EPFL

# CS-473: System programming for Systems on Chip

Practical work 5

# PIO, interrupts, and latency's

**Version:**
1.0

# Contents

# 1 Introduction

## 1.1 Prerequisites

On moodle you find the file `pw5.zip`; it contains the complete source code of the brute force accelerated version of the Mandelbrot set. Download this file. This template contains a directory called `buttons`, where the `makefile` can be found as for our previous PW.

## 1.2 PIO, interrupts, and latency

In the theory we have seen the PIO. The final goal of this PW would be to make an interrupt driven application that allows us to zoom in/out of the fractal, and to move around. Of course you are allowed to use your own implementation of the fractal generator that you made in a previous PW. Please note that the VGA-controller *fetches* each line in a burst of `nrOfPixelsPerLine/2` 32-bit words.

# 2 Exercises

## 2.1 Prerequisites

Familiarize yourself with the accelerated version of the Mandelbrot set. Compile the program and make sure that you do not have any errors. Then, download the `buttons.cmem` file to your virtual prototype and see the result.

Before continuing, read the document `switches.pdf` found on moodle.

## 2.2 Polling

The most easy way of reading out a PIO is to just poll it's contents, and print the result if there is a change. In the file `switches.h` you find some predefined constants. To be able to do the polling, we need a pointer to the PIO. In `main.c` add after:

```
volatile unsigned int *vga = (unsigned int *) 0X50000020;
```

the line:

```
volatile uint32_t * switches = (uint32_t *) SWITCHES_BASE_ADDRESS;
```

Now perform following task:

> In the endless loop, write all code that is required to read out the dip-switche, buttons, and joystick. Only in case that there is a change in the state of the dip switche/buttons/joystick, print out the new state. **Hint:** you need for each of the buttons, dip-switch, and joystick, two variables, one holding the old state, and one holding the current state.

## 2.3 IRQ-driven

Polling is a very inefficient way of reading out a PIO. Most of the time the CPU is just reading the old value, and not able to do anything else. More efficient is when the CPU is only *called to do something* when there is a change, hence an external interrupt. To be able to enable an external interrupt on your system, several steps need to be taken to be able to enable the interrupt. Read:

- Chapter 14 of the document `openrisc-arch-1.4-rev0.pdf` found on moodle in the week indicated by **Bios and exceptions**.
- Section 4.6 on the *supervision register (SR)* in the above mentioned document.

> Which bit needs to be set in the *supervision register (SR)* and which bits in the *PIC mask register* to be able to receive the interrupts of the dip switch, buttons, or the joystick?
> **Hint:** read also again the document `switches.pdf` to know the answer to the bits in the *PIC mask register*.

Now that we can enable the external interrupt, we have also to enable the interrupt sources on the PIO.

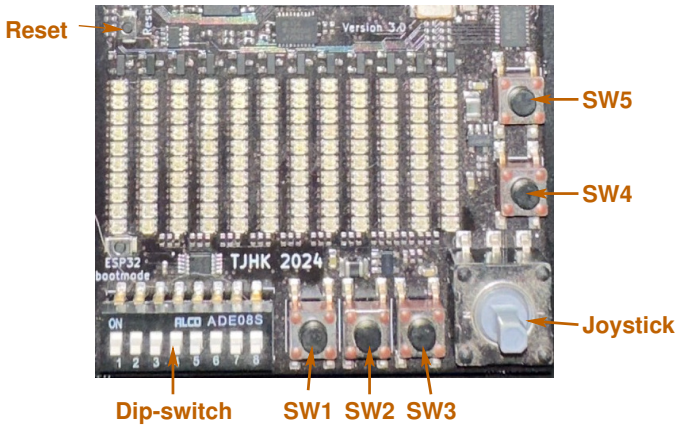


Figure 2.1: Button definitions on the GECKO5Education.

Write all the code that is required to enable an interrupt on the pressing on SW2, or setting dip-switch to 1 (see figure 2.1 for the definitions).

When executing your program, the moment you press SW2, or you set dip-switch1 to 1, you will observe that your system will indefinitely put the text ping on your screen. This comes from the external interrupt handler, defined as:

```
__weak void external_interrupt_handler() {
  puts("ping\n");
}
```

and the fact that the generated IRQ is never cleared. Hence we have to do another step, we have to write our own handlers that clears the external interrupt.

Write your own external_interrupt_handler() in main.c (hence do not modify exceptions.c), that calls the function *buttons_handler()*, in case of an interrupt generated by the buttons, and the function *dipswitch_handler()*, in case of an interrupt generated by the dip-switches. In the function *buttons_handler()*, read out the *buttons IRQ generator*. This will clear the generated interrupt and will indicate which button was pressed. Also add to *buttons_handler()* a puts("button handler"). Do the same thing for the functions *dipswitch_handler()* and *joystick_handler()*, but put here a puts("dipswitch handler"), respectively puts("joystick handler").

When executing this program, you should now see each time you press SW2 the text button handler. Also when you put dip-switch to 1, you should see the text *dipswitch handler*.

## 2.4 Latency's

Now that we have the interrupt-driven buttons working, it is interesting to know, what is the cost involved (*latency*). The buttons-PIO implements a latency counter that counts the number of system cycles between activating an IRQ, and it's deactivation. We are going to use this to determine the latency.

Add at the end of your *buttons_handler()* a printf statement that prints the value in decimal of the latency counter. Run the program and press several times on SW2 in different intervals, what can you observe?

We see that the values printed above quite vary. We do the same experiment by disabling the vga-screen by commenting following line in main.c:

```
vga[3] = swap_u32((unsigned int)&frameBuffer[0]);
```

Comment the above line in `main.c`. Run the program and press several times on `SW2` in different intervals, what can you observe? How can you explain the differences with the above experiment?

It is also interesting to know how long the CPU is busy handling the interrupt. To measure this, we are going to use the performance counters. To do this experiment, we have to add following changes:

- Add at the end of the *buttons_handler()* the statement:

```
redraw = 1;
```

  This will trigger the redraw of the fractal in the main endless loop.

- Modify:

```
if (redraw == 1) {
  redraw = 0;
  drawFractal(frameBuffer);
}
```

  to:

```
if (redraw == 1) {
  redraw = 0;
  perf_print_cycles(PERF_COUNTER_RUNTIME,"irq runtime");
  drawFractal(frameBuffer);
}
```

Run the same experiments as before. What can you observe? How many interrupts/sec would the CPU be able to handle in theory?

## 2.5 (optional) Using it all

Now that we have all working on interrupt bases, we are going to implement an application, with which we can move and zoom in to/out of our Mandelbrot. To be able to do this, there are three global variables:

- `delta`, this one you already know, it defines the "zoom factor".
- `cxOff`, this is a positive number that defines the offset to `CX_0`.
- `cyOff`, this is a positive number that defines the offset to `CY_0`.

Before continuing please remember to un-comment following line:

```
// vga[3] = swap_u32((unsigned int)&frameBuffer[0]);
```

We are going to define following functions (see also figure 2.1 for the definition of the buttons):

- `joystick-left`: Go left. This can be implemented by subtracting the step-size from `cxOff`.
- `joystick-right`: Go right. This can be implemented by adding the step-size to `cxOff`.
- `joystick-up`: Go up. This can be implemented by subtracting the step-size from `cyOff`.

- `joystick-down`: Go down. This can be implemented by adding the step-size to `cyOff`.
- `SW5`: Zoom in. This can be implemented by dividing `delta` by 2.
- `SW4`: Zoom out. This can be implemented by multiplying `delta` by 2.

**Hints:**

- For moving the step-size `delta` represents one pixel. It is easier to navigate if you choose a step-size of 5*`delta` or 10*`delta` (moving by 5 or 10 pixels).
- When zooming in, you will zoom into the left-top region of the visible fractal. It is with a simple calculation possible to zoom into the center of the visible fractal.

Implement your nice Mandelbrot application.