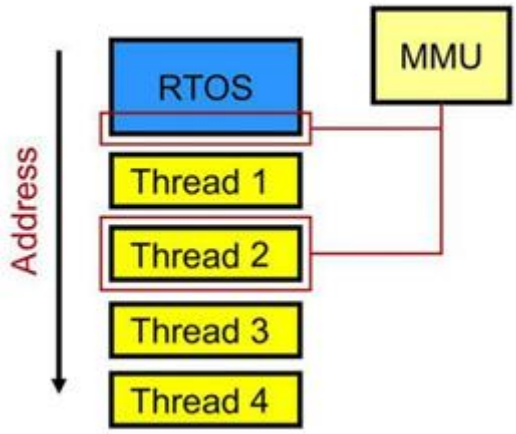

Figure 7 Light-weight process model [6]

0.5. CONTEXT SWITCHING

Changing the execution environment of one process/task/thread to that of another is called context switching, which is the basic mechanism of multitasking. That is, we save a point of the execution program, and load another one, previously saved.

0.5.1. Execution Image of a C program

An execution image, generated after compiling, assembling, and linking a C program has the following general layout [5], depicted on Figure 8.

This layout can contain other sections, depending on the linker script. We will explore more of this later. Now if you don't already know:

1. **Text (or Code) Section (.text):** This section contains the machine code for the executable instructions generated from the C source code. It includes the actual executable code for functions and instructions that the CPU will execute.
2. **Data Section (.data):** This section contains initialized global and static variables defined in the C code. It reserves memory for these variables and initializes them with predefined values. For example, int x = 10; would be stored in the .data section.
3. **BSS (Block Started by Symbol) Section (.bss):** This section contains uninitialized global and static variables. It allocates space for these variables but doesn't initialize them with any specific values. In the C code, variables declared with static int y; or int z; (without initialization) will typically be stored here.
4. **.isr_vector:** Is an array of addresses typical of the ARM Cortex-M CPUs. The first address is the top of the stack. The second address is the Reset_Handler address. Depending on the silicon vendor, an specific interrupt handler might be or not implemented.

The *linker* is the wizard who "knows" about everything in the text, bss and data sections and generates specific addresses for everything that goes into them. Binary code - or declared variables - have had their addresses resolved by the linker - even before the program is downloaded [5]. Respect the linker.

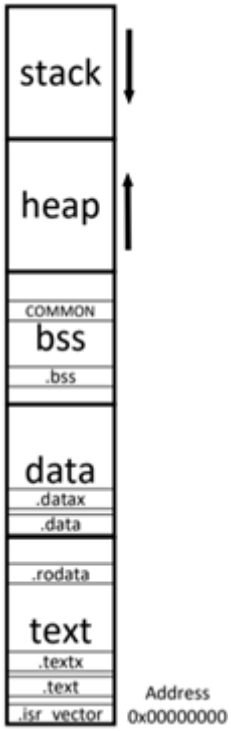

Figure 8 An execution image of a C program [5]

The stack is that memory region where function local data is pushed to and popped from, as it needs to be stored on and loaded from the core registers. As mentioned, a stack can have many stackframes, and there is a 1-1 map between a function call and a stackframe. Note that some kind of data is not stored on the stack even though when declared within a function. A static variable, for instance, makes a function not thread safe. The stack management is the cause and the solution of many system programming problems.

The heap is a part of the memory that can be seen as a pool of byte chunks, we can allocate, usually on run-time, to, for instance, accommodate data we can't know the size on compile-time – and we deem it is feasible to do that in run time. A problem inherent to heaps is that its fragmentation might reduce its real capacity: the kernel won't find a continuous chunk of a given size, although the total free memory is sufficient.

Worth mentioning that managing the heap on a multithread environment requires extra caution to ensure data integrity.

0.6. MULTITASKING

Multitasking refers to the ability of a system to run several tasks concurrently. On a single-processor system, this illusion of parallelism is achieved by multiplexing the CPU over time. The component of the kernel responsible for deciding which and when a task will run is the Scheduler. There are several kinds of schedulers and scheduling algorithms, the subject is extensive and complex. Generalizing the nature of schedulers, we can split in two classes - preemptive: a task might interrupt another if the scheduler policy says so and takes CPU over. Non-preemptive, or cooperative scheduling, that relies on the running task itself to yield the processor so the next task can run. Preemptive scheduling always has context-switching. That is not the case for cooperative schedulers, where in some implementations tasks run to completion before yielding the CPU – like the famous

super-loop calling functions from the top, the function itself returns, another is called, returns, and so forth (Figure 9).

Figure 9 Run-to-Completion scheduler [6]

There are other general patterns of schedulers [6]:

(1) *Cooperative round-robin* differs from RTC in the sense it has the ability of pausing a task and getting back to where it stopped (Figure 10)

(2) *Preemptive round-robin* scheduler. Each task has an amount of time to execute before being interrupted by the kernel. The time slice can be the same for every thread or individualised. (Figure 11)

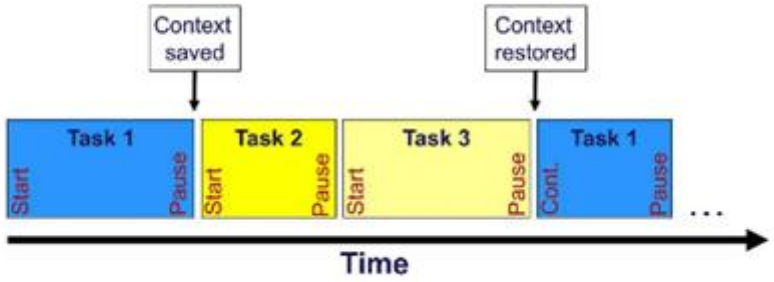

Figure 10 . Cooperative with context-switch - some authors might also call this a round-robin pattern (Figure from [6])

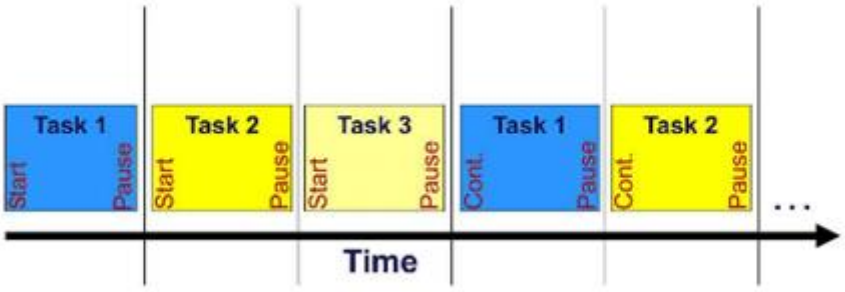

Figure 11 Round-robin time-sliced scheduler (Figure from [6])

(3) *Priority Scheduler*: tasks have priorities, and given a set of ready tasks, the one with highest priority is always chosen.

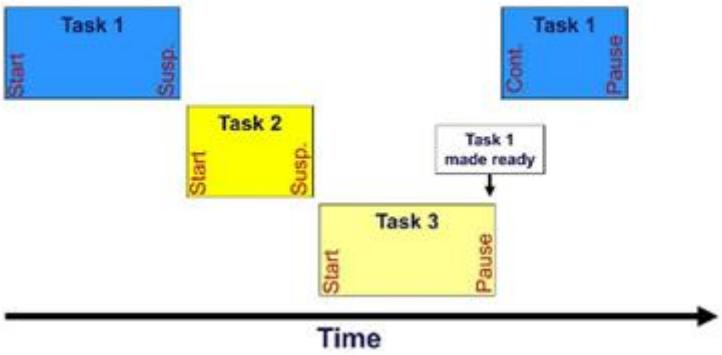

Figure 12. Priority scheduler - Task 1 interrupts Task 3 as soon as it is ready to run again (Figure from [6])

0.6.1. Context-Switch on ARMv7m

One might say the *ARMv7m* architecture is tailored for multitasking. One of the many characteristics for that is the way CPU handles interrupts entry and exit, which makes it easier to switch contexts when needed. Figure 13 depicts a full context-switch operation.

The mechanism is as follows: when an Interrupt Service Routine is fired, the gray registers on the upper left of the picture are all pushed (saved) from core to memory by the CPU itself. Then we need to push the remaining registers and our task will be ready to be resumed when it is time. Therefore, after manually pushing R4-R11, we can safely switch to another task. How? Making SP to point to the memory address aligned to the lowest address of the task's stack we wish to resume. This register is R4. *Which* task is that depends on the scheduler policy.

Then, we POP what is in the memory onto CPU registers, R4-R11. The return from the interrupt will make the CPU itself POP the remaining register values from the memory into the core, moving the stack pointer to highest address of the task's stack, and the task will run. Note a task is only active when its content is on the core registers. When switching to the R4 of the next task (the bottom of the stackframe), we are setting the scene for the next burst of popping values from the memory onto the core registers to activate a task.

On an ARMv7m core, when an interrupt handler takes place the LR will assume a special value*, depending on what kind of task was interrupted:

(1) If a task running using the Main Stack Pointer, but not on an interrupt handler, the value is 0xFFFFFFF9.
(2) If it was running using the Process Stack Pointer, the value is 0xFFFFFFFD.
(3) If it was in another interrupt handler, the value is 0xFFFFFFF1.

**There can be other return values if the CPU is using extended features*

0.7. IMPLEMENTATION

0.7.1. The Thread Control Block

The fundamental data structure on any kernel is the one that keeps track of each concurrent unit the system is managing. It is the Process/Task/Thread Control Block. On

a real kernel, it can be quite large. For our immediate purposes, we only need two fields: the stack pointer of the current stack frame, and the stack pointer of the next stack frame, that must be of a distinct task. In this case, we are going to implement a *Round-Robin* scheduler, a first-in first-out fashion with the same time slice for every thread. TCBs are arranged on a circular linked list, and the next task to run is simply the one in the next node of the list.

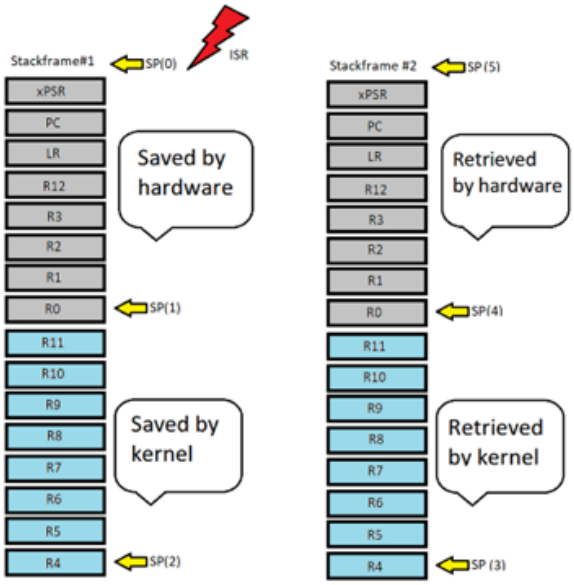

Figure 13 Switching contexts. The active task is saved by the hardware and the kernel. The stackpointer is re-assigned to the R4 of the next stackframe to be activated. The data is retrieved. Task is resumed. (Figure adapted from [3])

The stacks for each thread also need to be explicitly declared and initialized. Let's run 3 threads, with 64 words of stack space, which is 256 bytes.

We need a global pointer constantly holding the address of the running TCB. This pointer will be called *RunPtr* and system will often write/read from it to operate on a running task.

Note the thread control block (Listing 3) is a self-referential data structure because it is meant to be member of a queue (list). Queues are essential data structures on kernel design, from scheduling to inter-task communication.

```
#define NTHREADS 3 /*number of threads*/
#define STACKSIZE 64 /*stack size of each thread in words*/
struct tcb /*thread control block*/
{
       int32_t* sp; /*current stack pointer*/
       struct tcb* next; /*next stack pointer*/
}
typedef struct tcb TCB_t;
TCB_t tcbs[NTHREADS]; /*pool of threads*/
int32_t stacks[NTHREADS][STACKSIZE];
TCB_t* RunPtr;
```

Listing 3. TCB, stacks and RunPtr

0.7.2. Stacks Initialization

We need to assemble the stacks, so they are ready to be popped onto the core. To do so, we might initialize a stack with a stackframe that looks like one that ran before and is going to be resumed now. Looking at Figure 13, it means the address with offset of 16 words below the top register will be the initial stack pointer of a thread stack that has just been assembled. It is the lowest register of a stackframe. "Task 0" would have its stack pointer initially assigned to the following address: *&stacks[0][STACKSIZE-16]*. Also, there should be a valid Program Counter address.

The Program Counter takes the address of the "main" task function. In intermediate states, this PC can be pointing to other values, within the body of this function, or any other instruction address the task is about to branch to (remember: the program counter always points to next instruction to be fetched, not the current one). The xPSR needs to be (1<<24) to ensure we work on Thumb2 mode.

That said, the following function can be used to assemble an initial task stack. It is very important you understand what this function is doing, why values are initialised this way (explained above).

```
/*
 * @brief Core registers offset from the top of stack
 * */
#define PSR_OFFSET    1
#define PC_OFFSET     2
#define LR_OFFSET     3
#define R12_OFFSET    4
#define R3_OFFSET     5
#define R2_OFFSET     6
#define R1_OFFSET     7
#define R0_OFFSET     8
#define R11_OFFSET    9
#define R10_OFFSET    10
#define R9_OFFSET     11
#define R8_OFFSET     12
#define R7_OFFSET     13
#define R6_OFFSET     14
#define R5_OFFSET     15
#define R4_OFFSET     16

typedef void  (*TaskPtr)(void*); //<* Thread function pointer
/*
 @brief Assembles initial task stack
 @param i Task ID
@param task Task function address.
*/
void SetInitStack(uint32_t i, TaskPtr task)
{
	tcbs[i].sp = &stacks[i][STACKSIZE - R4_OFFSET];
	stacks[i][STACKSIZE - PSR_OFFSET] = 0x01000000; //**PSR**
	stacks[i][STACKSIZE - PC_OFFSET] = (int32_t)task; //r15 **PC**
	stacks[i][STACKSIZE - LR_OFFSET] = 0x14141414; //r14      **LR**
	stacks[i][STACKSIZE - R12_OFFSET] = 0x12121212; //r12
	stacks[i][STACKSIZE - R3_OFFSET] = 0x03030303; //r3
	stacks[i][STACKSIZE - R2_OFFSET] = 0x02020202; //r2
	stacks[i][STACKSIZE - R1_OFFSET] = 0x01010101; //r1
	stacks[i][STACKSIZE - R0_OFFSET] = 0x00000000; //r0
	stacks[i][STACKSIZE - R11_OFFSET] = 0x11111111; //r11
	stacks[i][STACKSIZE - R10_OFFSET] = 0x10101010; //r10
	stacks[i][STACKSIZE - R9_OFFSET] = 0x09090909; //r9
```

```
        stacks[i][STACKSIZE - R8_OFFSET] = 0x08080808; //r8
        stacks[i][STACKSIZE - R7_OFFSET] = 0x07070707; //r7
        stacks[i][STACKSIZE - R6_OFFSET] = 0x06060606; //r6
        stacks[i][STACKSIZE - R5_OFFSET] = 0x05050505; //r5
        stacks[i][STACKSIZE - R4_OFFSET] = 0x04040404; //r4
}
```

Listing 4. Function to assemble the initial stack of a task

The remaining registers that we did not assign meaningful values, were assigned values to easy debug - such as 0x04040404 for the memory address that should be aligned to R4. So, if we see register R7 storing 0x04040404 we can promptly see our system is malfunctioning. These values can be overwritten as the program flows, depending on your stack usage.

0.7.3. The Context-Switcher

Usually, a kernel deployed on an ARMv7m will use the *SysTick* hardware as a tick for the system. A simple scheduler will switch context at every system tick. The interrupt service routine will handle the context switching for now. Later we might defer this context-switching to a very low-priority software interrupt handler, so *SysTick Handler* is offloaded, and context-switching is prevented from happening in the middle of other interruptions. Because it accesses the core registers, it needs to be written in assembly. I suggest creating. s files instead of embedding ASM in C code (Listing 5).

```
.syntax unified //<< needed for Thumb2
.global SysTick_Handler
.type SysTick_Handler, %function
SysTick_Handler : /* R0-R3, R12, LR and xPSR from the interrupted task are pushed
on stack  */
PUSH {R4-R11} // therefore, we save the remaining stack frame
LDR R0, =RunPtr //R0 = &RunPtr
LDR R1, [R0]  /* R1 = RunPtr->sp */
STR SP, [R1] /*save current stack pointer into TCB, to be resumed later*/
LDR R1, [R1, #4] // R1=RunPtr->next
STR R1, [R0] // Then we store  the next tcb block in RunPtr
LDR SP, [R1] // SP=RunPtr->sp
POP {R4-R11} // Pop the software stackframe on registers
BX LR // Return will pop {R0-R3}, R12, LR, PSR
```

Listing 5. SysTick Handler routine

0.7.4. The Start-Up

How to start the first task? An approach can be as on Listing 6 [4]:

```
.global StartUp
.type StartUp, % function
StartUp :
LDR R0, =RunPtr // R0 = &RunPtr
LDR R1, [R0]    // R1 = RunPtr->sp
LDR SP, [R1]    // SP=RunPtr->sp = &stacks[0][STACK_SIZE-16]
POP {R4-R11}
POP {R0-R3}
POP {R12}
ADD SP, SP, #4
POP {LR} // pop PC onto LR
ADD SP, SP, #4
CPSIE  I // enable global interrupts
```

```
BX LR // return
```

Listing 6. Start-up routine #1

What this routine is doing is popping R4-R11, R0-R3, R12 into the core registers, discarding the LR of the initial stack by adding 32-bits to the stack pointer, popping the initial PC onto the LR core register, then moving the SP up more 32 bits and finally branching to the link register, that has the task function address. Therefore, we moved the stack pointer to the top of the stack either by popping the values from the stack to the CPU register file, or simply incrementing its value for the registers we didn't care, and . It will be dispatched.

Another approach could be as on Listing 7:

```
.global StartUp
.type StartUp, % function
StartUp :
LDR     R0, =RunPtr
LDR     R1, [R0]
LDR     SP, [R1]
POP     {R4-R11}
POP     {R0-R3}
POP     {R12}
POP     {LR}
CPSIE   I
POP     {PC}
```

Listing 7 Start-up routine #2

Popping (loading) a new value to the Program Counter (PC) register always causes a branch.

0.8. DEPLOYMENT

I will be using a NUCLEO-F103RB (Figure 14) board. This board is based on the MCU STM32F103RBT6, which incorporates an ARM Cortex-M3, that can run up to 72MHz. Relatively small, it has 128KB of Flash memory and 20KB of RAM. It does not have a Memory Protection Unit. On the top of the board, we have debugging circuitry for ST-LINK. Nice. But I warn you that the debugger is rather weak, and if you are up to designing a kernel for multithread applications, consider acquiring better debugging capabilities.

I'll also be using the STM32CubeIDE. These IDEs usually save a lot of work on the low-level end, since they generate the linker script, the start-up code and initial peripheral configuration. It also provides a Hardware Abstraction Layer, compliant with the ARM CMSIS standard. You simply select the board you're using and the tool will generate all the low-level software needed to start writing your application code (Figure 15). Taking these "details" for granted is not a good practice though. On a real project, you will often need to tweak linkers, HALs, and so forth. A good exercise is to get everything up and running from the scratch, with a toolchain and a plain text editor.

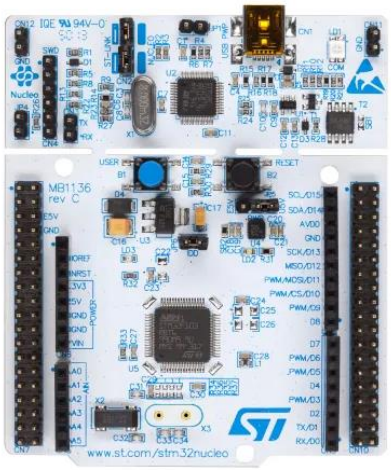

Figure 14. My ten-dollar computer. The Nucleo F103RB board

You can use any other target if it is an ARM Cortex-M3, M4 or M7.

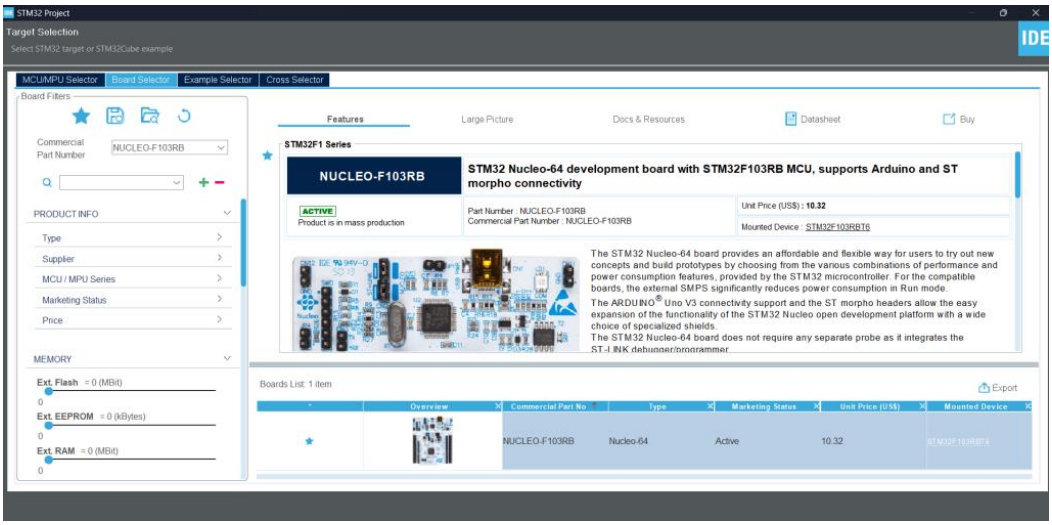

Figure 15 Selecting board for the STM32 project.

Now, the code. First, you need to get rid from the automatic generated code for the *SysTick Handler*, since we are going to write our own in assembly. In the file *stm32f1xx_it.c* comment out the provided *SysTick Handler*. I usually comment code out by using #if (0) / #endif guards (Figure 16).

Create a *kernel.s* file in Core/Src and put the *SysTick_Handler* and *StartUp* of Listing 5 and Listing 6.

Create a file *tcb.h* and *tcb.c* to accommodate the function prototype and definition of Listing 4.

Create a file *tasks.h* and *tasks.c*. Put them on Core/Inc and Core/Src respectively (Listing 8).

Our tasks will simply increment a variable. Declare the variables as *volatile* so the compiler won't optimize them out, since we are not using them to anything else.

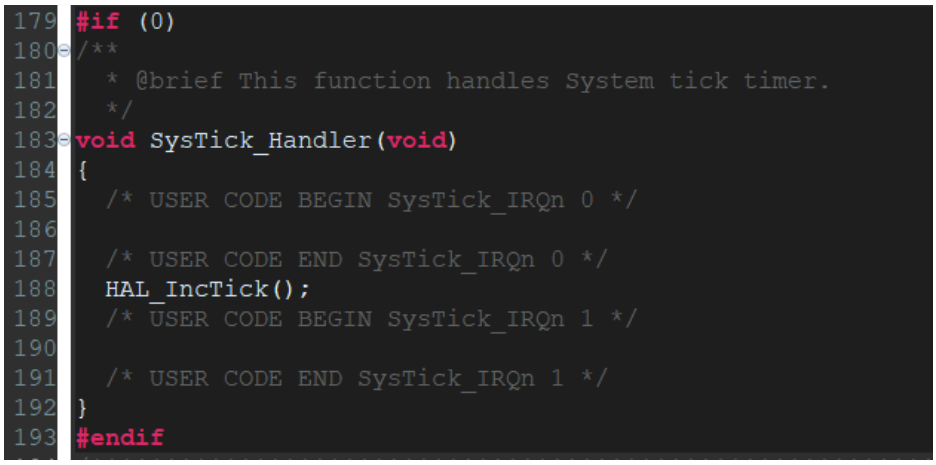

Figure 16. Automatic generated SysTick handler commented out

```
/*
/*
@file tasks.h
*/
#ifndef TASKS_H_
#define TASKS_H_
void Task0(void* args);
void Task1(void* args);
void Task2(void* args);
#endif
/*
@file tasks.c
*/
#include <stdint.h>
volatile int32_t counter0;
volatile int32_t counter1;
volatile int32_t counter2;
void Task0(void* args)
{
	while (1)
	{
		counter0++;
	}
}
void Task1(void* args)
{
	while (1)
	{
		counter1++;
	}
}
void Task2(void* args)
{
	while (1)
	{
		counter2++;
	}
}
```

Listing 8. tasks.h and tasks.c

It is interesting to note that tasks have a while(1) loop, that is, they never return. Also the main program after calling the kernel will have a while(1) at the bottom, meaning the

program can't go further that point. We have split the system in background (the kernel) and foreground (the threads) tasks, which take turns to run every SysTick interrupt.

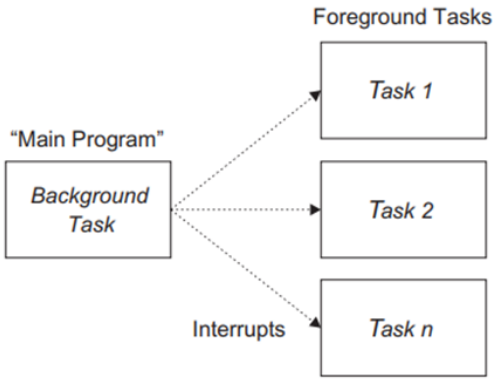

Figure 17. Background and foreground tasks [7]

The main program will look something like Listing 9:

```
#include "main.h"
#include "tcb.h"
#include "tasks.h"
TCB_t tcbs[NTHREADS]; /*pool of threads*/
int32_t stacks[NTHREADS][STACKSIZE];
TCB_t* RunPtr;
int main(void)
{
	HAL_Init(); /* init HAL library*/
	SystemClock_Config(); /* Configure sys clock */
	SysTick_Config(SystemCoreClock / 100); /*Tick every 10ms: every task will
run for 10ms before being preempted */
	__disable_irq(); /* disable global interrupts */
	/*initialize tasks circular linked list */
	tcbs[0].next = &tcbs[1];
	tcbs[1].next = &tcbs[2];
	tcbs[2].next = &tcbs[0];
	/*assemble initial stacks for the tasks*/
	SetInitStack(0, Task0);
	SetInitStack(1, Task1);
	SetInitStack(2, Task2);
	RunPtr = &tcbs[0]; //<* we start on Task 0
	/*start the scheduler */
	StartUp();
	while(1);
}
```

Listing 9. The main program

Let's run it:

The StartUp routine debugging window shows that our stack is correctly being popped onto the core registers (Figure 18)

When the breakpoint is at line 28 of the StartUp routine, if you look at the stack on the left-side at the debugging window, you'll (see Figure 19) 0x8000518 right after the StartUp() routine on the thread stack, meaning it will be the next point of execution when StartUp() returns.

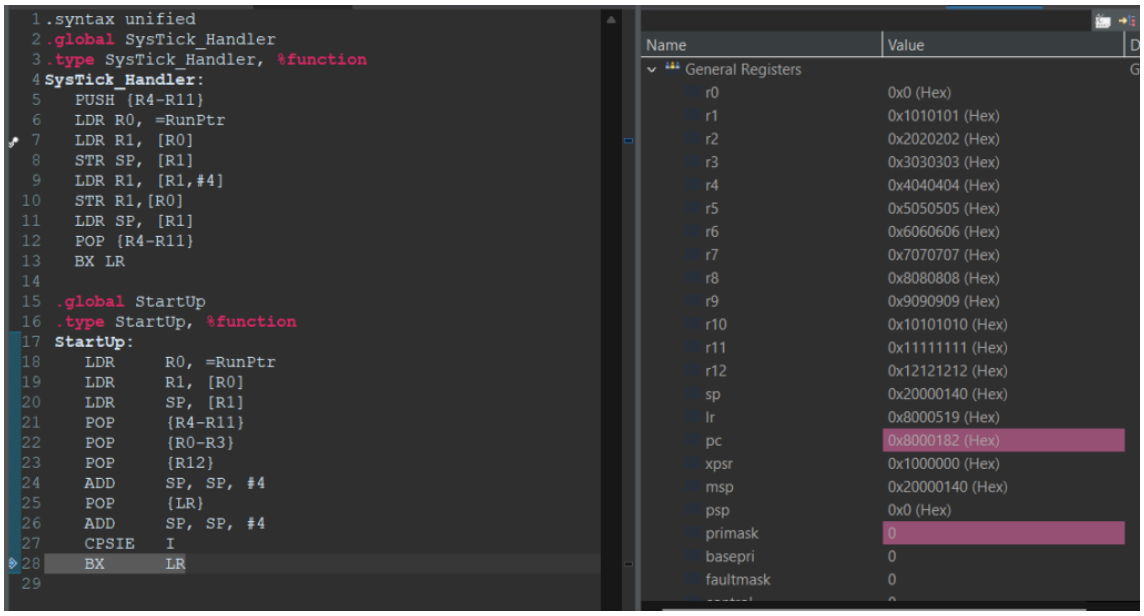

Figure 18 Debugging window of StartUp routine.

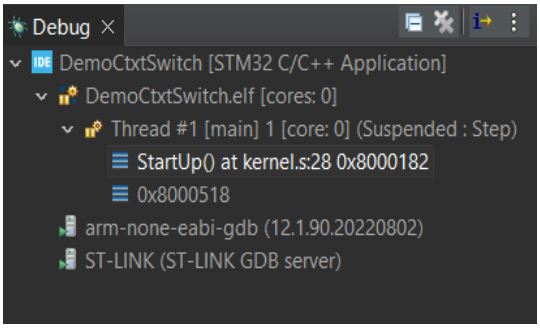

Figure 19 Stack elements right before branching to Task0.

Following through, it will dispatch *Task0 (*Figure 20*).* If everything is working out fine, every time you pause the system, the counters will have similar values (Figure 21).

0.8.1. Cooperative Round-Robin

With this very same scheduler we can make tasks to yield CPU cooperatively, that is, when they finish what they are supposed to do, they will voluntarily return the CPU control to the scheduler. To do that, we can write '1' to bit 26 of the *Interrupt Control and State Register* (address 0xE000ED04), and it will force a SysTick interrupt to be triggered (Listing 10).

```
/*this macro converts an address value to a pointer and dereferences it so we can
write into it
*/
#define HW_REG(addr) (*((volatile unsigned long *)(addr)))
void Yield(void)
{
	HW_REG(0xE000ED04) |= (1 << 26);
}
```

Listing 10. Forcing SysTick to trigger

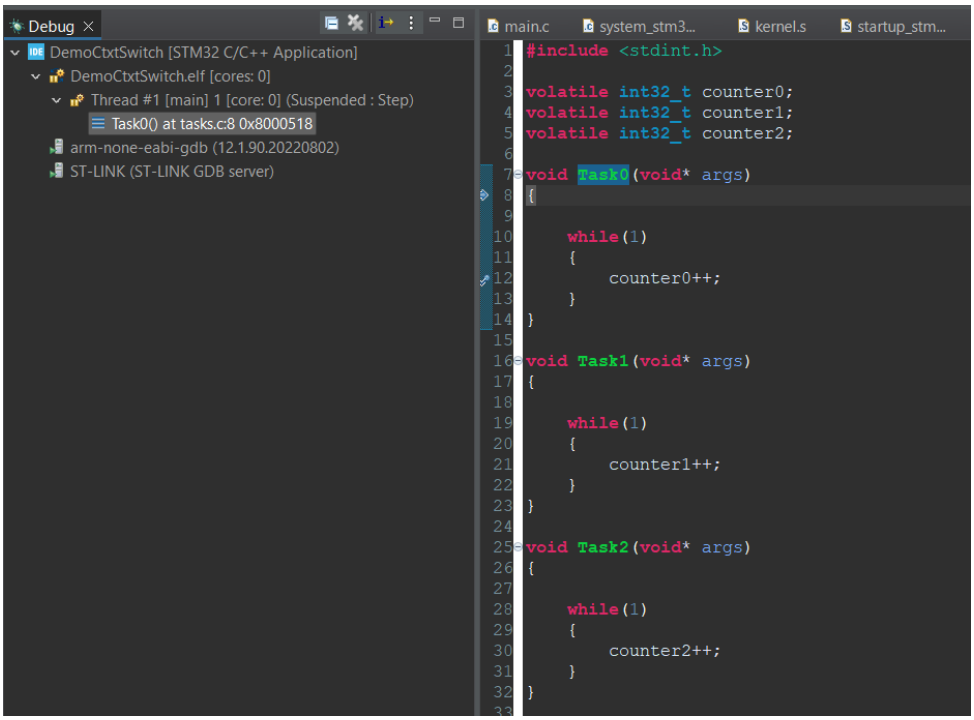

Figure 20 Task0 at address 0x8000518.

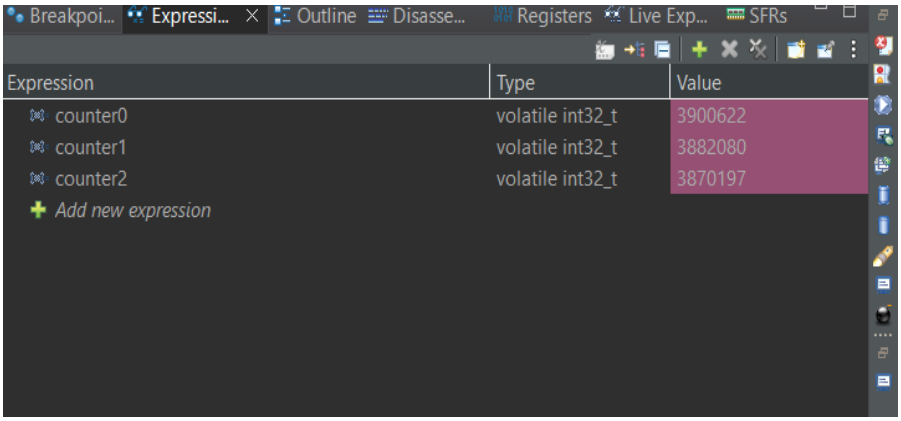

Figure 21 Counters will have roughly the same value every time you pause the system.

You could write the same function using the CMSIS Hardware Abstraction Layer, provided with the STM library (Listing 11):

```
#include <stm32f1xx_hal.h>
void Yield(void)
{
	SCB->ICSR |= SCB_ICSR_PENDSTSET_Msk;
}
```

Listing 11. Forcing SysTick to trigger using the CMSIS HAL

Our tasks cooperatively working can be written as on Listing 12. The application printing characters on the PC terminal is on Figure 22.

Note *Task0* restarting from the point where it gave CPU up, as a proof the scheduler is correctly saving and restoring context.

```c
void Task0(void* args)
{
    while(1)
    {
        HAL_UART_Transmit(&huart2, "Task0 will yield\n\r", 18, 1000);
        HAL_Delay(100);
        Yield();
        HAL_UART_Transmit(&huart2, "Task0 resumed\n\r", 15, 1000);
        HAL_Delay(100);

    }
}

void Task1(void* args)
{
    while(1)
    {
        HAL_UART_Transmit(&huart2, "Task1\n\r", 8, 1000);
        HAL_Delay(100);
        Yield();
    }
}
void Task2(void* args)
{

    while(1)
    {
        HAL_UART_Transmit(&huart2, "Task2\n\r", 8, 1000);
        HAL_Delay(100);
        Yield();
    }
}

```

Listing 12. Tasks yielding CPU cooperatively

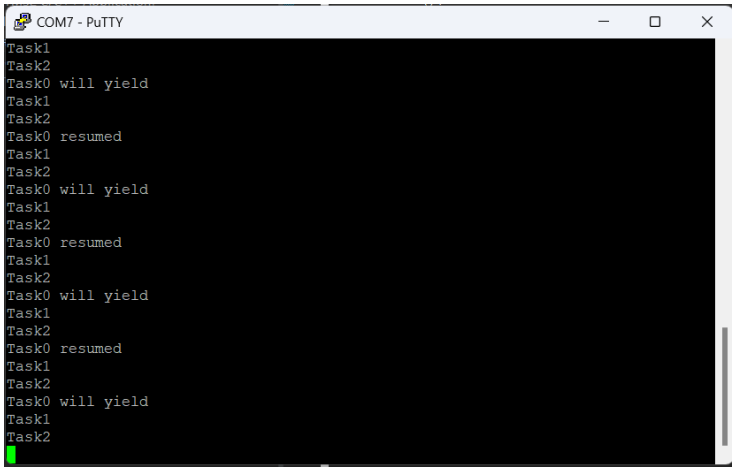

Figure 22. Characters printed on the terminal.

0.9. FINAL COMMENTS

The concept of process and task/thread were defined and related to an execution image. A process is the execution of an image. A thread or a task is a running unit of a process, sort of a semi-independent program running on the address space of a process. To create this abstraction – the thread – a mechanism is needed so the process can run several functions concurrently. The bookkeeping for each thread, so they can be dispatched, suspended, and resumed by the kernel, following a criterion, is kept on a data structure called Thread Control Block (TCB).
Based on these concepts and addressing the characteristics of the ARMv7m architecture we have designed and implemented a time-sliced cooperative-preemptive round-robin scheduler. It was deployed on an ARM Cortex-M3 based MCU and will set the foundation upon which we will keep improving our kernel design and implementation.

REFERENCES

[1] Yiu, Joseph. The definitive guide to ARM Cortex-M3. Newnes.
[2] Valvano, Jonathan. Embedded Systems: Real-Time Operating Systems for ARM Cortex-M Microcontrollers. Jonathan Valvano. Kindle Edition.
[3] https://www.embedded.com/taking-advantage-of-the-cortex-m3s-pre-emptive-context-switches/
[4] Lamie, Bill. Real-Time Embedded Multithreading with Azure RTOS ThreadX. 4th Edition.
[5] Elk, Klaus. Microcontrollers With C: Cortex-M and Beyond. Kindle Edition.
[6] Walls, Colin. Embedded RTOS Design: Insights and Implementation. Elsevier Science. Kindle Edition.
[7] Laplante, Phillip A. Real-time systems design and analysis : tools for the practitioner. 4th ed.
[8] Wang, K. C. Embedded and Real-Time Operating Systems. Springer International Publishing. Kindle Edition.
[9] Martin, Trevor. The Designer's Guide to the Cortex-M Processor Family. Elsevier Science. Kindle Edition.
[10] Tanenbaum, Andrew S.; Bos, Herbert. Modern Operating Systems, Global Edition. Pearson Education. Kindle Edition.