PART 1 – INTER-TASK SYNCHRONISATION

"Finally, we learned the art of reasoning by which we could prove that the society composed of processes thus mutually synchronised by each other would indeed in its time behaviour satisfy all requirements."
(The structure of "THE" Multiprogramming System, Dijkstra, 1968)

OUTLINE

After understanding the abstraction of a process running multiple threads and relating it to the underlying hardware via an execution image, we were able to create a (rather simple) multitasking mechanism – we are multiplexing CPU time. Besides CPU time, we need to share data, and hardware resources. Synchronisation mechanisms are about that.

In this text the term process, task and thread may be used interchangeably meaning a concurrent unit.

1.0.THE NEED FOR SYNCHRONISATION

On Part 0 we have split a process on several threads, now they are spinning over the processor, from power-up to power-down, but what if we want them to cooperate to perform a computation?

Synchronisation points are those where tasks will share information and/or ensure that certain preconditions are satisfied before proceeding [1].

On Figure 1 the synchronisation barrier only allows the program to proceed when all processes have reached that point – the conditions for all system actions to be correctly synchronised are true.

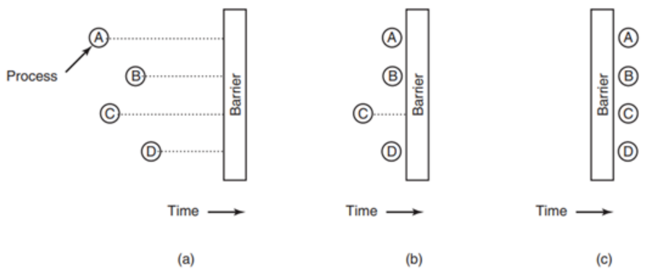


Figure 1. a) Running processes b) All processes except C are synchronised c) Once process C reaches the synchronisation point, the program moves on. (Figure from [2])

1.1. SOME KERNEL DESIGN IMPROVEMENTS

In the code snippets provided, the rule is to avoid convoluted code, to convey the core idea. As a result, certain good practices for production-grade code, like returning values for every operation (and checking return values), have been omitted. In situations when conditions are unhandable or would cause a hardware fault, I call an ErrorHandler() function.

Before presenting the synchronisation mechanisms, some adjustments on the kernel design are needed to accommodate these mechanisms. The modifications are as follows:

1.1.1. Task States

Any task can assume one of the following states:

(1) READY: It is waiting to be dispatched. If a task is running and is preempted or yields voluntarily, it will be set as READY.
(2) SLEEPING: It is blocked by a Sleep call (on an *event)*, waiting to be released by a Wake call.
(3) BLOCKED: It is blocked by a locking protocol, such as a Semaphore or a Mutex. A differentiation between SLEEPING and BLOCKED is not necessary but increases readability and debuggability. They could be joined on a single state called WAITING, for instance.
(4) RUNNING: Using the CPU.

1.1.2. A Ready Queue

Queues (or lists) are fundamental data structures to manage tasks on an operating system kernel. We will have a global *Ready Queue*, on which tasks ready to be dispatched will be placed. This queue will be a singly linked list. The first task to enter the ready queue will be the first to be dispatched when it is time. Two operations are performed on this queue:

(1) Enqueue: insert a TCB on the queue tail.
(2) Dequeue: removes a TCB from the queue head.

Data structures are out of the scope of this article, but it is worth mentioning that the key idea behind linked lists is the manipulation of the nodes' addresses. Each node comprises a data element and a pointer to the next node in the sequence (Figure 2). This structure enables the creation of ordered queues by altering the pointers associated with the queue head, tail, and the *next* pointer of each node. The head, tail and all the TCBs addresses are on the *.bss* memory segment of our execution image. (As you can see, nodes of a dynamic data structures don't have to be dynamically allocated on the heap, by using *malloc* – an approach overly emphasized on data structure books.)

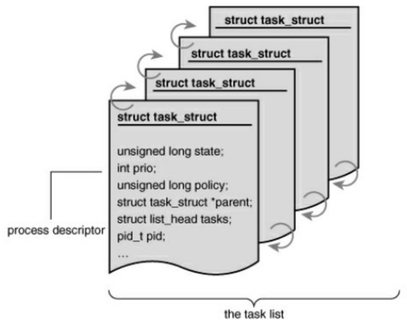


Figure 2 Linked-List of TCBs on Windows NT [6]

1.1.3. An IDLE task

An ***IDLE*** task is convenient for many reasons, and if you don't have room for running an idle task on a preemptive kernel - red flag - you are working with a very high CPU load. To our purposes, when all tasks are blocked, the idle task will run and put the CPU on a low-power mode waiting for an interruption, instead of letting the kernel spinning over the scheduler waiting to get a task ready. So far, the only interrupt we are dealing with is our own system tick. The idle task never blocks, it is always ready or running. In the implementation shown here it is not placed on the ready queue with other regular tasks, for easier management: if the ready queue is empty, the idle task takes over. When the system has task priorities, naturally the idle task will have the lowest priority. Listing 1 shows the proposed changes.

```
/**
* @file tcbs.h
*/
#ifndef TCB_H_
#define TCB_H_
#include <stdint.h>
#define NTHREADS 4
#define STACKSIZE 64
typedef enum
{
        READY,
        RUNNING,
        SLEEPING,
        BLOCKED
} TASK_STATUS_t;

typedef uint32_t EVENT_t;
struct tcb /*thread control block*/
{
        int32_t* sp;
        struct tcb* next;
        TASK_STATUS_t  status;
        EVENT_t event;
        uint8_t pid;
};
typedef struct tcb TCB_t;


/*
* @file kernel.c
*/
TCB_t* readyQueuePtr=NULL;

/*
* @brief     get a tcb from the head of the ready queue
* @param     queuePPtr - pointer to the queue head address (a pointer-to-pointer)
* @retval    the address of the element on the head,
*            NULL if empty
*/
TCB_t* Dequeue(TCB_t** queuePPtr)
{
        if (*queuePPtr == NULL)
        {
                return NULL;
        }

        TCB_t* headPtr = *queuePPtr;
        *queuePPtr = headPtr->next; // moving the head to the next element
        return headPtr;
}
/*
* @brief            Place an element on the tail of the queue
* @param queuePPtr  pointer to the queue address (a pointer-to-pointer)
* @param tcbPtr     pointer to the TCB to be added
* @retval            none
*/
void Enqueue(TCB_t** queuePPtr, TCB_t* tcbPtr)
{
        TCB_t* tailPtr = NULL;
        if (*queuePPtr == NULL) /* queue is empty*/
        {
                *queuePPtr = tcbPtr;
```

```c
        }
        else
        {
                tailPtr = *queuePPtr;
                while (tailPtr->next != NULL)
                {
                        tailPtr = tailPtr->next;
                } /*found tail*/
                tailPtr->next = tcbPtr;
        }
        tcbPtr->next = NULL; /* tcbPtr is the last element now*/
}

/*
* @brief           Set the initial task stack
* @param i         task id
* @param taskPtr   pointer to the main task function
* @retval          none
*/
void SetInitStack(uint32_t i, TaskPtr taskPtr)
{
        tcbs[i].sp = &stacks[i][STACKSIZE - R4_OFFSET];
        stacks[i][STACKSIZE - PSR_OFFSET] = 0x01000000; //**PSR**
        stacks[i][STACKSIZE - PC_OFFSET] = (int32_t)taskPtr; //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
        tcbs[i].status = READY;
        tcbs[i].pid = i;
}


#define IDLE_TASK     &tcbs[0]
#define IDLE_TASK_PID 0
/*
* @brief Called by the SysTick handler to switch tasks
* @retval none
*/
void TaskSwitch(void)
{
    TCB_t* runPtrNext = NULL;
    if ((runPtr->status == RUNNING))
    {
        runPtr->status = READY;
        if (runPtr->pid != IDLE_TASK_PID)
            Enqueue(&readyQueuePtr, runPtr);
    }
    if (readyQueuePtr == NULL)
    {
        runPtrNext = IDLE_TASK;
    }
    else
    {
        runPtrNext = Dequeue(&readyQueuePtr);
    }
    runPtr = runPtrNext;
    runPtr->status = RUNNING;
}
/*
* @file tasks.c
*/
void IdleTask(void)
{
       while (1)
       {
              __DSB();
              __WFI();
              __ISB();
       }
}
/*
* @file system.s
*/
SysTick_Handler:
CPSID I
PUSH {R4 - R11}
LDR R0, =runPtr
LDR R1, [R0]
STR SP, [R1]
PUSH {LR}
BL  TaskSwitch
POP {LR}
LDR R0, =runPtr
LDR R1, [R0]
LDR SP, [R1]
POP {R4 - R11}
CPSIE I
BX LR
```

Listing 1. Kernel improvements to support synchronisation mechanisms

Regarding the IDLE task implementation, before entering low-power mode by calling the instruction WFI (*Wait For Interrupt)*, it synchronises bus data by using the DSB (*Data Synchronisation Barrier)*. It is a good practice to force the CPU to use a Data Barrier, to ensure no pending operation on data is left out *before* entering a low-power mode. The ISB instruction (*Instruction Set Barrier*) guarantees instructions to be fetched *after* leaving low-power mode, will be executed *after* the barrier. The CPU can be mischievous trying to optimize this fetching on the pipeline. Also, make sure your debugger is configured to work when the CPU is on low-power mode; otherwise, it might fail, complaining the CPU is not responding.

The SysTick Handler is now split in two parts. For convenience, one part is implemented in C. The assembly code calls the *TaskSwitch* function. Before branching to another function, we should push the registers which data might be changed by the called function, and when returning we pop it, to get our local data back. The ARM Procedure Call Standard (AAPCS) defines the following:

(1) When a function (caller) calls another function (callee), the callee expects its arguments to be in R0-R3.
(2) Likewise, the caller expects the callee return value to be in R0. R4-R11 must be preserved between calls.
(3) R12 is the scratch register and can be freely used.

Why am I saving only LR then? Well, LR needs to be preserved so the interrupt handler returns correctly. The other registers can be modified with no harm in this specific code because we are not relying on them to anything else – we would waste CPU time by saving and restoring them, and as a rule for interrupt handlers, the shorter the better. The context of the task running before the tick interrupt is already saved. We get the next stack pointer and return from the interrupt.

When starting, we need to initialize the ready queue, and dequeue the first task to be dispatched, assigning it to *runPtr*. Note the task queue is not a circular linked-list anymore, it is a regular singly linked-list: the last element points to null rather than pointing back to the first. Finally, on the start-up routine we need to assign the RUNNING status to the dispatched task (Listing 2).

```
/*
* @file main.c
*/

SetInitStack(1, Task0);
SetInitStack(2, Task1);
SetInitStack(3, Task2);
SetInitStack(0, IdleTask);
for (size_t i = 1; i < NTHREADS; i++) // skip task id 0
        Enqueue(&readyQueuePtr, &tcbs[i]);
runPtr = Dequeue(&readyQueuePtr);
StartUp();
while (1);

 /*
* @file system.s
*/
StartUp :
LDR     R0, =runPtr
LDR     R1, [R0]
LDR     SP, [R1]
POP    {R4 - R11}
POP     {R0 - R3}
POP     {R12}
POP    {LR}
MOV     R12, #0x01 //RUNNING==1
LDR     R0, =runPtr
LDR     R1, [R0]
STR     R12, [R1, #8] //status is the 3rd field on the TCB
CPSIE  I
POP     {PC}
```

Listing 2 Modifications on the start-up code

1.2.SLEEP/WAKE-UP

The simplest form of task synchronisation is to sleep/wake-up on *events*, as used on the original UNIX kernel [3]. The concept of an event is quite loose. It can be defined as "an occurrence that requires a system reaction" [5]. In [3] it is defined as "a '*thing'* that is generated by a source and recognized by a recipient, causing the latter to take action to handle the event."

Anyway, it is an essential "*thing"* on embedded systems, because instead of continually checking for inputs, a system can be designed to be event-driven, i.e., it acts only in response to events. Such a system is said to be *reactive.*

Back to the Sleep/Wake-up mechanism, when a task is waiting for a condition to be true – e.g., a resource to be available - it suspends itself (goes to sleep) until another part of the code (not within the same task), such as another task or an interrupt handler, wakes it up when the condition is met.

The algorithm of the sleep/wake-up mechanism, is as shown on Listing 3 [3]:

```
Sleep(EVENT)
{
        record EVENT value in TCB event field;
        change thread status from RUNNING to SLEEPING;
        call task switcher;
}

Wake(EVENT)
{
        for every task do:
        {
                if ((task->status == SLEEPING) AND (task->event == EVENT))
                {
                        task->status = READY;
                        task->event = 0;
                }
        }
}
```

Listing 3. Sleep/Wake-up algorithm.

Note that when a *Wake* operation is called on an event, every task that is sleeping on that event will be set as *READY*. These operations must be atomic.

A note for the reader: the mechanisms presented here are not optimised for real-time systems (although I bet, they'd be enough for many applications). We are exploring general concepts first. Later, after discussing real-time concepts, we will adapt the machinery as possible.

1.2.1. Implementation

The Sleep/Wake-up functions are shown on Listing 4. EnterCR() and ExitCR() (enter/exit critical region) disable and enable all interrupts, respectively, so the operations are atomic.

```
#define EnterCR()      __disable_irq()
#define ExitCR()       __enable_irq()

void ErrorHandler(void)  /* generic error handler */
{
  __disable_irq();
  while (1);
}
/**
 * @brief Sleep on an event
 * @param event Event number (unsigned 32-bit integer)
*/

void Sleep(EVENT_t event)
{
        if (event == 0)
                ErrorHandler();
        EnterCR();
        runPtr->event = event;
        runPtr->status = SLEEPING;
        Yield();/* switch task */
        ExitCR();
}
/**
 * @brief Wake all tasks that are sleeping on an event
 * @param event Event number (unsigned 32-bit integer)
*/
void Wake(EVENT_t event)
{
        if (event == 0)
                ErrorHandler();
        EnterCR();
        for (int i = 1; i < NTHREADS; i++)
        {
                if (tcbs[i].event == event && tcbs[i].status == SLEEPING)
                {
                        tcbs[i].event = 0;
                        tcbs[i].status = READY;
                       Enqueue(&readyQueuePtr, &tcbs[i]);
                }
        }
        ExitCR();
}
```

Listing 4. Sleep/Wake-up implementation.

1.2.2. Example of Sleep/Wake-up synchronisation

As a smoke test for the synchronisation with Sleep/Wake-up, the following is proposed:

(1) Task0 and Task1 start and sleep to an event that will happen on Task2.
(2) This event is simply that counter2 has reached the value of 90000.

Listing 5 shows the implementation.

```
volatile uint32_t counter0;
volatile uint32_t counter1;
volatile uint32_t counter2;
EVENT_t reachCountEvent = 0xAABBCCDD; /*an event is just a number! there is no
meaningful value*/
void Task0(void* args)
{
        while (1)
        {
                Sleep(reachCountEvent);
                counter0++;
        }
}

void Task1(void* args)
{
        while (1)
        {
                Sleep(reachCountEvent);
                counter1++;
        }
}
void Task2(void* args)
{
        while (1)
        {
                counter2++;
                if (counter2 == 90000)
                {
                        Wake(reachCountEvent);
                }
        }

}
```

Listing 5. Smoke-test for the Sleep/Wake-up synchronisation

It is worth noting two things:

(1) The event value (0xAABBCCDD) is random. There is no meaning in it, it is just a number.
(2) *When* counter2 reaches 90000 it will wake-up Task0 and Task1 altogether. Then, when dispatched, these tasks will increment their counters and given the loop, will sleep on the event again. They will wake-up another time – but not too soon - when counter2 overflows and starts from 0, reaching 90000 again. This cycle repeats.

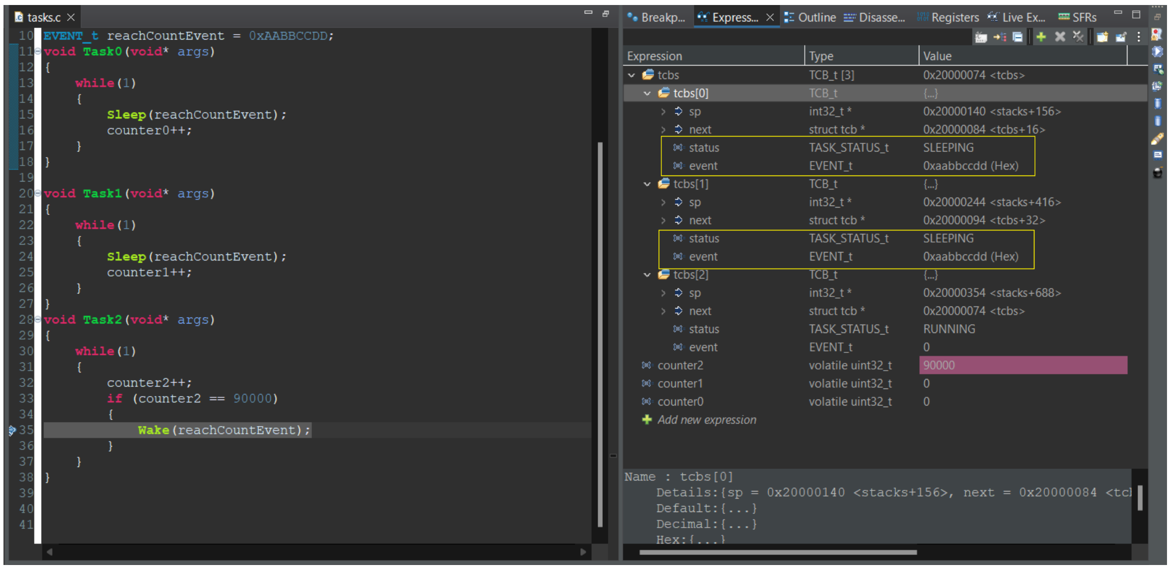
Let's see it running. I have placed the breakpoint exactly when counter2 reaches 90000. You can see on the right, that Task0 and Task1 are still sleeping (Figure 3).

Stepping into Wake, it will make Task0 and Task1 *READY (*Figure 4*)*.

Finally, letting the system run, counter0 and counter1 are incremented only once, and Task0 and Task1 are sleeping again (Figure 5).

1.2.3. Shortcomings of Sleep/Wake-up

In practice, this synchronisation primitive is rarely used as presented on embedded systems software. Normally an RTOS API will have a **Sleep(ticks)** function just to suspend a task for an amount of time.



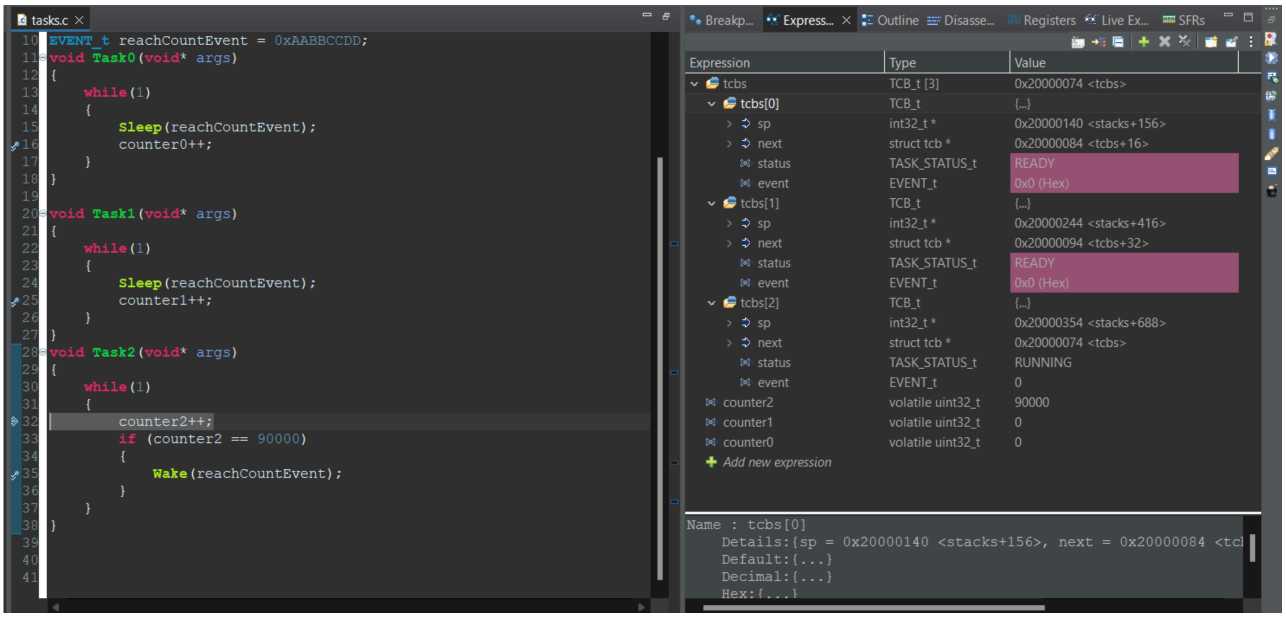
```
EVENT_t reachCountEvent = 0xAABBCCDD;
void Task0(void* args)
{
    while(1)
    {
        Sleep(reachCountEvent);
        counter0++;
    }
}

void Task1(void* args)
{
    while(1)
    {
        Sleep(reachCountEvent);
        counter1++;
    }
}
void Task2(void* args)
{
    while(1)
    {
        counter2++;
        if (counter2 == 90000)
        {
            Wake(reachCountEvent);
        }
    }
}
```

Expression	Type	Value
tcbs	TCB_t [3]	0x20000074 <tcbs>
tcbs[0]	TCB_t	{...}
sp	int32_t *	0x20000140 <stacks+156>
next	struct tcb *	0x20000084 <tcbs+16>
status	TASK_STATUS_t	SLEEPING
event	EVENT_t	0xaabbccdd (Hex)
tcbs[1]	TCB_t	{...}
sp	int32_t *	0x20000244 <stacks+416>
next	struct tcb *	0x20000094 <tcbs+32>
status	TASK_STATUS_t	SLEEPING
event	EVENT_t	0xaabbccdd (Hex)
tcbs[2]	TCB_t	{...}
sp	int32_t *	0x20000354 <stacks+688>
next	struct tcb *	0x20000074 <tcbs>
status	TASK_STATUS_t	RUNNING
event	EVENT_t	0
counter2	volatile uint32_t	90000
counter1	volatile uint32_t	0
counter0	volatile uint32_t	0
Add new expression		

```
Name : tcbs[0]
    Details:{sp = 0x20000140 <stacks+156>, next = 0x20000084 <tc
    Default:{...}
    Decimal:{...}
    Hex:{...}
```

Figure 3. Running the Sleep/Wake-up synchronisation smoke-test



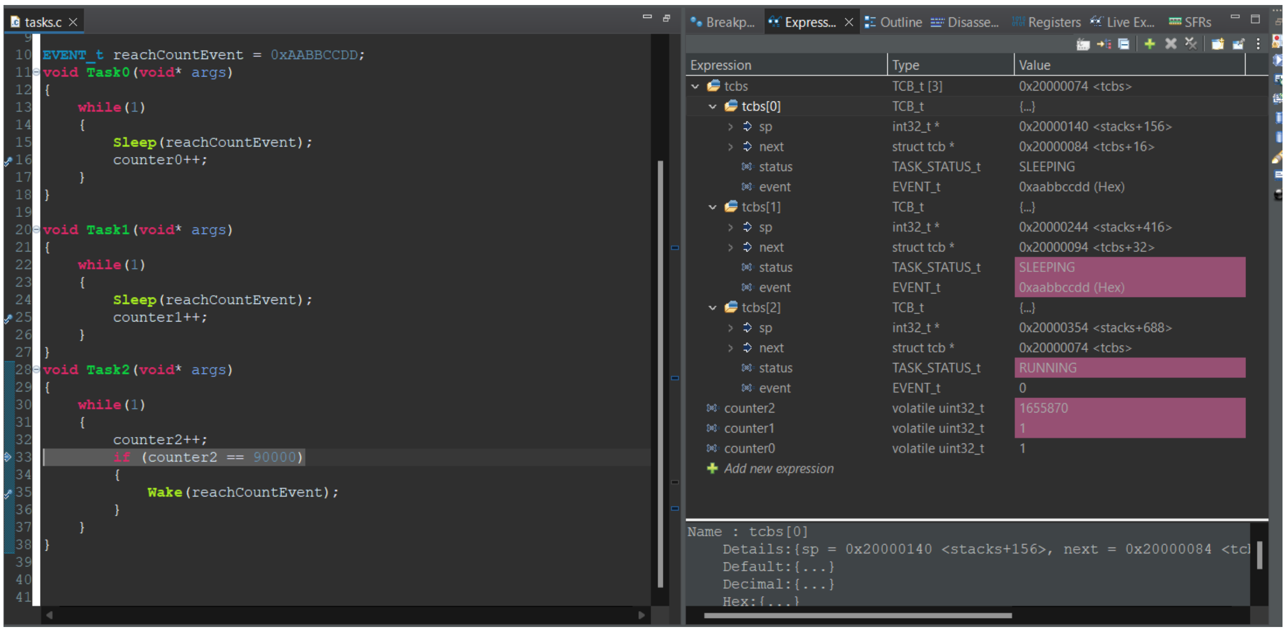
```
EVENT_t reachCountEvent = 0xAABBCCDD;
void Task0(void* args)
{
    while(1)
    {
        Sleep(reachCountEvent);
        counter0++;
    }
}

void Task1(void* args)
{
    while(1)
    {
        Sleep(reachCountEvent);
        counter1++;
    }
}
void Task2(void* args)
{
    while(1)
    {
        counter2++;
        if (counter2 == 90000)
        {
            Wake(reachCountEvent);
        }
    }
}
```

Expression	Type	Value
tcbs	TCB_t [3]	0x20000074 <tcbs>
tcbs[0]	TCB_t	{...}
sp	int32_t *	0x20000140 <stacks+156>
next	struct tcb *	0x20000084 <tcbs+16>
status	TASK_STATUS_t	READY
event	EVENT_t	0x0 (Hex)
tcbs[1]	TCB_t	{...}
sp	int32_t *	0x20000244 <stacks+416>
next	struct tcb *	0x20000094 <tcbs+32>
status	TASK_STATUS_t	READY
event	EVENT_t	0x0 (Hex)
tcbs[2]	TCB_t	{...}
sp	int32_t *	0x20000354 <stacks+688>
next	struct tcb *	0x20000074 <tcbs>
status	TASK_STATUS_t	RUNNING
event	EVENT_t	0
counter2	volatile uint32_t	90000
counter1	volatile uint32_t	0
counter0	volatile uint32_t	0
Add new expression		

```
Name : tcbs[0]
    Details:{sp = 0x20000140 <stacks+156>, next = 0x20000084 <tc
    Default:{...}
    Decimal:{...}
    Hex:{...}
```

Figure 4. Sleep/Wake-up smoke-test: tasks waking to an event



```
EVENT_t reachCountEvent = 0xAABBCCDD;
void Task0(void* args)
{
    while(1)
    {
        Sleep(reachCountEvent);
        counter0++;
    }
}

void Task1(void* args)
{
    while(1)
    {
        Sleep(reachCountEvent);
        counter1++;
    }
}
void Task2(void* args)
{
    while(1)
    {
        counter2++;
        if (counter2 == 90000)
        {
            Wake(reachCountEvent);
        }
    }
}
```

Expression	Type	Value
tcbs	TCB_t [3]	0x20000074 <tcbs>
tcbs[0]	TCB_t	{...}
sp	int32_t *	0x20000140 <stacks+156>
next	struct tcb *	0x20000084 <tcbs+16>
status	TASK_STATUS_t	SLEEPING
event	EVENT_t	0xaabbccdd (Hex)
tcbs[1]	TCB_t	{...}
sp	int32_t *	0x20000244 <stacks+416>
next	struct tcb *	0x20000094 <tcbs+32>
status	TASK_STATUS_t	SLEEPING
event	EVENT_t	0xaabbccdd (Hex)
tcbs[2]	TCB_t	{...}
sp	int32_t *	0x20000354 <stacks+688>
next	struct tcb *	0x20000074 <tcbs>
status	TASK_STATUS_t	RUNNING
event	EVENT_t	0
counter2	volatile uint32_t	1655870
counter1	volatile uint32_t	1
counter0	volatile uint32_t	1
Add new expression		

```
Name : tcbs[0]
    Details:{sp = 0x20000140 <stacks+156>, next = 0x20000084 <tc
    Default:{...}
    Decimal:{...}
    Hex:{...}
```

Figure 5. Task0 and Task1 woke-up and went to sleep again. Task2 is running.

Nevertheless, I chose to present Sleep/Wake-up on a good level of detail, because its simplicity straight demonstrates what a synchronisation tool is about, and most importantly, because its shortcomings can teach a lot about synchronisation issues and provide a good rationale for the idea of semaphores, and then mutexes, that will be presented later in this part.

As a synchronisation tool, sleep/wake-up has the following shortcomings [3]:

(1) An event is represented as a meaningless value. It does not have any memory location to record the occurrence of an event.

(2) Process must go to sleep first before another process or an interrupt handler tries to wake it up. The sleep-first-wakeup-later order can always be achieved in a

uniprocessor (UP) system but not necessarily in multiprocessor (MP) systems. In a MP system, processes may run on different CPUs simultaneously (in parallel). It is impossible to guarantee the execution order of the processes. Therefore, sleep/wakeup are suitable only for UP systems.

(3) When used for resource management, if a process goes to sleep to wait for a resource, it must retry to get the resource again after waking up, and it may have to repeat the sleep-wakeup-retry cycles many times before succeeding (if ever). The repeated retry loops mean poor efficiency due to excessive overhead in context switching.

Before presenting the next synchronisation primitive, let me introduce a classical synchronisation problem and how it can (*not*) be solved using sleep/wake-up.

1.3.THE PRODUCER-CONSUMER PROBLEM

A set of "producer" processes and a set of "consumer" processes communicate by means of inserting and extracting items on a buffer. The buffer has size N. There are two main criteria to achieve synchronisation [4]: producers cannot write on a full buffer and consumers cannot read from an empty buffer. That is for the number of inserted data $\boldsymbol{i}$ and extracted data $\boldsymbol{e}$: $\boldsymbol{0 < i - e < N}$. Furthermore, there must be some kind of mutual exclusion, so only one process can access the buffer at a time.

Using sleep/wake-up synchronisation to solve this problem, yields the following solution (Listing 6) [2]:

```
#define N 100 /* number of slots in the buffer */
int count = 0; /* number of items in the buffer */
EVENT_t eventSpace = 1;
EVENT_t eventItem  = 2;
void ProducerTask(void)
{
       int item;
       while (1)
       {
              item = ProduceItem() /* generate next item */
              if (count == N)
                Sleep(eventSpace); /* if buffer is full, go to sleep */
              InserItem(item); /* put item in buffer */
              count = count + 1; /* increment count of items in buffer */
              if (count == 1) /* was buffer empty? */
                Wake(eventItem);
       }
}
void ConsumerTask(void)
{
       int item;
       while (1)
       {
              if (count == 0)
                Sleep(eventItem); /* if buffer is empty, go to sleep */
              item = RemoveItem(); /* take item out of buffer */
              count = count - 1; /* decrement count of items in buffer */
              if (count == N - 1) /* was buffer full? */
                Wake(eventSpace);
              consume_item(item); /* print item */
       }
}
```

Listing 6. Pseudo-code for the producer-consumer problem using sleep/wake-up (Adapted from [2])

Although this solution seems neat, it has a race-condition [2]. Now, regarding software, a ***race condition*** occurs when multiple tasks or threads access shared resources concurrently without proper synchronisation, leading to unpredictable or erroneous behaviour due to the timing or interleaving of these accesses. This lack of synchronisation can result in conflicting operations when the outcome of a computation depends on the order or timing of execution.

Race conditions might not always be immediately evident and can lead to elusive bugs that are challenging to identify and reproduce during software development and testing. A synchronisation tool might not be able to prevent all race conditions, either because of its inherent construction or the way and the environment it is employed.

On the above algorithm proposal, the following race-condition exists [2]:

(1) The buffer is empty, and the consumer has just read *count*. At that instant, the scheduler decides to stop running the consumer and starts running the producer.
(2) The producer enters an item in the buffer, increments count and notices it is now 1. So, the consumer might be sleeping – because the counter was just 0.
(3) The producer calls wake to wake-up the consumer.
(4) This wake-up signal is lost because the consumer is not sleeping yet.
(5) When the scheduler dispatches the next consumer run, it will then test the *count* it previously read, and notice it is zero. So, it goes to sleep.
(6) The producer will eventually fill the buffer up and goes to sleep.
(7) Both will sleep forever, like Cinderellas.

1.4.SEMAPHORES

As pointed in section 1.2.2., item (1), an event is just a number it does not have any memory location to record the occurrence of an event. This fact leads that its usage on situations like the Producer-Consumer problem, a task *infers* the other is sleeping indirectly by the number of items on the buffer. In 1965, *Dijkstra* proposed the idea of semaphores [7], suggesting an integer variable to count the number of wake-ups saved for future use. A semaphore, then, could have the value of 0 indicating no wake-ups were saved, or some positive value, indicating one or more wake-ups are pending. The two operations on a semaphore are *Wait* and *Signal* – originally referred as P and V respectively – (because of some Dutch words). When a task calls *Wait* on a semaphore whose value is 0, it blocks on that semaphore until another task calls a *Signal* on that semaphore, releasing the blocked task.

Compared to Sleep/Wake-up, semaphores have the following advantages [3]:

(1) Semaphores combine a counter, testing the counter and making decision based on the testing outcome all in a single indivisible operation. The Signal operation unblocks only one waiting process, if any, from the semaphore queue. After passing through the Wait operation on a semaphore, a process is guaranteed to have a resource. It does not have to retry to get the resource again as in the case of using sleep and wakeup.

(2) The semaphore's value records the number of times an event has occurred. Unlike sleep/wakeup, which must obey the sleep-first-wakeup-later order, processes can execute SIGNAL/WAIT operations on semaphores in any order.

1.4.1. Semaphores Implementation

A *counting semaphore* is a signed integer variable. Every time it is affected by a Signal operation, it is increased. Every time it is affected by a Wait operation, it is decreased. A specialization of the counting semaphore is the binary semaphore, that is simply a semaphore which value is a Boolean, not a signed integer. A binary semaphore is mostly used for locking.

A task can be blocked on a single semaphore at a time, and many different tasks can be blocked on the same semaphore.

So, one could add a field on the TCB structure to record which semaphore the task is blocked on – just as we did for events. The standard approach though, is the semaphore to be implemented as a data structure containing a signed integer to represent its value, and a queue on which the blocked tasks are placed.

When not using an explicit queue, we will not have bounded waiting for the tasks blocked on a semaphore, since they will not necessarily be unblocked on a defined order: a signal operation when unblocking a task would fetch on the task queue for the very next task blocked on that semaphore. Therefore, the unblocking depends on the position of the task who is signalling the semaphore, which would increase the chances of a task starving. In [4] this simpler implementation is followed, and the author claims it works fine for a system with less than 20 threads, although no reasoning is presented.

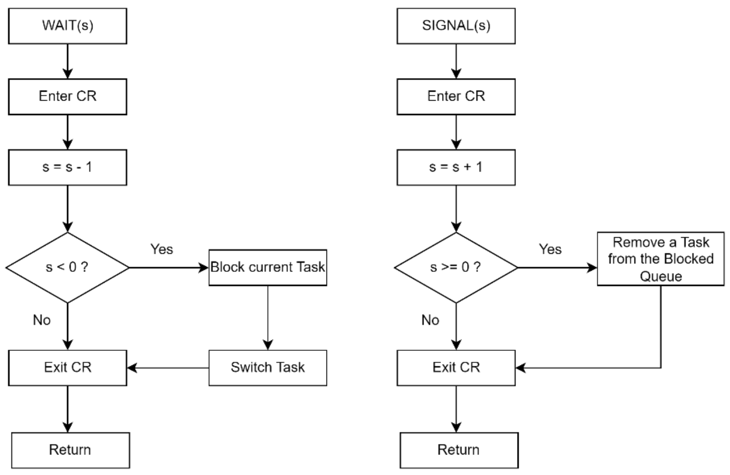


Figure 6 Signal and Wait operations on semaphores.

Again, the implementation depicted here does use a queue (Listing 7), and the first task to block on a semaphore will also be the first to be unblocked, independent of which task signalled the semaphore.

```
/**
```

```
 * @brief Counting Semaphore structure containing a value and
 *         a blocked queue
*/
typedef struct
{
       int32_t  value;
       TCB_t*   queuePtr;
} SEMA_t;
```

Listing 7 Data structures implementing a semaphore

The **Signal** and **Wait** operations are depicted on Figure 6. They are atomic, therefore are within critical regions (CR).

When a task calls a Wait on a semaphore and semaphore value assumes a number less than 0, the task has now the status *BLOCKED* and is placed on the blocked queue of that semaphore. If not, it just gets access to the resource/data guarded by that semaphore and moves on.

For a Signal operation, if incrementing the semaphore value results on a value equal or greater than 0, a task blocked on that semaphore will be granted access to that resource/data: it is removed from the blocked queue and is now *READY*.

Worth saying that some subtle variations can happen on the implementation of the Signal operation depending on the scheduling policy. On a priority-based scheduler for instance, after removing the task from the blocked queue and making it *READY*, the Signal operation might call a "reschedule" function, to reorder the Ready Queue – because if the just unblocked task has the highest priority amongst all the ready tasks, it might the one dispatched. The implementation shown (Listing 8) suffices the round-robin policy though, there is no need for a rescheduling.

```
/**
 * @brief Initialises a semaphore
 * @param self Semaphore address
 * @param value Initial value
*/
void SemaInit(SEMA_t* self, int32_t value)
{
    if (self == NULL)
        ErrorHandler();
    EnterCR();
    self->value = value;
    self->queue = NULL;
    ExitCR();
}
/**
 * @brief Wait on a semaphore
 * @param self Semaphore address
*/
void SemaWait(SEMA_t* self)
{
    if (self == NULL)
        ErrorHandler();
    EnterCR();
    (self->value) = (self->value) - 1;
    if ((self->value) < 0)
    {
        runPtr->status = BLOCKED;
        Enqueue(&self->queuePtr, runPtr);
        Yield();
    }
    ExitCR();
}

/**
 * @brief Signals a semaphore
 * @param self Semaphore address
*/
void SemaSignal(SEMA_t* self)
{
    if (self == NULL)
        ErrorHandler();
    EnterCR();
    TCB_t* nextTcbPtr;
    (self->value) = (self->value) + 1;
    if ((self->value) <= 0)
    {
        nextTcbPtr = Dequeue(&self->queuePtr);
        nextTcbPtr->status = READY;
        Enqueue(&readyQueuePtr, nextTcbPtr);
    }
    ExitCR();
}
```

Listing 8 Semaphores Signal and Wait implementation

The operations Signal and Wait mirrors what was depicted on Figure 6. Before being used, each semaphore must be initialized using the Init function, that sets its initial value and initialise the blocked queue.

1.4.2. Synchronisation with semaphores

The simplest form of synchronisation is that a process A should not proceed beyond a point Ll until some other process B has reached L2 [4]. Examples of this situations arise whenever A requires information at point L1 which is provided by B when it reaches L2. The synchronisation of the two processes below is achieved if ***proceed*** is a semaphore initialized as 0 (Figure 7).

It can also be said these processes are *cooperating* through semaphores. Another common term used when two tasks synchronise is to say they are performing a *rendezvous*.

One might naively think that to replace the Sleep/Wakeup example of section 1.1.2. to achieve the same synchronisation with Semaphores would be as on Listing 9.

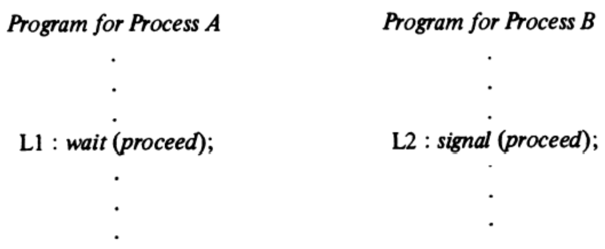


Figure 7 Semaphores synchronizing two processes.

```
volatile uint32_t counter0;
volatile uint32_t counter1;
volatile uint32_t counter2;
SEMA_t reachCountSema;
void Task0(void* args)
{
      SemaInit(&reachCountSema, 0);
```

```
        while (1)
        {
                SemaWait(&reachCountSema);
                counter0++;
        }
}

void Task1(void* args)
{
        while (1)
        {
                SemaWait(&reachCountSema);
                counter1++;
        }
}
void Task2(void* args)
{
        while (1)
        {
                counter2++;
                if (counter2 == 90000)
                {
                        SemaSignal(&reachCountSema);
                }
        }
}
```

Listing 9 Wrong attempt of replacing sleep/wake-up with semaphores

But it is not. As mentioned, different from sleep/wake-up that a Wake call wakes-up all tasks sleeping on that event, the Signal operation unblocks only one task. Since the tasks are being taken off from the blocked queue on a first-in first-out fashion, the signal operation on Task2 only unblocks the Task0, since it is always the first one to block on the semaphore and only one Signal operation exists. So, the following behaviour is observed (Figure 8). Note, on the right, only counter0 was incremented.

Therefore, to achieve the same behaviour, we would need two distinct semaphores or to signal the same semaphore twice.

1.4.3. Resource Management with Semaphores

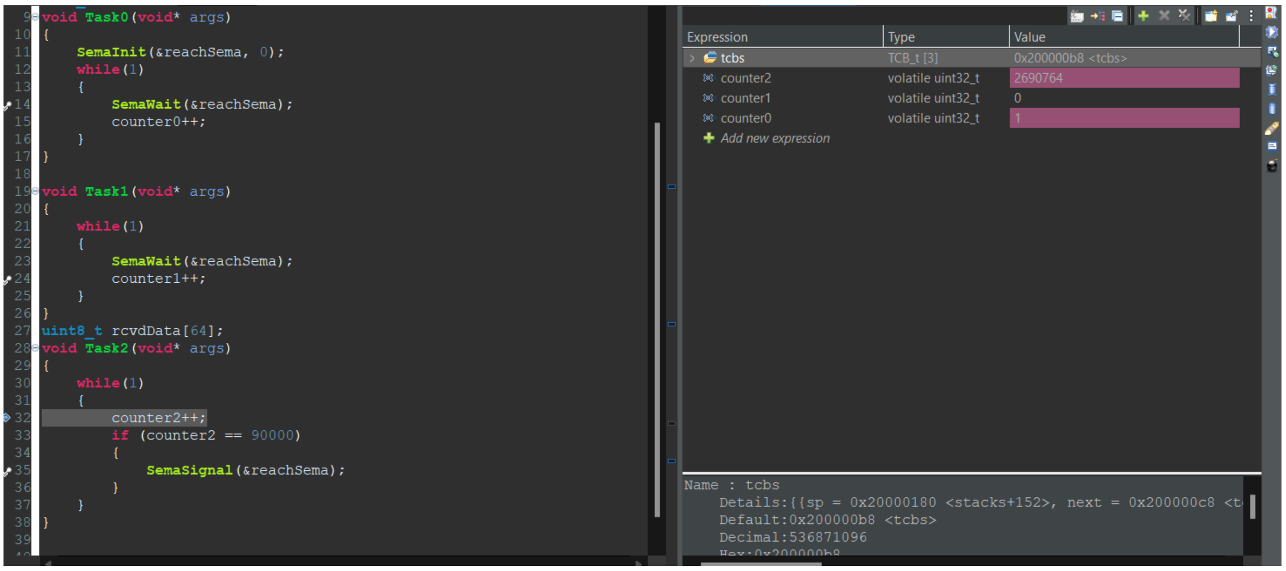
Having a signed integer as a value for a counting semaphore, a given resource might have N slots and use a semaphore initialized as N (Listing 10). To manage the resource access on time, every task will Wait on a semaphore related to that resource when trying to access it.

If the result value is more than or equal to zero, this task is granted access to the resource. If not, it will be placed on the blocked queue, waiting for a task to finish using the resource and Signal the related semaphore. A negative semaphore value, say, -2, means there are two tasks on the waiting queue.

1.4.4. Locking Critical Regions with Semaphores

I have already mentioned the term “critical region” (CR) several times, and you might already have grasped its meaning. So far, CRs are those code parts where all interrupts must be disabled, to perform operations that must be indivisible. A more general definition is as follows:

Non-shareable resources, whether peripherals, files, or data in memory, can be protected from simultaneous access by several processes by preventing the processes from concurrently executing the pieces of program through which access is made. These pieces of program are called **critical sections** or **critical regions** [4].



```c
void Task0(void* args)
{
    SemaInit(&reachSema, 0);
    while(1)
    {
        SemaWait(&reachSema);
        counter0++;
    }
}

void Task1(void* args)
{
    while(1)
    {
        SemaWait(&reachSema);
        counter1++;
    }
}
uint8_t rcvdData[64];
void Task2(void* args)
{
    while(1)
    {
        counter2++;
        if (counter2 == 90000)
        {
            SemaSignal(&reachSema);
        }
    }
}
```

Expression	Type	Value
tcbs	TCB_t [3]	0x200000b8 <tcbs>
counter2	volatile uint32_t	2690764
counter1	volatile uint32_t	0
counter0	volatile uint32_t	1
Add new expression		

```
Name : tcbs
    Details:{{sp = 0x20000180 <stacks+152>, next = 0x200000c8 <t
    Default:0x200000b8 <tcbs>
    Decimal:536871096
    Hex:0x200000b8
```

Figure 8 The signal operation unblocks only one task blocked on that semaphore

When a semaphore is initialized with value 1, it can be used to guard critical sections, since only one task will be granted access to that piece of executable code. The pattern is shown on Listing 11.

```c
SEMA_t sema;

/* sema.value = N; Semaphore initialized as N where convenient */

SemaWait(&sema); /* a task when trying to access the resource first Wait on the
semaphore */
          .
/* execution code */
          .
SemaSignal(&sema); /* a task when leaving the resource Signal on the semaphore */
```

Listing 10. Semaphore usage for resource management

```c
SEMA_t sema;

/* sema.value = 1; Semaphore initialized as 1 where convenient */

SemaWait(&sema); /* a task when trying to access a CR first Wait on the semaphore
*/
          .
/* CR code */
          .
SemaSignal(&sema); /* a task when leaving a CR Signal on the semaphore */
```

Listing 11. Semaphore usage for locking critical regions.

1.4.5. Semaphore usage for Event-Driven tasks

A common pattern to design event-driven tasks using semaphores, is for a task to start and Wait on a semaphore initialized as 0, so when an event happens, the interrupt handler, event callback or similar, signals this semaphore releasing the task. After completing its duty, the task will block on the semaphore again waiting for the next signal.

1.4.5.1.Example of Event-Driven tasks using semaphores

The following example reads a number inserted on a PC terminal. The reception of this number, via UART, generates an interrupt. This interrupt handler, after receiving the character, signals Task0, which is blocked waiting for this information. Then, depending on the number received (valid numbers are ‘1’ or ‘2’), Task0 will now signal the semaphore related to Task1 or Task2. Each task simply prints a line on the PC terminal. The UART driver implementation shown depends on the HAL provided by the Board Support Package for the Nucleo-F103RB board. Listing 12 and Listing 13 show the code. Figure 9 shows the terminal on the PC screen with the inputs and output lines. Note that UART interrupts happen for a reception when the UART peripheral data register is *not empty*, and for transmissions, when it *is* empty – i.e., when data arrives and when data leaves.

```
/*
* @brief UART Driver
*/

#include <stdint.h>
#include <stm32f1xx_hal.h>
#include "kernel.h"

SEMA_t rcvdUartSema;
SEMA_t task1Sema;
SEMA_t task2Sema;
SEMA_t uartMutex;

#define BUFFER_SIZE 8
struct circularBuffer /* circular buffer used to improve throughput */
{
    uint32_t putIndex;
    uint32_t getIndex;
    uint32_t nCharacters;
    uint8_t data[BUFFER_SIZE];
};

static volatile struct circularBuffer sendBuffer = { 0, 0, 0, {'\0'} };
static volatile uint8_t rcvdData[BUFFER_SIZE] = {0};
static volatile size_t rcvdgetIndex = 0;
static volatile size_t rcvdputIndex = 0;
#define USART2_DisableRXNEIRQ() huart2.Instance->CR1 &= ~USART_CR1_RXNEIE
#define USART2_EnableRXNEIRQ()  huart2.Instance->CR1 |= USART_CR1_RXNEIE
#define USART2_EnableTXEIRQ()   huart2.Instance->CR1 |= (USART_CR1_TXEIE)
#define USART2_DisableTXEIRQ()  huart2.Instance->CR1 &= ~(USART_CR1_TXEIE)

/*  UART driver lower level
******************************/
/**
 * @brief This function uses the HAL to initialise USART2 as Sender/Rcvr at
115200bps, 1 byte per data unit, 1 stop-bit and no parity check. The oversampling
is handy to reject noise*/

void USART2_Init(void)
{

    huart2.Instance = USART2;
    huart2.Init.BaudRate = 115200;
    huart2.Init.WordLength = UART_WORDLENGTH_8B;
    huart2.Init.StopBits = UART_STOPBITS_1;
    huart2.Init.Parity = UART_PARITY_NONE;
    huart2.Init.Mode = UART_MODE_TX_RX;
    huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart2.Init.OverSampling = UART_OVERSAMPLING_16;
    if (HAL_UART_Init(&huart2) != HAL_OK)
    {
        ErrorHandler();
    }
}
/**
 * @brief This function handles USART2 global interrupt.
 */

void USART2_IRQHandler(void)
{
    if (huart2.Instance->SR & USART_SR_RXNE) /* rx not empty */
    {
        EnterCR();
        rcvdData[rcvdPutIndex] = (uint8_t)huart2.Instance->DR & 0xFF;
        rcvdPutIndex += 1;
        rcvdPutIndex %= BUFFER_SIZE;
        SemaSignal(&rcvdUartSema); /* signal task0 there is data */
        ExitCR();
        return;
    }
    if (huart2.Instance->SR & USART_SR_TXE) /* tx empty */
    {
        if (sendBuffer.nCharacters == 0) /* no more chars? */
        {
            USART2_DisableTXEIRQ();
            return;
        }
        //Send char
        EnterCR();
        huart2.Instance->DR = sendBuffer.data[sendBuffer.getIndex];
        ++sendBuffer.getIndex;
        sendBuffer.getIndex %= BUFFER_SIZE;
        --sendBuffer.nCharacters;
        if (sendBuffer.nCharacters == 0)
            USART2_DisableTXEIRQ();
        ExitCR();
        return;
    }
}

/*
*  UART driver upper level
******************************/
/*
 * @brief Transmits a single unsigned char to USART2
 * @param ch unsigned char to be transmitted
 */
void PutChar(const uint8_t ch)
{
    while (sendBuffer.nCharacters == BUFFER_SIZE); /* no room, busy-wait (?) */
    EnterCR();
    sendBuffer.data[sendBuffer.putIndex] = ch;
    ++sendBuffer.putIndex;
    sendBuffer.putIndex %= BUFFER_SIZE;
    ++sendBuffer.nCharacters;
    USART2_EnableTXEIRQ();/* interrupt me when data leaves */
    ExitCR();
}
/*
```

```
 * @brief Transmits a string of chars to USART2
 * @param str pointer to a string of chars
 */
void Puts(const char* str)
{
    SemaWait(&uartMutex);
    for (; *str != '\0'; ++str)
        sPutChar(*str);
    SemaSignal(&uartMutex);
}
```

Listing 12 UART Driver to receive a single char and print a string.

```
/*
* @file tasks.c
*/
SEMA_t rcvdUartSema;
SEMA_t task1Sema;
SEMA_t task2Sema;

void Task0(void* args)
{
    SemaInit(&task1Sema, 0);
    SemaInit(&task2Sema, 0);
    SemaInit(&rcvdUartSema, 0);
    SemaInit(&uartMutex, 1);
    USART2_EnableRXNEIRQ();
    while (1)
    {
        SemaWait(&rcvdUartSema);
        switch (rcvdData[rcvdGetIndex])
        {
        case '1':
            SemaSignal(&task1Sema);
            break;
        case '2':
            SemaSignal(&task2Sema);
            break;
        default:
            Puts("\n\rInvalid number\n\r");
            break;
        }
        EnterCR();
        rcvdGetIndex++;
        rcvdGetIndex %= BUFFER_SIZE;
        ExitCR();
    }
}

void Task1(void* args)
{
       const char t1SendMsg[] = "\n\rTask1 says: the boggie-oogie\n\r";

       while (1)
       {
              SemaWait(&task1Sema);
              Puts(t1SendMsg);
       }
}
void Task2(void* args)
{
       const char t2SendMsg[] = "\n\rTask2 says: a toast to the boogie\n\r";
       while (1)
       {
              SemaWait(&task2Sema);
            Puts(t2SendMsg);

      }
}
```

Listing 13. Tasks for the event-driven example

1.4.6. Using semaphores to avoid busy-waiting

The skilled reader might be wondering why in the UART driver above, I chose to busy-wait on a multithreaded system right after presenting and implementing semaphores. One of the characteristics of the synchronisation primitives presented, is that they prevent a task from running when a certain condition they need to keep going is not met. In the case of the UART driver, it is taking CPU time just to *wait* for room on the buffer. With semaphores, we can implement the driver on a way that if there is no room in the buffer, the task is suspended to be resumed when there is room.

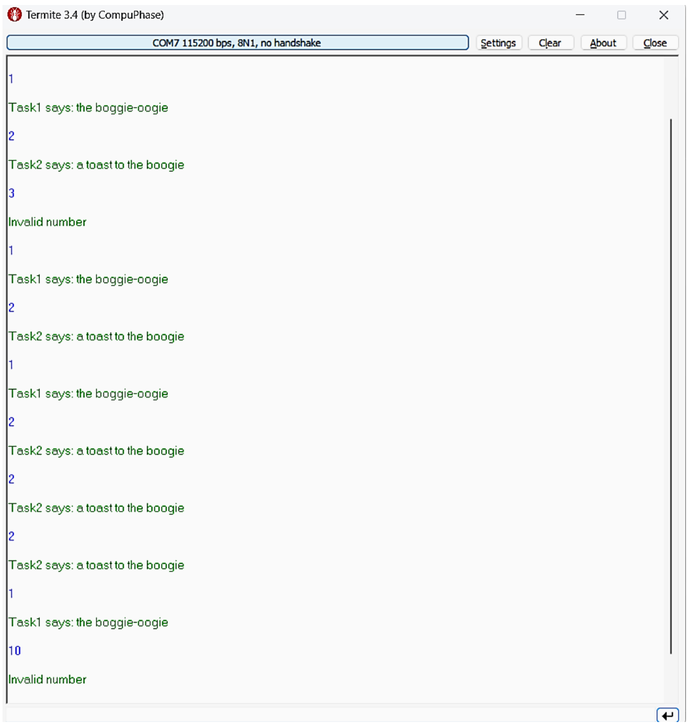
Note that what signifies a buffer slot has been taken is the increment of `sendBuffer.nCharacters`. Every time a transmission happens, it is then decremented. When it reaches 0, there is nothing else to transmit, the TXE interrupt is disabled. To get rid from the busy-wait we can create a semaphore to track buffer space. So, a semaphore named like `uartSpaceSema` is initialized with the value of `BUFFER_SIZE`. On the beginning of the `PutChar` function, call a Wait on the semaphore. And on IRQ Handler, every time `sendBuffer.nCharacters` is decremented, signal the semaphore. These modifications are shown on Listing 14 with a boldface.

1.4.7. Using semaphores to implement a mailbox

Two threads can exchange messages (one as a sender and the other as a receiver) by properly synchronising access to a *mailbox*. Some authors and RTOSes refer to mailboxes as queues (FIFOs) in general, others as a queue that can hold one single message. This article series takes the latter. A simple mailbox implementation can be as follows (`Listing 15`):

(1) A mailbox is *full* after a sender writes to it. A mailbox is *empty* after a receiver reads from it.
(2) A *sender* task will block when trying to write on a full mailbox. A receiver task will block when trying to read from an empty mailbox.
(3) A sender task *posts* a message to a mailbox by calling the function *MailboxPost*, passing the mailbox address and the message content.
(4) If the mailbox is full, it will be blocked on the *empty* semaphore, because there is a message waiting to be retrieved by a receiver. If it is empty, it follows through.
(5) The semaphore *mutex* adds a guarantee two different tasks will not access the mail data at the same time.
(6) After sending the message, the semaphore *full* is signalled, indicating there is a message on the box, and potentially unblocking a receiver task that might have been blocked when trying to read from an empty mailbox.
(7) A receiver task calls *MailboxPend* passing the mailbox address and the address of the variable that will hold the received message contents.
(8) If there is a message posted but not read yet, it will pass the *full* semaphore. Otherwise, it will block until the sender task posts a message.

After reading the message, the receiver task signals *empty*, to make the mailbox able to receive another message, and potentially unblocking a sender task that has been blocked when trying to write on a full mailbox.


Termite 3.4 (by CompuPhase)
COM7 115200 bps, 8N1, no handshake
Settings
Clear
About
Close

```
1
Task1 says: the boggie-oogie
2
Task2 says: a toast to the boogie
3
Invalid number
1
Task1 says: the boggie-oogie
2
Task2 says: a toast to the boogie
1
Task1 says: the boggie-oogie
2
Task2 says: a toast to the boogie
2
Task2 says: a toast to the boogie
2
Task2 says: a toast to the boogie
1
Task1 says: the boggie-oogie
10
Invalid number
```

Figure 9. UART event driven tasks running

```
void PutChar(const uint8_t ch)
{
    SemaWait(&uartSpaceSema); /*initial value == BUFFER_SIZE*/
    EnterCR();
    sendBuffer.data[sendBuffer.putIndex] = ch;
    ++sendBuffer.putIndex;
    sendBuffer.putIndex %= BUFFER_SIZE;
    ++sendBuffer.nCharacters;
    USART2_EnableTXEIRQ();/* interrupt me when data leaves */
    ExitCR();
}

void USART2_IRQHandler(void)
{
    if (huart2.Instance->SR & USART_SR_RXNE)
    {
        EnterCR();
        rcvd_data[rcvd_putIndex] = (uint8_t)huart2.Instance->DR & 0xFF;
        rcvd_putIndex += 1;
        rcvd_putIndex %= BUFFER_SIZE;
        SemaSignal(&rcvdUartSema);
        ExitCR();
        return;
    }
    if (huart2.Instance->SR & USART_SR_TXE)
    {
        if (sendBuffer.nCharacters == 0)
        {
            USART2_DisableTXEIRQ();
            return;
        }
        //Send
        EnterCR();
        huart2.Instance->DR = sendBuffer.data[sendBuffer.getIndex];
        ++sendBuffer.getIndex;
        sendBuffer.getIndex %= BUFFER_SIZE;
        --sendBuffer.nCharacters;
       SemaSignal(&uartSpaceSema); /* signals space */
        if (sendBuffer.nCharacters == 0)
            USART2_DisableTXEIRQ();
        ExitCR();
        return;
    }
}
```

Listing 14 Using semaphore to avoid busy-waiting

```
/* @brief Simple mailbox (error handling omitted) */
typedef struct mailbox
{
    SEMA_t  empty;
    SEMA_t  mutex;
    SEMA_t  full;
    uint32_t  mail; /* for simplicity, the msg is just a u32 variable */
} __attribute__((aligned)) MAILBOX_t;

/* @brief Initialises a mailbox
 * @param self Address of a mailbox structure
 */
void MailBoxInit(MAILBOX_t* self)
{
    SemaInit(&self->mutex, 1); /* data init as free to access */
    SemaInit(&self->empty, 1); /* mailbox init as empty */
    SemaInit(&self->full, 0);  /* mailbox init not full */
    self->mail = 0U;
}
/* @brief Post a message on a mailbox
 * @param self Address of a mailbox structure
 * @param mail Message
 */
void MailboxPost(MAILBOX_t* self, uint32_t mail)
{
    SemaWait(&self->empty); /* is it empty? */
    SemaWait(&self->mutex);
    self->mail = mail; /* fills mail by copying the message */
    SemaSignal(&self->mutex);
    SemaSignal(&self->full); /* signals there is message on the box */
}
/* @brief Retrieves a message from a mailbox
 * @param self Address of a mailbox structure
 * @param rcvd_msg Address of the variable to hold the received message
 */
void MailboxPend(MAILBOX_t* self, uint32_t* rcvd_msg)
{
    SemaWait(&self->full); /* there is a message? */
    SemaWait(&self->mutex);
    *rcvd_msg = self->mail; /* copy message to receiver local variable */
    SemaSignal(&self->mutex);
    SemaSignal(&self->empty); /* signals it is empty */
}
```

Listing 15 Simple mailbox using semaphores

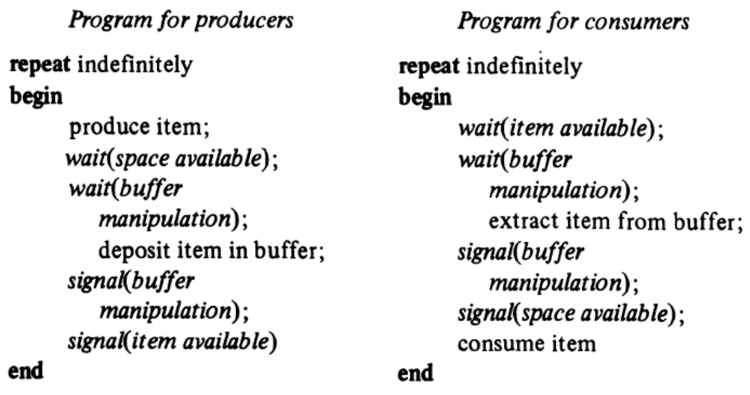
1.4.8. Solving the Producer-Consumer problem with Semaphores

In [4] the author demonstrates the following algorithm can solve the consumer-producer problem with semaphores.

On Figure 10 the processes use a semaphore to protect the buffer critical region excluding each other from accessing the buffer at the same time, and they cooperate through the semaphores that keep track of space and items. That is, it uses semaphores for both cooperation and mutual-exclusive locking.

Using this algorithm, we can write a FIFO queue with *put* and *get* methods to be free of race conditions, as on Listing 16.

You can appreciate the Mailbox algorithm (Listing 15) is a special case for a queue with a single position.



```
        Program for producers

repeat indefinitely
begin
        produce item;
        wait(space available);
        wait(buffer
            manipulation);
            deposit item in buffer;
        signal(buffer
            manipulation);
        signal(item available)
end
```

```
        Program for consumers

repeat indefinitely
begin
        wait(item available);
        wait(buffer
            manipulation);
            extract item from buffer;
        signal(buffer
            manipulation);
        signal(space available);
        consume item
end
```

Figure 10. Producer-Consumer problem. Semaphore based solution. [4]

```c
/* @brief Safe FIFO for multithreading (error handling omitted) */
typedef struct
{
    uint32_t   head; /* get index; init as 0 */
    uint32_t   tail; /* put index; init as 0 */
    SEMA_t     item; /* number of chars in the buffer; init as 0 */
    SEMA_t     space; /* empty space in the buffer; init as BUFFER_SIZE */
    SEMA_t     mutex; /* init as 1, to manage concurrent access to the buffer */
    char       buffer[BUFFER_SIZE];
}__attribute__((aligned)) FIFO_t;

/* @brief Inserts a char on a FIFO queue
 * @param self Pointer to a FIFO structure
 * @param data character to be inserted
 */
void FifoPut(FIFO_t* self, char data)
{
    SemaWait(&(self->space));
    SemaWait(&(self->mutex));
    self->buffer[self->tail] = data;
    self->tail = (self->tail + 1) % BUFFER_SIZE;
    SemaSignal(&(self->mutex));
    SemaSignal(&(self->item));
}
/* @brief Gets a char from a FIFO queue
 * @param self Pointer to a FIFO structure
 * @retval retrieved character
 */
char FifoGet(FIFO_t* self)
{
    char get_data;
    SemaWait(&(self->item));
    SemaWait(&(self->mutex));
    get_data = self->buffer[self->head];
    self->head = (self->head + 1) % BUFFER_SIZE;
    SemaSignal(&(self->mutex));
    SemaSignal(&(self->space));
    return get_data;
}
```

Listing 16 Circular Buffer (FIFO) encapsulating synchronisation

1.4.9. Semaphores drawbacks

The role of semaphores is to coordinate, enforce mutual exclusion, and precedence between tasks. The mechanism is used to ensure that accesses to a shared variable takes place in the right sequence and at the appropriate time. Semaphores are an evolutionary idea to solve *almost* any concurrency problems. But it also comes with several drawbacks that a system programmer needs to be aware of [10]. Semaphores misuse leads to deadlocks, inconsistent data, and other poltergeists.

(1) Semaphores are not automatically connected with the protected items.
(2) The wait and signal operations form an essential pair. However, there is no automatic enforcing of such pairings; each can be used without regard to the other. So, for example, a semaphore locked by one task could be unlocked quite incorrectly by another.
(3) Semaphores must be visible and in the scope of all tasks that share the protected resource. This means that any task can issue semaphore calls at any time – and they will be acted on (even if they produce nonsense actions).
(4) The semaphore does not guarantee to prevent unauthorized access to protected resources. This comes about because the semaphore does not hide the resource; in reality, it is visible and in scope of all the user tasks. Thus, tasks can bypass semaphore locks and directly use critical resources.

1.5.THE READER-WRITER PROBLEM

Another classic inter-process communication problem is the ***Reader-Writer*** problem, when multiple readers get data from a memory location one or multiple writers write to. It is about databases on its general meaning. It is acceptable to have multiple readers reading the data, but multiple writers at the same time can never occur. The problem can be summarised as follows [3]:

A set of reader and writer processes share a common data object, e.g., a variable or a file. The requirements are as follows: an active writer must exclude all others. However, readers should be able to read the data object concurrently if there is no active writer. Furthermore, both readers and writers should not wait indefinitely (starve). The solution is as follows [2, 3]:

```
SEMA_t rcSema = 1; /*controls access to rc */
SEMA_t dataSema = 1; /* controls access to the data */
SEMA_t rwSema = 1; /* controls access for both readers and writers */
int rc = 0; /* number of processes reading or wanting to */
DATA data;
```

```
void reader(void)
{
        while (1)
        {
                SemaWait(&rwSema); /* ensure fifo order for r/w */
                SemaWait(&rcSema); /*get exclusive access to rc */
                rc = rc + 1; /*one reader more now */
                if (rc == 1)
                  SemaWait(&dataSema); /* if this is the first reader */
                SemaSignal(&rcSema); /* release exclusive access to rc */
                SemaSignal(&rwSema);
                       read(data); /* access the data */
                SemaWait(&rcSema); /* get exclusive access to rc */
                rc = rc - 1; /* less one reader */
                if (rc == 0)
                  SemaSignal(&dataSema); /* if this is the last reader */
                SemaSignal(&rcSema); /* release exclusive access to rc */
                process(data);
        }
}
```

```
void writer(void)
{
        while (1)
        {
                SemaWait(&rwSema); /* ensure fifo order for both r/w */
                SemaWait(&dataSema); /* get exclusive access */
                process(data); /* update the data */
                SemaSignal(&dataSema); /* release exclusive access */
                SemaSignal(&rwSema);
        }
}
```

Listing 17 Reader-writer solution with semaphores

The semaphore *rwSema* enforces FIFO order of all incoming readers and writers, which prevents starvation. *rcSema* is for readers to update the *rc* variable in a CR. The first reader in a batch of readers locks the *dataSema* to prevent any writer from writing while there are active readers. On the writer side, at most one writer can be either actively writing or waiting in *dataSema* queue. In either case, new writers will be blocked in the *rwSema* queue. Assume that there is no writer blocked at *rwSema*. All new readers can pass through both *SemaWait(rwSema)* and *SemaSignal(rwSema),* allowing them to read the data concurrently. When the last reader finishes, it issues *SemaSignal(dataSema)* to allow any writer blocked on this semaphore to continue. When the writer finishes, it unlocks both *dataSema* and *rwSema*. As soon as a writer waits at *rwSema*, all newcomers will be blocked at *rwSema* also. This prevents readers from starving writers [3].

1.6. MUTEXES

A *mutex* is like a binary semaphore in the sense it is suitable for mutual-exclusive locking and assumes values 0 or 1. These values are defined as unlocked and locked respectively, so, the mutex semantics is different from semaphores. The major difference between a mutex and a semaphore is that a mutex has an *owner* associated to its locking state – the task which locked the mutex *owns* it, and a locked mutex can only be unlocked by its owner.

When a mutex is initialised, it has no owner and is unlocked. A task that successfully locks it is then the owner. When a task tries to lock an already locked mutex, and it is not the owner, this task will block on the mutex. A release operation on the mutex, will unlock the mutex and unblocks one task, if any. While semaphores can be used for both locking and synchronisation/cooperation, a mutex is a mechanism used solely for mutual exclusive locking.

Worth mentioning that the (not uncommon) claim that a *mutex is a binary semaphore* is wrong. A binary semaphore is a special case of a counting semaphore while a mutex is a specialized mechanism for mutual-exclusion, and they have different semantics: if a task waits on a binary semaphore which value is 0, it is still 0 and the task will be blocked. If a mutex is zero, it is unlocked, and a lock operation will result on owning the mutex and the critical region it is guarding.

1.6.1. Mutex Implementation

The proposed implementation for simple mutexes that suffices a round-robin preemptive-cooperative scheduling is on Listing 18.

```
/**
 * @brief Mutex structure holding a blocked queue, lock status, and owner thread.
 *        (Error handling omitted).
 */
typedef struct {
    TCB_t* queuePtr;
    uint8_t lock; /**< 0=unlocked, 1=locked */
    TCB_t* ownerPtr; /**< Pointer to current owner */
} __attribute__((aligned)) MUTEX_t;
/**
 * @brief      Initialises a mutex.
 * @param self Mutex address
 */
void MutexInit(MUTEX_t* self)
{
    EnterCR();
    self->lock = 0;
    self->queuePtr = NULL;
    ExitCR();
}
/**
 * @brief      Locks a mutex.
 * @param self Mutex address
 */
void MutexLock(MUTEX_t* self)
{
    EnterCR();
    if (self->lock == 0) //unlocked
    {
        self->lock = 1; //lock
        self->ownerPtr = runPtr; //set owner
        ExitCR();
        return;
    }
    else //locked
    {
        if ((self->ownerPtr != runPtr) && (self->ownerPtr != NULL)) //not owner
        {
            runPtr->status = BLOCKED;
            Enqueue(&self->queuePtr, runPtr);
        }
        Yield();
        ExitCR();
    }
}

/**
 * @brief      Releases a mutex.
 * @param self Mutex address
 */
```

```
void MutexUnlock(MUTEX_t* self)
{
    EnterCR();
    TCB_t* taskPtr=NULL;
    if ((self->ownerPtr == runPtr) || (self->ownerPtr == NULL))
    {
        if (self->queuePtr == NULL) //no waiters
        {
            self->lock = 0;
            self->owner = 0;
        }
        else //unblock, set new owner
        {
            taskPtr = Dequeue(&self->queuePtr);
            taskPtr->status = READY;
            self->ownerPtr = taskPtr;
            Enqueue(&readyQueuePtr, taskPtr);
        }
    }
    ExitCR();
}
```

Listing 18 Simple Mutex implementation

Despite the different semantics, mutex usage patterns are the same as for locking semaphores.

1.7. HIGHER-LEVEL SYNCHRONISATION CONSTRUCTS

To circumvent the easy misuse of synchronisation primitives, many higher-level mechanisms have been proposed.

1.7.1. Condition Variables

A Condition Variable is an object that is logically associated with a shared resource, allowing a task to wait for a desired condition to occur before using the resource. In practice, a condition variable is a queue of processes waiting for a condition to be met [11]. It is built upon a mutex object, called its guarding or binding mutex. Prior to evaluating a condition about the shared resource, a task must have exclusive access to it. This is enforced by the guarding mutex. A task blocks and enters the waiting list of the guarding mutex if it is currently locked by another task. At the time of evaluation, if the condition is not true, the task blocks and is placed on the waiting queue of the Condition Variable. When the condition is met another task *signals* the Condition Variable, unblocking at least one thread that is waiting for the specified condition to be met.

1.7.1.1. Condition Variables Implementation

The POSIX Threads API, as well as the C11 (and later) standard library, takes the mutex related to a condition variable as an input parameter. The implementation proposed here encapsulates a mutex and a linked-list of TCBs on a data structure (Listing 19). A semaphore could have been used instead of a raw queue, but it would deviate from regular Condition Variables, that do not have counters, i.e., they do not accumulate signals. That's important, because the order of *Wait* and *Signal* matters, like on *Sleep-WakeUp.*

Condition Variables usage and behaviour is as follows [3]:

(1) When a thread wants to access a shared data/resource it locks the mutex first. Then it checks the condition. If the condition is not met, the calling thread blocks on the

condition variable and the mutex is unlocked, all in an atomic operation. The mutex is unlocked so another thread can operate on the structure.

(2) While a thread is locked on the condition variable another thread might call *signal* to unblock a waiting thread if the condition is met. A function *SignalBroadcast* can be used to signal all blocked threads, which is like *Wake*.

(3) When an unblocked thread runs, the mutex is automatically and atomically locked, allowing the unblocked thread to resume in the critical region of the mutex.

```c
typedef struct {
    MUTEX_t condMutex;
    TCB_t*  condQueuePtr;
} COND_VAR_t;

/**
 * @brief      Initialises a condition variable.
 * @param self Condition variable address
 */

void CondVarInit(COND_VAR_t* self)
{
    MutexInit(&self->condMutex);
    condQueuePtr = NULL;
}

/**
 * @brief      Wait on a condition variable.
 * @param self Condition variable address.
 */
void CondVarWait(COND_VAR_t* self)
{
    EnterCR();
    MutexUnlock(&self->condMutex);
    runPtr->status = BLOCKED;
    Enqueue(&self->condQueuePtr, runPtr);
    Yield();
    ExitCR();
    MutexLock(&self->condMutex);
}
/**
 * @brief      Signals a condition variable,
 *             to release one queued task, if any.
 * @param self Condition variable address.
 */
void CondVarSignal(COND_VAR_t* self)
{
    TCB_t* nextTcbPtr = NULL;
    if (self->queue == NULL)
        ErrorHandler();
    MutexLock(&self->condMutex);
    EnterCR();
    nextTcbPtr = Dequeue(&self->condQueuePtr);
    nextTcbPtr->status = READY;
    Enqueue(&readyQueuePtr, nextTcbPtr);
    ExitCR();
    MutexUnlock(&self->condMutex);
}
/**
 * @brief      Signal a condition variable,
 *             releasing all queued tasks.
 * @param self Condition variable address.
 */
void CondVarSignalBroadcast(COND_VAR_t* self)
```

```
{
    MutexLock(&self->condMutex);
    TCB_t* nextThread = self->condQueuePtr;
    size_t nOfThreads = 0;
    while (nextThread != NULL)
    {
        nOfThreads++;
        nextThread = nextThread->next;
    }
    if (nOfThreads == 0)
        ErrorHandler();
    for (size_t i = 0; i < nOfThreads; i++)
    {
        CondVarSignal(self);
    }
    MutexUnlock(&self->condMutex);
}
```

Listing 19 Condition Variable implementation

The programmer needs to guarantee the correctness of the code that evaluates if the condition is met and signals the condition variable.

1.7.1.2. Using a Condition Variable to implement a Synchronisation Barrier

As an example of condition variable usage, let us implement a synchronisation barrier as depicted on Figure 1. The idea is that all threads will run up to a synchronisation point, a barrier, and they only can proceed when a specific number of threads, equal or greater than a threshold, have had reached the barrier too. We define a Synchronisation Barrier as a data structure with a threshold number, a counter, and a condition variable. Every time a task reaches the barrier it must call the function *SynchBarrierSynch*, to increase the counter of synchronised threads. This function also checks for the condition variable predicate, that is, the number of synchronised threads on the barrier must be greater than or equal to the threshold. If false, it will block the task on the condition variable. If true, it calls *CondVarSignalBroadcast* to unblock all threads waiting for the condition.

What we expect to see is that only after all threads synchronised, they will be free to go further. To emulate different computation times for each thread before reaching the barrier, a *SleepTicks* function is used – the thread suspends for a given number of ticks. Listing 20 shows the implementation of the Synchronisation Barrier.

```
struct SynchBarrier
{
    COND_VAR_t barrierCondVar;
    uint32_t   barrierCount;
    uint32_t   barrierThreshold;
};
typedef struct SynchBarrier SynchBarrier_t;

void CondVarSignalBroadcast(COND_VAR_t* self)
{
    MutexLock(&self->condMutex);
    TCB_t* nextThread = self->condQueuePtr;
    size_t nOfThreads = 0;
    while (nextThread != NULL)
    {
        nOfThreads++;
        nextThread = nextThread->next;
    }
    if (nOfThreads == 0)
        ErrorHandler();
    for (size_t i = 0; i < nOfThreads; i++)
    {
        CondVarSignal(self);
    }
     /* reset condition*/
    SynchBarrier.barrierCount = 0; /* a more elaborate condition may require
                                      a dedicated function */
    MutexUnlock(&self->condMutex);
}
void SynchBarrierInit(SynchBarrier_t* self, uint32_t barrierThreshold)
{
    CondVarInit(&(self->barrierCondVar));
    self->barrierThreshold = barrierThreshold;
    self->barrierCount = 0;
}

void SynchBarrierSynch(SynchBarrier_t* self)
{
    MutexLock(&self->barrierCondVar.condMutex);
    self->barrierCount++;
    if (self->barrierCount >= self->barrierThreshold)
    {
        CondVarSignalBroadcast(&self->barrierCondVar);
    }
    else
    {
        CondVarWait(&self->barrierCondVar);
        /*
         * You got the resource locked here to
         * work exclusively on it. In this case, we got
         * nothing else to do.
         */
    }
    MutexUnlock(&self->barrierCondVar.condMutex);
}
void Task0(void* args)
{
    SynchBarrierInit(&SynchBarrier, 3);
    SemaInit(&uartMutex, 1);
    while (1)
    {
        SleepTicks(8);
        Puts("Task0 is synching...\n\r");
        SynchBarrierSynch(&SynchBarrier);
        Puts("Task0 freed\n\r");
        HAL_Delay(1);
        Yield();
    }
}

void Task1(void* args)
{
    while (1)
    {
        SleepTicks(6);
        Puts("Task1 is synching...\n\r");
        SynchBarrierSynch(&SynchBarrier);
        Puts("Task1 freed\n\r");
        HAL_Delay(1);
        Yield();
    }
}
void Task2(void* args)
{
    while (1)
    {
        SleepTicks(15);
        Puts("Task2 is synching...\n\r");
        SynchBarrierSynch(&SynchBarrier);
        Puts("Task2 freed\n\r");
        HAL_Delay(1);
        Yield();
    }
}
```

Listing 20 Synchronisation Barrier using Condition Variable

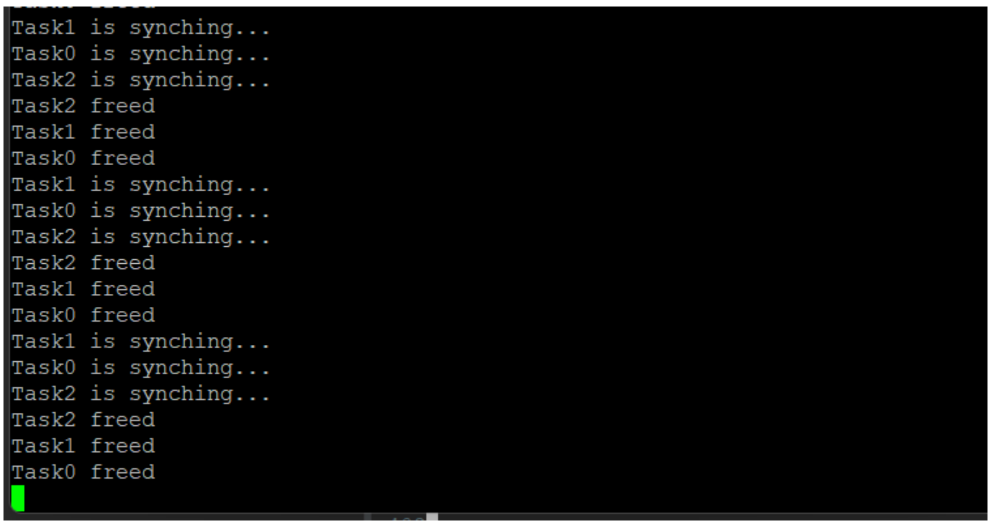
By inspecting Listing 20 we should expect *Task1* to be the first to synchronise on the barrier, followed by *Task0,* and both will block on the condition variable. The last one is *Task2* that increments the counter to 3 and therefore, does not block on the Condition Variable. The dispatching of *Task1* and *Task0* follows the order they were blocked on the Condition Variable, first *Task1* and then *Task0*. This is what we see on Figure 11.

Note that when returning from the waiting queue the mutex is locked again, so the thread was resumed inside the critical region of an associated resource, and you can use it safely. But nothing guarantees the condition is still true. Indeed, in the case of the barrier, the condition is not true anymore, because we reset the barrierCounter variable to zero after signalling. In this case it is ok that the condition *was* met and is not true *anymore*.

If the condition shall still holds true when returning from the waiting queue, such as a flag on a device register, the programmer should implement a loop to check, like *do { wait } while (!condition);*. Note that if we had used this loop in our implementation, the system would enter on a deadlock state. That's the pain of synchronisation, it is hard to establish a one-fits-all pattern.

1.7.2. Monitors

For the sake of completeness, the concept of Monitor [11, 14] is briefly addressed. Monitors are *language constructs*, i.e., a programming language feature for concurrency. The monitor construct ensures that only one process at a time is active within the monitor. Consequently, the programmer does not need to code this synchronisation constraint explicitly. The C language does not support monitors. Java, Ada are examples of languages that support a kind of monitor. Monitors are used mostly in concurrent programming, but rarely on operating systems design [9].



```
Task1 is synching...
Task0 is synching...
Task2 is synching...
Task2 freed
Task1 freed
Task0 freed
Task1 is synching...
Task0 is synching...
Task2 is synching...
Task2 freed
Task1 freed
Task0 freed
Task1 is synching...
Task0 is synching...
Task2 is synching...
Task2 freed
Task1 freed
Task0 freed
```

Figure 11 Synchronisation Barrier with Condition Variable (running)

The concept of Monitor was first introduced by Brinch Hansen [14], and further developed by Hoare [11]. In his book on operating systems [12], Hansen describes a monitor as "(…) *a shared variable and the set of meaningful operations on it. The purpose of a monitor is to control the scheduling of resources among individual processes according to a certain policy."* For sure, he is describing what we know as an Abstract Data Type – a type of data that is completely defined by the operations acting on it. As well, he is describing the role of a kernel. Indeed, Hansen was the first to propose that operating systems could be developed around a *Nucleus* [14], that he also regarded as a Monitor.

Hoare introduced *Condition Variables* along the operations *wait* and *signal,* issued from the procedures within the monitor. Quoting his paper [11]: "*Note that a condition "variable" is neither true nor false; indeed, it does not have any stored value accessible to the program. In practice, a condition variable will be represented by an (initially empty) queue of processes which are currently waiting on the condition."* Figure 12 is a representation of a Monitor.

When a process signals a condition variable, at least one process that is waiting on the condition will wake up. Since two processes cannot be active inside a monitor at the same time, what happens to each process might be defined. Hoare's proposal was to suspend the process which issued the signal letting the awakened to run. Hansen's proposal was that a process that issues a signal, must leave the monitor, i.e., a signal must always be the final statement of a procedure within a monitor [15].

Finally, [10] introduces the idea of "simple monitors" (Figure 13) for embedded system software. It has the following features:

(1) Provides protection for critical regions of code.
(2) Encapsulates data together with operations applicable to this data.
(3) Is highly visible. Is easy to use. Is difficult to misuse.
(4) Simplifies the task of proving the correctness of a program.

Fundamentally, the simple monitor prevents tasks directly accessing a shared resource by doing the following:

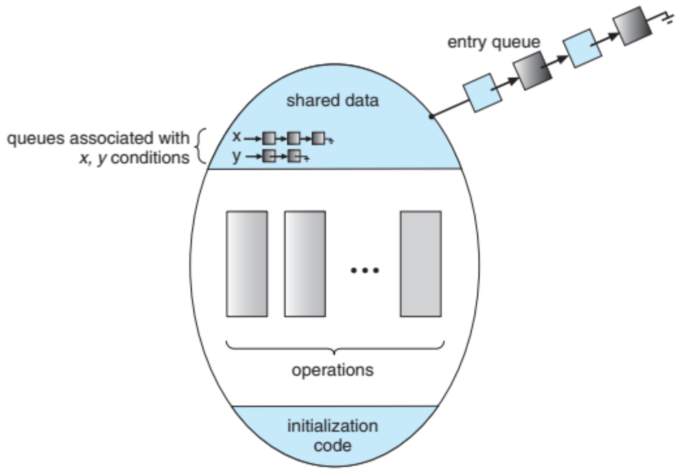


Figure 12 Conceptual representation of a Monitor with Condition Variables [13]

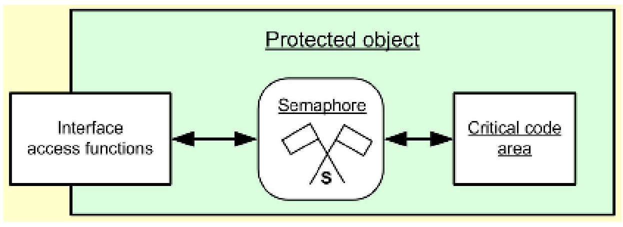


Figure 13 A "simple monitor" as regarded by [10]

(1) Encapsulating the resource with its protecting semaphore/mutex within a program unit.
(2) Keeping all semaphore/mutex operations local to the encapsulating unit.
(3) Hiding these from the "outside" world, that is, making them private to the program unit.
(4) Preventing direct access to the semaphore/mutex operations and the critical code section Providing the means to indirectly use the shared resource.

You can see, a new name for an old thing. It is a Class whose interface hides from the programmer the synchronisation complexities of the operations. Accordingly to this definition, the FIFO queue of Listing 16 and the Mailbox of Listing 15 are *"simple monitors";* so are the Condition Variables.

1.8.FINAL COMMENTS

After splitting the execution of a program image on several threads, and creating a mechanism to dispatch, suspend and resume them, thus, multiplexing CPU allocation over time, we took the next step on our kernel development and created synchronisation mechanisms.

These mechanisms to work together, by coordinating, enforcing mutual exclusion, and precedence on the access to shared resources - so computations happen on the right sequence at the right time.

These mechanisms facilitate the development of complex applications, by effectively allowing the decomposition of a complex job on several tasks. Our kernel is taking shape. On the next article we will discuss and implement inter-task communication.

REFERENCES

[1] Design Patterns for Embedded Systems in C (Douglass Powell, 2009)
[2] Modern Operating Systems (Andrew Taneunbaum, Herbert Boss, 2023)
[3] Embedded and Real-Time Operating Systems 2nd Ed., (Wang, K.C., 2023)
[4] Fundamentals of Operating Systems (Lister, A.M., 1984)
[5] Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. (Buttazzo, Giorgio., 2023)
[6] Windows Internals: System architecture, processes, threads, memory management, and more, Part 1 (Developer Reference). (Pavel, Yosifovich, 2017)
[7] Co-operating sequential processes. (Dijkstra, 1966)
[8] Real-Time Embedded Systems: Design Principles and Engineering Practices. (Fan, Xiaocong, 2015)

[9] Design and Implementation of the MTX Operating System (K.C. Wang, 2015)
[10] Software Engineering for Real-Time Systems: A software engineering perspective toward designing real-time systems (Cooling, Jim., 2019)
[11] Monitors: An Operating System Structuring Concept (Hoare, C.A.R., 1974)
[12] Operating System Principles (Hansen, P.B., 1973)
[13] Operating System Concepts, 9th edition (Silberschatz, 2013)
[14] RC 4000 Software: Multiprogramming System (Hansen, P.B, 1969)
[15] The Programming Language Concurrent Pascal (Hansen, P.B., 1975)