Lab 1.2
Joystick Interface

Lab 1.0 + 1.1 – PWM Software/Hardware Design (recap)

The previous labs in the 1.x series put you through the following progression:

- Lab 1.0 – You learnt some theory behind how one interacts with a hardware peripheral, then went on to practice what you learnt by means of a programming exercise. The practical task involved writing the control software in C for a *black-box* PWM generator connected to a pan-tilt module.
- Lab 1.1 – The control software written in lab 1.0 was left untouched, but you replaced the black-box PWM generator with a *custom* hardware design you wrote yourself in VHDL. You also learnt how to write a VHDL testbench so you can test the hardware components you will be developing throughout the course.

Lab 1.2 – Joystick Interface

Theory

Introduction

The previous labs were centered around *output* devices, so it’s now time to switch gears and explore *input* devices. We use the scenario below as motivation for this lab:

Until this point, the control software was issuing the movement pattern of the servomotors by means of a hard-coded algorithm in the C source code. We now want to update the movement, but it is quite tedious due to the hard-coded nature of the algorithm. Furthermore, each modification requires extensive debugging to be sure it works as expected, so a lot of work needs to be done for a simple change in the movement algorithm. Last week we saw David Wheeler’s “fundamental theorem of software engineering” which states that “all problems in computer science can be solved by another layer of indirection”. We follow the steps below to apply the words of wisdom contained in the theorem in order to fix the issue described above:

1. Remove the hard-coded movement pattern from the control software and push the burden of providing the expected position of the servomotors to the *user*. This can be done by adding a new layer of indirection by means of an *input device*.
2. The user interactively manipulates the input device to provide his/her desired position for the servomotors.
3. The control software *reads* the input device to obtain the user’s desired position.

4. The control software uses the desired position to update the position of the servomotors accordingly by *writing* to the PWM unit's registers.

Essentially, our control software will be movement-pattern-agnostic with these changes as it no longer has any hard-coded pattern in its source code since the user provides the pattern at runtime.

Joysticks

The input device we will use in this lab the *PlayStation 2 joystick* shown in Figure 1. I sense nostalgia must already be building up!

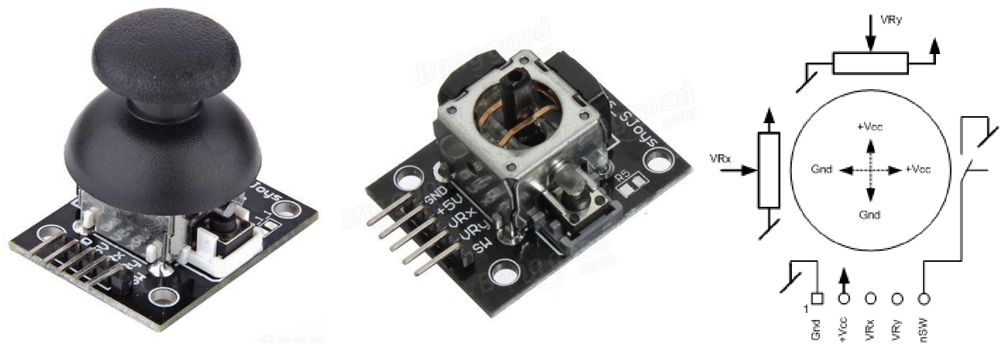


FIGURE 1. PLAYSTATION 2 JOYSTICK

The device consists simply of an input-controlled resistor on each axis, where the resistance ranges from approximately 30 Ω to 3.6 *k*Ω depending on the position of the joystick. The device has a 5-pin connector with the following connections:

- GND
- +5V
- VRx: *Analog* X-axis
- VRy: *Analog* Y-axis
- SW: *Digital* button

Analog to Digital Conversion

Hardware

Since we want to connect the joystick to our FPGA's *digital* GPIO pins, we need to convert the *analog* output of the VRx and VRy pins to a digital signal before we can do anything with it. We will use an *Analog to Digital Converter (ADC)* for this purpose.

The extension board has 2 joysticks, each with 2 analog outputs, so we will need a 4-channel ADC. The extension board used in this course uses the MCP3204, a 4-channel, 12-bit ADC. Figure 2 shows the MCP3204's package, and Figure 3 shows how the joysticks are connected to it on the extension board.

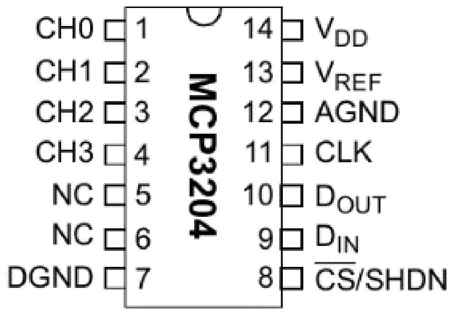


FIGURE 2. MCP3204

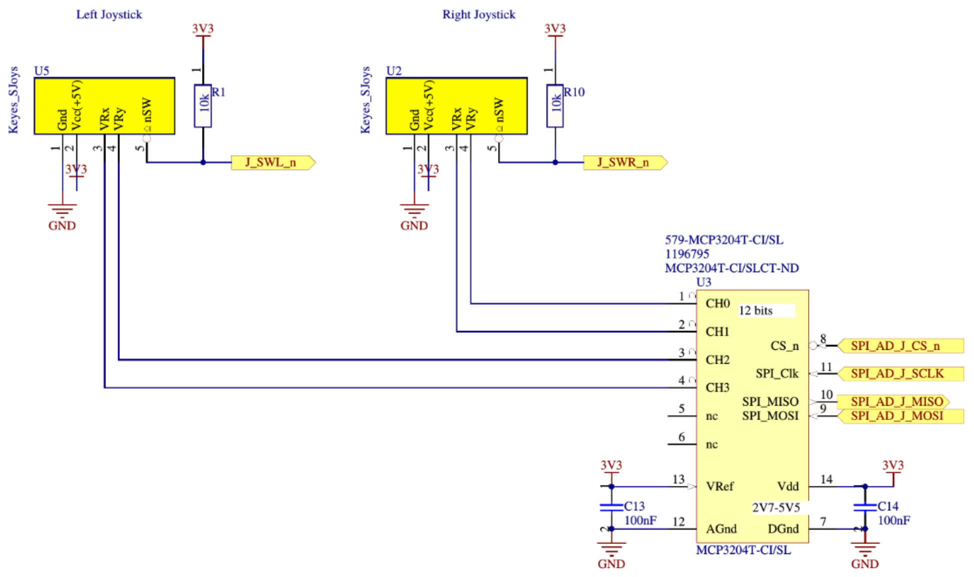


FIGURE 3. JOYSTICKS + MCP3204 CONNECTIONS ON EXTENSION BOARD

Communication protocol

The ADC interfaces with the analog outputs of the joysticks and converts the analog value to a digital one. What remains to be determined is how this digital value is transmitted to the FPGA. This problem occurs quite often in computer systems and manufacturers have, over time, proposed standard *communication protocols* for this purpose. There are numerous communications protocols out there (SPI, I^{2}C, 1-Wire, UART, …), and every device provides one or more such interfaces in order to be compatible with as many SoCs as possible.

The communication interface available on the MCP3204 is the *Serial Peripheral Interface (SPI)* bus. Figure 4 shows the details of the communication protocol between the ADC (SPI *slave*) and the FPGA (SPI *master*)[1].

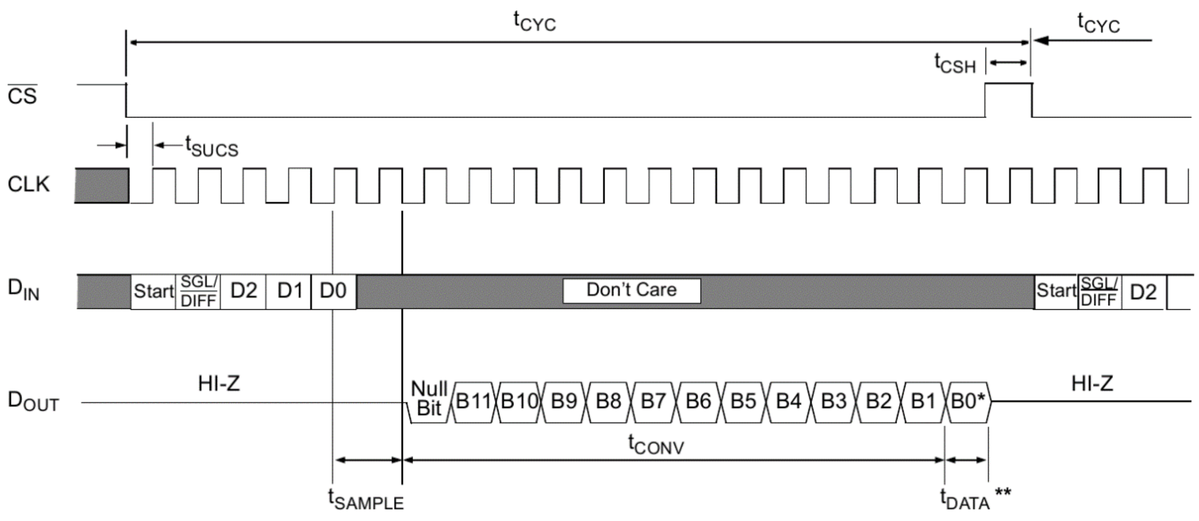
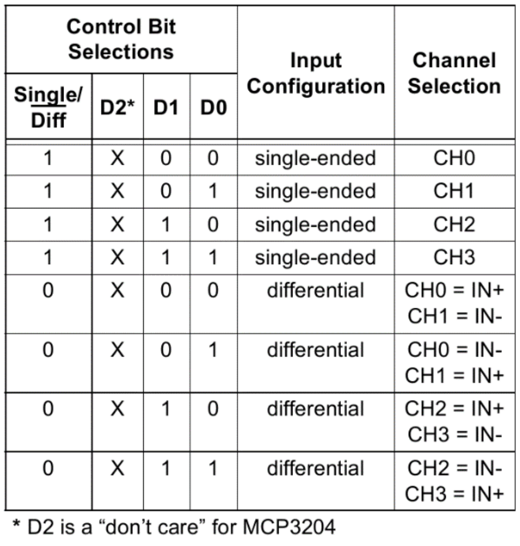


FIGURE 4. MCP3204 SPI COMMUNICATION PROTOCOL

[1] Note that the terms “master” and “slave” come up often in this field and are not Avalon-specific.

In the specific case of the MCP304, the same SPI bus is used both for *commands* and *data*. To obtain the digital value corresponding to one of the analog channels, you need to send a sequence of commands to the device, then the device responds with the corresponding data. Table 1 shows the configuration bits of the command sequence that needs to be sent to the MCP3204.



Control Bit Selections				Input Configuration	Channel Selection
Single/Diff	D2*	D1	D0		
1	X	0	0	single-ended	CH0
1	X	0	1	single-ended	CH1
1	X	1	0	single-ended	CH2
1	X	1	1	single-ended	CH3
0	X	0	0	differential	CH0 = IN+ CH1 = IN-
0	X	0	1	differential	CH0 = IN- CH1 = IN+
0	X	1	0	differential	CH2 = IN+ CH3 = IN-
0	X	1	1	differential	CH2 = IN- CH3 = IN+

* D2 is a “don’t care” for MCP3204

TABLE 1. CONFIGURATION BITS FOR THE MCP3204

FPGA-ADC interface

Top-level design

In order to communicate with the ADC, we need to design a VHDL module that can supply commands to the ADC and that can read the converted values back through the SPI bus. We implement the *Avalon-MM slave* module shown in Figure 5 for this purpose.

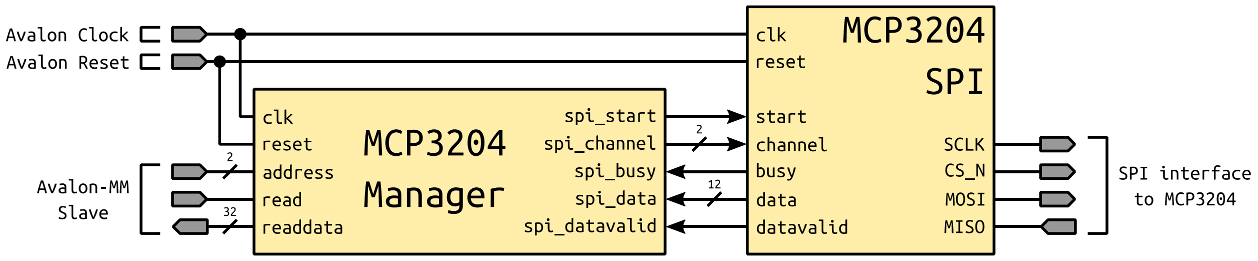


FIGURE 5. MCP3204 AVALON-MM SLAVE BLOCK DIAGRAM

The module consists of 2 parts:

- *MCP3204 Manager*: this unit interfaces with the host processor through an Avalon-MM slave interface and forwards requests to a custom SPI controller which then communicates with the MCP3204 and returns the conversion results. Table 2 shows the details of the unit’s register map. Note that all the unit’s registers are *read-only*, as it is impossible to “write” to the joysticks, so we can omit any write-related signals from the Avalon bus to simplify the design. The unit is supplied with a *50 MHz* clock.

Byte offset (from base)	Name	Access	Description
0	CHANNEL_0	RO	12-bit digital value of channel 0
4	CHANNEL_1	RO	12-bit digital value of channel 1
8	CHANNEL_2	RO	12-bit digital value of channel 2
12	CHANNEL_3	RO	12-bit digital value of channel 3

TABLE 2. MCP3204 MANAGER REGISTER MAP

- *MCP3204 SPI Controller*: this unit is responsible for performing the actual SPI communication to retrieve the converted values from the ADC. The unit is supplied with a *50 MHz* clock, but all *SPI communication* must be done at *1 MHz*.

Note that *conversions* are *not* done when a *request arrives* through the Avalon-MM slave interface, but are rather done *continuously* in the background: The *Manager* contains a state machine where it constantly instructs the *SPI controller* to perform a conversion when it is idle. The result of each conversion is stored in one of the *Manager's* 4 internal registers and incoming requests on the Avalon-MM slave interface are served directly from the contents of these registers. This results in a more reactive system, as the master communicating with the MCP3204 Avalon-MM slave unit does not have to wait for a complete conversion cycle.

SPI controller

Your goal in this lab is to implement the SPI controller, so it is important to understand how it works. The *SPI controller* works in 2 conceptual phases:

1. The *Manager* instructs it which channel to convert.
2. The *SPI controller* communicates with the MCP3204 to obtain a converted value, then sends the received data back to the *Manager*.

Figure 6 explains the first phase of *SPI controller's* operating procedure in more detail:

- The *Manager* waits as long as needed for the *SPI Controller* to be ready. The *SPI Controller* is considered unavailable if the `busy` signal is asserted.
- As soon as the *SPI controller* deasserts `busy`, it means that it is ready, so the *Manager* instructs it to start a new conversion. A new conversion is started by asserting the `start` *pulse* and supplying a 2-bit `channel` to the *SPI controller*. The *SPI Controller* captures the new instruction and becomes busy again since it starts converting the appropriate channel.

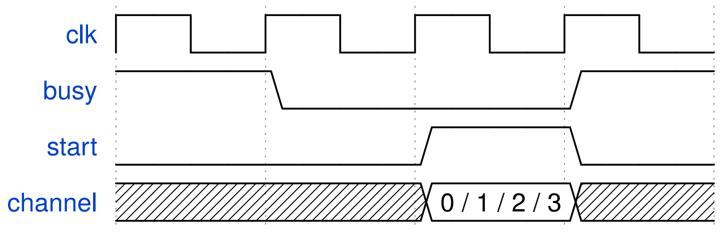


FIGURE 6. SPI CONTROLLER — PHASE 1

Figure 7 explains the second phase of the *SPI controller's* operating procedure.

- Once your *SPI Controller* has successfully converted a value, it passes the data to the *Manager* simply by asserting `data_valid` and putting the value on the `data` bus during one cycle.

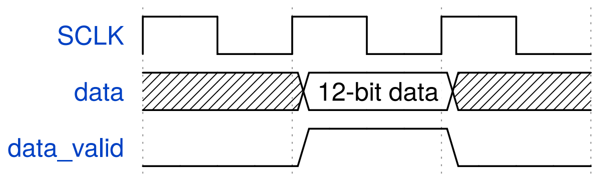


FIGURE 7. SPI CONTROLLER — PHASE 2

Important: Note that the `start` signal is a *50 MHz* pulse, whereas the `data_valid` and `data` signals are *1 MHz* pulses. You can see this due to the "clk" signal in Figure 6, whereas Figure 7 uses "SCLK".

Clock dividers

The SPI controller is clocked at *50 MHz*, but the SPI communication itself must be done at *1 MHz*, so we need to figure out some way to slow down the clock frequency *inside* the SPI controller.

Students generally make the mistake of generating a slow "clock" by using logic inside the FPGA. For example, to generate a clock that runs 3x slower than `clk_in`, they would generate the `clk_out` waveform shown in Figure 8 and connect components as shown in the schematic below.

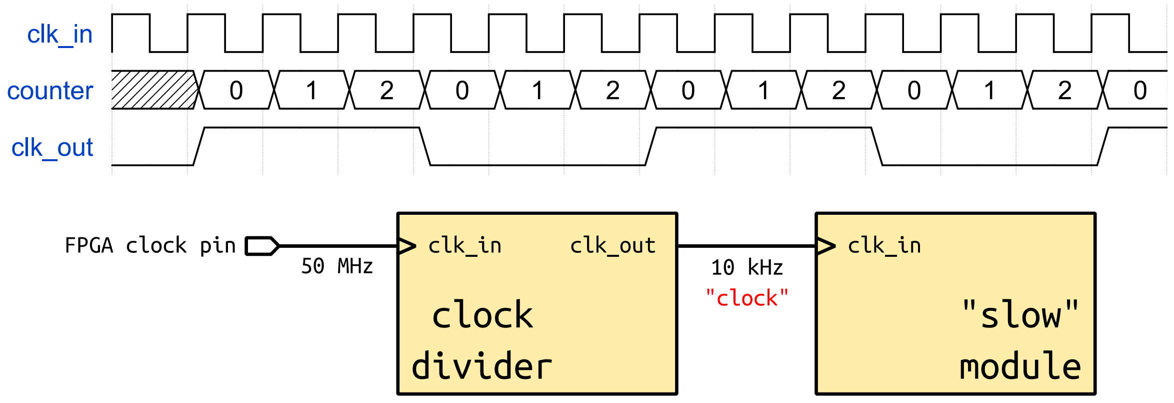


FIGURE 8. *INCORRECT* CLOCK DIVIDER (3X)

However, this is *not* the correct way to do things on FPGAs. Indeed, clock signals are routed through special channels to guarantee that the clock arrives at all components roughly at the same time (avoids clock skew). If you instead decide to "create" a clock manually by using logic within the FPGA, your clock will not be routed on the dedicated clock channels and will suffer from clock skew, therefore causing all logic driven by the custom clock to be unstable.

The correct way to generate slow clocks in FPGAs is to *not* generate a *clock*, but to instead generate a *clock divider*. A clock divider is a component that periodically generates a *pulse* which is then fed to the slower components. Figure 9 shows an example waveform and schematic of a correct divider that divides the clock frequency by 3.

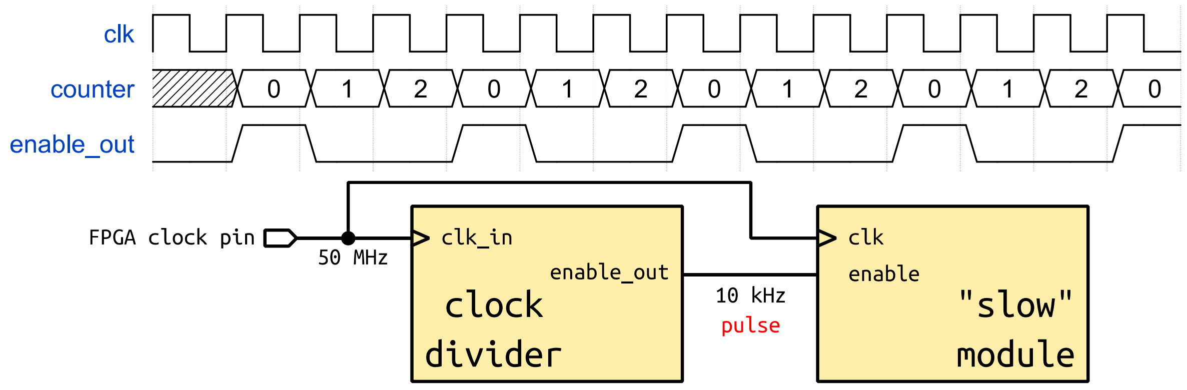


FIGURE 9. *CORRECT* CLOCK DIVIDER (3X)

Each component that requires a slow clock takes as input the standard clock of the FPGA (the one correctly routed through dedicated clock channels) along with the `enable_out` pulse generated by the clock divider. This pulse acts as an activation signal and triggers the operation of a component requiring a slow clock.

Note: We are telling you this, because you will be using a clock divider we provide you later on in the assignment, so it is important to understand how they work so you can use them correctly.

ADC output

The converted digital value you obtain as the output of the ADC depends on the position of the joystick when a conversion occurs. Figure 10 shows the values you should obtain when performing conversions.

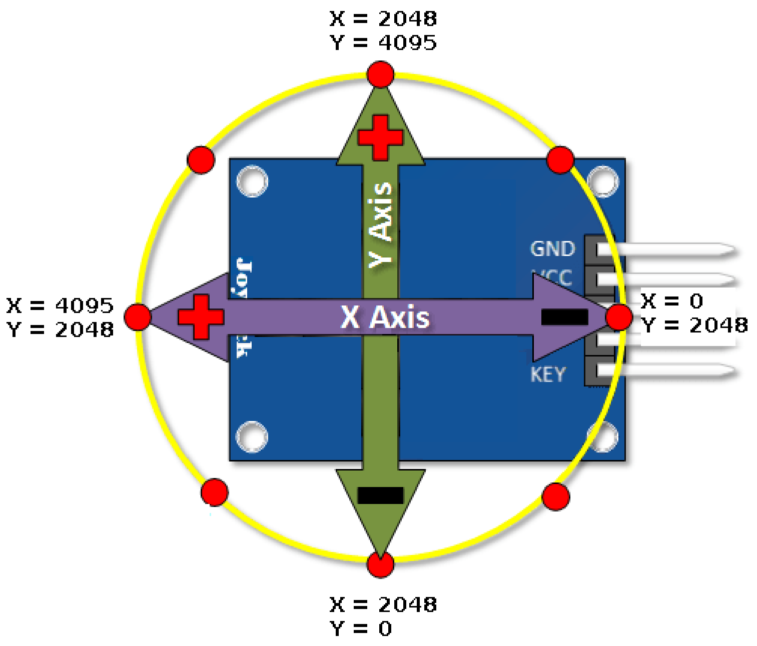


FIGURE 10. MCP3204 DIGITAL OUTPUT DEPENDING ON JOYSTICK POSITION

Practice

There is a lot to do in this lab, so let's list the things so you can advance incrementally.

1. Implement the *hardware design* of the *SPI controller* in "`hw/hdl/joysticks/hdl/mcp3204_spi.vhd`".
    a. We gave you all the details you need to know in the assignment above, but you should still read section 5 of the MCP3204 datasheet to let everything sink in and to be sure you understand the protocol.
    b. To help you out, we provide the clock divider used for the SPI communication (and all associated signals). We suggest you read and make sense of the existing design before continuing.
    c. Draw the finite state machine (FSM) of your SPI controller on **PAPER**.
    d. Implement your FSM in the process named "STATE_LOGIC" in `mcp3204_spi.vhd`.
        i. **HINT 1**: Using ModelSim can truly be beneficial when implementing your design. You can use the stimulus generation file "`tb_mcp3204_spi.vhd`" we provide to generate instructions for the controller. Note that the file is not a testbench, but rather just generates instructions for the *SPI controller*, so you'll have to compare the output of your design against Figure 4 to see if it is functioning correctly.
        ii. **HINT 2**: Pay attention to the fact that some signals need to be sent out on the *falling edge* of the *SPI clock (SCLK)*, while others on the *rising edge* of *SCLK*. This can lead to difficult problems to debug! Use the "`reg_rising_edge_sclk`" and "`reg_falling_edge_sclk`" flags provided in `mcp3204_spi.vhd` to detect the correct edge!
2. Implement the *control software* for the MCP3204 Avalon-MM slave.
    a. Fill-in the following function in "`sw/nios/application/joysticks/mcp3204/mcp3204.c`". This function is responsible for reading one of the 4 registers in the MCP3204's Avalon-MM slave unit. **HINT**: It's a 1-liner (or 2-liner to be safe ☺).

```
uint32_t mcp3204_read(mcp3204_dev *dev, uint32_t channel);
```

    b. Fill-in the following 4 functions in "`sw/nios/application/joysticks/joysticks.c`". These are helper functions which wrap around the low-level "`mcp3204.c`" implementation and provide functionality at the joystick level of abstraction.

```
uint32_t joysticks_read_left_vertical(joysticks_dev *dev);
uint32_t joysticks_read_left_horizontal(joysticks_dev *dev);
uint32_t joysticks_read_right_vertical(joysticks_dev *dev);
uint32_t joysticks_read_right_horizontal(joysticks_dev *dev);
```

We want the values obtained from the 4 functions above to satisfy the property shown on the left diagram of Figure 11 (values *increase* from left to right, bottom to top). However, the physical readings obtained from the ADC are similar to the right diagram of Figure 11 (values *decrease* from bottom to top), so you'll need to find an algorithm (in software) to invert the reading of the Y axis for it to match the expected behavior of the system.

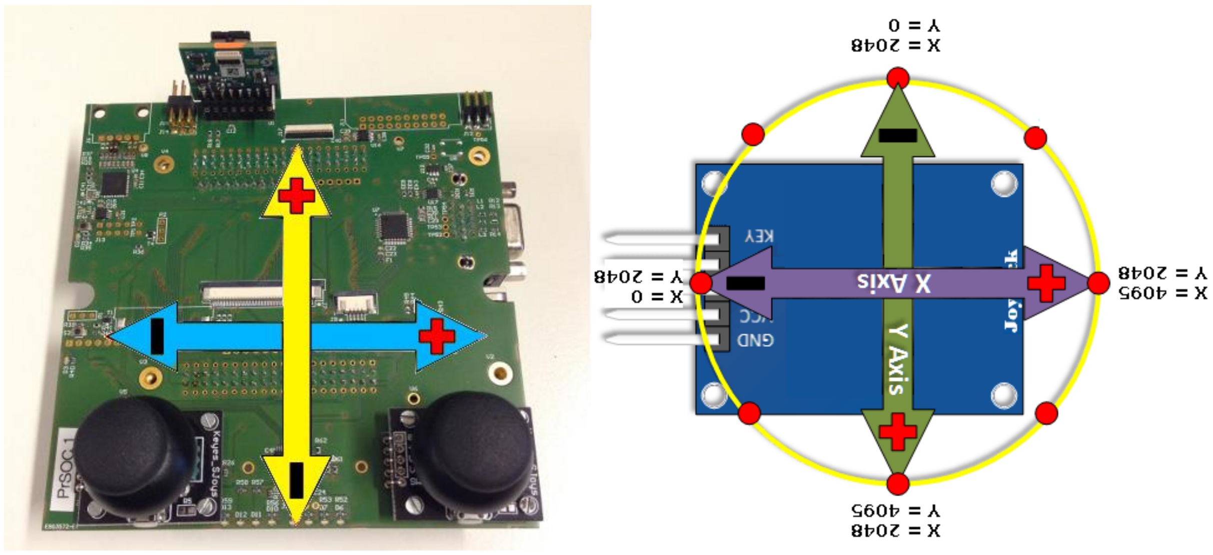


FIGURE 11. EXPECTED BEHAVIOR (LEFT), ACTUAL BEHAVIOR (RIGHT)

c. Fill-in the following function in “sw/nios/application/app.c”.

```
uint32_t interpolate(uint32_t input,
                     uint32_t input_lower_bound,
                     uint32_t input_upper_bound,
                     uint32_t output_lower_bound,
                     uint32_t output_upper_bound);
```

This function is used to map the range of values obtained from the ADC readings of the joysticks to the range of values used by the PWM generator. This is a generic function which takes as arguments the upper and lower bounds of the input and output domains.

HINT: If you are skilled in the art of *“Google-fu”*, you will find implementations of this function on stackoverflow (under other names of course ☺), but we suggest you try to write it yourself.

Conclusions

At the end of this lab, you should have a system capable of controlling the direction of a servomotor based on the live position of a joystick. Congratulations for making it this far ☺. More cool stuff coming up in the next weeks.