Aleksandrs Serbajevs

Development of an embedded system for ventilation control

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

24 November 2016

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Aleksandrs Serbajevs Development of an embedded system for ventilation control 35 pages 24 November 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Embedded Systems Engineering
Instructor(s)	Keijo Länsikunnas, Senior Lecturer

The goal of the project was to implement a system that would control a ventilation system in order to equalize the pressure between the inside and outside of the building.

The system consists of one or several units that measure the difference in pressure between the inside and outside of the building, and a unit that uses the data obtained from those sensors to adjust the output of the ventilation system.

Each unit is based on a LPCXpresso microcontroller and uses an XBee module for wireless communication. Sensor units use a differential pressure sensor to detect the pressure difference, and send the data to the ventilation control unit. The ventilation control unit reads control signals of a ventilation system, makes adjustments to them based on the pressure difference, and sends the adjusted pressure signals to the intake and exhaust fans of the ventilation system.

The system was successfully implemented and the testing confirmed its effectiveness in achieving its purpose.

Keywords	embedded systems, lpcxpresso, xbee, ventilation

Contents

List of abbreviations

ADC	analog-to-digital converter
API	application programming interface
DAC	digital-to-analog converter
I²C	Inter-Integrated Circuit
IC	integrated circuit
IDE	integrated development environment
ITM	Instrumentation Trace Macrocell
Op-amp	operational amplifier
PID control	proportional-integral-derivative control
RC filter	resistor-capacitor filter
PWM	pulse width modulation
SAR	successive approximation register
UART	Universal asynchronous receiver/transmitter

1 Introduction

An embedded system is a computer system that, unlike personal computers, is dedicated to performing specific functionality within a larger mechanical or electronic system. Embedded systems are ubiquitous in modern everyday life, found in most household appliances, personal computers, as well as in infrastructure and industrial applications. Embedded systems possess less capabilities and processing power than general-purpose computers and are harder to program, but are widely used because of their low cost, power consumption and small size. Since embedded systems are dedicated to a single task, they tend to be highly efficient, and due to them utilizing generic components, easily mass-produced.

Development of an embedded system includes design of its overall architecture, choice of platform and hardware components, circuit design and development of the microcontroller or microprocessor software.

The goal of this project is to develop an embedded system capable of controlling fans of a ventilation system in order to equalize the pressure between the inside and outside of the building. The pressure difference causes air to move inside or outside through places other than the ventilation system, which can cause undesirable effects, such as moisture from outside causing degradation of building materials.

2 Embedded systems

2.1 Structure of embedded systems

The common feature of all embedded systems is interaction with the real world. Embedded systems gather data about their environment using sensors and manipulate the environment according to the collected data using actors. [1, 5]

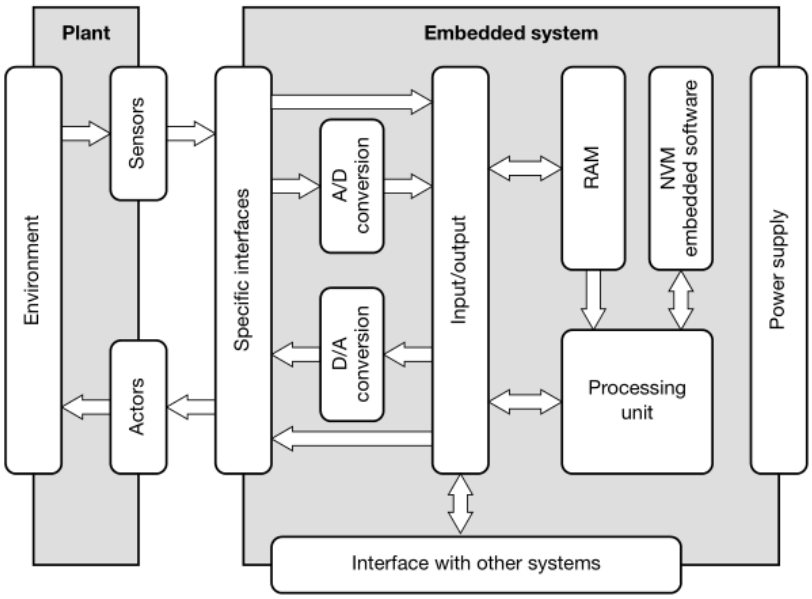

Figure 1. Generic layout of an embedded system. Copied from [1].

Figure 1 illustrates a generic layout of an embedded system that shows all necessary components of an embedded system, and accurately describes almost all of embedded systems. The environment of the system, including actors and sensors, is referred to as the plant. The processing unit, using embedded software stored in its non-volatile memory, reads data from sensors into its volatile memory, performs the necessary calculations with that data and outputs the resulting data to actors that manipulate the environment. [1, 5-6]

Depending on the scale and performance requirements of the system, different components can be located on different integrated circuits (ICs) or integrated into a single one. Typically, simple systems are highly integrated in order to decrease production costs, physical dimensions and power consumption, while high-performance systems consist of a greater number of highly specialized components. [2, 129-130]

The processing unit can be a microcontroller, microprocessor or a digital signal processor. A microprocessor is a general-purpose electronic device that incorporates all functions of a central processing unit on a single IC. A microcontroller, as opposed to a microprocessor, also integrates volatile and non-volatile memory as well as input and output peripherals onto the same IC as the processor core. A digital signal processor is a microprocessor with an architecture optimized for measurement, filtering and processing of continuous analog signals. [2, 129-130]

Since the processing unit can only process and output digital data, analog-to-digital conversions are necessary on sensor inputs and digital-to-analog conversions are necessary on outputs. In some instances, sensors packages include ADCs and output digital data using some data interface, while in others sensors convert physical quantities such as luminance, pressure or temperature into voltage that has to be measured by an ADC. [2, 88].

2.2 Embedded system development

Embedded systems design can be divided into four major stages: design of its architecture, implementation of its architecture, testing and maintenance. [2, 8]

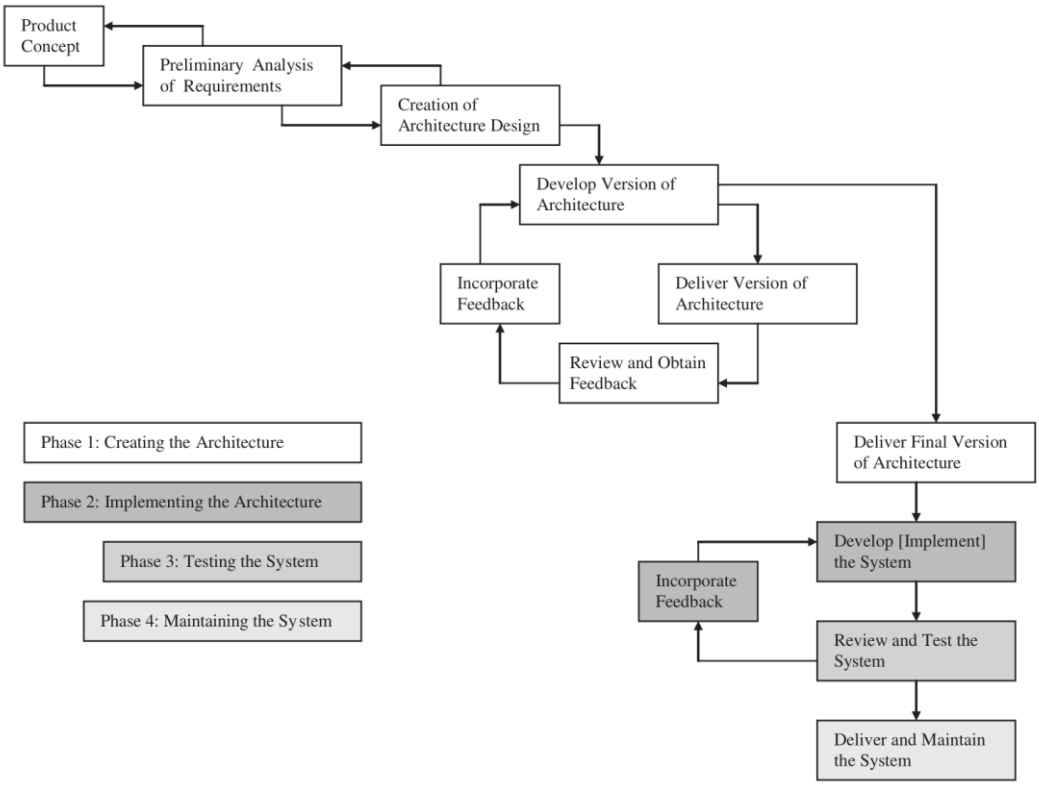

Figure 2. Embedded systems design and development model. Copied from [2]

Figure 2 illustrates the complete design, development and lifecycle model of embedded systems, and how the four stages interact with each other. This model includes elements of waterfall and spiral development processes. The creation of architecture is described in detail due to the fact that in complex embedded systems the design of the architecture is the major factor in the success or failure of the project, as well as the time it takes to complete it [2, 7-8].

The architecture creation stage includes defining requirements for the system, listing all of the internal elements of the system and the external ones that interact with it, picking the platform and components that satisfy the stated requirements for both the final product and development process, choice of the programming language and planning of the general structure of the embedded software that the system will require to function. [2, 7-8]

Once the architecture is defined, implementation of the system consists of assembly of its hardware components and implementation of its software. Once a functioning prototype is developed, testing may reveal some flaws of the implementation that may require changes to the hardware layout of the system or its software, which, after implementation, would need to be tested again. [2, 7-8]

3 Project overview

3.1 System architecture

The system consists of two types of units: one or several sensor units that measure pressure difference and a unit that controls the fans of the ventilation system using data received from sensor units.

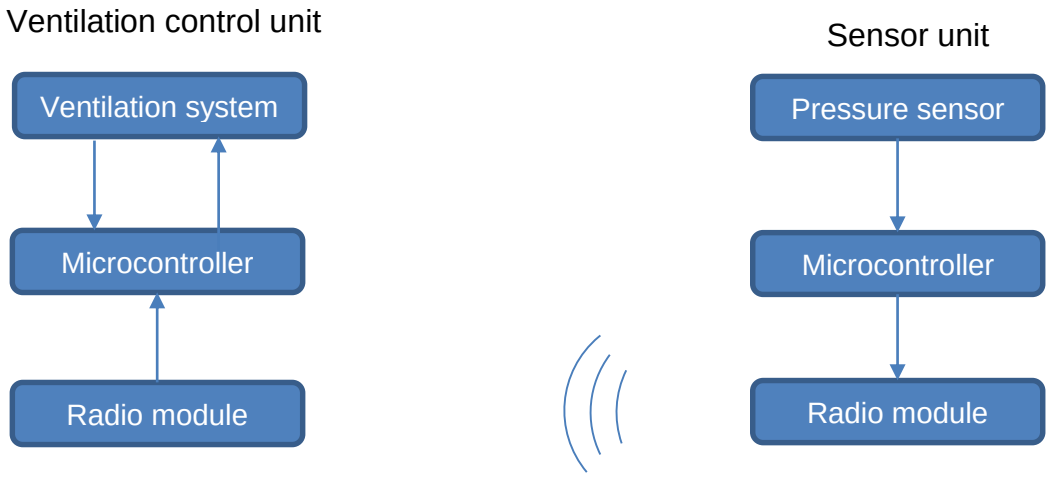

Figure 3. Block diagram of the system

Figure 3 illustrates the basic architecture of the system, with arrows indicating direction of communication between different modules. The plant of this system consists of a pressure sensor, fan control signals of the ventilation system, and ventilation fans that are controlled by the output of the system.

3.2 Sensor unit

Sensor units are responsible for measuring the pressure difference between the inside and outside of the building, and transmitting measured values to the ventilation control unit. Since distances between the ventilation system and different measurement points can be from several meters to several dozens of meters, depending on the layout of the building, a wired connection is not a viable option, as it is difficult to install in a way that would not create inconveniences for people who would use the building. Wireless radio communication is therefore necessary between sensor units and the ventilation control unit.

3.3 Ventilation control unit

The ventilation control unit is responsible for adjusting airflow in the ventilation system in order to equalize the pressure between the inside and outside of the building. It is connected to the ventilation system directly. Using a radio module, it receives data from one or several sensor units, reads the fan control signals of the ventilation system, makes individual adjustments to the power of intake and exhaust fans, and outputs adjusted control signals to the fans.

3.4 Feedback control

Feedback control is necessary for this project in order to determine the right adjustments to the operation of the ventilation system in order to achieve the desired effect quickly and once its achieved, maintain stability. Without proper feedback control the pressure difference may not equalize at all, or regularly oscillate between positive and negative pressures.

Proportional-integral-derivative (PID) control is the most common form of feedback control. It is ubiquitous in all kinds of applications, from simplest devices to most complex industrial systems. Most PID controllers do not a utilize derivative component, including one used in this project, and those could be called PI controllers. [3, 293]

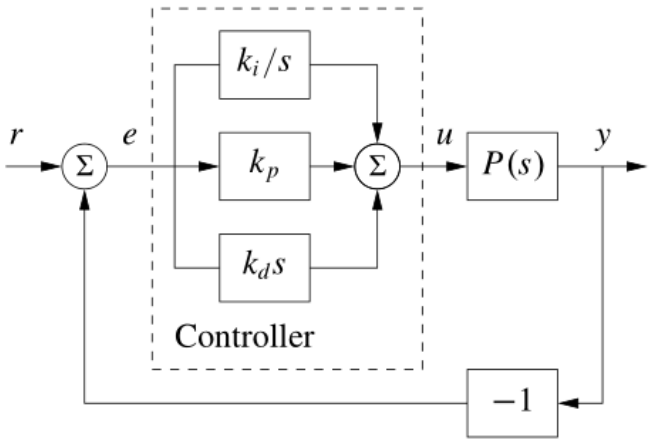

Figure 4. Closed-loop feedback system with a PID controller. Copied from [3]

Figure 4 illustrates a basic closed loop system with a PID controller. The controller has one input, which is error feedback. The error represents how much output of the system

deviates from the desired output. In this project, the error is the pressure difference between the inside and outside of the building. The controller has three main parameters: proportional gain k_p, integral gain k_i, and derivative gain k_d, and the output of the controller is the sum of the proportional, integral and derivative terms that are calculated in different ways based on error value, and multiplied by their respective gains. The controller is tuned by adjustment of the gain values. [3, 293-294]

The proportional term, as its name suggests, is directly proportional to the error. The proportional term is responsible for making the system react to the present value of the error.

The integral term is based on the sum of previous error values, typically constrained to some maximum value to guard against positive feedback causing the system to lose stability [3, 306-307]. The integral term accounts for the past values of the error. If the current output is not sufficient to reduce error value, the integral term will accumulate over time in order to compensate the error, and stabilize at a certain value once the error becomes zero.

The derivative term is based on the rate of chance of the error. It is used to account for the possible future value of the error in order to increase stability of the system and decrease the amount of time it oscillates before settling. The derivative term is rarely used in practice due to its inconsistent impact on system stability in real-world applications. For instance, noise, which is to some degree present in all measurement systems, can cause the derivative term to introduce undesired changes to the control signal [3,308].

Before a PID controller can be used to control a system, its parameters need to be set to appropriate values. There are various techniques used to tune PID controllers. The most common way to tune a PID controller is formally referred to as Ziegler–Nichols method.

In this project, the PID controller would be implemented with software on the microcontroller of the ventilation control unit. The pressure difference is taken as the error value, and the output of the controller would control the difference between the intake and exhaust fan power. The proportional term of the controller would compensate for minor fluctuations in the pressure difference, as well as partly compensate for the constant

difference. The integral term would be responsible for compensating the constant pressure difference that the proportional term cannot compensate.

As the integral term accumulates over time based on the pressure difference, it would be able to adjust itself to compensate for any pressure difference that the ventilation system is capable of counteracting. In the instance that the system is incapable of compensating the pressure difference, the integral term would eventually cause the system to create the maximum pressure difference it is capable of producing by running an intake fan at maximum power and an exhaust fan at minimum power or vice versa, resulting in the system compensating as much of the pressure difference as it is capable of.

4 Hardware overview

4.1 LPCXpresso platform

LPCXpresso is a low-cost development platform that uses ARM-based microcontrollers. It provides developers with an end-to-end solution for development from initial evaluation to production. The platform uses an Eclipse-based IDE that includes everything necessary for development and testing. [4]

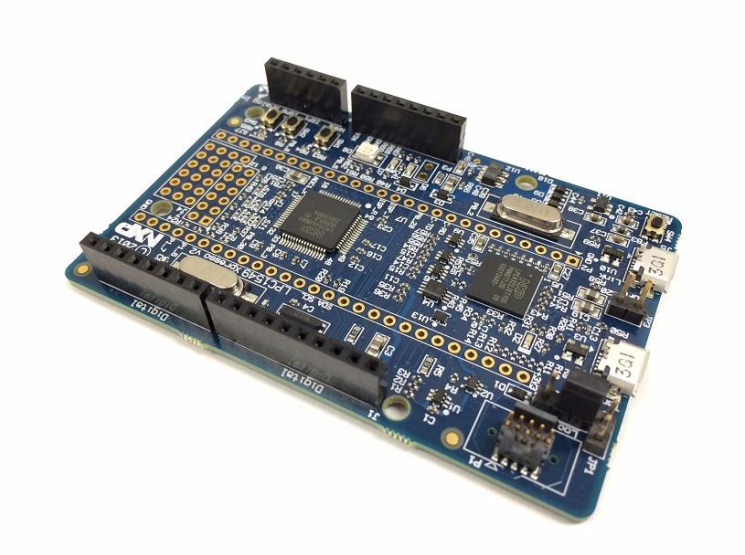

Figure 5. OM13056 board. Copied from [4]

OM13056 board for LPC1549 (shown in Figure 5) uses Arduino UNO form factor, which allows it to be compatible with a wide range of expansion boards, including XBee shields that are used to interface with XBee modules in this project.

LPC1549 is a 32-bit microcontroller based on ARM Cortex-M3. It operates at clock frequencies of up to 72 MHz, offers high power efficiency and its instruction set as well as 3-stage pipeline offer fast data processing capabilities. On-chip drivers of LPC1549 include UART and I²C, which are used for communication in this project.

A major advantage of the LPCXpresso platform for this project is that it has the Instrumentation Trace Macrocell (ITM), which allows it to be easily debugged without requiring any additional hardware or impacting microcontroller's performance during debugging. A developer can use standard C language printf and scanf functions to transmit data between a microcontroller that is executing code and a PC. Another advantage of ITM is that functions used for debugging could be left in the code after development is complete,

as the microcontroller will just ignore calls to those functions if no debugging tools are attached. [7]

4.2 XBee

XBee is a family of radio modules from Digi International. They provide a low-cost and simple to implement solution to wireless communication within the range of 60 m [2]. They require a small number of connections to function, with a minimum being power, ground, data in and data out.

Figure 6. XBee module. Copied from [5]

XBee modules (one model shown in Figure 6) can operate in transparent mode, where all data they receive on the input is transmitted immediately, and all data they receive is available at the output pin through the UART serial interface, or in the application programming interface (API) mode, where commands are used to facilitate communication with data packets and provide more functionality. The transparent mode allows devices to remotely communicate while requiring no change to the device's programming compared to two devices being connected with UART directly, while also providing features of XBee platform such as addressing, framing and data verification, as well as optional configurable features like encryption.

4.3 SDP610 differential pressure sensor

SDP600 is a family of differential pressure sensors from Sensirion. They offer high accuracy and no drift throughout their entire range, as well as internal compensation for temperature difference. The sensor chip includes an ADC and outputs measurement

results in digital format. I²C serial data interface is used for communication by the sensor, including configuration and transmission of measurement results.

Figure 7. SDP610 differential pressure sensor. Copied from [6]

Housing of the sensor, which can be seen in Figure 7, features two ports for tube connection. SDP610-125Pa model used in this project provides accuracy of 0.1 Pa + 3% of reading over the range of -125 Pa to +125 Pa pressure, which is more than sufficient for this project, as typical pressures encountered during this project are within the 20 Pa range [6].

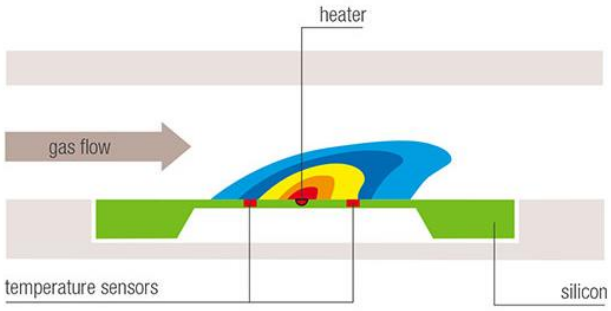

Figure 8. Thermal measurement principle. Copied from [9]

The sensor operates on thermal measurement principle, shown in Figure 8. A heating element is positioned between two temperature sensors. The pressure difference between two measurement ports creates gas flow over the silicon membrane. The gas flow causes different magnitudes of heat transfer between areas upstream and downstream of the heating element, which creates a precisely measurable temperature difference that can be used to determine the pressure difference between two measurement ports. [9]

4.4 AD5593R

AD5593R is a 12-bit, configurable, 8-channel ADC/DAC with on-chip reference. Each of its 8 channels can be independently configured as either analog-to-digital converter

(ADC) or digital-to-analog converter (DAC). AD5593R uses I²C serial data interface for configuration and transmission of input and output voltage values. [8]

For this project, two ADC and two DAC channels are necessary to read and adjust analog fan control signals of the ventilation system. This project also benefits from an internal reference and buffers on inputs and outputs that AD5593R has, which removes the need to include those components separately. Operational amplifier (op-amp) buffers provide high input impedance and low output impedance in order to transfer voltage between different circuits without input and output resistances of those circuits affecting it.

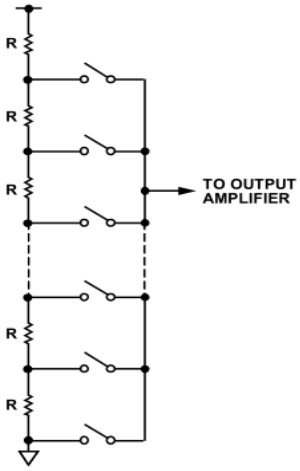

Figure 9. Resistor string DAC. Copied from [8]

To output specific voltage, DACs of the AD5593R utilize strings of equal value resistors connected in series (shown in Figure 9). A switch is connected to each resistor, that when closed, allows the voltage to bypass that resistor, and when open, forces the voltage to pass through the resistor, which reduces the voltage of the final output. The requested voltage output is achieved by closing a certain number of switches.

To measure voltage, AD5593R uses successive approximation register (SAR) ADCs. SAR ADCs operate by successively comparing input voltage to the voltage generated by its internal DAC, and writing the results of each comparison as bits in the output of the DAC. The first comparison is between the input and half of the reference voltage. If the input voltage is higher, bit 1 is written to the output, zero otherwise. The second comparison is between 3/4 of the reference voltage if first bit was 1, and 1/4 if was 0, and the result of the comparison is written as the second bit of the output. This repeats until all bits are written. For a 12-bit ADC, 12 successive comparisons are made for each measurement.

4.5 Ventilation system

The ventilation system that was used in this project is located in a laboratory (ETYA0117) in the Vanha maantie campus of Metropolia University of Applied Sciences.

Figure 10. Ventilation system

The system (shown in Figure 10) was installed by Datasteel Oy. The ventilation control module is connected to the control signals of the intake and exhaust fans. The system can be remotely controlled though a web server with an interface that gives access to its sensors and direct control over the fan speed.

4.6 Hardware layout

The hardware components previously described in this section use specific interfaces to connect and communicate.

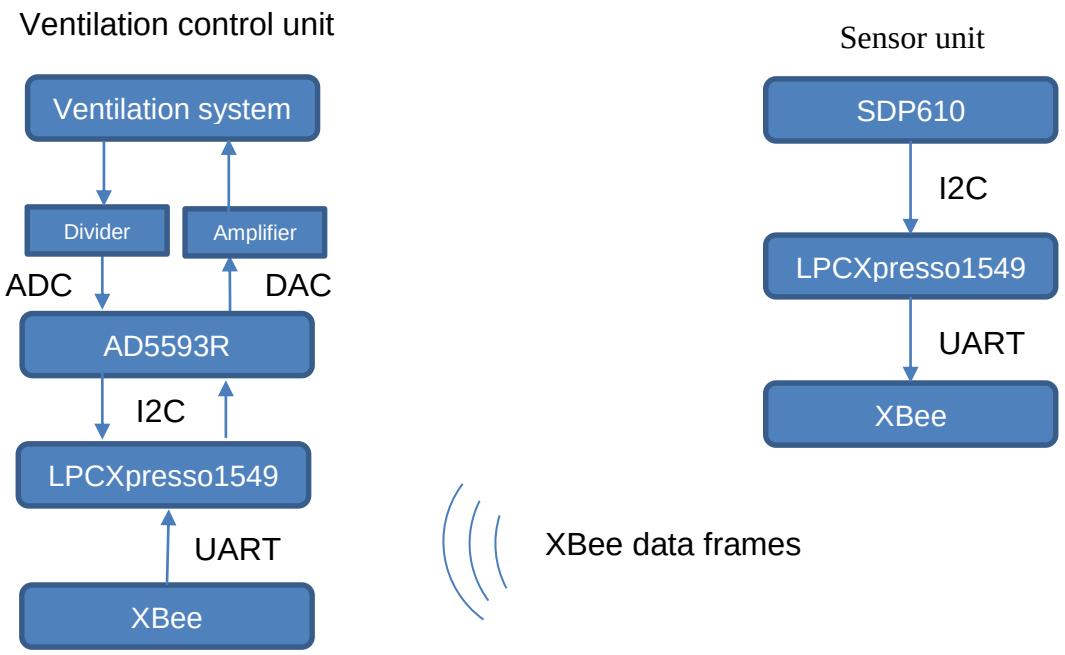

Figure 11. Hardware layout of the system

Figure 11 illustrates which hardware components connect to each other and how they communicate with one another. LPCXpresso microcontrollers use I²C serial interface to communicate with SDP610 sensor and AD5593R, and UART to send and receive data using XBee modules. The inputs and outputs of AD5593R connect to the ventilation system through divider and amplifier circuits which are necessary to match the voltage ranges that they use.

5 Communication

5.1 I²C

I²C (Inter-Integrated Circuit) is a bidirectional serial communication bus developed by NXP Semiconductors. It is used for comparatively low-speed communication between hardware components in close proximity to each other, typically on the same circuit board. Over 1000 different ICs use I²C for communication. [10, 3]

I²C bus uses two wires: a serial data line (SDA) and a serial clock line (SCL). I²C supports multiple masters and slaves on the same bus. The number of devices on one bus is not explicitly limited, though limiting factors are the maximum allowed bus capacitance and address space (with most commonly used 7-bit addresses, 128 devices). Each transmission must be at least one byte long, and there is no upper limit on the number of bytes per transmission.

I²C bus supports arbitration in case multiple masters attempt to initiate transmission at the same time. The master that initiated transmission second will detect that the state of the SDA line does not match the expected value and wait for STOP condition before retrying [10, 11-12].

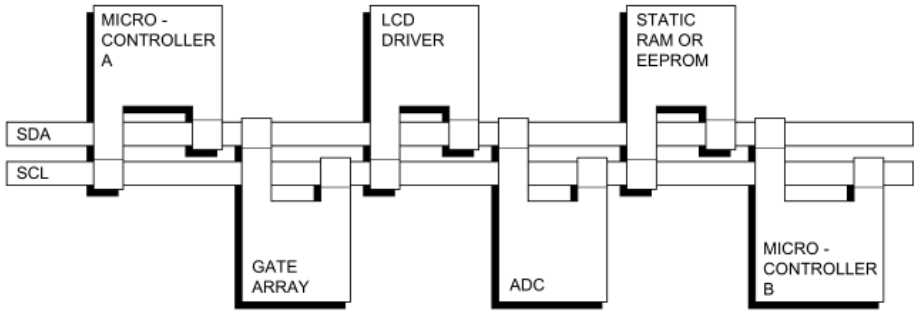

Figure 12. I²C bus with multiple masters and slaves. Copied from [10]

Figure 12 illustrates how multiple devices can be connected to an I²C bus. Every transmission is started by one master and addresses one unique slave address. Therefore, the presence of any number of other slave devices on the bus does not in any way affect the transmission procedure for the master initiating it. Additionally, each microcontroller can switch between being a master or a slave and use that to facilitate communication between microcontrollers. Combined with the physical layout of the bus connection, this

allows devices to be easily added to or removed from the bus without requiring other changes to hardware layout or software implementation of the system.

5.2 UART

Universal asynchronous receiver/transmitter (UART) is a hardware device used for serial communication. Transmission speed, as well as data format are configurable to some degree. Due to UART being asynchronous, there is no common clock signal. Therefore, devices that need to communicate with each other must be configured to use the same baud rate. Data flow is controlled with start and stop bits that precede and follow each byte. The data is transmitted sequentially bit by bit and is reassembled into bytes by the receiving device.

A UART requires one wire in order to transmit data and one wire in order to receive data, as well as common ground between two communicating devices. Tx pin of one device is connected to Rx pin of another. Transmission and reception can be performed simultaneously without affecting each other.

Figure 13. UART data packet structure

Figure 13 illustrates the structure of a UART data packet. The exact structure of the packet is defined by the configuration of the communicating devices and must be identical for devices that intend to communicate with each other.

5.3 XBee

While XBee modules use UART to communicate with their hosts, when transmitting data between each other, they encapsulate the transmitted data into packets in order to facilitate reliable communication. In the transparent mode XBee modules generate those packets automatically, while in API mode the packets have to be constructed and decoded by the host. Manually constructing packets, as opposed to using the transparent mode, allows the host device to individually address different packets to different destination modules.

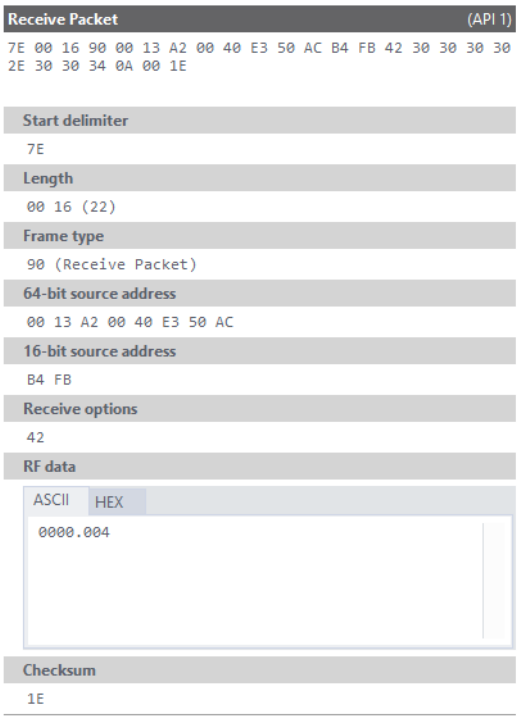

Figure 14. XBee packet interpretation

Figure 14 illustrates an example of a packet split into its components. Each packet starts with a start delimiter, followed by its total length (including packet metadata itself). The data itself is after the metadata and before the checksum. If a packet is malformed or either its length or checksum does not check out, it will be discarded.

6 Project implementation

The implementation of an embedded system can be split into multiple stages, some of which are completely independent of each other, while others require one or several previously implemented modules to function. This section is going to describe the implementation of each part of the project in detail, in the order in which they were implemented.

6.1 I²C communication

I²C implementation is required for both the sensor and ventilation control unit in order to communicate with SDP610 pressure sensor and AD5593R. LPCXpresso1549 does have an on-chip I²C driver. However, it only provides low-level functions and requires setup and configuration, as well as implementing higher-level functions to utilize it.

```
void Init_I2C_PinMux()
{   //Initializes pin muxing for I2C interface
    Chip_IOCON_PinMuxSet(LPC_IOCON, 0, 22, IOCON_DIGMODE_EN | I2C_MODE);
    Chip_IOCON_PinMuxSet(LPC_IOCON, 0, 23, IOCON_DIGMODE_EN | I2C_MODE);
    Chip_SWM_EnableFixedPin(SWM_FIXED_I2C0_SCL);
    Chip_SWM_EnableFixedPin(SWM_FIXED_I2C0_SDA);
}

void Setup_I2C_Master()
{   //Initializes and configures I2C peripheral
    Init_I2C_PinMux();                                      //Setup I2C pin muxing
    Chip_I2C_Init(LPC_I2C0);                                //Enable I2C clock and reset I2C peripheral
    Chip_I2C_SetClockDiv(LPC_I2C0, I2C_CLK_DIVIDER);        //Configure I2C clock rate
    Chip_I2CM_SetBusSpeed(LPC_I2C0, I2C_BITRATE);           //Configure I2C transfer rate
    Chip_I2CM_Enable(LPC_I2C0);                             //Enable master mode
    NVIC_EnableIRQ(I2C0_IRQn);                              //Enable I2C interrupts
}

void I2C_Transfer(uint8_t address, uint8_t *tx_data, uint16_t tx_size, uint8_t *rx_data, uint16_t rx_size)
{   //Set up and execute I2C transfer
    I2C_TransferRecord.slaveAddr = address;             //Set up I2C transfer record
    I2C_TransferRecord.status = 0;
    I2C_TransferRecord.txSz = tx_size;
    I2C_TransferRecord.rxSz = rx_size;
    I2C_TransferRecord.txBuff = tx_data;
    I2C_TransferRecord.rxBuff = rx_data;

    Chip_I2CM_Xfer(LPC_I2C0, &I2C_TransferRecord);          //Start transfer
    Chip_I2C_EnableInt(LPC_I2C0, I2C_INTENSET_MSTPENDING | I2C_INTENSET_MSTRARBLOSS | I2C_INTENSET_MSTSTSTPERR);
    I2C_WaitForCompletion(&I2C_TransferRecord);             //Wait until completion
    Chip_I2C_ClearInt(LPC_I2C0, I2C_INTENSET_MSTPENDING | I2C_INTENSET_MSTRARBLOSS | I2C_INTENSET_MSTSTSTPERR);
}
```

Listing 1. I²C initialization and data transfer functions

Listing 1 illustrates the implementation of the functions required to enable, configure and utilize I²C driver on the LPCXpresso1549. The I²C_Transfer function allows sending and receiving sequences of bytes from I²C devices. However, to perform specific tasks, like

doing pressure readings, it is necessary to follow a procedure that is specific to the device and the task. The required sequences of actions to perform each task are described in the datasheets of the respective components.

```
uint16_t SDP610_Read_Pressure ()
{
    uint8_t send, receive[2];
    uint16_t data;

    Chip_I2CM_SendStart (LPC_I2C0);                          //Send start condition
    send = SDP_610_TRIGGER_MEASUREMENT;
    I2C_Transfer(SDP_610_ADDR, &send, 1, &receive, 0);       //Send header byte and trigger measurement command
    __WFI();                                                 //wait for sensor to complete conversion
    Chip_I2CM_SendStart (LPC_I2C0);                          //Send start condition
    I2C_Transfer(SDP_610_ADDR, send, 0, &receive, 2);        //Send header byte and receive measurement data
    Chip_I2CM_SendStop (LPC_I2C0);                           //Send stop condition
    data = receive[0]*256 + receive[1];                      //Convert received two bytes to an integer value
    return data;
}
```

Listing 2. Function to perform pressure measurement

Listing 2 illustrates a function that is used to send a command to the pressure sensor to perform measurement and receive the measured value. The command to trigger measurement is sent, then the microcontroller waits for the conversion to complete, after which the sensor sends back two bytes of data that contain the measurement.

6.2 XBee configuration

Even though XBee's transparent operation allows wireless data transmission to be achieved just by implementing UART support in the system that it is connected to, in order for an XBee module to be able to transmit and receive data, XBee modules that need to communicate with each other have to be properly configured. The modules have to be programmed with firmware for their role in the network (coordinator, router or end device) and be assigned a network ID and destination addresses.

Appropriate firmware has to be uploaded to devices using the XCTU program, while the rest of the configuration can be performed either with it or directly from the devices it is connected to using the command mode.

It is necessary to test and confirm that XBee modules are able to communicate with each other before connecting them to the system, as in the instance where communication does not function, it is necessary to know which part of the system is faulty in order to fix it.

6.3 UART communication

Like with I²C, UART requires initialization and configuration in order to function, and additionally two ring buffers to hold the data queued to send, and the received data before it is processed.

```
void Setup_UART()
{
    Init_UART_PinMux();
    Chip_Clock_SetUARTBaseClockRate(Chip_Clock_GetMainClockRate(), false);
    Chip_UART_Init(LPC_USART);
    Chip_UART_ConfigData(LPC_USART, UART_CFG_DATALEN_8 | UART_CFG_PARITY_NONE | UART_CFG_STOPLEN_1);      //8 bits, no parity bit, 1 stop bit
    Chip_UART_SetBaud(LPC_USART, UART_BAUDRATE);            //Set baud rate
    Chip_UART_Enable(LPC_USART);
    Chip_UART_TXEnable(LPC_USART);

    RingBuffer_Init(&rxring, rxbuff, 1, UART_RB_SIZE);     //Before using the ring buffers, initialize them using the ring
    RingBuffer_Init(&txring, txbuff, 1, UART_RB_SIZE);     //buffer init function

    Chip_UART_IntEnable(LPC_USART, UART_INTEN_RXRDY);       //Enable receive data and line status interrupt
    Chip_UART_IntDisable(LPC_USART, UART_INTEN_TXRDY);      //May not be needed
    NVIC_EnableIRQ(LPC_IRQNUM);                             //Enable UART interrupt
}
```

Listing 3. UART initialization

Listing 3 shows the function that initializes the UART peripheral and ring buffers it uses to store transmission data, as well as configures parameters that it uses for data transmission.

6.4 AD5593R configuration and usage

Before AD5593R can be used to measure or output voltage, at least some of its channels need to be configured as DAC or ADC. There are also other configurable options, such as buffers on ADC channels, internal reference and load DAC mode.

To make changes to AD5593R's configuration, it is necessary to transmit three bytes: pointer byte, which indicates which settings (such as pin configuration, general settings or power options) are going to be changed, and two bytes that contain new settings. [8, 23]

For instance, pointer byte 00000011 points to general-purpose configuration, and bit 8 of the general-purpose control register indicates whether the ADC buffer is enabled. [8, 24].

```
void AD5593R_Write (uint8_t pointer_byte, uint8_t MSB, uint8_t LSB)
{
    /*  Sequence:
     *  Send start condition
     *  Send 7-bit address followed by direction bit
     *  Send pointer byte and data
     *  Send stop condition
     */

    uint8_t send[3];
    uint8_t receive;

    Chip_I2CM_SendStart (LPC_I2C0);

    send[0] = pointer_byte;
    send[1] = MSB;
    send[2] = LSB;

    I2C_Transfer(AD5593R_ADDR, send, 3, &receive, 0);
    Chip_I2CM_SendStop (LPC_I2C0);
    __WFI();
}
void AD5593R_Configure() {
    AD5593R_Write(0b00001111, 0, 0);                        //soft reset
    AD5593R_Write(0b00000011, 0b00000001, 0b00110000);      //set ADC range to 2xVref, set DAC range to 2xVref, enable ADC buffer
    AD5593R_Write(0b00000101, 0b00000000, 0b00000011);      //set I/0 pins 0,1 to DAC
    AD5593R_Write(0b00000100, 0b00000000, 0b00011100);      //set I/0 pins 2,3 to ADC
    AD5593R_Write(0b00001011, 0b00000010, 0b00000000);      //enable internal reference
}
```

Listing 4. AD5593R configuration functions

Listing 4 illustrates a function that is used for configuration and a sequence of commands used to configure AD5593R for this project. The first command resets settings to default to prevent any misconfiguration and the following commands configure internal reference, enable ADC buffer and designate pins as ADC or DAC.

```
float AD5593R_ADC_ReadVoltage(uint8_t channel) {
    uint16_t data;
    uint8_t LSB;
    float voltage;

    LSB = 1 << channel;                             //set the bit that indicates channel to read
    AD5593R_Write(0b00000010, 0b00000000, LSB);     //transmit read command
    data = AD5593R_Read(0b01000000) & 0xFFF;        //read measurement result and remove unnecessary data from result
    voltage = (float)data / 4095 * AD5593R_VREF;    //convert result to a decimal number
    return voltage;
}

void AD5593R_DAC_SetVoltage(float voltage, uint8_t channel) {
    uint8_t MSB, LSB, pointer;
    uint16_t data;

    if (channel <= 7) {
        pointer = 0b00010000 | channel;             //add DAC channel to the pointer byte
        MSB = 0b10000000;
        MSB = MSB | (channel << 4);
    } else
        MSB = 0b10000000;

    data = (int)((voltage/AD5593R_VREF)*4095);      //convert voltage to 12-bit binary number
    MSB = MSB | (data >> 8);                        //split the number between first
    LSB = data & 0b11111111;                        //and second byte
    AD5593R_Write(pointer, MSB, LSB);
}
```

Listing 5. AD5593R DAC write and ADC read functions

Listing 5 illustrates the functions that perform voltage measurement and voltage output using AD5593R. To perform a measurement, it is necessary to transmit a pointer byte

that indicates the ADC read command, and a byte that contains the address of the ADC channel to read. After the command is transmitted, AD5593R will return the measured voltage in a 12-bit number, along with the channel of the ADC. Then the bits indicating the channel are removed from the number, and number is converted to a voltage value.

In order to output voltage with a DAC, a pointer byte with a DAC write command and address of the DAC channel needs to be sent, along with a 12-bit number that represents the output voltage. The number needs to be split into two separate bytes before it can be sent to the DAC. Once AD5593R receives the command, it either changes voltage automatically, or waits for the command to apply all previously sent voltage values, depending on whether the load DAC (LDAC) mode is enabled. In this case, the LDAC mode is disabled and changes are applied immediately.

6.5 Interfacing with ventilation system

In order to control the ventilation system, its fan control signals were disconnected from their respective inputs in order to measure them, make adjustments to fan speed and output adjusted control signals to the inputs that the original signals were connected to.

Since ventilation fans use 0-10V control signals, and AD5593R can measure and output up to 5V, it was necessary to reduce the voltage of the fan control signals coming from the ventilation system before they could be measured by an ADC, and double voltage output from DAC in order to have full range of control over the fans.

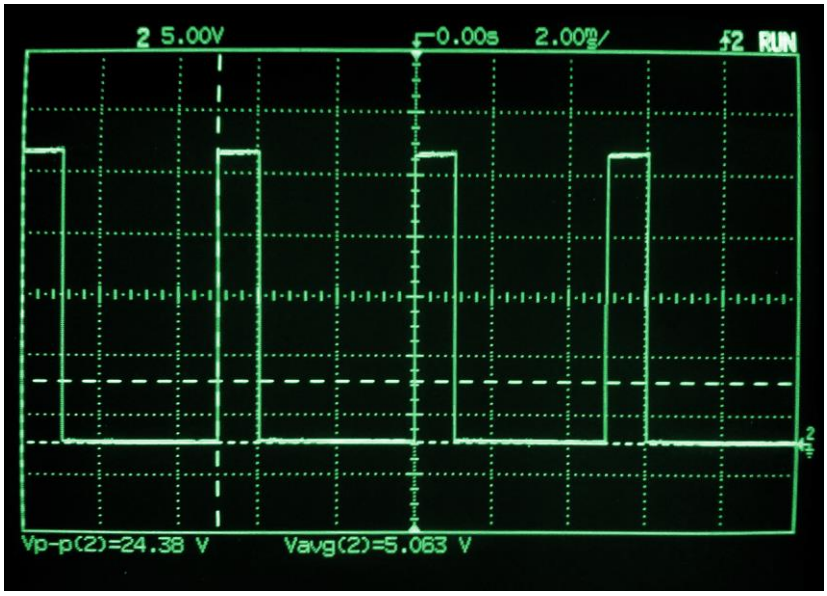

Figure 15. Oscilloscope measurement of a fan control signal at 50% power.

Additionally, due to lack of space, automation engineers from Datasteel Oy had used pulse width modulation (PWM) instead of analog output card to control fans. Instead of analog 0-10V signal, there is a 24V pulse width modulation (PWM) signal with an average magnitude of 0-10V, the measurement of which can be seen in Figure 15. Therefore, it was necessary to add a filter to rectify the PWM signal before it could be measured by an ADC.

In order to halve the voltage, a voltage divider was used. The voltage divider consists of two resistors of equal resistance in parallel, one of which is connected to the ground, and the voltage across the other is measured.

One way to read an average voltage of a PWM signal is to pass it through a low-pass resistor-capacitor (RC) filter with resistance and capacitance that are high enough for the output signal to become sufficiently uniform. Insufficient resistance or capacitance of the filter will result in high ripple that would decrease the quality of the measurements.

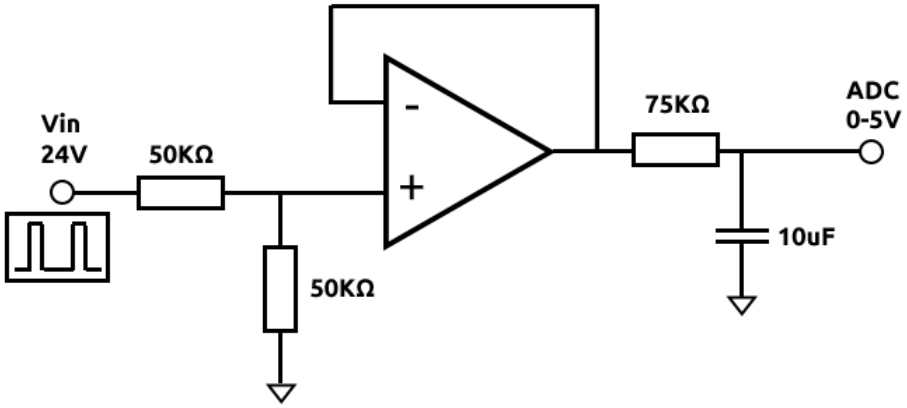

Figure 16. Diagram of the voltage divider and filter circuit

Figure 16 illustrates the complete circuit used to transform 24V PWM signal from the ventilation system to 0-5V analog signal that can be measured by AD5593R. After the voltage is halved, the signal passes through an op-amp buffer in order to prevent unwanted voltage division between the voltage divider and the filter. After the buffer, the signal passes through a RC filter in order to turn it into an analog signal, which can then be measured by AD5593R.

Since AD5593R has its own internal buffers, the output impedance of the circuits connected to its inputs does not need to be taken into account.

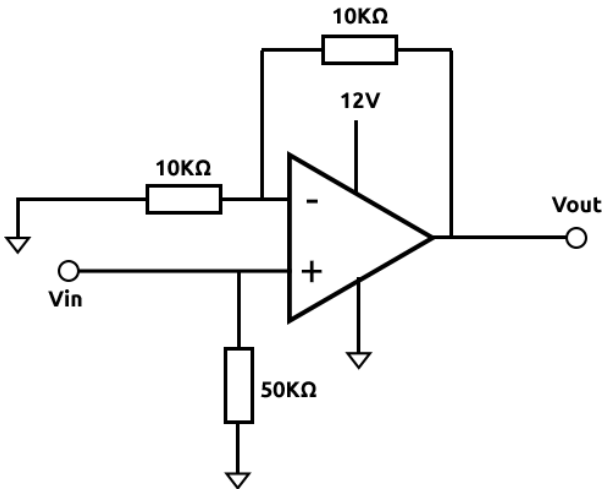

Figure 17. Diagram of an operational amplifier circuit

To double the voltage, an op-amp is necessary. LM324 was chosen, as it is cheap, very common, operates within the voltage ranges necessary for this project and operates from a single power supply. A single LM324 chip provides four independent op-amps. Figure 17 illustrates the circuit used to double the voltage using a single-supply op-amp.

6.6 PID controller

At a set interval, a function that updates the PID controller would be called by the main function of the microcontroller, with pressure measurement being passed as the error, as it represents how much the current state of the system deviates from the desired state of the system (zero pressure difference).

```
typedef struct {
    float windup_guard;
    float proportional_gain;
    float integral_gain;
    float derivative_gain;
    float prev_error;
    float int_error;
    float control;
    float fan_min;
    float fan_max;
} PID;

void PID_update(PID* pid, float error) {
    float diff, P_term, I_term, D_term;

    pid->int_error += error;                        //integration with windup guarding
    if (pid->int_error < -pid->windup_guard)
        pid->int_error = -pid->windup_guard;
    else if (pid->int_error > pid->windup_guard)
        pid->int_error = pid->windup_guard;

    diff = error - pid->prev_error;                 //differentiation

    P_term = pid->proportional_gain * error;    //scaling
    I_term = pid->integral_gain * pid->int_error;
    D_term = pid->derivative_gain * diff;

    pid->control = P_term + I_term + D_term;    //summation of terms

    float limit = pid->fan_max - pid->fan_min;      //limiting maximum control output

    if (pid->control > limit)
        pid->control = limit;
    else if (pid->control < -limit)
        pid->control = -limit;

    pid->prev_error = error;                    //saving current error as previous error for next iteration

    printf("PID: P = %0.3f, I = %0.3f, D = %0.3f, Control =  %0.3f\r\n", P_term, I_term, D_term, pid->control);
}
```

Listing 6. Implementation of a PID controller

Listing 6 illustrates the implementation of the PID_update function, as well as lists all of the variables that are used by the PID controller. The PID_update function calculates the proportional term by multiplying the error by the proportional gain, the integral term by adding the current error to the sum of the previous errors and multiplying the sum by the integral gain, and the derivative term by subtracting the previous error from current one and multiplying the result by the derivative gain. The terms are then summed, and the resulting value is checked to be within the bounds of what can be output to the system.

After the controller is implemented, in order to perform its function, its parameters will need to be tuned. The main parameters are three gain values, and additional ones specific to this implementation are windup guard (the maximum allowed accumulated value of integral error), as well as the upper and lower limits of the fan speed.

6.7 Main functions

With all individual components of the system implemented, all that was remaining to implement were main functions of two microcontrollers that would control the operation of the system.

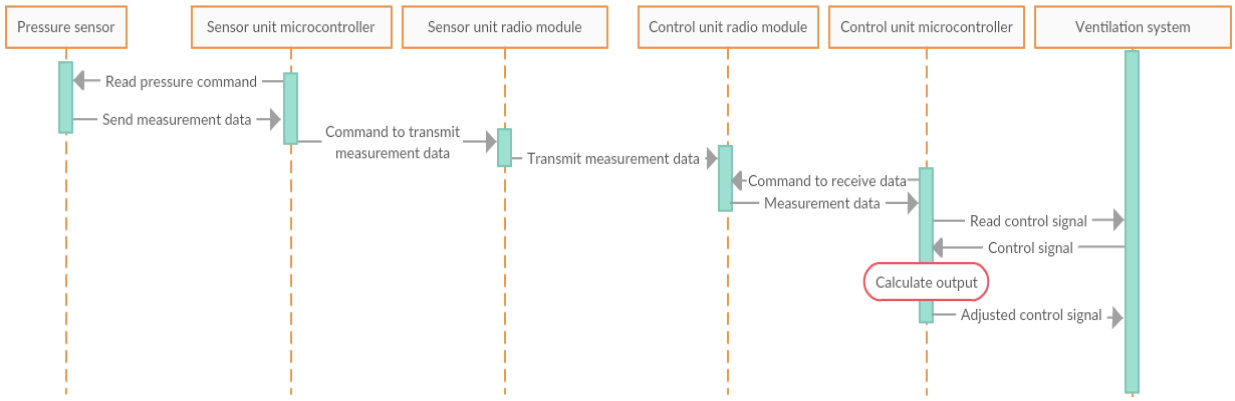

Figure 18. Sequence diagram of the system

Figure 18 illustrates the sequence of events between the main components of the system during regular operation. Both microcontrollers perform their parts of the sequence inside an infinite loop, and after each microcontroller completes its part of the sequence, it goes into sleep mode for a specified amount of time, after which the sequence is repeated again.

```
    Init_Board();
    Setup_I2C_Master();
    Setup_UART();

    while (1) {
        while (lastState == state) __WFI();                                    //Sleep until a state change occurs in SysTick

        pressure = SDP610_Read_Pressure();                                     //Read pressure from the sensor
        pressure_pa = (float)Unisgned16IntToSignedInt(pressure) / SDP_610_SCALE_FACTOR; //convert received value to Pa

        printf("Pressure difference is %.4f Pa\r\n", pressure_pa);

        sprintf(send, "%08.3f\n", pressure_pa);                                //format data for transmission
        Send_UART(&send, sizeof(send));                                        //trasnsmit data

        lastState = state;                                                     //Reset lastState to allow for WFI
    }
    return 0;
}
```

Listing 7.Main function of the sensor unit

Listing 7 shows the implementation of the sensor unit's main function. The unit initializes its hardware, then enters the infinite loop, where it reads data from the sensor, converts it into pascals in a specified format, and transmits converted data to the ventilation control unit.

```
Init_Board();
Setup_I2C_Master();
Setup_UART();

ADS593R_Configure();
PID_initialize(&PID_data);

while (1) {
    while (lastState == state) __WFI();                                        //Sleep until a state change occurs in SysTick

    int bytes = Receive_UART(&read, sizeof(read));                             //check for data from XBee
    if (bytes > 0) {                                                           //if anything received
        printf("Received %d bytes: %s \n", bytes, read);
        if (bytes % MESSAGE_SIZE == 0) {                                       //if one or several data packets received
            char readbuffer[MESSAGE_SIZE];
            int measurements = bytes / MESSAGE_SIZE;
            int valid_count = 0;
            float pressure_buffer = 0;

            for (int i = 0; i < measurements; i++) {
                memcpy(readbuffer, read + i * MESSAGE_SIZE, MESSAGE_SIZE);         //reading each message into a buffer
                if ((readbuffer[4] == '.') && (readbuffer[8] == '\n')) {           //verifying contents
                    sscanf(read, "%f%c", &pressure_pa);                            //converting string to float
                    pressure_buffer += pressure_pa;                                //adding to sum to calculate average
                    valid_count++;                                                 //keeping count of valid measurements
                }
            }
            pressure_pa = pressure_buffer / valid_count;                           //calculating average of received measurements
            PID_update(&PID_data, pressure_pa);                                    //sending the value to PID controller
        }
    }

    voltage_intake = ADS593R_ADC_ReadVoltage(2);                               //reading fan control signals
    voltage_exhaust = ADS593R_ADC_ReadVoltage(4);
    printf("ADC 2 (intake) data: %0.3f V, ADC 4 (exhaust) data: %0.3f V \r\n", voltage_intake, voltage_exhaust);

    voltage_intake += PID_data.control/2;                                      //splitting control signal value between both fans
    if (voltage_intake > PID_data.fan_max) {                                   //ensuring that intake output is in allowed range
        voltage_exhaust = voltage_exhaust + PID_data.fan_max - voltage_intake;
        voltage_intake = PID_data.fan_max;                                     //if intake is over maximum, subtract from intake and add to exhaust
    } else if (voltage_intake < PID_data.fan_min) {
        voltage_exhaust = voltage_exhaust - voltage_intake + PID_data.fan_min;
        voltage_intake = PID_data.fan_min;                                     //if intake is below minimum, subtract from exhaust and add to intake
    }

    voltage_exhaust -= PID_data.control/2;                                     //splitting control signal value between both fans
    if (voltage_exhaust > PID_data.fan_max) {                                  //ensuring that exhaust output is in allowed range
        voltage_intake = voltage_intake + PID_data.fan_max - voltage_exhaust;
        voltage_exhaust = PID_data.fan_max;                                    //if exhaust is over maximum, subtract from exhaust and add to intake
    } else if (voltage_exhaust < PID_data.fan_min) {
        voltage_intake = voltage_intake - voltage_exhaust + PID_data.fan_min;
        voltage_exhaust = PID_data.fan_min;                                    //if exhaust is below minimum, subtract from intake and add to exhaust
    }

    printf("Intake output: %0.3f V, exhaust output: %0.3f V \r\n", voltage_intake, voltage_exhaust);
    ADS593R_DAC_SetVoltage(voltage_intake, 0);                                 //outputting adjusted signals
    ADS593R_DAC_SetVoltage(voltage_exhaust, 1);

    lastState = state;                                                             //Reset lastState to allow for WFI
}   //while (1)
```

Listing 8. Implementation of the ventilation control unit's main function

Listing 8 shows the implementation of the ventilation control unit's main function. After initializing hardware and peripherals, it enters an infinite loop. The first action inside the loop is a check for new received data from sensor units.

If any data is received, its length will be checked, and if multiple data packets are received, each packet will be processed separately. The contents of each packet are verified to be of the correct format, and if they are, they are converted into a variable. As the received messages are processed, their average value will be calculated, and once all messages are processed, the average value will be sent to the PID controller. This allows this implementation to correctly work regardless of how many sensor units send it measurement values simultaneously.

Afterwards the function reads the fan control signals of the ventilation system, makes adjustments to them according to the output of the PID controller, ensures that new control signals are within the bounds and sends them to the fans of the ventilation system.

The control value received from the PID controller is split between the input fan and the exhaust fan equally. For example, a control value of 0.5 would subtract 0.25 volts from the exhaust fan and add 0.25 volts to the intake fan control voltage. In cases where after this operation a control signal exceeds the minimum or maximum allowed values, the value that exceeds maximum or is below the minimum is set to the limit that is exceeded, and the other value is adjusted to maintain the difference set by control value. For instance, if the controller reads the fan control signals of 4.5V and calculates a control value of -2, the initial split would provide 3.5V intake fan control signal, and 5.5V for exhaust. Since 5.5V is above the limit of 5V, the 0.5V above the limit would be subtracted from the intake control signal, resulting in the output of 3V to intake fan and 5V to the exhaust fan.

The PID controller implementation is programmed to avoid outputting the control signal that would exceed the difference between the upper and lower limits of the fan control signals in order to prevent undefined behavior that would be caused by overflowing voltage values being sent to a DAC.

After the adjusted control signals are output, the microcontroller will enter the sleep mode for a specified amount of time.

7 Testing

7.1 Testing during development

During development of the project, each part of the system was tested independently in order to ensure that it performs as intended, and to assess its performance without other elements of the system potentially affecting the results.

As soon as I²C communication was implemented, the communication with the pressure sensor was tested directly in order to confirm that the microcontroller receives correct measurement data that corresponded to the pressure that was being applied to the measurement ports of the sensor.

Communication of XBee modules was tested separately before the sensors were connected to the system to ensure that in case communication was not functioning properly, they were not caused by misconfiguration or defects in the modules. After UART communication was implemented, communication was also tested by sending arbitrary data from one microcontroller and confirming that the other microcontroller received the same data.

Voltage division, filtering and amplification circuits were tested with an oscilloscope in order to ensure that they provided the desired output across the entire voltage range of control signals, and that their output never exceeded the absolute maximum limits of the system, as exceeding the maximum ratings could cause damage to the hardware they were connected to.

Performance of ADC and DAC was tested both by comparing their readings and outputs to the readings performed with an oscilloscope, and by connecting a DAC output to an ADC input and checking that the ADC reads an identical value to one that the DAC was instructed to output. After testing by connecting a DAC to an ADC directly, the ADC and DAC were also tested in the same way with the voltage divider and amplifier circuits connected to them. The testing showed that the difference between the signal output by a DAC, which was then amplified, passed through the voltage divider circuit and measured by an ADC was less than 10 mV.

The testing and debugging of the software was performed using the ITM functionality of the microcontroller. Whenever it was necessary to observe the state of any variable, the printf function was used in order to output the variable in the desired format to the console output of the LPCXpresso IDE.

```
printf("Received %d bytes: %s \n", bytes, read);
printf("PID: P = %0.3f, I = %0.3f, D = %0.3f, Control =  %0.3f\r\n", P_term, I_term, D_term, pid->control);
printf("ADC 2 (intake) data: %0.3f V, ADC 4 (exhaust) data: %0.3f V \r\n", voltage_intake, voltage_exhaust);
printf("Intake output: %0.3f V, exhaust output: %0.3f V \r\n", voltage_intake, voltage_exhaust);

Received 20 bytes: -005.438
PID: P = -1.088, I = -0.148, D = -0.000, Control =  -1.235
ADC 2 (intake) data: 3.504 V, ADC 4 (exhaust) data: 3.504 V
Intake output: 2.887 V, exhaust output: 4.122 V
```

Listing 9. Debugging functions and console output.

Listing 9 shows some of the fuctions that were used to monitor the system and gather data during testing, as well as the output they produced in the ITM console of LPCX-presso IDE.

Before final testing, the response of the ventilation system was tested by sending arbitrary control signals to it and verifying that both the intake and exhaust fans correctly adjusted their speed according to the control signals.

7.2 PID controller tuning

Tuning of the PID controller requires running it in a real environment, as at least within the scope of this project, it would not be feasible to simulate how the pressure difference would change based on output of the ventilation system. Tuning of the controller is performed by adjusting its gain values (k_p, k_i, and k_d).

Since this particular implementation does not use a derivative component, k_d is set to zero. The first parameter that is tuned is k_p. While k_p is being tuned, k_i is set to zero. In order to find the appropriate value of k_p, the k_p is progressively increased until control signal it produces is sufficient to counteract the pressure difference. The value of k_p necessary to achieve this is called the critical gain k_c. With $k_p = k_c$ the pressure difference tends to oscillate between positive and negative values. After k_c was determined, k_p was set to approximately $0.5k_c$. [3, 302-303]

After k_c value is determined, k_i needs to be adjusted. A low value of k_i causes the system to stabilize very slowly while the integral term accumulates with each cycle until it is sufficient to compensate for the error. A high value of k_i causes the system to reach zero error quickly, but causes overshoot afterwards. Using trial and error, the value of k_i is set to one that provides desirable balance between the speed of the response and the magnitude of overshoot.

After k_i value is determined, windup guard value needs to be set. As the integral term accumulates the previous values of the error, there is a possibility of the term accumulating beyond the intended values, dominating the system and locking it into a minimum or maximum output state. To prevent such a scenario, a maximum limit is set on the value that an integral term may accumulate. The value is set to one that, when multiplied by k_i, would produce the maximum desirable value of the integral term.

7.3 Final testing

Once all parts of the project parts were successfully implemented, the entire system was tested while being connected to the ventilation system. As the system was running, the data was gathered from the ITM console.

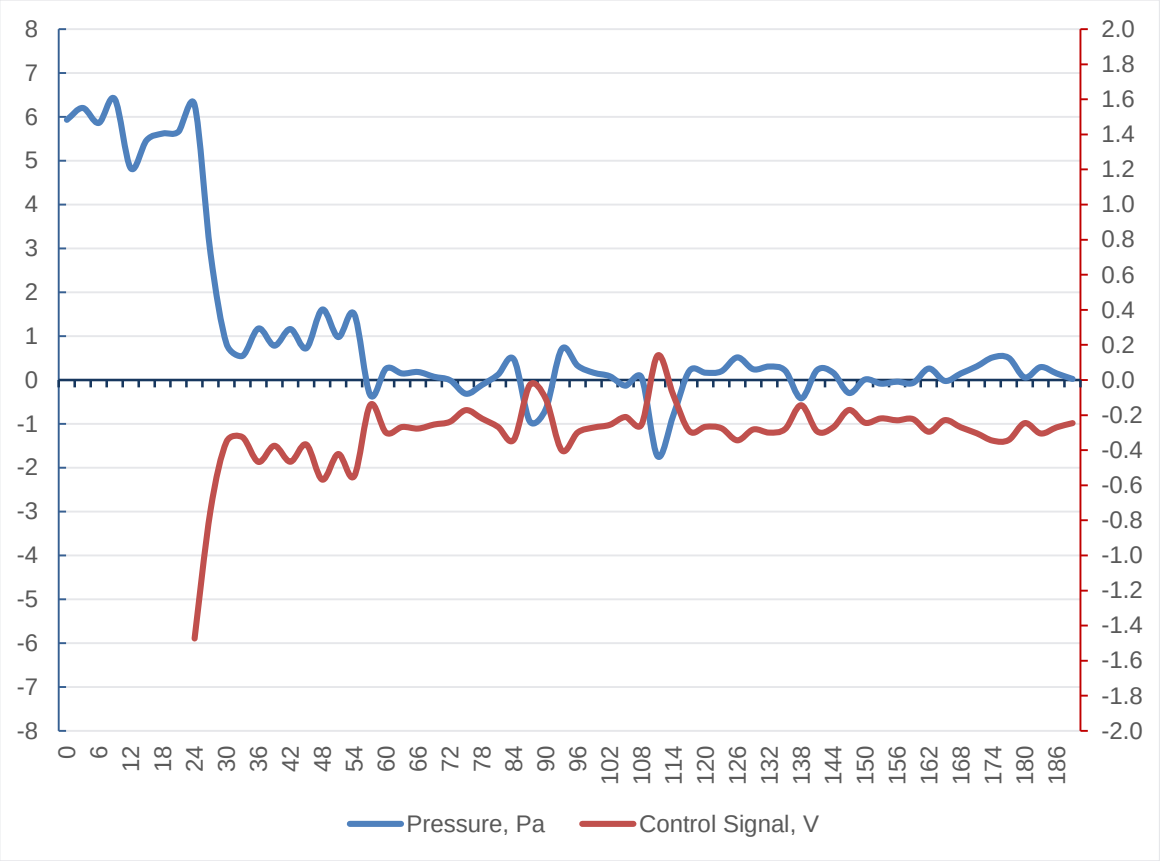

Figure 19. Pressure difference and control signal data

Figure 19 illustrates the data from the pressure sensor on the left vertical axis (blue line) and the control signal that adjusts the relative speed of the intake and exhaust fans on the right vertical axis (red line) over time, which is shown on the horizontal axis in seconds. A pressure reading and adjustment of the control signal was performed every three seconds.

The graph starts with 20 seconds of pressure data prior to activation of the ventilation system, where pressure difference was 6 Pa. As soon as the system is activated pressure difference immediately drops, and after roughly 30 seconds since activation stabilizes within ±0.5 Pa, outside of occasional spikes, which are typically caused by external factors. The control signal can be seen stabilizing at -0.3 volts, which represents the difference between the intake and exhaust fan speeds necessary to compensate for the initial pressure difference. The noise that can be seen on the graph after the system stabilizes is caused by the sensor, as well as temporary pressure disturbances like wind.

The proportional term of the PID controller compensates for the spikes in pressure, while the long-term stable value that compensates for constant pressure difference comes from the integral term.

8 Conclusion

The goal of this project was to design and implement an embedded system that would control a ventilation system in order to equalize pressure between inside and outside of a building.

The system was successfully implemented and the testing confirmed that had achieved its purpose. As soon as the system is enabled, the pressure difference very quickly reaches zero and remains stable afterwards. The system was implemented using common and relatively cheap components. The project was concluded with the system in a prototype stage using evaluation boards. However, it would be possible to design custom circuit boards for it.

Even though the system was developed to interface with a particular ventilation system, most of it could be reused when connecting it to a different ventilation system. The specific parts that would need to be adjusted or replaced are the circuit responsible for processing ventilation control signals in order for them to be readable by the system and the circuit responsible for outputting control signals for the fans.

Completion of this project required knowledge and skills in three distinct fields: programming, which was necessary to write software for the microcontrollers, embedded systems, which was needed to design the architecture of the system and choose necessary hardware components, and electronics, which was necessary in order to design some of the circuits and build the prototype.

References

1. Bart Broekman, Edwin Notenboom. Testing Embedded Software. Harlow: Pearson Education Ltd., 2003.

2. Tammy Noergaard. Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers. Oxford: Elsevier; 2005

3. Karl Johan Åström, Richard M. Murray. Feedback Systems. Version v2.11b. [Online] Santa Barbara: Princeton University Press; 2008
   URL: http://www.cds.caltech.edu/~murray/books/AM08/pdf/am08-complete_28Sep12.pdf
   Accessed November 16 2016

4. LPCXpresso™ Board for LPC1549. [Online].
   URL: http://www.nxp.com/products/software-and-tools/hardware-development-tools/lpcxpresso-boards/lpcxpresso-board-for-lpc1549:OM13056
   Accessed November 6 2016

5. XBee® ZigBee. [Online].
   URL: https://www.digi.com/products/xbee-rf-solutions/rf-modules/xbee-zigbee
   Accessed November 6 2016

6. SDP600 Series Low-cost Digital Differential Pressure Sensor datasheet [Online].
   URL: https://www.sensirion.com/fileadmin/user_upload/customers/sensirion/Dokumente/Differential_Pressure_Sensors/Sensirion_Differential_Pressure_Sensors_SDP6x0series_Datasheet_V.1.9.pdf
   Accessed November 6 2016

7. LPCWare support: How to use ITM Printf. [Online].
   URL: https://www.lpcware.com/content/faq/lpcxpresso/how-use-itm-printf
   Accessed November 13 2016

8. AD5593R Datasheet [Online].
URL: http://www.analog.com/media/en/technical-documentation/data-sheets/AD5593R.pdf
Accessed November 13 2016

9. CMOSens® Technology for Gas Flow and Differential Pressure [Online].
URL: https://www.sensirion.com/technology/cmosensr-technology-for-gas-flow/
Accessed November 19 2016

10. I²C-bus specification and user manual UM10204 [Online].
URL: http://www.nxp.com/documents/user_manual/UM10204.pdf
Accessed November 21 2016