ECE 491: Real-Time Operating Systems

Andy Jaku (EE'25) and Fred Kim (EE'26)

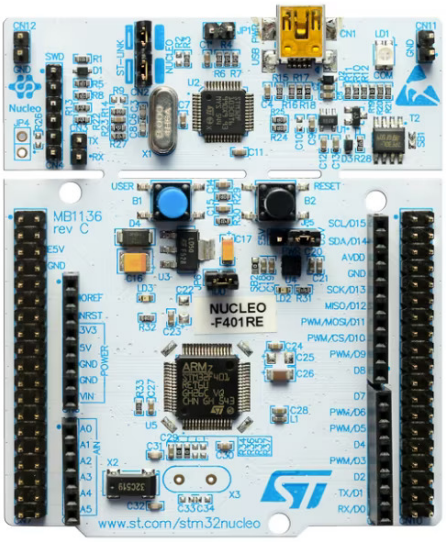
1 Introduction

The purpose of this independent study, **ECE-491: Real-Time Operating Systems**, was to develop a Real-Time Operating System (RTOS) for the **Nucleo-F446RE** development board, which features ARM's Cortex-M4 processor.

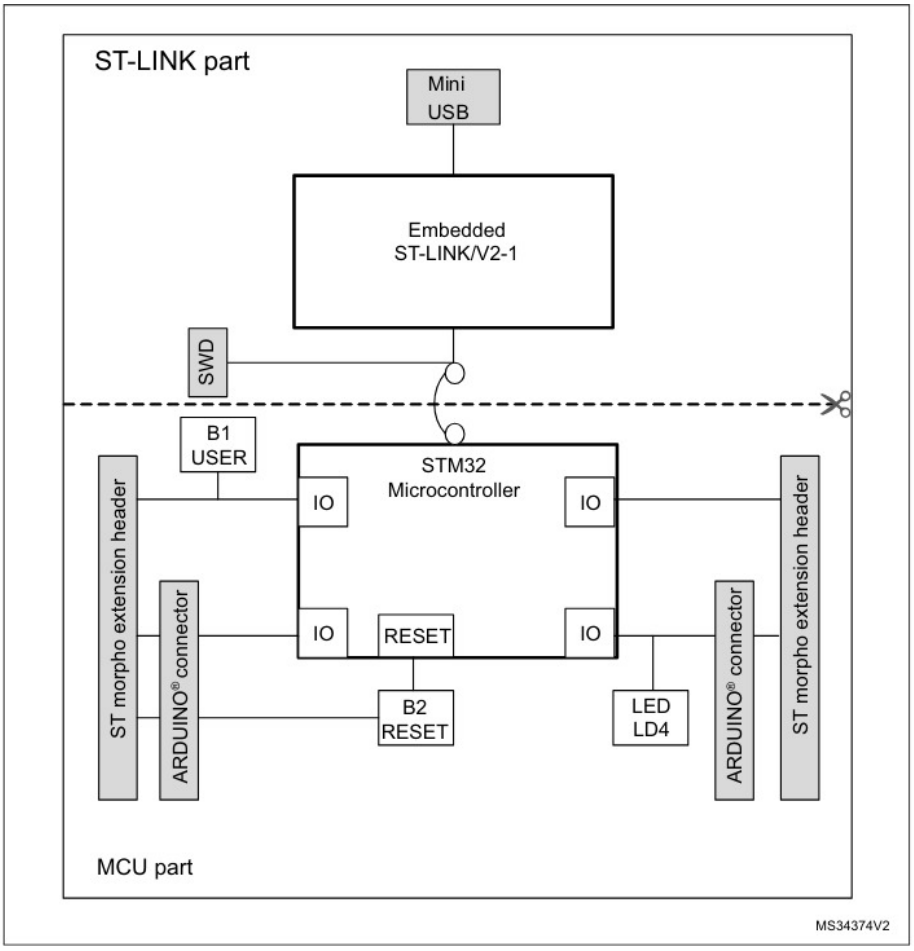
This project involved the following key tasks:

- Implementing bare-metal initialization for the processor.
- Separating the stack for the kernel and each task to prevent tasks from modifying the kernel's memory region.
- Creating processor context backup and restoration routines to enable context switching.
- Building a preemptive scheduler that cyclically schedules tasks based on their priority and respective wait times.
- Implementing basic synchronization primitives like mutexes and semaphores to facilitate resource sharing between tasks.

2 The Nucelo-F446RE



This project features the Nucelo-F446RE as the development board of choice, seen above [1]. It comes equipped with a Cortex-M4 microcontroller that has 512KB of FLASH and 128KB of SRAM. To program the device, the development board comes includes an ST-LINK/V2-1 which utilizes either JTAG or SWD to load firmware onto the device, as seen in the split below [2].



Nucelo's development board also has a Mini-USB header which serves as the aforementioned programming port or USART passthrough (which will be covered later) and an on-board LED (LD4) which was used for demonstration and debugging.

3 Environment Configuration

As this is a baremetal utilization of the processor, no standard CMSIS libraries were used to configure the build enviornment and were instead written from scratch using them as a resource [3]. The majority of the similarities can be

found in the peripheral `.h` files which lay out register maps based on addresses specific to the hardware.

A custom linker script, `cm4.ld`, is used when building firmware for the device. It defines two regions in memory, SRAM and FLASH which have lengths of 128k and 512k respectively. Beyond the standard operations of a linker script, like defining the `data` and `bss` regions, this script allocates a region in memory for the **Nested Vector Interrupt Controller**, `NVIC` in short, which needs to be located at the start of the SRAM. When the processor begins initialization, the starting point of the main stack pointer is the first entry within the `NVIC`, in this case the start of SRAM. The second entry within the `NVIC` is the default routine to run during a system reset (or initialization) [8].

Thus, to begin system initialization, the `_start` assembly routine is stored as the second vector within the `NVIC` and is the first routine that will run upon system initialization. That routine, located within the `startup_cm4.s` file, will copy the flash's `data` section into the SRAM section and then clear out the `bss` region within SRAM - both approaches are inspired by CMSIS's [3] method for doing so. The program then initializes the `MSP` (main stack pointer) and `PSP` (processes stack pointer) to their regions in memory and branches to main with the `MSP` active.

With that set up, it's time to start building! The `Makefile` is quite generic - it takes in desired `CFLAGS` and `LDFLAGS`, then gets sources from the `src` directory and includes files from the `inc` directory. Simply running `make` will create a `.elf` file which can be flashed to the board using a debugger (GDB or the CubeIDE) while utilizing OpenOCD. Performing `make flash` will write the binary onto the chip without any form of debugging active.

4 The STOS Kernel

4.1 Kernel and Task Stacks

The Cortex-M4 provides two different stack pointer registers, the Main Stack Pointer (`MSP`) and the Process Stack Pointer (`PSP`). On initalization the processor utilizes the `MSP` and switching between the `MSP`/`PSP` can only happen during a return from an exception, as shown in the figure below [4]. The current implementation does not feature floating-point support, thus exceptions will always return with either `0xFFFFFFF9 (MSP)` or `0xFFFFFFFD (PSP)`Tasks.

EXC_RETURN[31:0]	Description
0xFFFFFFF1	Return to Handler mode, exception return uses non-floating-point state from the MSP and execution uses MSP after return.
0xFFFFFFF9	Return to Thread mode, exception return uses non-floating-point state from MSP and execution uses MSP after return.
0xFFFFFFFD	Return to Thread mode, exception return uses non-floating-point state from the PSP and execution uses PSP after return.
0xFFFFFFE1	Return to Handler mode, exception return uses floating-point-state from MSP and execution uses MSP after return.
0xFFFFFFE9	Return to Thread mode, exception return uses floating-point state from MSP and execution uses MSP after return.
0xFFFFFFED	Return to Thread mode, exception return uses floating-point state from PSP and execution uses PSP after return.

Distinctions between the kernel and task stacks are defined within the `cm4.ld` file as `_mstack` and `_pstack` which are mapped to the beginning and middle of SRAM respectively. Thus, at the moment, each stack region takes half of the stack space, but this could be refined further depending on the needs of a given program. On initialization, the processor uses the main stack pointer (the kernel pointer), until an SVC call is performed. An SVC call (aka a Supervisor call) is an exception that provides an opportunity to switch the active stack pointer by writing to the link register.

```
// From stos.c

void svc_handler(void) {
    __asm volatile(
        "MOV LR, #0xFFFFFFFD \n"
        "BX LR \n");
}
```

Determing which region of the stack is being used is quite simple. The Cortex-M4 exception frame states that all exceptions will required elevated priority and will use the `MSP`. Thus any exception (like `SYSTICK` - discussed later) and subsequent kernel level functions will take place within the `MSP`, while an exception that intends to return to a task (like `PENDSV` - discussed later) will modify the link register and return into the context of the `PSP` [4].

4.2 Kernel Structure

The stos kernel `stos_kernel_t` is the main data structure repsonsible for managing the status of the operating system. It is laid out as follows:

```
// From stos.h

typedef struct stos_kernel {
```

```
    stos_tcb_t *list_ready_head;
    stos_tcb_t *list_blocked_head;

    stos_tcb_t *next_task;
    stos_tcb_t *active_task;
    stos_tcb_t idle_task;
} stos_kernel_t;
```

The kernel tracks a list of task control blocks (`stos_tcb_t`), each of which stores critical task information.

```
// From task.h
typedef struct stos_tcb {
    void        *sp;
    void        (*func)(void);

    uint32_t    state;

    uint32_t    pri;
    uint32_t    timeout;
    uint32_t    sleep;

    struct stos_tcb *next;
    struct stos_tcb *prev;
} stos_tcb_t;
```

Where each task keeps track of it's stack pointer, function, state (which will be one of `TASK_READY`, `TASK_BLOCKED`, `TASK_RUNNING`, or `TASK_SUSPENDED`), priority, the amount of time it should timed out/sleeping for, and then two pointers to the next and previous task, that should always be sorted in terms of priority.

The `list_ready_head` tcb points to the head of the ready list of tasks, which should always be the highest priority task that is ready to run (excluding the active task). Similarly, it keeps track fot he `list_blocked_head` which is the highest priority task that is has been put on timeout/sleep for a set duration and cannot be scheduled until it is placed back on the ready list.

Internally, the kernel calls on the `STOS_AddTask` and `STOS_RemoveTask` functions take in these task control blocks and the respecitve task lists to append/remove them from and maintain that ordering.

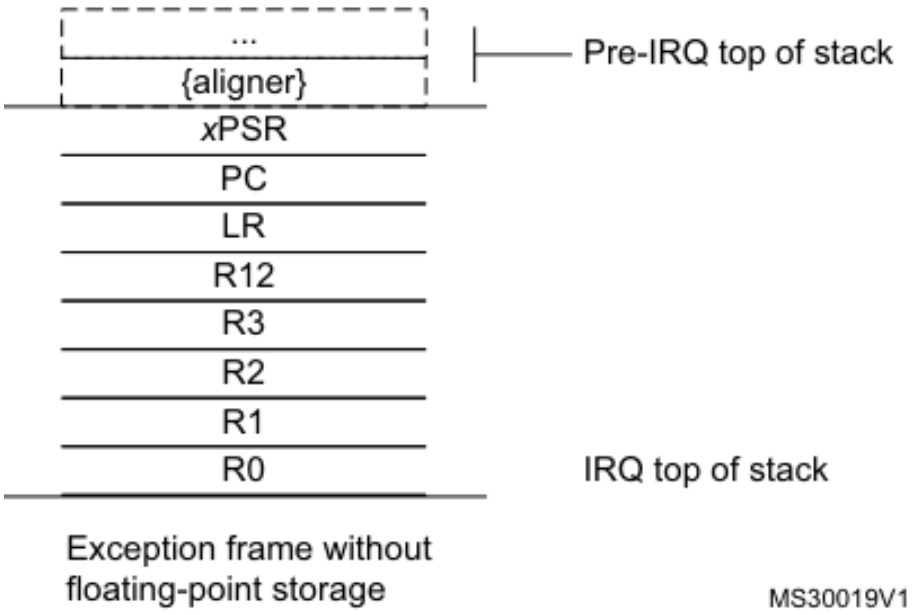
The last three tasks that the kernel stores are the `next_task`, which is the highest priority ready task that will be scheduled next (same as `list_ready_head`) and the `active_task` which points to the task that is currently running (it shouldn't be on either the ready or blocked list), and lastly the idle task, which should be the last lowest priority task within the system. There is certainly room for some more optimization by removing some bloat from what the kernel is keeping track of.

4.3 Task Creation

Creating a task is as simple as initalizing an `stos_tcb_t` and then calling:

```
// From stos.h
void STOS_CreateTask(stos_tcb_t * const task,
                     void       (*handler)(void),
                     uint32_t   pri,
                     uint32_t   size);
```

This function creates space by decrementing `PSP` to establish the stack region and frame of the task within the process stack. The ARM AACPS [5] defines what a standard stack frame should look like to ensure that returning from an exception maintains the context of the function which was interrupt (seen below) [4]. Part of the inspiration for how to establish this context frame is attributed to Miro Samek's great guide on developing an RTOS, the modern-embedded-programming-course [6].



Once all of the desired tasks have been added, calling `STOS_Init` with the default configuration parameter will add an idle task with the lowest priority to the linked list and set up a `SYSTICK` timer.

5 Fixed-Priority Scheduler

5.1 SYSTICK Configuration

The core of a Real-Time Operating System consists of having a scheduler that runs at a set rate and decides which task to schedule at each instant.

Fundamental to the functionality of a Real-Time Operating System is having a scheduler that re-evaluates which task should be currently running at a set rate. Thankfully, the Cortex-M4 provides a timer that can be used to perform this task, the `SYSTICK` timer. As previously mentioned, the `STOS_Init` function sets up the `SYSTICK` peripheral to call an interrupt every millisecond (which is configurably based on the processors clock speed). With each systick interrupt, the scheduler is called.

The initialization for the `SYSTICK` to occur millisecond is seen below:

```
// From interrupts.c

void SYSTICK_Config(void) {
    /*  SYSTICK CTRL: 31 - 0
        [0] -> Enable (counter loads reload to load and counts down)
        [1] -> Tick Int (counting down to zero asserts exception request)
        [2] -> Clk Source (AHB/8 or Processor clock (AHB) as clock source)
    */
    SYSTICK->CTRL &= ~(1UL << 0);  // Don't enable
    SYSTICK->CTRL |= 1UL << 1;     // Enable exception
    SYSTICK->CTRL |= 1UL << 2;     // Select processor clock

    /*  SYSTICK LOAD: 23 - 0
        [23:0] -> RELOAD Value
        If we want 1ms ticks, set to the maximum timer count value 15.9e3
    */
    SYSTICK->LOAD |= (CORE_FREQ / 10) - 1;  // Set the LOAD value

    SYSTICK->VAL &= ~(0xFFFFFFUL);  // Clear VAL value= 0;

    SYSTICK->CTRL |= 1UL;  // Enable SYSTICK
}
```

With each `SYSTICK`, a handler function gets called the decrements the sleep and timeout of tasks that are blocked, and then calls the scheduler.

```
// From stos.c

void sys_tick_handler(void) {
    if (stos_ker.active_task->sleep > 0) stos_ker.active_task->sleep--;

    stos_tcb_t *runner = stos_ker.list_blocked_head;
```

```
    while (runner != NULL) {
        stos_tcb_t **head = &runner;
        (*head)->timeout--;
        if ((*head)->timeout == 0) {
            STOS_RemoveTask((*head));
            STOS_AddTask((*head), TASK_READY);
        }
        runner = runner->next;
    }

    STOS_Schedule();
}
```

5.2 Beginning the RTOS

After completing the initialization, calling `STOS_Run` will begin the RTOS operation. It loads the context frame of the `active_task`, updates the `PSP` to reflect that value, and then performs an `SVC` call to switch execution to the process stack pointer, which should be the context of the `active_task`.

Once this routine has completed, the processor will shift execution to the function defined within the `active_task`, performing that operation until getting interrupted by the `SYSTICK` timer, which will call the scheduler to evaluate which task should continue.

5.3 STOS Scheduler

The scheduler itself is quite simple, it compares the priority of the current task (assuming it hasn't been forced out of the running state) and compares it with the head of the ready list. If the ready list has a higher priority or an equal one, it will be the next task to run. In this sense, it maintains a form of round-robin scheduling between tasks that share the highest priotiy. It will cycle between these tasks with fair time slicing, based on when the task was last scheduled. To switch out the running task, it's necessary to set the `next_task` within the kernel to point to the task that will replace the running one - the rest will be handled within a `PENDSV` exception that the scheduler will call if it determines that the running task needs to be changed.

5.4 Context Backup/Restoration Routines

If the active task has been scheduled out, the scheduler will trigger a `PENDSV` exception, which is the lowest priority interrupt within the `NVIC`. Once triggered the `pend_sv_handler` will go about swapping the active task with the next task, adjusting the stack pointers to match. The mechanism for doing so consists of saving the context of the `active_task` and then updaing the stack pointer of

the `active_task`. It will then swap the `active_task` with the `next_task` and restore the context of the "new" `active_task` (which was the `next_task`).

6 Synchronization Primitives

6.1 Atmoic Memory Access

To perform atomic operations on memory, the Cortex-M4 provides two unique instructions that guarantee atomicity - `LDREX` and `STREX`. The `LDREX` instruction loads a word from ememory and initalizes a hardware monitor keep track of any modifcations made to it. A following `STREX` instruction performs a conditional store based on the condition that nothing has modified the word since the previous `LDREX` instruction [7].

6.2 Mutexes

A mutex is defined as follows:

```
// From sync.h

typedef struct stos_mutex {
    uint32_t lock;
} stos_mutex_t;
```

It contains three functions which modify it, `STOS_MutexTryLock`, `STOS_MutexLock`, and `STOS_MutexUnlock`. Under the hood, all three utilize `__stos_check_lock` which performs a `LDREX` an returns the result - a 0 if unlocked and 1 if locked. Likewise, the `__stos_write_to_lock` performs an `STREX` and returns a 0 if succseful and 1 if failed.

`STOS_MutexTryLock` function first checks the lock, tries to write to it, and returns the result (0/1). `STOS_MutexLock` is a blocking lock, which continually calls `STOS_MutexTryLock` until successful. Naturally, there needs to be a method to unlock the mutex, which `STOS_MutexUnlock` provides by resetting the value of the lock to 0.

An interesting thing to note, is that the `STREX` instruction can fail! That's why when unlocking the mutex, it's necessary to first load the value once more, because the outcome of `STREX` requires no modifications since the previous `LDREX`, which is unlikely to be true when trying to unlock a mutex. Thus a "useless" `LDREX` instruction is wasted just to validate the following `STREX`.

6.3 Semaphores

Semaphores share the same structure as mutexes, just with an additional `typedef`. In this context, the only practical difference between sempahores and mutexes is that a semaphore can be initialized to a value larger than 1, while a mutex will cap at 1. This is reflected by the addition of a `STOS_SemInit` function which allows the user to set the initial value of the sempahore. The `STOS_SemWait` function is

a blocking wait that will decrement the value of the sempahore, assuming that it is greater than 0, or continually wait for the value to become greater than 0 and then perform the decemrent. On the otherhand, the `STOS_SemPost` function will check the lock and increment that value by 1.

7 USART Debug Ouput

The STM32F446RE features four Universal Synchronous/Asynchronous Receiver Transmitters (USART) and two Universal Asynchronous Receiver Transmitters (UART) [sec 3.23 datasheet]. By default the USART peripherals operate at the internal 16 MHz [sec 2.3 ref manual]. It is important to note that the USART2 interface is available on the PA2 and PA3 of the STM32 Microcontroller, which is by default connected to the ST-LINK MCU used in our ST-LINK USB Connection.

As the USART2 peripheral is part of the Advanced Peripheral Bus 1 (APB1) matrix, it lies in the memory address (0x40000000) [sec 2.2.2 ref manual]:

```
// From cm4_periphs.h
#define PERIPH_BASE     (0x40000000UL)
#define APB1PERIPH_BASE (PERIPH_BASE)
```

Thus, the USART2 peripheral resides in the 0x4000 4400 - 0x4000 47FF region in memory provided by the Register Boundary Address table [Table 1 ref manual].

The USART register map is as follows:

Offset	Register
0x00	USART_SR (Status Register)
0x04	USART_DR (Data Register)
0x08	USART_BRR (Baud Rate Register)
0x0C	USART_CR1 (Control Register 1)
0x10	USART_CR2 (Control Register 2)
0x14	USART_CR3 (Control Register 3)
0x18	USART_GTPR (Guard Time and Prescaler Register)

The following equations from STM32CubeIDE driver code & [sec. 25.4.4 reference manual] were used to calculate the register values from the baud rate and process clock running at 16 MHz:

$$
\begin{aligned}
USART_{DIV} &= \frac{PCLK \times 25}{4 \times BAUDRATE} \\
USART_{DIV\,MANTISSA} &= \frac{USART_{DIV}}{100} \\
USART_{DIV\,FRACTION} &= \frac{((USART_{DIV} - (USART_{DIV\,MANTISSA} \times 100)) \times 16) + 50}{100} \\
BRR &= Mantissa + Overflow + Fraction \\
USART_{BRR\,DIV} &= (USART_{DIV\,MANTISSA} << 4) \\
&\quad + (USART_{DIV\,FRACTION}\,\&\,0xF0) \\
&\quad + \ldots \\
&\quad + (USART_{DIV\,FRACTION}\,\&\,0x0F)
\end{aligned}
$$

The following procedure is executed to configure the USART transmitter:

- The USART2 APB1 peripheral clock must be enabled through the `RCC->APB1ENR` register.
- The PA2 and PA3 and GPIO pins must be anbled to the alternate function 7 pin-mode, which is used for the USART2 peripheral on the SMT32F446RE.
- The baud rate frequency is written onto the `USART_BRR` register.
- Set the `USART_ENABLE (UE)`, `RECIEVER_ENABLE (RE)`, and `TRANSMITTER_ENABLE (TE)` bits in the Control 1 register.

Bit 7 or the Transmit Data Register Empty (TXE) bit in the Status Register of the USART device can be used to transmit bytes from the STM32. This bit is set by hardware when the content from the TDR has been transferred into the shift register, effectively notifying us when the USART transaction is complete.

The following function transmits a byte:

```
// From usart.c
void USART_transmit_byte(USART_t *port, uint8_t byte)
{
    while(!(port->USART_SR & USART_SR_TXE))
    {
        (void) 0;
    }
    port->USART_DR = byte;
}

// Note: This is not an interrupt driven implementation
```

Furthermore, the _write system call is modified to redirect printf() to our USART device, ultimately allowing us to print information from the mcu to our main computer device over serial. Currently, the USART device is configured at

a 115200 baud rate, and serial information is read from TIO, a serial terminal I/O tool. Simple usage of the TIO tool is as follows: `$ tio /dev/ttyACM0 -b 115200`

More information about tio can be found here [7].

8 References

[1] https://www.st.com/en/evaluation-tools/nucleo-f446re.html

[2] https://www.st.com/resource/en/user_manual/um1724-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf

[3] https://github.com/STMicroelectronics/cmsis-device-f4/tree/cdbad761857acedcdd07ece7939b4cb209ed826a

[4] https://www.st.com/resource/en/programming_manual/pm0214-stm32-cortexm4-mcus-and-mpus-programming-manual-stmicroelectronics.pdf

[5] https://developer.arm.com/documentation/107656/0101/Getting-started-with-Armv8-M-based-systems/Procedure-Call-Standard-for-Arm-Architecture--AAPCS-

[6] https://github.com/QuantumLeaps/modern-embedded-programming-course/blob/main/lesson-26/tm4c123-gnu/miros.c

[7] https://developer.arm.com/documentation/dht0008/a/arm-synchronization-primitives/exclusive-accesses/ldrex-and-strex

[8] Joseph Yiu. 2009. The Definitive Guide to the ARM Cortex-M3, Second Edition (2nd. ed.). Newnes, USA.