

Lab Work

CSA – Architecture, System, Networks

The lab work is performed on ARM / AT91SAM7S simulator called *BoardSim*. The lab work assessment is based on the delivering of sources made during the lab work and running on *BoardSim*.

1 Using Boardsim

BoardSim is a versatile microcontroller board simulator. It can be configured to support several types of boards and particularly the AT91SAM7S. It is contained in the Lab Environment provided on the Moodle page of the module.

TODO download it and unpack it.

```
tar xvf ASN.tgz
```

You will get a directory named ASN containing *BoardSim* and the sources used in each lab work exercise.

1.1 Preparing a project

To run BoardSim with a specific board and a specific program, you have to type:

```
boardsim board-file-path executable-file-path
```

To get an executable file, you have to use a cross-compiler to produce ARM code. You can use the `arm-elf-gcc` cross-compiler provided on your OS. A classic cycle of development includes the following steps:

1. Edit the source file with your preferred editor (geany or kate for example).
2. Cross-compile the source file (.c) to get an ARM executable file (.elf).
3. Run the obtained executable with *BoardSim* and with a board configuration file.
4. Test and debug the obtained program in the *BoardSim* environment.
5. If the program still contains bug, restart at step (1).

To help you a bit, we have prepared Makefile scripts to make automatic these steps as much as possible:

TODO

1. Type command: `cd lab1/lab11`

2. Type command: `geany lab11.c &`

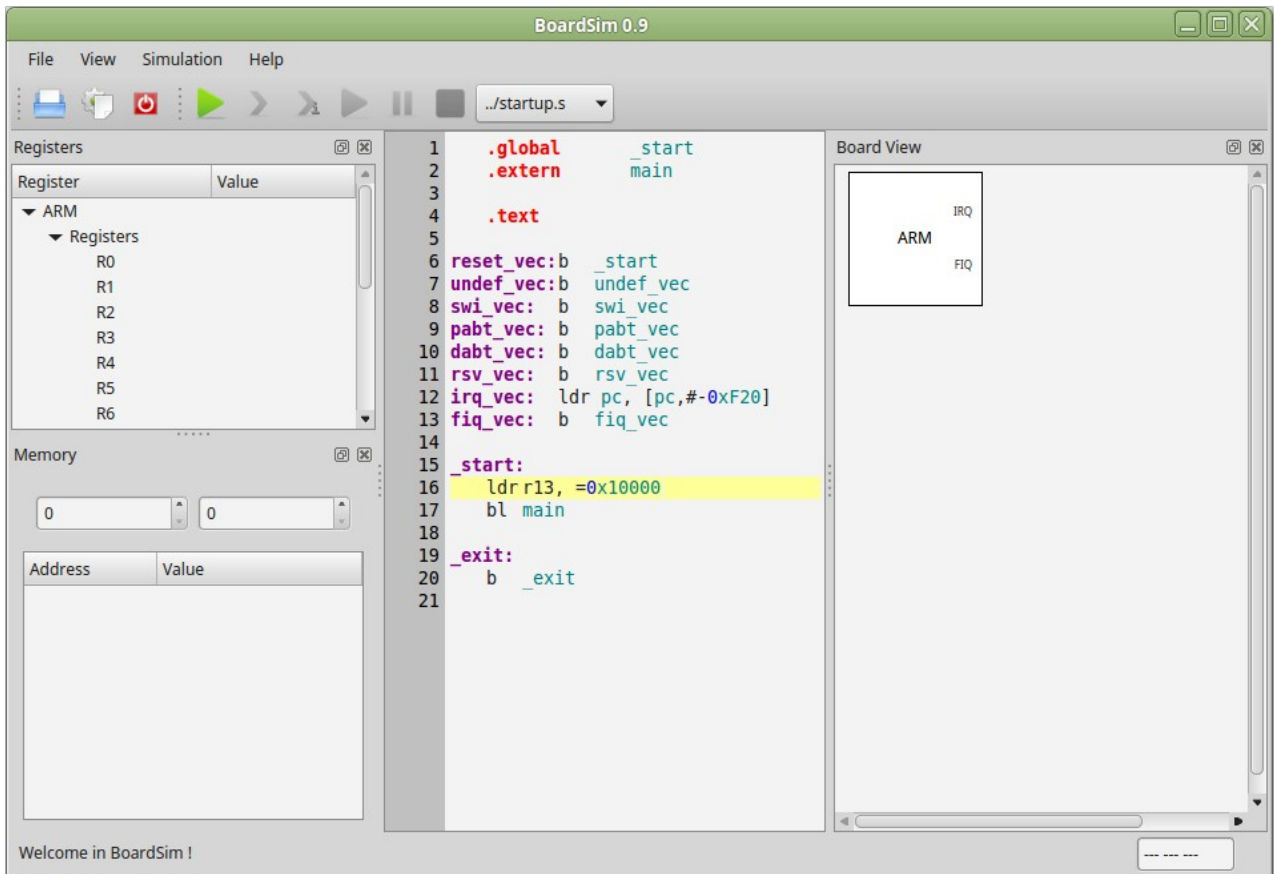
3. Examine this little C program. You have to notice that embedded systems usually does not provide `printf`/`scanf` functions and save it if you modify it.

4. Coming back to the console, built it with the command: `make`

5. Run *BoardSim* with command: `make run`

1.2 Using BoardSim

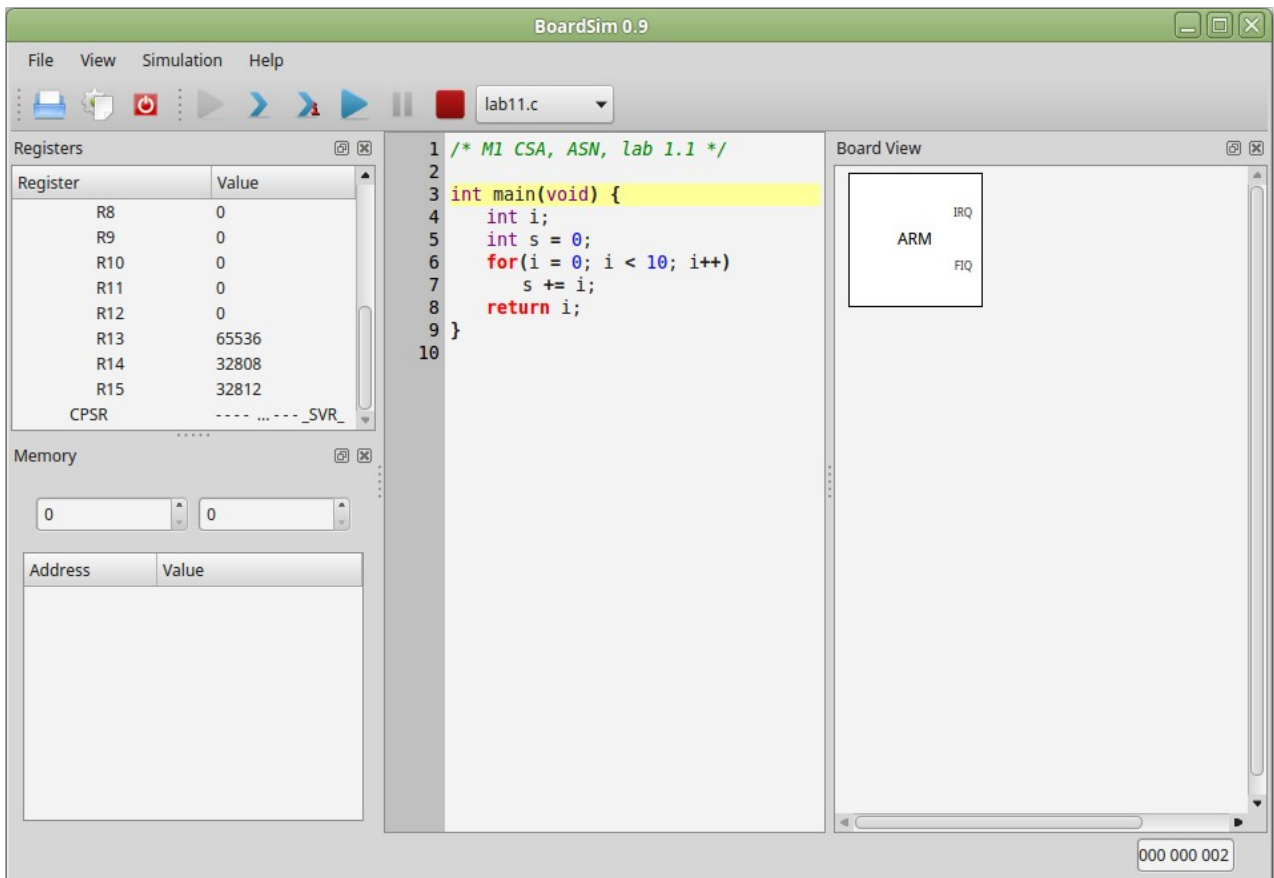
BoardSim provides a Graphical User Interface providing a board and microcontroller simulator and an interface to view its work. The main window looks like this:



This window is divided in the following view:

- left-top – view providing mostly in real-time the value of the microprocessor registers and of the peripheral controllers registers (on real hardware, it is not always possible to see them).
- left-bottom – view of the memory of the current program.
- middle – the view of the program sources or the disassembled view. Don't be afraid if the current view displays assembly, this is the start up code that mainly sets the exception table.
- right – view of the board itself showing the components of the board. For now, it only display the ARM core.

To start the simulation, click on the green  arrow and the window will become:



Now the displayed source is your source C file and the simulation is stopped at the entry of the main function: the initialization code has been performed and the main program is about to be executed. The current execution point in the C source is marked by the line with a yellow background.

New buttons are now active to perform the simulation:




Simulate to the next source line: this may represent the execution of several machine instructions.



Simulate 1 machine instruction: a source line is usually made of several machine instructions.

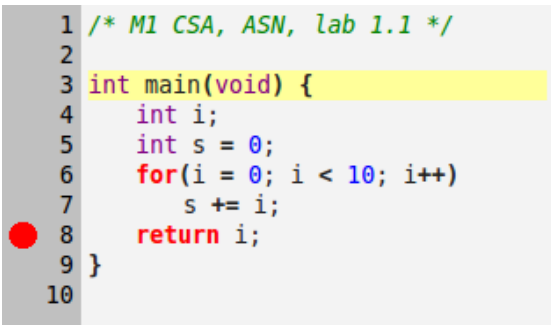


Launch the simulation: it will stop only if it encounters a break point, or if it is paused with button .



Stop the simulation that, then, can be restarted with .

A break point is a special position in the program where the simulation stops as soon as the corresponding code is executed. To set a break point, double-click in the source view on the left of the line (before line numbers) where you want to stop the simulation. Break points make possible to execute lots of code and to stop the execution at a point you want to observe (to debug for example):





```

1  /* M1 CSA, ASN, lab 1.1 */
2
3  int main(void) {
4      int i;
5      int s = 0;
6      for(i = 0; i < 10; i++)
7          s += i;
8      return i;
9  }
10

```

TODO

1. Using , execute several iterations of the loop and observe how the ARM registers evolve. Could you identify which registers contain `i` and `s`?
2. Put a break point on line 8 and run the program to this break point using .
3. Stop the simulation and exit from BoardSim.

2 Programming the PIO

The PIO controller provides access to external peripheral bound on the parallel interface of the microcontroller. For now, we are only interested about 2 LEDs (GREEN and YELLOW) and 2 push-buttons (PUSH1 and PUSH2). They are soldered, respectively, on pins 16, 17, 18 and 19. They work in negative logic, 0 for switched on / pushed, 1 for switched off / false.

This exercise aims to drive the PIO using the I/O registers definitions provided with the board in the file `lab/at91sam7s.h`.

TODO Open the file `lab1/at91sam7s.h` and examine it to find back the definitions of the PIO registers.

Now we want to implement a small application using the PIO. We want to switch on/off the LEDs according to the user action on the push-buttons:

- the YELLOW LED is switched on when SW1 is pushed, switched off else,
- the GREEN LED switches between on and off on each click on SW2 (a click is the sequence of press then release on SW2).

We will use the PIO to drive LEDs and push-buttons. The AT91SAM7S has a PIO with 32 pins. Therefore, the 32-bits of the PIO registers corresponds each one to one of the 32 pins. Writing a 1 to a register bit trigger the action corresponding to the register on the corresponding pin. The pins which bit are 0 are unaffected by the assignment to the register. This is why we say that a mask is written to these registers where the bits corresponding to the pin we want to use are set to 1. Such a mask is easy to build in C using left-shift operations: `1<<b` builds a mask whose bit `b` is set to 1 and other bits are cleared.

The file `at91sam7s.h` contains the masks for using the LEDs and the push-buttons:

```
#define YELLOW 0x10000
#define GREEN 0x20000
#define PUSH1 0x40000
#define PUSH2 0x80000
```

The program has to be written and compiled in the directory `lab1/lab12` in the file `lab12.c`. and you have to complete the main function.

First we have to initialize the PIO:

```
PIO_PER    = YELLOW | GREEN | PUSH1 | PUSH2;  (a)
PIO_OER    = YELLOW | GREEN;                  (b)
PIO_ODR    = PUSH1 | PUSH2;                    (c)
PIO_SODR   = YELLOW | GREEN;                   (d)
```

(a) The four pins will be managed by the PIO.

(b) Pins YELLOW and GREEN are used as output.

(c) Pins PUSH1 and PUSH2 are used as input.

(d) Pins YELLOW and GREEN are set to 1 (LED switched off).

To produce a 1 (5V) on a pin n , one has to write to `PIO_SODR` with the n th bit set to 1. To set a pin to 0 (0V), one has to write to `PIO_CODR` with the n th bit set to 1. The input state of the pins can be read from the `PIO_PDSR` register using a mask with n th bit to 1 to mask out other bits. For example, to wait for a 0 on PUSH1 pin, we can write:

```
while((PIO_PDSR & SW1) != 0) ;
```

TODO Design the state/transition diagram (with paper and pen) for each pair YELLOW / PUSH1 and GREEN / PUSH2.

TODO Complete the source file `lab1/lab12.c` to implement the exercise program and test it on *BoardSim* (notice that the button from the component view can be clicked).

3 Using Interrupts

The ARM core has only two interruption lines: IRQ (Interrupt ReQuest) and FIQ (Fast Interrupt reQuest). A microcontroller as AT92SAM7S contains lots of peripherals, each one providing their own interrupt. To manage them together efficiently, AIC (Advanced Interrupt Controller) is used to identify the interrupt sources and to provide the precise interrupt source to the program.

From a hardware point of view, the AIC has 32 input lines where an event (descending front) represents a raised interrupt. The connection between the AIC and the peripheral interrupt lines depends on the configuration of the micro-controller. The wiring of interrupt is documented inside the technical manual of the micro-controller, in our case, the manual of AT91SAM7S. From this documentation, we can learn that:

- line 2 (`ID_PIO`) provides PIO interrupts,
- line 12 (`ID_TC0`) provides Timer 0 interrupts.

When the interrupt are programmed, one has first to configure the AIC to be sure that any interrupt will be well handled, then to configure the interrupt in the peripheral controller that will produce them and finally to unmask the interrupts in the core (that is to enable the interrupt in the status register bits).

3.1 Configuring the Interrupt

In this exercise, we use the push-buttons to generate the interrupts, that is, we will handle the interrupts coming from the PIO. So we have to:

1. configure in the PIO which lines will cause an interrupt,
2. configure the AIC to accept interrupts coming from the PIO,
3. unmask the interrupt inside the status register.

An interrupt will be raised each time a change happen (ascending or descending front) on the PIO input line configured to support interrupts. In turn, the core will stop and switches to interrupt state and then call the interrupt handler, a C function provided by the AIC.

To initialize an interrupt handling by AIC, we have first to disable this interrupt in the AIC:

```
AIC_IDCR = 1 << ID_PIO;
```

Then, the interrupt vector (SVR) and the interrupt mode (SMR) corresponding to the PIO can be configured in the AIC (as the PIO is an internal device, the mode is not used):

```

void handle_pio(void);
AIC_SVR[ID_PIO] = (uint32_t)handle_pio;
AIC_SMR[ID_PIO] = 0;

```

Then the PIO itself can be configured (as previously done). The register `PIO_IER` is used to enable interrupts on the push-buttons lines:

```

PIO_PER = YELLOW | GREEN | PUSH1 | PUSH2;
PIO_OER = YELLOW | GREEN;
PIO_ODR = PUSH1 | PUSH2;
PIO_SODR = YELLOW | GREEN;
PIO_IER = PUSH1 | PUSH2;

```

In the end, we reset the AIC to a stable state without any pending interrupt or acknowledge and then we enable the interrupts on the PIO:

```

x = AIC_EOICR;
AIC_ICCR = 1 << ID_PIO;
AIC_IECR = 1 << ID_PIO;
UNMASK_IRQ ;

```

3.2 The Interrupt Handler

The interrupt handler is a C function that is invoked when an interrupt is raised. It is advised to keep handler function as small as possible to avoid to spend too much time inside an interrupt and to loose some other waiting interrupts. Another reason to keep interrupt handler small is that interrupts are usually hard to debug. Depending on the architecture of the core, the interrupt handler function has special initialization and termination code: this is why we have to inform the compiler about the nature of this function with a special attribute:

```

void __attribute__((interrupt("IRQ"))) handle_pio(void) {
    int v;

    /* interrupt processing */

    v = PIO_ISR;
    v = AIC_EOICR;
}

```

At the end of the interrupt handler, we have to signal the AIC and the PIO that we have complete the process of the interrupt. We have to read the register `PIO_ISR` for the PIO and the register `AIC_EOICR` for the AIC.

TODO Change to directory `lab1/lab13` and edit the file `lab13.c`. Inside this file, implement the 2-LED 2-push-button program from the previous exercise using interrupts. To check the interrupt, you can put a break point at the entry of the interrupt handler.