Babix: an Educational Multitask Kernel for Arduino Due

François Pessaux
firstname_lastname@yahoo.fr / firstname.lastname@ensta.fr

May 5, 2022

I want to thank **Maxime Ayrault** who worked on the second version of Babix during his internship and fruitfully contributed to the implementation of mutex and semaphores. He also debugged few insidious bugs I didn't see or didn't have time to kill.

1 What is Babix?

Babix is a little preemptive multitask kernel for Arduino Due I wrote to illustrate the basic principles of task-switching in an operating system. Its purpose is first of all *educational*, hence Babix remains *simple* enough to be taught. Babix is written in C as far as possible, with a little bit of assembly inlines (since it is impossible to escape from them when writing low-level operating system stuff ☺). Processes can be dynamically created simply providing the address of their entry function.

Therefore, Babix is **not** a complete, industrial or state-of-the-art operating system, **not** *real-time* and **not** fully optimized.

It should normally not turn your Arduino Due into a hot and smoking device, nor make it explode. I have tortured several boards a long time without killing any of them. But, just in case, if yours gets transformed into a "I-don't-know-what", please do not send lawyers at my door ☺.

Downloads

Babix sources are freely available at `http://francois.pessaux.perso.sfr.fr/arduino.html#Babix`. Two versions can be downloaded. The "lite" version is a bit simpler, without processes termination. The "normal" version adds processes termination. The "lite" version is intended not to change o much, in order to keep it simple to understand. The "normal" version may be subject to evolutions and extensions.

2 Aim of this Document and Lecture Guide

This document explains the basics of a preemptive multitask kernel, from the concepts up to the implementation. Prerequisites are very poor. The reader is expected to fluently program in C (especially master pointers), to roughly know the structure of a μ-processor (registers, stack, state register) and to have at least heard about assembly code. All the rest will be described here.

The section 3 introduces the concept of multitask and the high-level view of what to do to run "simultaneously" several "programs". A presentation of the Arduino Due board, and especially its μ-controller is addressed in section 4. The section 5 is the step-by-step implementation of the simplest kernel. In section 6 we refine the implementation to allow "programs" to end (not any more being infinite loops). Synchronization primitives (mutex and semaphores) are addressed in section 7.

3 The Basic Concepts

First of all, let's get rid of this bad expression "a running program". A program is a *static* thing: either a source code (written in any language, but OCaml is among the best ones, just before C ☺) or its compiled form, i.e. an inert executable binary file. When a binary executable is loaded in memory (of any kind, RAM, ROM, EPROM…) to be ran, its "living" instance, running on the processor (CPU), is rather called a *process* (or sometimes *task* with some subtle differences for specialists).

3.1 Multitask

By multitask, one mostly (and intuitively) means "several processes running at the same time". However, **one** CPU can only execute the code (instructions) of **one** process at the same moment. The magic is that the operating system will make so that the CPU will execute a little bit of instructions of each processes in turn. Hence, globally, the user will feel that all his processes are running at the same time. But, at each instant, only **one** process is really running.

There are several ways to run several processes:

- *Cooperative multitask*: In such a model, a process is responsible for leaving the CPU and let it available for another process. In other words, processes must be friendly together. A "nasty" process may decide to never give the hand to the others. When a process is about giving the hand, it is responsible for preparing its leaving.
- *Preemptive multitask*: The operating system decides itself to interrupt the currently running process and gives the hand to another one. Processes are not "masters" and never know when then will be interrupted.

Since a process never knows when it will be interrupted it cannot prepare itself for leaving. *A fortiori*, it must even not be conscious that it can / will be interrupted. All the interruption, removal from the CPU and return on the CPU must be fully transparent for a process.

Which process is chosen at each turn and who long the CPU will run its instructions before switching to another process is a matter of *scheduling policy*. The literature is gorgeous on this complex topic and we will adopt a very simple policy.

Conclusion: Babix is a preemptive multitask kernel and we only address this kind of multitask in this document.

3.2 Time Sharing

We must ensure that each "a certain duration" the operating system stops the current process and runs another one, until the next time. The "certain duration" is called a *quantum of time*. It is the amount of time attributed to a process to run before it gets asleep.

While a process is running, the CPU is occupied on it: there is no way to magically have the operating system "running in background" to monitor the elapsed time and to stop the process once its quantum is elapsed. We need another mean to "interrupt" the CPU once the quantum of time is elapsed: an *interrupt*, and more specifically a *timer* interrupt.

An interrupt is a hard-wired mechanism of the CPU allowing it to stop its execution when an event occurs and jump to execute a routine "at fixed address" depending on the kind of interrupt. Such a routine is called an *interrupt handler*. Once the handling of the interrupt is finished, the CPU returns to the code it came from. Available interrupts depend on the kind of CPU as well as what is saved by this latter before entering the interrupt handler (to be able to return to the code to execute where it was interrupted).

A *timer* is a kind of hardware clock "ringing" once the time it was configured for is elapsed. "Ringing" means . . . raising a dedicated interrupt.

Conclusion: with the duo timer / interrupt, we have a way to stop the CPU from executing a process every quanta of time.

3.3 Scheduling Policy

Once the operating system has taken the hand it must asleep the process that was running but also choose a new process to make it running. The simplest policy is "each process" in turn, in equal portions of time and in circular order: *round-robin* policy. Think to a ring where each process passes in front of the CPU to run:

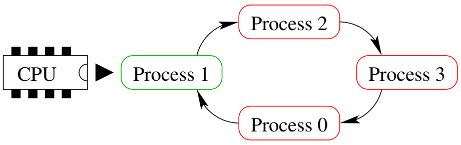


Figure 1: Simple Round-Robin Scheduling Policy

The standard data-structure used to implement such a mechanism is a *queue*, i.e. a FIFO (First In First Out) data-structure. A queue has two operations: *enqueue* to insert an element at the end of the queue, *take* to extract the first element of the queue. Some implementations can also add a function to test whether the queue is empty.

Conclusion: Babix uses a simple round-robin scheduling policy implemented by a simple queue.

3.4 Context Switch

When a process is interrupted its "state" must be saved to be able to restore it when it will get the CPU the next time. In the same spirit, the process chosen to be ran must retrieve the state in which it was the last time it was interrupted. These two actions are called the *context switch*.

The context is the "state" of a process at the time it is interrupted and is simply made of all the CPU's registers values. In a "real" operating system, the context may also include some extra information like coprocessors registers, memory management unit (MMU) tables, etc. Babix does not implement memory protection nor memory virtualization, hence only the CPU registers will be saved.

- **Question 1**: "what are the registers to save?"
  All the *general purpose registers* (used to perform arithmetic, logical, moves etc. operations), the *stack pointer* (often called `SP`) storing the current address of the stack top, the *status register* (often called `SR`) storing the condition bits of the last executed instruction, the *program counter* (often called `PC`) storing the address of the next instruction to execute. Saving the `PC` may raise questions in the reader's mind since ... once arrived in

the context switch routine, the PC has already changed compared to when the CPU was executing the process's code. We will see further that the hardware helps us.

- **Question 2**: "where to save these registers?"
  There are at least two solutions. First one, save in a memory area owned by the kernel and specific for each process. This means that when the kernel creates a new process, it must allocate some memory to store its context. Second one, directly on the stack of the process. This is the retained solution for Babix since it is very simple. However, the nasty point is that if the process heavily uses the stack, if at the context switch moment there is not enough stack remaining to push the context, the process will simply crash. If the kernel does not check the available space on the stack it will also crash. Babix does not perform this check, again it aims at being simple to understand.

The outgoing process's context must be saved and the incoming process's context must be restored. Restoration is fully symmetric. Registers are written with the values from the saved context. In particular, to give the hand to the new process code, the PC is written with the (saved) address of the next instruction to execute when this process was interrupted.

Conclusion: The context switch consists in saving all the registers of the CPU at the moment where the outgoing process is interrupted, then choose a new process to run and finally restore the registers of the CPU with the saved context of the new process.

4 The Arduino Due and its μ-controller

Now that basics are set, we need to address a little the internals of the Arduino Due and especially its μ-controller (MCU) since to properly context switch, handle interrupts, etc. the kernel has to be aware of the structure and processes of the MCU (what are the registers, what does happen when an interrupt occurs, how stack is organized, etc.).

This presentation is very light and cannot replace the official datasheet of the MCU [2] and the one of the board [1].

The core of the Arduino Due is a μ-controller SAM3x8E (SAM3X ARM Cortex-M3) at 84 MHz from Atmel Corporation. This μ-controller hosts a ARM 32 bits architecture and several input/output interfaces (digital, analogical, CAN, USB, etc.).

4.1 Registers

The registers of the SAM3x8E can mostly be summarized as (c.f. SAM3x8E datasheet section 12.4.3 page 61):

- 13 general purpose registers (R0 - R12).
- A stack pointer register (R13 or SP).
- A link register (R14 or LR) used to store the return address of a subroutine call.
- A program counter register (R15 or PC) where the MCU stores the address of the next instruction to execute.
- A status register PSR where the condition bits of the last executed instruction are and some other stuff about the state of the MCU.
- 4 other registers used to control the MCU and its exceptions (we won't care of them for sake of simplicity).

Note that the SP is a banked register, i.e. depending on the MCU state ("user" or "supervisor", i.e. "unprivileged" or "God mode") this register automatically "points" to two different registers. This allows to have separate stacks, one for the “normal” processes and one for the kernel (operating system), which enhances the security of the kernel. Babix does simple and does not have separate stacks.

4.2 Stack

The `SAM3x8E` implements a full descending stack. This means that the stack pointer indicates the **last stacked item** on the stack memory (SAM3x8E datasheet section 12.4.2 page 60). This is not like the usual implementation of stacks in software where the stack pointer indicates the next free “cell” on the stack!

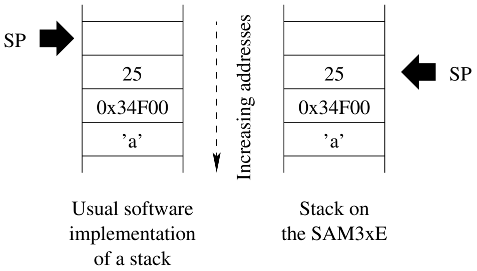


Figure 2: Usual Software Versus `SAM3x8E` Stacks

The stacks grows towards the **decreasing** addresses. In other words, when pushing a data, the stack pointer decreases. When popping a data, the stack pointer increases. Hence, for an empty stack, the stack pointer designates the ended of the memory allocated for the stack.

4.3 Exceptions / Interrupts

In the remaining of this document we may also use the term *interrupt* for exception. There is no difference on this MCU and the `SAM3X8E` datasheets prefers the term `exception`.

When an exception occurs, a complex mechanism is triggered. We address here a slightly simplified presentation, leaving some dark details untold. These omitted details do not have a primordial importance for our simple kernel. The complete description of the exception mechanism can be found in the `SAM3x8E` datasheet section 12.6.

4.4 Exception Entry

When an exception occurs, the MCU automatically pushes on the stack eight data words of 32 bits each (c.f. `SAM3x8E` datasheet section 16.6.7.5 page 85). This *stack frame* contains the values stored in the registers `R0 - R3`, `R12`, `LR`, `PC` and `PSR`. The order in which they are pushed, hence their layout in the stack will be addressed later in 5.2.2.

The fact that the `PC` is automatically stacked is the important point for having an exception mechanism working. More precisely, at the stacking moment, the `PC` already indicates the address of the next instruction (of the interrupter process) to execute. If this save was not done by the hardware, it would be impossible to save the `PC` without executing a “move” instruction . . . hence automatically modifying the `PC` before it is saved.

In parallel to the stacking mechanism, the MCU performs a lookup in the exception *vector table* to get the address of the code (handler) to execute. The vector table can be seen as an indirection array, indexed by the number of the occurred interrupt and containing the address of the related handler. The `PC` gets loaded with the address of the corresponding handler.

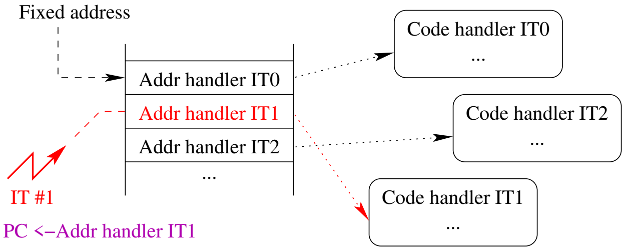


Figure 3: Exception Vector Table Mechanism

Also in parallel, the MCU writes an `EXC_RETURN` value to `LR`. Hopefully, `LR` is also saved automatically by the hardware! This value is a bitmask to later load in the `PC` to specify the end of the exception handler, hence the exception return.

4.5 Exception Return

When a handler has finished its processing, it must load the `EXC_RETURN` value into the `PC`. The `EXC_RETURN` is the one that was automatically stored in `LR` at exception entry (c.f. `SAM3x8E` datasheet section 16.6.7.6 page 86). This value is a kind of "marker" that the MCU recognizes as an illegal instruction address but as an "end of exception handler" tag. The MCU then automatically restores, from the stack, the registers that were automatically saved at exception entry. Once done, the `PC` recovered the address of the next instruction of the process to run, then the process continues without having even noticed that it has been interrupted.

5 Going to the Practice

It is now time to enter in the gory details, where these details have their importance. Implementing a multitask kernel on another MCU will inevitably introduce small but subtle and important differences due to the exception management, the MCU's instructions etc. However, a very small part of the kernel is impacted . . . mostly part written in assembly code. Indeed, in C there is no way to access the registers of a processor. The compiler know them and use them for the code it generates. But never the user has the hand on them . . . unless he introduces *assembly inlines* in his C code.

Attention: Source files must be C and not C++. In effect, C++ introduces some nasty mechanisms (overloading mostly) preventing some mechanisms we will use later. If you however want to use C++ features (like the most probable one: `Serial.print ()` for debug), you will have to enclose specific pure C material by:

```
#ifdef __cplusplus
extern "C" {
#endif
Blah blah pure C source.
#ifdef __cplusplus
}
#endif
```

Not following this rule will make some strange things happening, like impossibility to redefine "weak symbols". Be aware the `.ino` source files of Arduino are compiled as C++ files (as well as `.cpp` ones). `.c` files are compiled as regular C.

5.1 Basic Data-Structures

We first present the basic types and data-structures of the kernel. All of them are located in `kernel.h`.

typedef uint32_t mcu_word_t ;

This represents the machine word of the MCU. Since the `SAM3X8E` is a 32 bits processor, so is the machine word. This will especially be the size of the registers we will save and restore.

typedef int16_t proc_id_t ;

A process must be identified in a unique way. We choose to number them. Since the Arduino Due has 96 Kb of memory, we wont create tons of processes, hence 2^{16} identifiers will be largely sufficient!

#define MAX_PROCESSES (8)

Since the kernel will have to store the processes, we must set a maximal number of processes living at the same time. To ease extensibility, you could have used a linked list instead. Anyway, do not forget that each process will have a memory space for its stack. Memory is limited on the Arduino Due. If too many processes are allowed, you will run out of memory possibly just the the stacks segments.

```
struct pid_queue_t {
  int16_t cur_nb ;        /* Current number of elements. */
  int16_t first ;         /* Index of the first element. */
  proc_id_t pids[MAX_PROCESSES] ;
};
```

The scheduler being a round-robin one, we need a queue. This queue will store only the processes identifiers. The implementation of the queue is very standard and does not present any particularity. For this reason, we do not detail it.

```
#define STACK_SIZE (1 << 9)
struct process_t {
  proc_id_t pid ;
  mcu_word_t *top_stack ;
  mcu_word_t *sp ;
};
```

The kernel needs to record some information about each living process. It needs to keep its identifier, the starting address of the allocated stack in order to be able to free this memory once the process is ended and finally, the stack pointer of the process. This last information is crucial since it represents the address where the context of the process has been saved when it was switched from running to asleep. In effect, since the context of a process is saved onto its stack, one only needs to remind the process's stack pointer to retrieve the whole context.

```
struct kernel_t {
  /** Saved stack pointer of the main process. */
  mcu_word_t main_task_sp ;
  /** Array containing all the living processes. */
  struct process_t processes[MAX_PROCESSES] ;
  /** Identifier of the currently active process. */
  proc_id_t current_process_id ;
  /** FIFO for round-robin simple scheduling. */
  struct pid_queue_t queue ;
};
```

Finally, the kernel is represented by a unique structure, with an array of processes, the scheduling queue and the identifier of the currently running process.

The field main_task_sp seems a bit odd. It represents the stack pointer of the "main" task, i.e. the process parent of all the other processes. Indeed when your program starts, before creating processes, there is already one running. In fact, it is the invisible main () that the Arduino build process inserted to wrap the setup () and loop () standard functions. If no "new" process is created or if all the dynamically created ones are ended, your program is still running. What does it do? It simply runs the loop () function until the board is powered off. And when the last dynamically created process ends, the kernel must give back the hand to the initial "process". Since it is not really a process it has no related **struct** process_t and *a fortiori* is not in the array of processes. So we need save / restore its context in another location: in this field.

Note that it is double-satisfactory that this initial "fake process" is not considered as the others since it has nothing special to do and is ran only when there is no real processes. And we don't want to schedule (i.e. periodically give some time to) a useless piece of code doing nothing. This kind of initial parent of any other process also exists in "other" operating systems and is often call idle (to reflect that when it runs, there is nothing else to do: the machine is idle).

It would also be possible to keep the "idle" process in the table of processes, simply not put it in the queue to prevent from having it scheduled. This may have simplified a very little bit de context switch routine. For the next version... ☺

#define SYSTICK_FREQUENCY_HZ (1000)

We must finally define the quantum of time allocated to the processes, i.e. the delay between task switching. Since we use a timer, we rather define the frequency (the inverse of the quantum) at which an active process is switched off.

5.2 The Context Switch

We now address the context switch routine, leaving the creation of a process for later since it can be more easily understood once the context switch is understood. The implementation related to this section is in context.c.

5.2.1 Triggering a Context Switch

To periodically trigger the context switch we could have used one of the general purpose timers of the Arduino Due (known as TCx with x from 0 to 3). However the SAM3X8E has a dedicated timer for operating system stuff, leaving the other timers free for user applications. This timer is called SysTick and can be configured to generate a SysTick interrupt periodically. We then could have decided to perform the context switch in the handler of this interrupt. However, the best practices and guidelines of the SAM3X8E strongly advice to use instead a dedicated exception: PendSV. Handling an exception takes some time (depending of the length of the code to execute). The SysTick interrupt also serves for other functions of the "standard library" of the Arduino. Hence, it is better to keep it fast and short.

Let's summarize. A SysTick exception will be generated each quantum of time. The handler of SysTick will raise a PendSV exception. The handler of PendSV will perform the context switch.

It is not directly possible to attach a handler to the SysTick exception since the Arduino development environment already installs one in the compiled programs to make some "standard" functions of the "standard" library working properly. Instead, a smart mechanism is used: *weak symbols* supported by the gcc compiler.

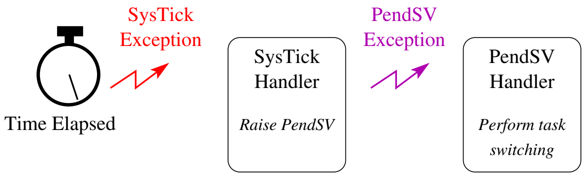


Figure 4: Context Switch Sequence

A weak symbol is a ... symbol that can be redefined by the programmer. Such a symbol may have a default value. If the programmer defines his own symbol (typically a function) with the same name than a weak symbol, instead of complaining “multiply defined symbol”, the compiler will use the user’s one instead of the default one. If a weak symbol does not have a default value, then if the user tries to use it (typically calling a function) then the compiler will raise an “undefined symbol” error. If the weak symbol does not have a default value but is not used in the program, then it is simply ignored.

The handlers defined in the `Arduino` library for the `SysTick` and `PendSV` call each a “hook” function which is weakly defined with a default behavior (doing nothing for the first one, and stopping the program – i.e. looping forever – for the second one). These definitions can be found in the `Arduino` standard library files `cortex_handlers.c` and `hooks.c`.

```
void SysTick_Handler(void)
{
  if (sysTickHook()) return;
  ...
}
void PendSV_Handler (void) { pendSVHook(); }

int sysTickHook(void) __attribute__ ((weak, alias("__false")));
void pendSVHook(void) __attribute__ ((weak, alias("__halt")));
```

Hence, we will simply redefine `sysTickHook ()` and `pendSVHook ()`. The first one will simply trigger a `PendSV` interrupt by writing in the right MCU register. The second one will be our context switch routine.

Note: attention to enclose these functions in an **extern** C” ” block as previously stated in 5, otherwise the redefinition of the symbols won’t work.

```
int sysTickHook ()
{
  SCB->ICSR |= SCB_ICSR_PENDSVSET_Msk ;
  return (0) ;
}

__attribute__ ( ( naked ) ) void pendSVHook (void)
{
  /* ... Our coming code to perform the context switch. */
}
```

5.2.2 What to Save / Restore

It is now time to understand what is saved by the hardware on the stack and what do we need to save in extra. As stated in 4.4, the hardware automatically saves the registers `R0` - `R3`, `R12`, `LR`, `PC` and `PSR` at exception entry. And we are in a function *called by* a handler. So, these registers have been saved. Moreover, our function `pendSVHook ()` having no arguments (of type `void`), we know that the stack won’t have changed since the hardware did its save.

The other registers must now be "manually" saved in the outgoing process's context. This includes the registers R4 – R11. But attention, as stated in the section 4.4, the register LR has been loaded with an EXC_RETURN value that will have to be loaded in the PC when the outgoing process will get the hand in the future. Hence, LR must also be saved in the context.

To summarize, we have 2 kinds of context saves: the one automatically done by the hardware at exception entry and the one the context switch routine must do in software. This lead to two structure in kernel.h:

```
struct soft_save_t {
  mcu_word_t r4 ;
  mcu_word_t r5 ;
  mcu_word_t r6 ;
  mcu_word_t r7 ;
  mcu_word_t r8 ;
  mcu_word_t r9 ;
  mcu_word_t r10 ;
  mcu_word_t r11 ;
  mcu_word_t lr ;
};
```

```
struct hard_save_t {
  mcu_word_t r0 ;
  mcu_word_t r1 ;
  mcu_word_t r2 ;
  mcu_word_t r3 ;
  mcu_word_t r12 ;
  mcu_word_t lr ;
  mcu_word_t pc ;
  mcu_word_t psr ;
};
```

The tricky question is "why must the fields of these structures be layout in this order, and not in the reverse one or even another order?". The answer comes from two points.

- The C point: when the compiler allocates a structure on the stack, the starting address is at the lowest address in a descending stack.
- The "push" instruction of the SAM3X8E (which is the STMDB instruction) performs "accesses in order of increasing register numbers, with the lowest numbered register using the lowest memory address and the highest number register using the highest memory address" (citation from the SAM3X8E datasheet section 12.12.6.2 page 110).

The combination of these two facts leads to the given correct layout of the C structure according to the way the MCU works. A complete context to save (and symmetrically, restore) for a process can hence be sketched by the following figure.

5.2.3 The Switch Routine

First of all, as sketched in the previous section 5.2.1, the definition of pendSVHook () starts by a strange annotation: __attribute__ ((naked)). This attribute tells to the compiler "do not generate any code saving registers or anything at the beginning of this function". Normally, the compiler knows in the code it generates for a function which are the registers used, hence whose content coming from the caller will be killed. Hence, to avoid spoiling the caller's state, the compiler saves these registers at the entry of the function then restore them at the exit.

In our case, our function has for unique aim to exactly start by saving the registers, then replacing their content by some other values. We do especially not want the compiler to modify these registers in our back! Moreover, the compiler does not know the full story: there are not only the registers used in the function to save, but all the registers. And it does not also know that is must not restore them: we do, with some particular values the compiler does not know about. With this attribute, we are then sure that when entering our function, no extra instructions will be added before our own code and no extra ones will be after our own code.

The context switch routine has a very simple structure separated in three steps.

1. Save the registers of the outgoing process that were not saved by the hardware at exception entry.

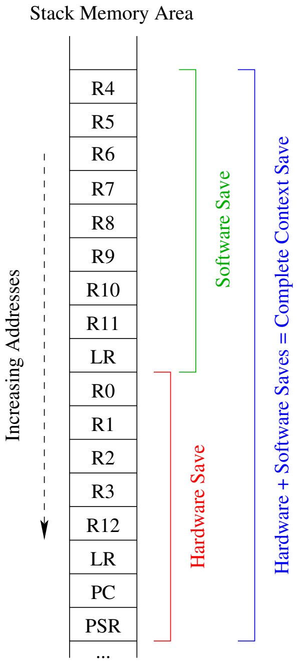


Figure 5: Context Structure in Memory on the Stack

2. Choose new process to run (incoming process).
3. Restore the registers that were not saved by the hardware from the incoming process's context.

Doing any other computation before the step 1 would kill the registers, hence definitively corrupt the system. During the step 2, we can do any computation we want without any fear to kill the registers: they all have been saved. After the step 3, nothing can be done since the MCU will have its registers reloaded with the correct content for the incoming and the MCU will continue its execution by resuming the incoming process.

To access the registers we need to inline some assembly code. `gcc` allows to interface "easily" C and assembly via the `asm` construct. This allows the assembly code to read and write C variables and perform jumps to C labels. A complete documentation on this construct can be found in [3]. We provide here the minimum information to understand the coming code.

An `asm` construct can have two general forms, depending on whether it is a "computational"instruction or a "jump" instruction:

- `asm` [`volatile`] `(` asm-template `:` out-operands `:` in-operands `:` clobbers `)`
- `asm` [`volatile`] `goto` `(` `:` in-operands `:` clobbers `)` `:` C-goto-labels `)`

`volatile` instructs the compiler to not remove the inlined assembly code for any reason and forbids some optimizations. *asm-template* is a string containing assembly instructions in which operands can refer to C variables listed in *out-operands* and *in-operands* (`%0` is the first of the out/in list, `%1` is the second and so on). *out-operands* and *in-operands* are lists of comma separated C expressions. `clobbers` is a comma separated list of registers and values modified by the assembly instruction in addition of those listed in the *out-operands*.

In a `asm goto`, the *C-goto-labels* represents the list of **all** the C labels where the inlined assembly code may jump.

It is important to understand that the compiler **does not** “read” the content of the inlined assembly code! It does not know nor understand what this code does. All the semantics of the effects done by the inlined assembly code must be reflected in the out, in and clobbers lists. Missing one of them may cause the compiler ignore that some values or registers changed (or not) during the instruction. This will spoil the compiler’s assumptions and it will generate incorrect code for the surrounding C code you wrote: the interface will be broken.

For the same reason, using a “branch” assembly instruction in a regular `asm` instead of in an `asm goto` will make so that the compiler does not “see” a jump, hence will mostly probably generate wrong code.

Now, back to the code... The `push` and `pop` instructions on the `SAM3X8E` are aliases to `STMDB` and `LDMIA`. We present here the skeleton of the context switch routine and we will complete it incrementally. The prelude saves the current (outgoing) process’s context. The postlude restores the new (incoming) process’s context and triggers the exception return.

```
__attribute__ ((naked)) void pendSVHook (void)
{
  /* Push the context on the current process's stack.
     push is a synonym for STMDB sp!, relist */
  asm volatile (".save_outgoin_context: \n\t") ;
  asm volatile ("push     {r4 - r11, lr} \n\t") ;

  ... Find a new process to run and get the pointer on its context ...

  /* Pop the context of the incoming process on its (the current process's)
     stack.
     pop is a synonym for LDMIA sp! reglist */
  asm volatile (".restore_incoming_context: \n\t") ;
  asm volatile ("pop {r4 - r11, lr}    \n\t" :::
    "r4", "r5", "r6", "r7","r8", "r9", "r10", "r11") ;
  /* Exit from exception by jumping at LR which must be 0xFFFFFFF9.
     See SAM3x8E datasheet 16.6.7.6 page 86. */
  asm volatile ("bx lr                          \n\t") ;
}
```

There remains three things to do: find a new process to run, then if some exists, record the outgoing process’s context address in the kernel and finally switch to the incoming process’s context address in order to have `SP` pointing at the right location before the final “pops”.

New process election Since processes are stored in a queue, this part simply relies on queue manipulation. The outgoing process must be re-inserted in the queue (but only if it is not the “idle” task : remember, as stated in 5.1, we do not want to schedule the initial “fake process”). The incoming process must be extracted from the queue and its identifier is reminded as the one of the new running process.

```
  ...
  asm volatile ("push     {r4 - r11, lr} \n\t") ;
►
  /* Guess the outgoing process id. */
  outgoing_proc_id = kernel.current_process_id ;
  /* Only enqueue the outgoing process if it is not the "Idle" task. */
  if (outgoing_proc_id != MAIN_PROCESS_ID)
    enqueue (&kernel.queue, outgoing_proc_id) ;
  /* Guess the incoming process id. */
  incoming_proc_id = take (&kernel.queue) ;
  /* Record the new running process. incoming_proc_id could be removed and
     replaced by kernel.current_process_id in fact. */
  kernel.current_process_id = incoming_proc_id ;
  ...
}
```

Saving the outgoing process's context address If the outgoing process is the idle task then is it not in the kernel's table of processes. Hence the context address must be saved in the separate location `kernel.main_task_sp` (c.f. section 5.1). Otherwise, the context address is saved in the field `sp` of the process's structure in the kernel array of processes. To do so, we load `R1` with the address where to store the context address.

Attention: we conceptually want to assign a variable which would correspond in C to outgoing_proc−>sp = ... ; or kernel.main_task_sp. However, this cannot be done directly in assembly. On must load the address of the variable in a register, then write **at the address** contained in this register. This is a kind of ∗ptr = ... ; in C, where ∗ptr is considered as an output. However, when loading the address in `R1`, this address is an **input**, not an output! In effect, the `mov r1, %0` instruction does not modify itself the memory location designated by `%0`.

Once the address where to store the value of `SP` is loaded, we must get the content of `SP`. This is done using the instruction `mrs` which stores this value in `R2`.

Finally there only remains to transfer `R2` at the address designated by `R1`. The particular clobber `"memory"` tells to the compiler that "the memory is modified somewhere" by this instruction.

```
  ...
  kernel.current_process_id = incoming_proc_id ;
►
  /* Save the outgoing process' SP.
     If the process is the main process, then save its SP in the kernel
     structure, otherwise in the process's structure.
     r1 = outgoing stack pointer. */
  if (outgoing_proc_id != MAIN_PROCESS_ID) {
    outgoing_proc = &kernel.processes[outgoing_proc_id] ;
    asm volatile ("mov r1, %0      \n\t" :: "r" (&outgoing_proc->sp) : "r1") ;
  }
  else asm volatile ("mov r1, %0  \n\t" :: "r" (&kernel.main_task_sp) : "r1") ;
  /* Really save outgoing SP. */
  asm volatile ("mrs r2, msp     \n\t" ::: "r2") ;
  asm volatile ("str r2, [r1]    \n\t" ::: "memory") ;
  ...
}
```

Restoring the incoming process context's address The outgoing process's context address being saved, we simply need to update the register `SP` with the incoming process context's address. This way, the coming "pops" will act on the incoming process's stack, where its context was saved. Symmetrically to the save, the restoration is done either from the idle task's storage location or the kernel's table of processes. We load in `R0` the address where to get the value to load in the `SP`. Then writing in the `SP` register is done using the dedicated `msr` instruction of the `SAM3X8E`.

```
  ...
  asm volatile ("str r2, [r1]    \n\t" ::: "memory") ;
►
  /* Restore the incoming process' SP.
     If the process is the main process, then load its SP from the kernel
     structure, otherwise from the process's structure.
     r0 = incoming stack pointer. */
  if (incoming_proc_id != MAIN_PROCESS_ID) {
    incoming_proc = &kernel.processes[incoming_proc_id] ;
    asm volatile ("mov r0, %0     \n\t" :: "r" (incoming_proc->sp) : "r0") ;
  }
  else asm volatile ("mov r0, %0  \n\t" :: "r" (kernel.main_task_sp) : "r0") ;
  /* Really write incoming SP. */
  asm volatile ("msr msp, r0    \n\t") ;
  ...
}
```

The context switch routine is now complete.

5.3 Creation of a Process

We have written a context switch routine allowing to periodically change the running process. However, there remains an important question: “how to start a process?”. We have seen the chicken, we must now address the egg!

Obviously, to create a process we will need to allocate a stack memory area for it and put it inside the queue to have it scheduled. The most interesting question is “how to do so that, once is the queue, it will get the CPU **as if it had been switched out** in the past?”. The answer is simply to simulate a “past witch out”. Hence, we need to put on its stack the equivalent of what would have been saved by a context switch.

But what are the values to put? Indeed, the process has not yet ran. Hence, its working registers do not matter. The only important registers values are:

- `PC` which is where the process must (re)-start. It is the address of the process starting function.
- `SR` which represents the condition flags of the last instruction the process executed. Since it never ran, we simply clear the register.
- `SP` which represents the stack pointer **after** the fake context save we want to simulate.
- `LR` which must contain the `EXC_RETURN` value that the `PendSV` exception would have automatically saved in the “hardware save” (c.f. 5.2.2).

To clear the `PSR` we write 0’s everywhere except in the bit 24 which must be always 1 according to the `SAM3X8E` datasheet section 12.4.3.8 page 66.

The initial `EXC_RETURN` is the bitmask value `0xFFFFFFF9` reflecting the fact that we do not implement a separate stack pointer register for ”system” and ”user” mode (c.f. `SAM3X8E` datasheet section 16.6.7.6 page 86).

The creation of a process is done by the function `create_process ()` in `process.cpp`. The only reason why this source file is a C++ instead of a C one is to be able to print messages using `Serial.print ()`. This function takes one argument, the address of the process entry function. It must first check that the number of living processes does not exceeds the maximum allowed. If there is still some room for a new process, it must be registered in the kernel and a stack must be allocated.

Attention: we voluntarily omit the need for an exclusive access to the kernel tables by this function for the moment. We will address this question once we will have seen *mutex*, in section 7.1.

```
proc_id_t create_process (void (*code)())
{
  /** Stack top as allocated with malloc. Reminded for liberation. */
  mcu_word_t *stack_top ;
  /** Pointer to the stack once a hardware stack frame has been pushed. */
  struct hard_save_t *sp_after_hard_save ;
```

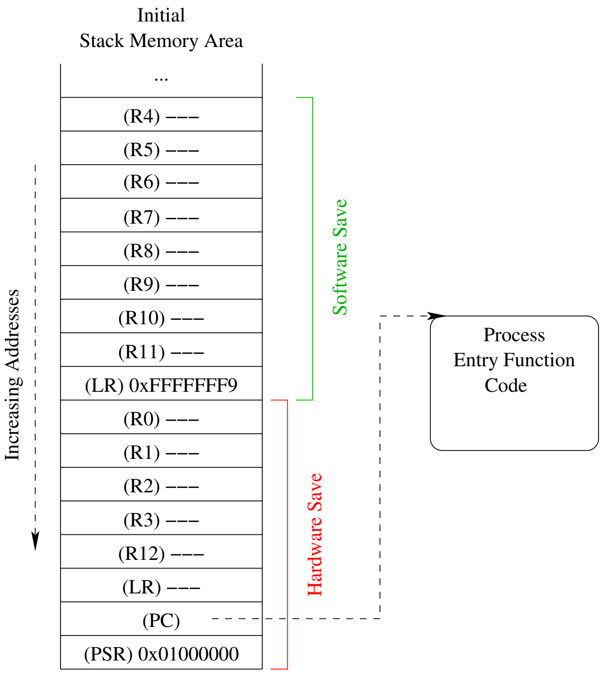


Figure 6: Initial Context Structure in Memory on the Stack

```
  /** Pointer to the stack once a software stack frame has been pushed. */
  struct soft_save_t *sp_after_soft_save ;
  int16_t nb_processes = kernel.queue.cur_nb ;

  /* Check for enough room for a new process. */
  if (nb_processes > MAX_PROCESSES) {
    return (MAIN_PROCESS_ID) ;
  }
  /* Register the process in the processes table. */
  kernel.processes[nb_processes].pid = nb_processes ;

  /* Allocate a stack segment. */
  stack_top = (mcu_word_t*) malloc (sizeof (mcu_word_t) * STACK_SIZE) ;
  if (stack_top == NULL) return (MAIN_PROCESS_ID) ;
  /* Record the top of the stack to free it later. */
  kernel.processes[nb_processes].top_stack = stack_top ;
  ...
}
```

Once the stack is allocated, we must fill its with the initial frame (i.e. hardware + software saves). We first fill the hardware save fields corresponding to `PC` and `PSR`.

Since the `SAM3X8E` implements a full descending stack, the stack pointer indicates the last stacked item on the stack memory. The stack grows towards the lower addresses. Hence, the address of the top of the stack when the hardware save "would have been done" corresponds to the beginning our `struct hard_save_t` (from which we access the fields of the structure). It is the first address after the end of the stack zone, minus the size of a "hardware save".

```
  ...
  kernel.processes[nb_processes].top_stack = stack_top ;
►
  sp_after_hard_save = (struct hard_save_t*)
    (((uint32_t) &stack_top[STACK_SIZE - 1]) - sizeof (struct hard_save_t)) ;
  /* Initialise PC with the process' function code. */
  sp_after_hard_save->pc = (mcu_word_t) code ;
```

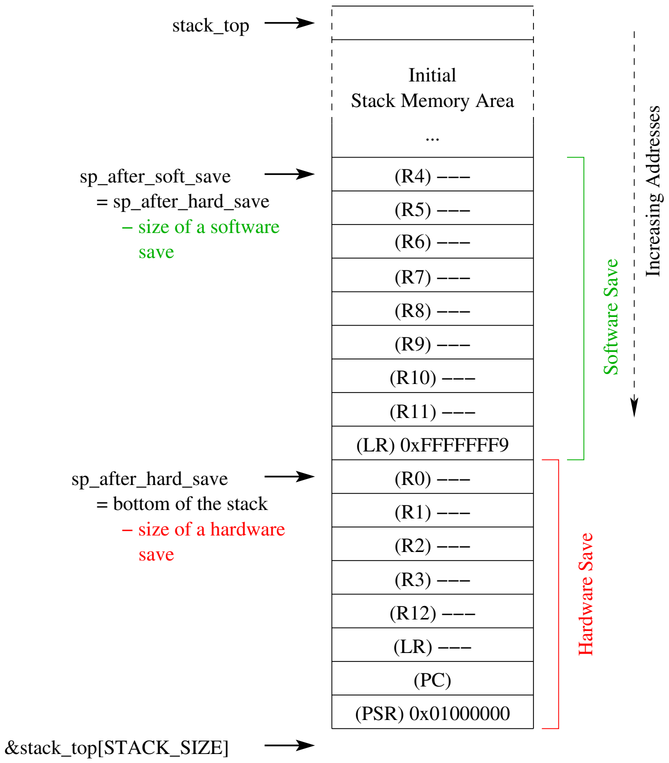


Figure 7: Initial Context Structure in Memory on the Stack (Addresses)

```
/* Clear the SR. */
sp_after_hard_save->psr = 0x01000000 ;
...
```

We proceed in the same way to fill the fields of the `struct soft_save_t` corresponding to the register `LR`. The address of the top of the stack when the software save "would have been done" corresponds to the beginning our `struct soft_save_t`. It is the address of the hardware stack frame, minus the size of a "software save".

This is also the address we store in the kernel table as the address of the process's context. Indeed, it is the final position of the stack pointer after a complete save and from where the context will be restored when the process will run for the first time.

```
...
sp_after_hard_save->psr = 0x01000000 ;
►
sp_after_soft_save = (struct soft_save_t*)
  (((uint32_t) sp_after_hard_save) - sizeof (struct soft_save_t)) ;
sp_after_soft_save->lr = 0xFFFFFFF9 ;
/* Record the process handler to the context, i.e. only its stack pointer. */
kernel.processes[nb_processes].sp = (mcu_word_t*) sp_after_soft_save ;

...
```

Finally, we just have to insert the new process in the queue in order to make it eligible for having the CPU and we increment the number of living processes.

```
...
►
/* One more process is in the pipe... Enqueue it. */
```

```
  enqueue (&kernel.queue, nb_processes) ;
  kernel.queue.cur_nb = nb_processes + 1 ;
  return (nb_processes) ;
}
```

5.4 The Initialization and "User" Tasks

Aside the core of the processes management, there remains a little bit of administrative stuff to do to have all of this working. First, a short initialization of the MCU must be done. Next, the user must launch his processes.

5.4.1 Configuration

For a kernel as simple as Babix, very few configuration is required. We must configure the interruptions to make them preemptive, to allow them to be handled and to make the exception `PendSV` having the lowest priority. This last point ensures that the task switching mechanism will not be more important than other interrupts (I/Os, etc.), hence prevents from losing other events.

The system tick frequency must be setup depending on the quantum of time wanted for the processes.

The initialisation must be done before the multitask can start. Typically, this is done in the usual `setup ()` function on the Arduino or in a separate function called by this latter.

```
void setup ()
{
  ...
  /* Set interrupts to be preemptive. Change the grouping to set no
     sub-priority.
     See SAM3x8E datasheet 12.6.6 page 84 and 12.21.6.1 page 177. */
  NVIC_SetPriorityGrouping (0b011) ;
  /* Configure the system tick frequency to adjust the time quantum allocated
     to processes. */
  SysTick_Config (SystemCoreClock / SYSTICK_FREQUENCY_HZ) ;
  /* Set the base priority register to 0 to allow any exception to be
     handled.
     See SAM3x8E datasheet 12.4.3.14 page 62. */
  __set_BASEPRI (0) ;
  /* Force the PendSV exception to have the lowest priority to avoid killing
     other interrupts.
     See SAM3x8E datasheet 12.20.10.1 page 168. */
  NVIC_SetPriority (PendSV_IRQn, 0xFF) ;
...
}
```

5.4.2 Creating Processes

This is done simply using the function `create_process ()`, providing it the address of the function to execute as a task. Since processes are not "called" by functions, they can't receive arguments. For the same reason, they do not return any result. Hence functions to run by processes must be of type **void** ... (**void**).

In this first presentation of Babix, processes are not allowed to end. This means that their functions must be endless loops. In section 6, we will relax this restriction.

```
void process2 ()
{
  for (;;) {
    Serial.print (”I’m 2 ”) ;
    delay (3700) ;
  }
}
```

```
void process3 ()
{
  for (;;) {
    digitalWrite (13, HIGH) ;
    delay (1000) ;
    digitalWrite (13, LOW) ;
    delay (1000) ;
  }
}
```

6 Allowing Process Termination

Processes are not always infinite loops, and this is even more interesting when they can be created dynamically (otherwise, once the maximal number of living processes is reached, none won’t be created later).

The “lite” version of `Babix` does not implement termination. The “normal” version does

6.1 The principle

If the function of a process ends (by an explicit **return** or simply by reaching the end of its body), the standard funcall mechanism will cause the execution to return at the caller (using a `bx LR` instruction). The problem is that in our case, there is **no** caller: we made so that the `PC` was rerouted at the function’s first instruction to run the function!

In a multitask system, the end of a process must trigger a system call to remove the process from the scheduler, free the resources it was owning (memory, stack, file handles, etc.), remove the process from the kernel tables, etc.

To trigger a system call, `Babix` must provide a `end_process ()` function dedicated to terminate the process using it. In some sense, the process commits suicide. When this function is called by the process, it must indicate that this process is about to end by any kind of marker, and must trigger a rescheduling (i.e. cause a task switching).

When the context switch routine will be invoked, it will distinguish the case of a regular context switch and the case of a process termination.

6.2 The Termination System Call

To cause a task switch, we simply do the same thing than what happens when a quantum of time is elapsed: trigger a `PendSV` exception.

To mark the process as ended, we simply chose to turn the current running process identifier in the kernel to a negative value. Hence, if this identifier is negative, the kernel knows that the process whose identifier is the absolute value must be terminated. This also means that the identifier of the “main” task (“idle”) must be 0 (it never ends).

Until the exception is really processed, the process is still running. In effect, if the process is able to call `end_process ()`, that’s because it is running. Since the process has nothing more to do, the function `end_process ()` makes it looping forever (indeed, until the task switching routine is called and see that the process is to be ended).

```
__attribute__ ( ( naked ) ) void end_process ()
{
  kernel.current_process_id = - kernel.current_process_id ;
  SCB->ICSR |= SCB_ICSR_PENDSVSET_Msk ;
  for (;;) ;
}
```

One more time, this function is marked “naked” to prevent the compiler from saving registers. This would not be a real issue since nevermore this process will run again. However, this would be totally useless.

6.3 Impact on the Context Switch Routine

In fact, the impact is very light. Changes are located **after the save of the registers** and **before the restoration of the incoming** process’s `SP`.

When determining the outgoing process we just need to test if the current process identifier is negative. If so, then we must remove it from the kernel table and free its stack. Before putting back the process, one must ensure that it is not terminated.

```
__attribute__ ( ( naked ) ) void pendSVHook (void)
{
  ...
  /* Guess the outgoing process id. */
  outgoing_proc_id = kernel.current_process_id ;
►
  /* If the PID is negative, then the process with a PID being the absolute
     value is ended and must be destroyed. */
  if (outgoing_proc_id < 0) {
    kernel.processes[- outgoing_proc_id ].pid = MAIN_PROCESS_ID ;
    free (kernel.processes[- outgoing_proc_id ].top_stack) ;
  }
  /* Only enqueue the outcomming process if it is not the "Idle" task. */
  if ((outgoing_proc_id != MAIN_PROCESS_ID) && (outgoing_proc_id > 0))
  ...
```

Then, the determination of the incoming process does not change at all. The last change is for the outgoing process’s `SP` save. We must save it only if there is one, i.e. only if we are not handling an end of process (which, in this case, doesn’t exist anymore). Hence, we must guard the code that was previously saving `SP` by a simple test ensuring that the outgoing process identifier is positive.

```
►
  if (outgoing_proc_id >= 0) {
◄
    /* Same code than previously... */
    if (outgoing_proc_id != MAIN_PROCESS_ID) {
      outgoing_proc = &kernel.processes[outgoing_proc_id] ;
      asm volatile ("mov r1, %0     \n\t" :: "r" (&outgoing_proc->sp) : "r1") ;
    }
    else
      asm volatile ("mov r1, %0  \n\t" :: "r" (&kernel.main_task_sp) : "r1") ;
    /* Really save outgoing SP. */
    asm volatile ("mrs r2, msp    \n\t" ::: "r2") ;
    asm volatile ("str r2, [r1]   \n\t" ::: "memory") ;
►
  }     /* End of added guard test. */
◄
  ...
```

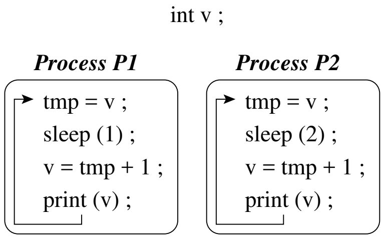
6.4 Ending a User Process

As previously stated, the process’s function simply has to call the function `end_process ()`. Attention, forgetting to call this function before the end of a process will crash as explained in 6.1.

7 Synchronization Primitives

When running several process “in paralel”, all goes right as long as none of them share resources. Resources can be files, peripherals, etc, . . . or even simply a variable.

Suppose that two process P_1 and P_2 share a common global variable `v` initialized to 0. Both of them endlessly increment this variable. Both of them read the value of `v`, wait a certain amount of time, then write back in `v` the incremented value.



```
int v ;

Process P1
tmp = v ;
sleep (1) ;
v = tmp + 1 ;
print (v) ;

Process P2
tmp = v ;
sleep (2) ;
v = tmp + 1 ;
print (v) ;
```

Figure 8: Concurrent Processes

We could expect `v` to get the values $1, 2, 3, 4, 5 \ldots$. Unfortunately one rather gets a silly sequence like $1, 1, 2, 3, 2 \ldots$. Indeed, each process makes a copy of `v` (in `tmp`) at the moment where it is running. If the process gets switched out and if the other one writes in `v`, the copy of `v` owned by the first process is now out-of-date. When this first process will get the hand, it will write its own incremented value, hence smash the one previously written. Accesses to `v` are said *concurrent*. In other words, the concurrency tends to reduce the effective *parallelism*.

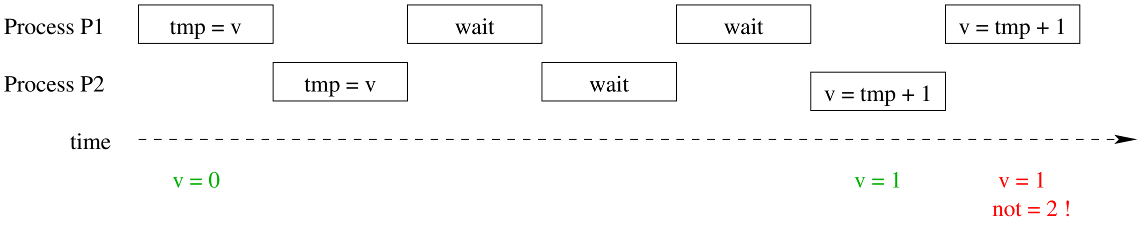


Figure 9: Concurrent Scenario for the Processes

This raises the need for *synchronization primitives*: processes must access shared resources respecting some rules. As shortly stated in 5.3, this problem already exists in the code we wrote for the kernel. Since processes can by dynamically created, if two processes try to create a process, the kernel data-structures will be accessed concurrently. For instance, one possible consequence can be to have two new processes with the same identifier.

7.1 Mutex

One of the simplest synchronization mechanism is the *mutex*. It ensures an exclusive access to a delimited zone of code: only one process can be in the part of code protected by the mutex. A mutex is a (very simple) data-structure with two operations:

- **Acquisition**: allows the calling process to attempt entering a protected section of code. While the acquisition fails, the process is not allowed to reach the section. Once the mutex is acquired by the process, it is sure that it will be the only one in the section of code.
- **Release**: allows a process having acquired the mutex to let it available to other processes. This operation never fails. It must be called at the end of the protected zone of code.

To guaranty an exclusive access to a section of code (called *critical section*), it is sufficient to enclose this section between an acquisition and a release of a mutex.

```
f ()
{
  do something ;
  mutex_acquire () ;
  do something in the critical section ;
  mutex_release () ;
  do something ;
}
```

Basically, the data-structure underlying a mutex is a simple boolean flag telling if the exclusive access is already in use. Hence, in C, it is a simple integer. The file `mutex.h` defines a type alias, **typedef volatile** uint32_t mutex_t ; to represent a mutex. Obviously, a **short** or even a **char** could have been chosen.

Conceptually, the acquisition function (`mutex_acquire ()`) must check if the mutex is `true` or `false`. If it is `true` then the mutex is already occupied and the acquisition fails. If it is false then the mutex is free, it must be toggled to `true` and the acquisition succeeds.

However, if two process are trying to acquire the mutex, between the reading of the mutex value and its writing, a context switch can occurs. This causes the same problem than the one described in the introduction of this section! The access to the mutex's variable is concurrent. Same problem than the initial one for our expected solution ☹.

The hardware provides the solution: the instruction sets of nowadays CPUs contain a few number of atomic (i.e. non-interruptible) instructions. Depending on the architecture, these instructions may vary. On the `SAM3X8E`, we will use the `ldrex` and `strex` instructions (c.f. `SAM3X8E` datasheet section 12.5.7 page 78 and section 12.12.8 page 112).

`ldrex Rt, [Rn]` loads a word from a memory address. `strex Rd, Rt, [Rn]` tries to store a word (contained in `Rt`) to a memory address (contained in `Rn`). The address used in the instruction must be **the same** as the address in the **most recently** executed `ldrex` instruction. If `strex` succeeds the store, it writes 0 to its destination register (`Rd`) and it is guaranteed that no other process has accessed the memory location between the `ldrex` and the `strex` instructions.

With these instructions, the pseudo-code algorithm for a mutex acquisition gets simple:

```
mutex_acquire (mutex)
{
restart:
  while 'ldrex (mutex)  == 1' ;  /* Wait for the mutex to be free. */
  if 'strex (1, mutex) == 1' goto restart ;
}
```

Once again, the real implementation requires to inline assembly code. To be efficient and to avoid saving the registers we use, we only uses the registers `R2` and `R3`. In effect, these registers (with `R0` and `R1`) are considered as *scratch registers*. This means that, by convention, a calling function must expect them to be destroyed by the called function.

The mutex is passed as argument by address. In effect `mutex_acquire ()` must write in the variable of the mutex. To do so, there is no other solution than passing its address.

The coming code is pretty simple and comments should be sufficient for the understanding. Note the use of the `asm goto` statement for branching instructions (as stated in 5.2.3).

```
void mutex_acquire (mutex_t *mutex)
{
  /* No need to save the registers used in this function. In effect, we only
     use scratch registers r2 and r3. Gcc seems to store our argument in r0
     which is also a scratch register. */
 take:
  /* Ask an exclusive access on the lock and get its value by the way
     (c.f. datasheet section 12.5.7 page 78 and section 12.12.8 page 112. */
  asm volatile ("ldrex r3, [%0]   \n\t" :: "r" (mutex) : "r3", "memory") ;
  /* Check the value of the lock. If it is taken (==1) we must retry. */
  asm volatile ("mov r2, #1   \n\t" ::: "r2") ;
  asm volatile ("cmp r2, r3   \n\t") ;
  asm volatile goto("beq %0   \n\t" :::: take) ;  /* Loop back. */

  /* Lock is not == 1, hence is free. Let's try to acquire it by trying
     storing 1 inside.
     Not-success value returned value in r3.
     Value to store in r2 (i.e. value 1).
     Address of the lock is 'mutex'. */
  asm volatile ("strex r3, r2, [%0]  \n\t" :: "r" (mutex) : "r3", "memory") ;

  /* Check return value: if 0 write succeeded otherwise failure. In case of
     failure we must retry the whole process (i.e. reading the lock's value
     and if possible trying to write 1 inside. */
  asm volatile ("mov r2, #1  \n\t" ::: "r2") ;
  asm volatile ("cmp r2, r3      \n\t") ;
  asm volatile goto("beq %0       \n\t" :::: take) ;  /* Failure: loop back. */
  return ;
}
```

The release of a mutex is trivial. Since the calling process is sure to have an exclusive access to the mutex, no *race condition* (concurrent access) is possible. Hence `mutex_release ()` simply resets the variable of the mutex, putting 0 inside. Its implementation is written with assembly inlines, but it could be pure standard C.

```
void mutex_release (mutex_t *mutex)
{
  /* No need to save the register, we use a scratch register. */
  asm volatile ("mov r2, #0    \n\t" ::: "r2") ;
  asm volatile ("str r2, [%0]  \n\t" :: "r" (mutex) : "memory") ;
  return ;
}
```

Obviously, a process releasing a mutex that it didn't obtain but that was obtained by another process will spoil the mechanism! It would unlock the critical section although a process is still inside. It is the responsibility of the programmer to take care of his usage of synchronization primitives.

To prevent race conditions on the kernel data-structures, the function `create_process ()` seen in 5.3 must start by a mutex acquisition and end by a release of this mutex. Attention, the mutex must be released in any case of exit of the function! This means that **each** return case must not forget to call `mutex_release ()` otherwise, the mutex will remain locked forever, preventing any future call to `create_process ()` to succeed.

7.2 Semaphore

A semaphore is simply a generalization of a mutex where the access to the protected section of code is restricted to a certain number of processes, not only one. The underlying data-structure of a semaphore is still an integer, initialized with the maximum number of allowed processes in the code section it guards.

A function `sem_acquire ()` (often caller `P ()` in the literature) tries to get the access to the semaphore. If the semaphore's value is not 0 then the access is granted and the value is decremented. Otherwise, the access is denied. Symmetrically, a function `sem_release ()` (often caller `V ()` in the literature) liberates an access to the semaphore, incrementing its value.

The implementation of semaphores in `semaphore.c` is based on the same instructions than for the mutex. The structure of the code is very similar, involving the `add` and `sub` assembly instructions instead of simply some `mov`.

References

[1] Arduino. *Arduino Due.* https://www.arduino.cc/en/Main/arduinoBoardDue.

[2] Atmel Corporation. *SAM3X/SAM3A Datasheet. SAM3X/SAM3A Series Complete.* http://www.atmel.com/devices/sam3x8e.aspx.

[3] GNU Software Fundation. *Extended Asm - Assembler Instructions with C Expression Operands.* https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html.