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.

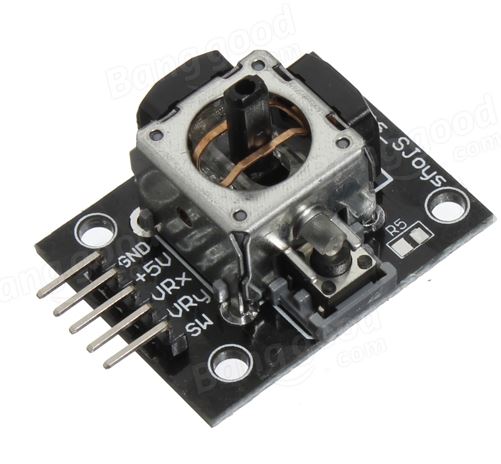
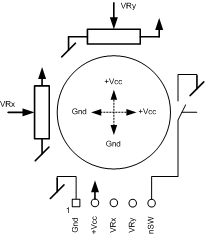
  

Figure 1. PlayStation 2 Joystick

The device consists simply of an input-controlled resistor on each axis, where the resistance ranges from approximately 30 Ω to 3.6 kΩ 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 ADC used on the extension board for this purpose is the [MCP3204](http://moodle.epfl.ch/mod/resource/view.php?id=945125). Figure 2 shows the MCP3204’s package, and Figure 3 shows how the joysticks are connected to it on the extension board.

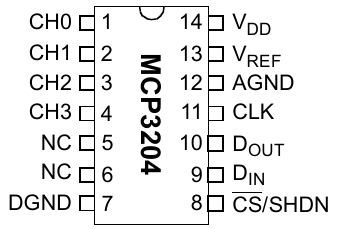


Figure 2. MCP3204

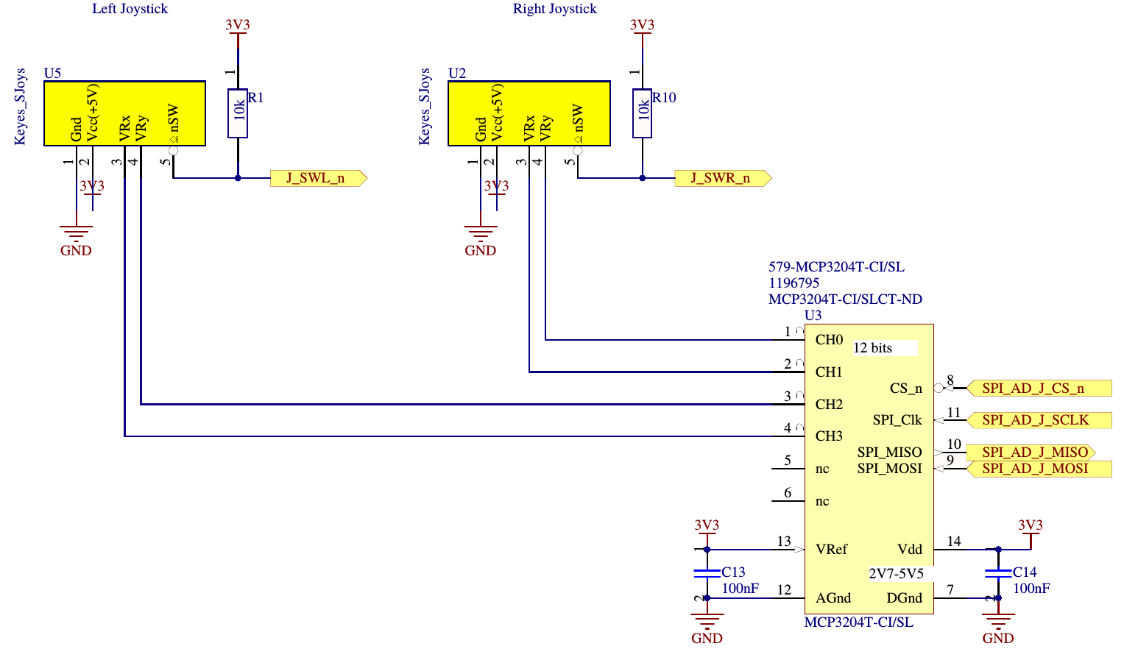


Figure 3. Joysticks + MCP3204 connections on extension board

#### 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, I2C, 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]](#footnote-1).

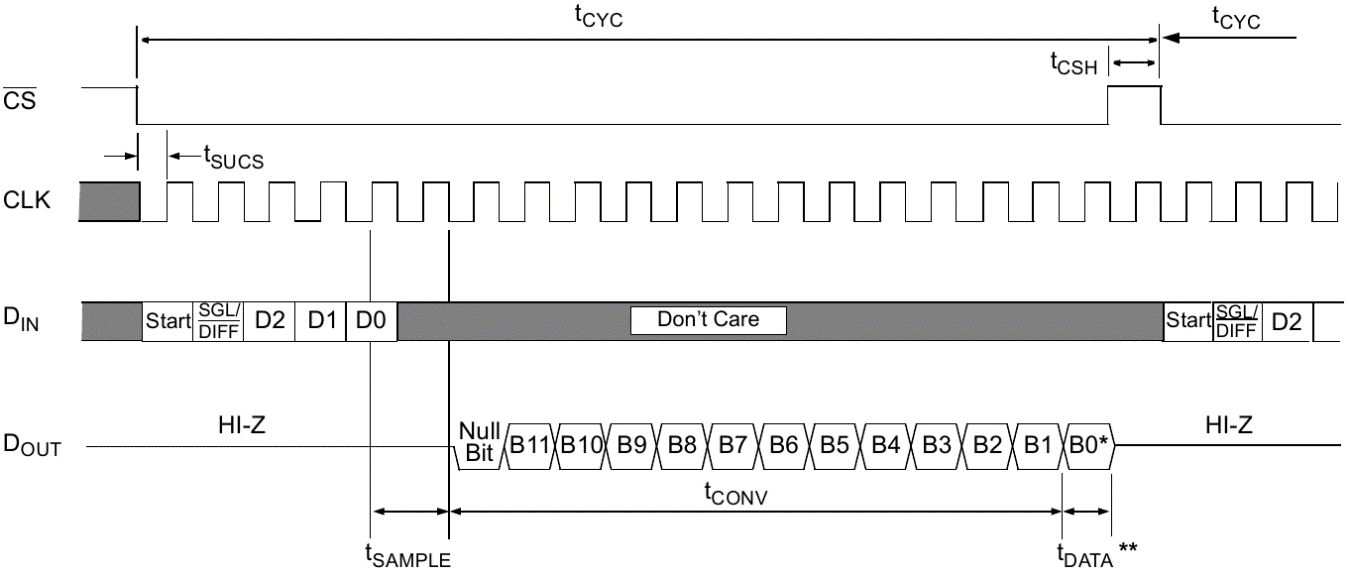


Figure 4. MCP3204 SPI Communication Protocol

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 need to be sent to the MCP3204.

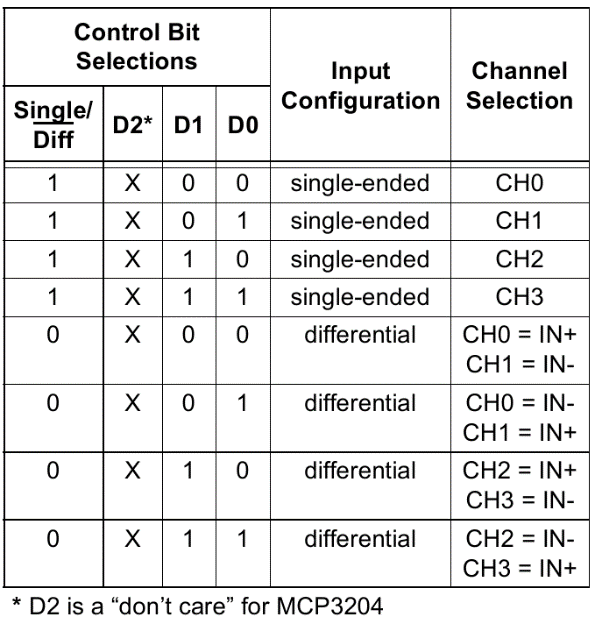


Table 1. Configuration bits for the MCP3204

### 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 a read-only Avalon-MM slave interface and provides four 32-bit registers that hold the digital values returned by the ADC. The unit is 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 forwards requests coming from the Avalon-MM slave interface to a custom SPI controller which then communicates with the MCP3204 and returns the conversion results. The unit is supplied with a 50 MHz clock.
* 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.

#### SPI controller

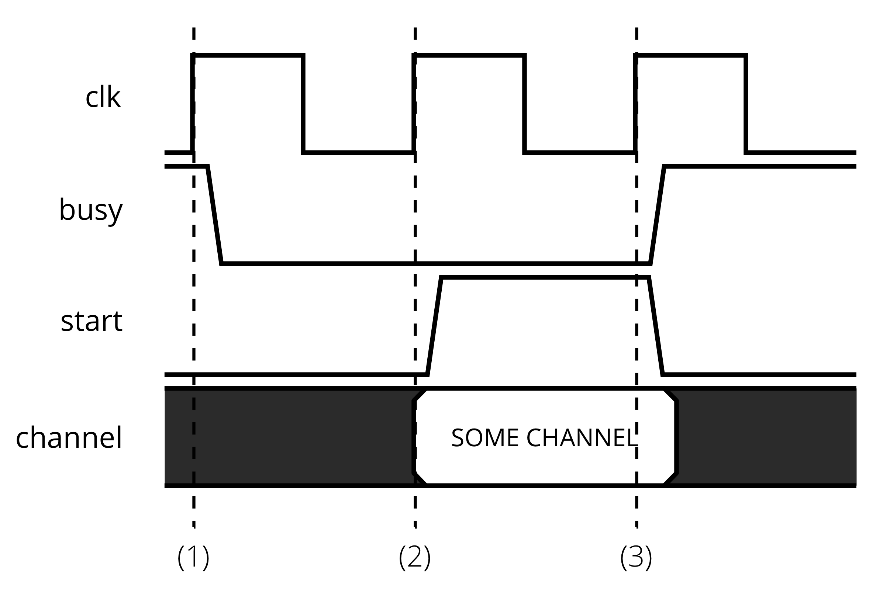
The SPI controller is clocked at 50 MHz, but the SPI communication itself must be done at 1 MHz, so we need to figure out some way to “slow down” the clock.

## Practice

The MCP3204 SPI Controller works in two phases:

1. The *Manager* instructs it which channel to convert.
2. The SPI communication is performed and the converted data is sent back to the *Manager*.

## Phase 1

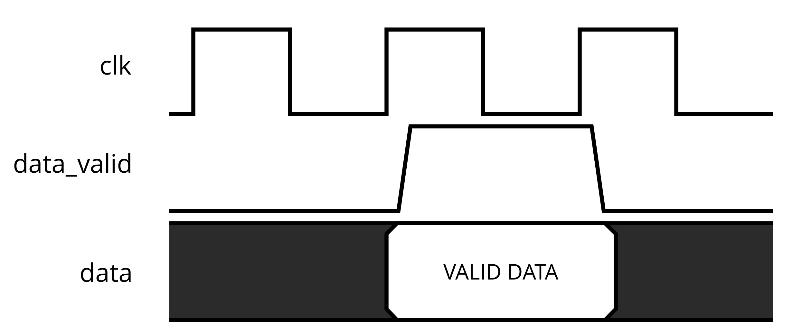


As long as the *SPI Controller* isn't ready, i.e. is busy converting, it asserts ***busy***. As soon as it finishes, it deasserts ***busy*** (see point 1 in the figure above).

The *Manager* waits for the *SPI Controller* to be ready. The *SPI Controller*'s 2-bit ***channel*** input is used by the *Manager* to instruct which channel must be requested from the ADC. Therefore, when the *Manager* detects that the *SPI Controller* is ready, it sets the appropriate ***channel*** and asserts the ***start*** strobe (see point 2 in the figure above).

Finally, the *SPI Controller* captures the new instruction and becomes busy again since it starts converting the appropriate channel (see point 3 in the figure above).

## Phase 2



Once your *SPI Controller* has successfully converted a value. It passes it to the manager simply by asserting ***data\_valid*** and putting the value on the ***data*** bus during one cycle.

## Methods

We suggest that you follow the steps below:

* We provide the clock divider used for the SPI communication (and all associated signals) in *mcp3204\_spi.*vhd. We suggest you read and make sense of the existing design before continuing.
* Draw the finite state machine (FSM) of your SPI controller on PAPER.
* Implement your FSM in the *STATE\_LOGIC* process in *mcp3204\_spi.vhd*. Using ModelSim is a non-negligible advantage. We provide the stimulus generation file *tb\_mcp3204\_spi.vhd* to generate instructions for the controller.

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 to debug problems! Use the *reg\_rising\_edge\_sclk* and *reg\_falling\_edge\_sclk* flags provided in *mcp3204\_spi.vhd* to detect the correct edge!

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