Expertise
and insight
for the future

Group 7: Taneli Voutilainen. Lauri Marjanen, Monika Radaviciute

Ventilation project

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Embedded Systems Programming

Project report

March 13, 2020

Contents

1 User manual

Introduction

This ventilation system is designed to allow the user to control the fan speed and pressure in a simple manner. The system is operated by three buttons. The screen displays the current pressure and selected pressure or selected fan speed

The system functions based on two modes: automatic and manual, which allow the user to adjust pressure or speed of fan respectively.

Startup

For the system to function, it needs to be plugged into a socket. Upon turning on, the ventilation system starts up in the automatic mode with a pressure value of 0 Pa (Fig. 1).

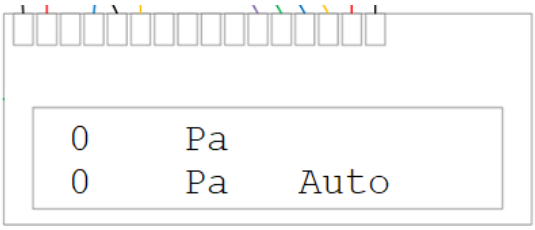

Figure 1. Startup screen

Switching between modes

To toggle the mode, press the middle button of the three.

Automatic mode

When automatic mode is entered, the user can adjust the pressure in the range of 0 to 120 Pascals. This range is differential, i.e. it represents the difference in pressure between current environmental pressure level and the level in the ventilation duct. Please keep in mind that it is not advisable to leave the ventilation duct open

significantly if a high pressure level is desired, as the system will not be able to maintain it.

The pressure value can be adjusted with the other two buttons in the increments of 5 pascals. The left button increases the value, while the right button decreases it (Fig. 2).

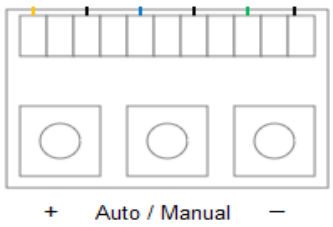

Figure 2. Buttons

The selected pressure is shown on the bottom left of the screen, whilst the current pressure level is on the top left. The inbuilt fan then works automatically to keep the specified pressure.

In some cases the selected pressure level can not be reached in due time, especially if the ventilation duct is left open too widely or there is a great difference between present pressure level and a desired one. In these instances, if the selected pressure is not reached within 5 seconds, an error message on the top right of the screen (Fig. 3).

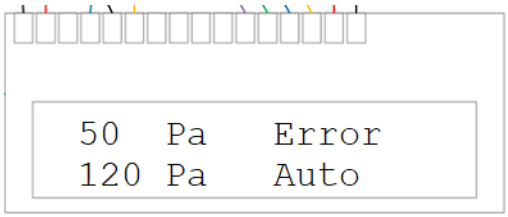

Figure 3. Error message

Manual mode

In the manual mode, the speed of the fan can be controlled by the user. The maximum possible speed of the fan is 500 Hz. This value is represented in percentage, ranging from 0 to 100 percent. Pressing the left button increases the speed by 5 percent, while

the right button lowers it in the same manner. The selected speed is reflected on the bottom left of the screen (Fig. 4).

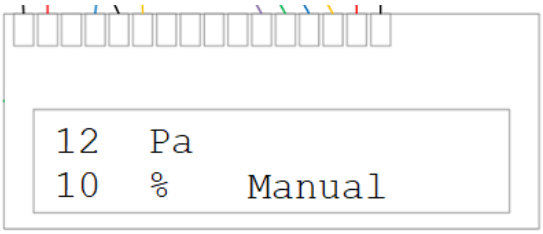

Figure 4. Manual mode

Please note the effectiveness of the fan is influenced not only by fan speed, but also by the airflow that is allowed through the duct opening.

The specified values of pressure (in automatic mode) and fan speed (in manual mode) remain until they are modified or the system is switched off.

2 Program documentation

2.1 Program overview

This program is intended to control a fan, depending on the selected mode, by speed or pressure measurements from the ventilation duct. It is accomplished by the help of a user interface that interacts with the micro-controller through three external buttons. The communication between the central devices is accomplished through I^2C and Modbus RTU protocols.

The central element of the project is the LPCXpresso1549 micro-controller. The program makes use of the board's built-in features, including system tick timer, I^2C bus interface, Repetitive Interrupt Timer (RIT), and Windowed Watchdog Timer (WWDT). This device hosts the program and manages user input and controls the rest of the system. In-depth information on the properties of LPCXpresso1549 can be found in the user manual: https://www.nxp.com/docs/en/user-guide/UM10736.pdf.

The project was built with the MCUXpresso IDE v11.0.0. For version control, GitHub was used; the project files can be found at https://github.com/lauriom/EmbeddedProject.

2.2 Program structure

The main part of the program consists of two parts: initialization and class instantiations, and an operation loop.

The initialization starts with a ring buffer instance for menu event handling and system timer configuration.

As timing precision to milliseconds is vital for the screen operation, Repetitive Interrupt Timer (RIT) is initialized before a class instance of *LiquidCrystal* is created. Moreover, the LCD display requires six pins (register select, enable, and four data pins) to be configured as output: it is done with the help of *DigitalIoPin* class.

The LPCXpresso1549 communicates with the pressure sensor on the I^2C bus, which requires two pins with a pull-up resistor. There are two suitable pins on the board (pin 22 and pin 23) that are configured as clock (SCL) and data (SDA) lines. After that I^2C can be initialized.

Next, before a *Fan* class object is instantiated, the Modbus protocol it necessitates is set up with a slave ID and a specified baud rate.

All the main components of the ventilation system — LCD display, pressure sensor, fan, along with a ring buffer — are setup before a *MainController* class instance is introduced in preparation for the second part.

The system is controlled by three buttons that are wired to D4, D6, and D7 pins. To make the system respond to presses upon release, interrupts *PIN_INT0_IRQn*, *PIN_INT1_IRQn,* and *PIN_INT2_IRQn* are initialized. The events they trigger are then passed on to be queued on the ring buffer and processed in the main loop.

A Watchdog timer is set up before the operation loop as well to reset the system in case of unexpected delays or crashes. It is configured with a one second timeout, within which it has be fed in the program loop.

The second part of the program is an endless loop that manages the pressure sensor, fan, user input, and screen updates. All of this is accomplished by a *MainController* class instance. The screen is updated every tenth loop with an *updateMenu* function. The *run* function of *MainController* handles the main process of the ventilation system: it controls the fan and reads pressure values accordingly to a chosen mode of operation and user value input.

2.3 Key elements and their operation description

2.3.1 Pressure sensor SDP 610-125Pa

The pressure sensor selected for this project is a differential one: it is designed to show the pressure difference between environmental pressure measured by one valve and inside the ventilation duct (as shown in Figure 7).

To utilize the sensor, the *PressureSensor* class is available. The class provides a few functionalities. As the development board communicates with the pressure sensor through an I^2C bus, this class implements a combined transfer function. The transfer function uses a 7-bit address of the sensor on the bus, as well as an input buffer (containing a command code) and an output buffer (intended for 16-bit pressure value and an 8-bit check sum value). Thus, the class allows the reading of pressure values, and conversion from signed 16 bit integers to pascals by dividing by a scale factor of 240 (as defined in SDP 610-125Pa documentation).

Additional technical information about SDP 610-125Pa and its communication process can be found in the documentation from the manufacturer's data sheet at http://www.farnell.com/datasheets/1720196.pdf.

2.3.2 Fan and ABB simulator

The functioning of the fan in this project relies on Modbus communication over serial lines via Remote Terminal Unit (RTU) protocol. In short, this type of inter-device communication requires a slave ID (ranging from 1 to 247), with which transmission starts; it is continued with a function code (which tells which table should be accessed, and whether it should be done so in read or write mode). With RTU protocol, the data is transmitted in binary format. A message ends with Cyclic Redundancy Check (CRC) in order to screen errors in the transmission.

In the program, Modbus protocol is managed with multiple files in the *Modbus* folder. The *ModbusMaster* class is based on Arduino library; it handles the communication basics, such as initialization, buffer management, and slave ID.

The *Fan* class utilizes these classes, and enables to set the frequency of the fan, to start the fan, and to get the speed of the fan. Modbus requires a constant interaction between devices: if the simulator is not receiving any signals, the communication with the device is terminated, and it is perceived as inactive. This is avoided in the program with consistent looped reading and setting of the fan speed.

It is important to note that for the system to work properly, the simulator has to be plugged in. The debug information from the ABB simulator is printed over UART, which is handled by *LpcUart* class; it is available through the Arduino USB port.

2.3.3 LCD display and external buttons

Both of the UI elements — the LCD and the buttons — are wired to the LPCXpresso1549 and their basic functionality depends on the *DigitalIoPin* class.

DigitalIoPin class configures a pin as either input or output, in regular or inverted mode. The input-configured pins can be additionally configures as pull-up or pull-down, and read; output-configured pins can be set with a boolean value.

The *LiquidCrystal* class (based on Arduino library), utilizes the *DigitalIoPin* class, as six output pins are used for the LCD display.

The display output and the user interaction through the external buttons is handled with the help of the *MainController* class. The *MainController* class acts as the central intersection between the input and the system response by sending commands to the LCD display and fan as well as acquiring information from the buttons and the pressure sensor. The main functioning of the program relies on *MainController* public functions, consequently the *main.cpp* file remains relatively brief..

As the digital pin of one of the buttons (located at port 1, pin 9) integrated into the LPCXpresso1549 board is used in Modbus communication, external buttons were instead chosen for this system.

3 Wiring diagrams

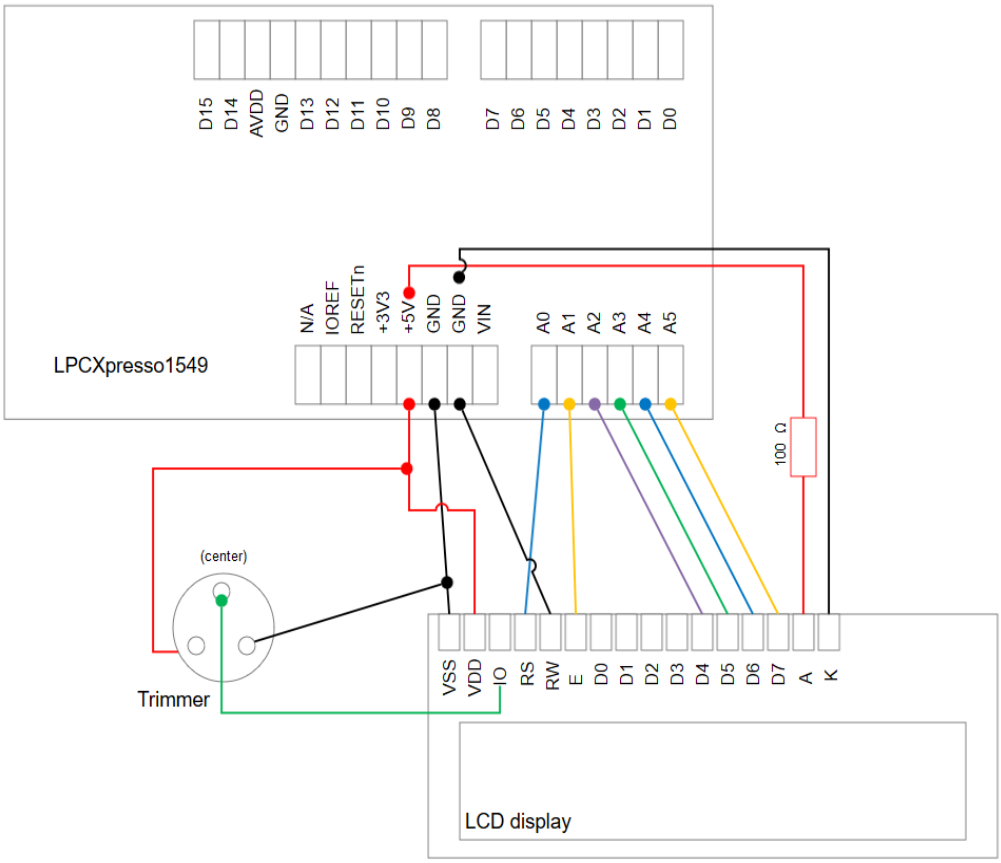

Figure 5. LCD display wiring diagram

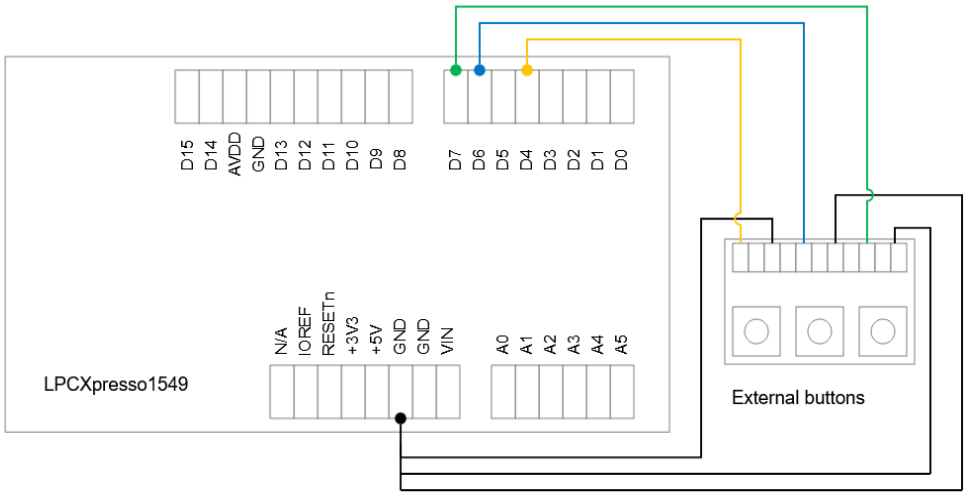

Figure 6. External buttons' wiring diagram

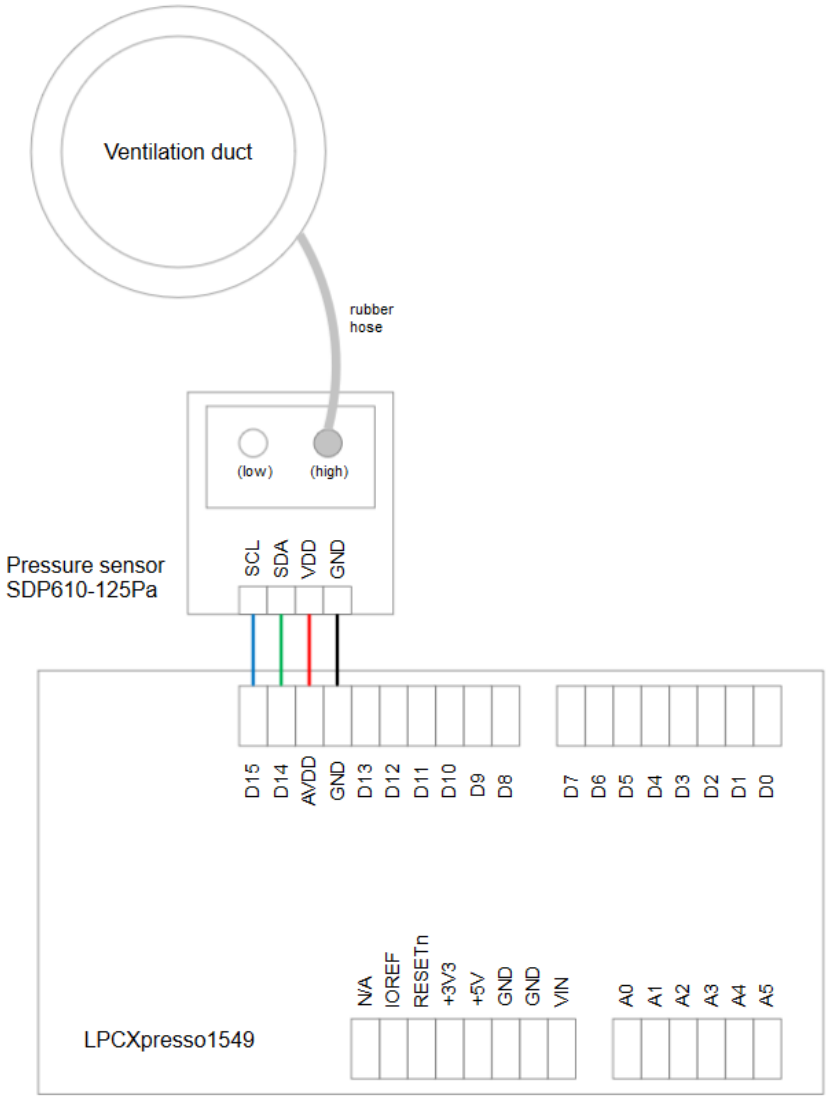

Figure 7. Pressure sensor wiring diagram

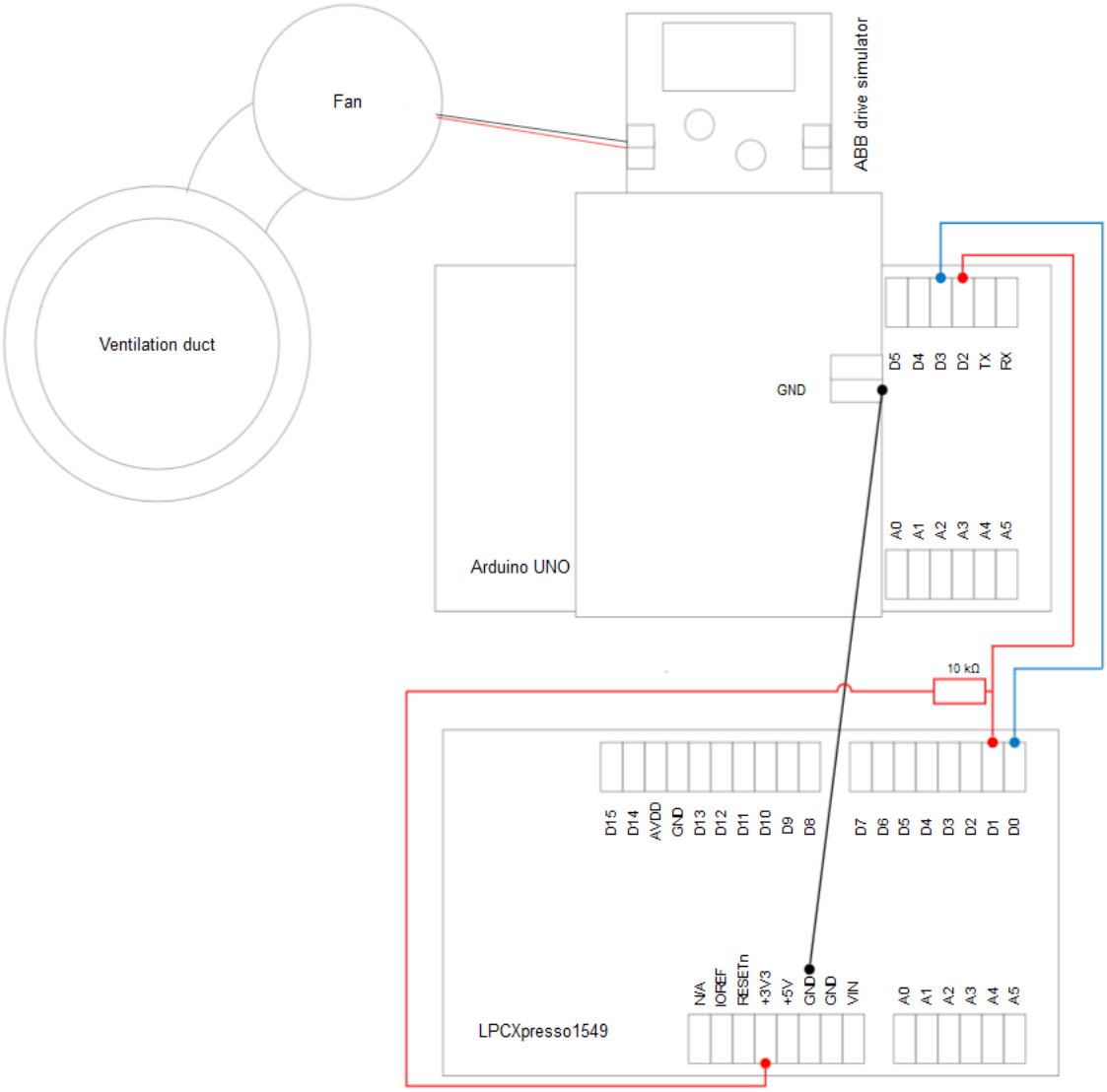

Figure 8. Arduino UNO and ABB drive simulator wiring diagram