Measurement and control of the angular velocity of a DC motor via a microcontroller and a Wi-Fi application

Aleksa Rančić
Department of Power, Electronic and Telecommunication Engineering
Faculty of Technical Sciences
University of Novi Sad
Novi Sad, Serbia
aleksarancic@gmail.com

Marjan Urekar
Department of Power, Electronic and Telecommunication Engineering
Faculty of Technical Sciences
University of Novi Sad
Novi Sad, Serbia
urekarkm@gmail.com
https://orcid.org/0000-0002-6089-9148

***Abstract* — This paper contains an explanation of how a fully functional control system is realized on a microcontroller and the different ways the user can interface with it. The main control loop is written in C language, specifically using the FreeRTOS embedded operating system. FreeRTOS enables the microcontroller to handle multiple tasks, seemingly at once. The tasks include handling the system control loop, the Wi-Fi functionality and other human – machine interface solutions this project includes. The PC application written in Python enables the user to communicate with the microcontroller wirelessly.**

Keywords — Application, Wi-Fi, GUI, microcontroller, DC motor, FreeRTOS

I. MOTIVATION AND INTRODUCTION

In the world of industrial automation and smart technologies, there is a growing need for systems that can be controlled remotely, in real time. Classic control systems often require physical access to equipment or complex wiring to control rooms, which becomes a real problem when the actuator is located in hard to reach or hazardous environments. The need for two-way wireless communication with the actuator is what represents the fundamental motivation of this work.

This paper describes the implementation of a system for wireless control and monitoring of the speed of a DC motor that responds to the aforementioned challenges. The basic idea was to develop a system with multiple control methods, where Wi-Fi communication is the main one.

Through the PC application, the user can not only see the speed of the motor, but also change PID parameters, activate autotuning and instantly see the effect of those changes in real time. The system automatically recovers from temporary connection loss if such event happens.

This research has been supported by the Ministry of Science, Technological Development and Innovation (Contract No. 451-03-34/2026-03/200156) and the Faculty of Technical Sciences, University of Novi Sad through project "Scientific and Artistic Research Work of Researchers in Teaching and Associate Positions at the Faculty of Technical Sciences, University of Novi Sad 2026" (No. 01-3609/1).

What makes this work unique is layering and integration. Wi-Fi functionality is realized as a separate FreeRTOS task with its own state machine, equal to the regulation task. At the same time the user is not only limited to the PC application. A local TFT touch screen and a physical joystick are also available, which ensures full functionality even without a Wi-Fi connection. The goal was to create a modern control system based on wireless connection that combines the power of microcontrollers with the modularity of FreeRTOS and the prevalence of Wi-Fi technologies into one coherent whole.

II. SYSTEM COMPONENTS

This project was realized on a PIC32MZ2048EFH144 32-bit microcontroller on an EasyPIC Fusion v7 development board. Additional components include an external power supply for powering the motor, JGA25 370 DC12V1364RPM DC motor with an encoder, EasyTFT ILI9341 display, DC motor click and WiFi ESP click. A CODEGRIP programmer and debugger was used for programming the microcontroller.

III. WORKING PRINCIPLE

This project contains two main parts, which will be covered in this chapter: the microcontroller firmware and the Python application.

A. INTRODUCTION TO PID CONTROL

The proportional-integral-derivative (PID) controller is one of the most common linear controllers in industrial and process automation. Its relatively simple structure, robustness and ability to provide satisfactory performance for a wide range of processes of diverse nature make it an ideal choice in a large number of applications. The main role of the PID controller is to maintain the output process

ISBN 978-86-6200-074-3 ©2026 AM & ETRAN
https://doi.org/10.69994/13Ic26018

variable at a desired value set by the user, even under the influence of external disturbances acting on the system. This is achieved by means of a feedback loop which is shown in Fig. 1.

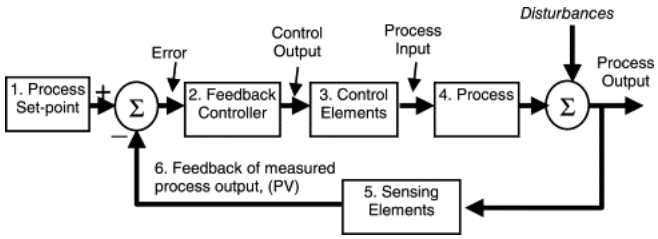

Fig. 1. Block diagram of a closed-loop feedback control system with PID controller, showing the signal flow from the setpoint *r(t)* to measured output *y(t)*

- Process set-point – the desired value of the process output variable,
- Error – the difference between the measured process output variable and the set-point,
- Feedback controller – consists of the P, I and D action,
- Control output – the three actions of the PID regulator combined. They are supposed to tell the control element how to act on the process,
- Control element – the physical element responsible for controlling the process. In this case, the DC motor 2 click board/driver,
- Process – the subject of regulation. In this case, the DC motor itself,
- Sensing element – sensor for the output variable. In this case, the sensing element is an encoder.

A simplified transfer function of a DC motor is shown in equation (1) (where K_{ss} is the steady state gain and τ is the motor's time constant) and the effects of the PID action's gains on a system's dynamics can be seen in table 1.

$$G(s) = \frac{K_{ss}}{\tau s+1} \quad (1)$$

The P action is proportional to the value of the error variable. The integral action I produces a control signal which corresponds to the accumulated value of the error over time. The differential D action approximates the future behavior of the error. These three actions combined make up the control output signal of the PID regulator. Considering this is a digital system, the three aforementioned actions need to be discretized and also modified for real practical use. After these modifications they are as shown in equations (2) – (5).

$$P(kT) = K(br(kT) - y(kT)) \quad (2)$$

$$I(kT+T) = I(kT) + \frac{KT}{T_i}e(kT+T) \quad (3)$$

$$D(kT) = \frac{2T_d - NT}{2T_d + NT}D(kT-T) - \frac{2KT_dN}{2T_d+NT}(y(kT) - y(kT-T)) \quad (4)$$

	Rise time	Overshoot	Settling time	Steady-state error
P	Decrease	Increase	Little to no effect	Decrease
I	Decrease	Increase	Increase	Decrease
D	Little to no effect	Decrease	Decrease	No change

Table 1. Effects of PID action's gains on system's response

$$PID\ (kT) = P(kT) + I(kT) + D(kT) \quad (5)$$

In these equations, k is the sample index, T is the sampling period, K is the proportional gain, b is the setpoint weighting factor, T_d and T_i are the derivative and integral time respectively, N is the derivative filter coefficient *u(kT)* is the control output, *y(kT)* is the measured speed and *e(kT)* is the error.

B. MICROCONTROLLER FIRMWARE – OPERATING SYSTEM AND MECHANISMS

This project's firmware was conceptualized as a multi-threaded program which is why it was written in FreeRTOS [1]. Embedded operating systems present an alternative to so called "bare metal" programming, allowing the main control loop to be split into multiple tasks and giving more control over the execution of the tasks to the programmer. Each task is independent from the other, having its own task priority, stack memory and function. This project is split into four tasks: vPIDtask, vMotorControl, vTFTtask, vWifiTask.

These tasks by themselves can't do much without communicating with each other. For that purpose, standard FreeRTOS mechanisms such as queues and notifications were used.

Communication between tasks in this project is largely done via queues. Queues are a mechanism used to send packets of data of variable size between tasks. For all the tasks to work as intended, they need to work with five variables: the setpoint, the current speed and the three PID parameters. For the convenience of not sending five different variables separately between tasks, a data structure was formed to combine all of them and send them through the different queues. Each task has a queue to communicate with the other tasks. Tasks in this project are like islands connected by bridges (queues) that are used to propagate the data from one to another, ensuring real time control over the process.

Considering that PIC32MZ2048EFH144 is a single core processor, it can't run multiple tasks at the same time. That's where FreeRTOS's scheduler purpose lies. The scheduler's job is to give choose which tasks get executed at what time and for how long. One of the most popular types of scheduling is Round Robin scheduling - shown in Fig. 2.

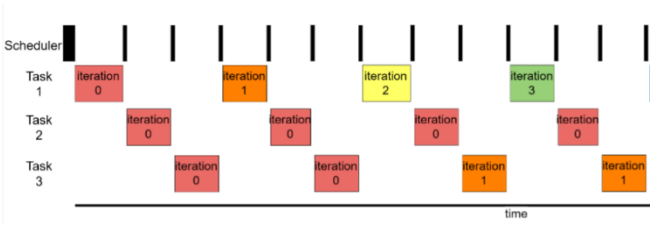

Fig. 2. Round Robin scheduling where each FreeRTOS task gets equal CPU time slices

When all of the tasks have the same level of priority, the scheduler gives each task the equal number of ticks to be executed by the processor. This results in seemingly simultaneous execution of all tasks, which is crucial for real time applications such as PID control.

Semaphores and mutexes are some of FreeRTOS's task synchronizing mechanisms used during the development of this project. A semaphore is used when a task is supposed to wait for a signal from another task or an interrupt routine. Semaphores can be classified as either binary or counting. Binary semaphores are generally used when a task needs to notify another task about a certain event. A counting semaphore is an expansion of the binary semaphore which enables monitoring multiple events at once. A "giving" of the semaphore increments the counter and a "taking" decrements it.

Mutexes are a special type of semaphore that acts as a sort of a token. A task that "takes" a mutex becomes its owner until it "gives" it back. Mutexes are usually used to prevent other tasks from using a resource that's already in use by another task thus ensuring thread safety.

The final FreeRTOS mechanism used in this project is a task notification or simply, notifications. Notifications are built into every task when it's created. This means that elements such as tasks, queues and semaphores need to be explicitly created, whereas notifications do not. They can be used as both a communication and a synchronization mechanism. Notifications can carry a fixed data value of 32 bits. On the contrary, if a notification is used as strictly a task synchronization mechanism it doesn't need to be given a value.

C. MICROCONTROLLER FIRMWARE – vPIDtask

This task contains the PID algorithm and the auto-tuning algorithm. The sample time of this regulator is 100 ms. When the system initializes and is given a setpoint and non-zero values for at least the proportional parameter the motor's shaft will start rotating. As it rotates, the encoder connected to it is generating pulses on its two channels. These impulses force the microcontroller into a short interrupt routine that increments a counting variable. When the sampling period of 100 ms is up, a semaphore is generated and then the task in charge of the PID algorithm becomes active. If the autotuning mode isn't on, the PID assumes the manually set parameters and continues working with them. After that the task checks if any other tasks sent new PID parameters. Then vPIDtask either updates them or keeps them the same, depending if any new parameters were received. After making sure it's working with the most up to date parameters it calculates the current speed, error and the control variable using the PID parameters. The control variable directly affects the duty cycle of the microcontroller's PWM module that supplies the motor with voltage via the DC Motor 2 click driver. PWM is a technique that enables us to mimic analog voltage values with a digital signal. It achieves that by quickly switching from high to low states. By altering the ratio of "on" time and "off" time effectively obtaining an analog value that's somewhere between the high and low digital supply voltages. Example of pulse width modulation shown in Fig. 3.

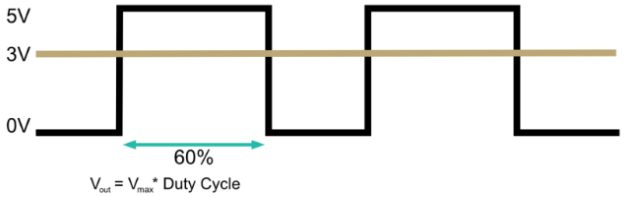

Fig. 3. Example of pulse width modulation (PWM)

The autotuning algorithm is activated by setting an appropriate flag to true. This can be done by pressing a physical GPIO pin that activates an interrupt in which the flag is set or by clicking the "autotune" button in the desktop application that sends a message to the microcontroller to set that flag to true, achieving the same thing. The system then switches to relay mode, which turns the motor on and off at full power to generate controlled oscillations. The oscillations force the current speed to cross the setpoint twice. The task measures the time between these crossings and records the oscillation periods. After two full periods are collected, the PID parameters are calculated using the Ziegler-Nichols formulas. After autotuning is complete, the parameter fine-tuning mode is entered. If the error is greater than 5% of the setpoint, the integral term is increased, while if the oscillations are too large, the proportional term is decreased. This mechanism allows the motor to reach the desired speed without any manual adjustment by the user.

D. MICROCONTROLLER FIRMWARE – vTFTtask

All aspects of the graphical interface of the touch sensitive display, shown in Fig. 4, are managed by this task. When the task function is first entered, the display is initialized. After the initial hardware initialization, a bitmap image is drawn on the display which serves as the background of the interface with already drawn elements that display parameters and an animated speedometer. This task is divided into two separate parts. The first part deals with the processing of the display touches. When the screen is touched (assuming the screen is already calibrated), the corresponding flag is activated, which signals the need to read the touch coordinates of the resistive layers of the display. When the screen is touched, the coordinates are read via the ADC, sampled 21 times, and filtered to remove noise.

The second part of the task manages the refresh of the parameters on the display and the speedometer. For each parameter, there exists a white triangle in the background

image that acts as a frame in which the parameter is presented. In every execution cycle, the parameter is overwritten with another, more up to date parameter received from other tasks.

Overwriting a parameter in this context implies drawing a white filled rectangle over it and then pasting a new parameter over that white triangle. The speedometer is conceptualized as a red rectangle that changes its height depending on the speed that this task constantly receives from vPIDtask.

The display interface is made in a way so that it's similar to the joystick, shown in the bottom right side of the display in Fig. 4, for more intuitive manipulation of the parameters. The way of adjusting parameters via the TFT screen and the joystick will be described in detail in the following chapter.

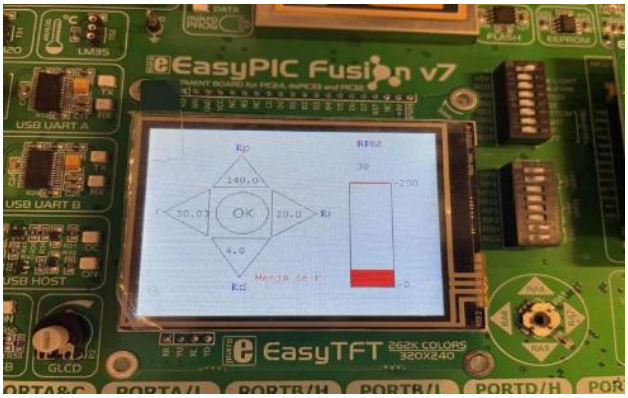

Fig. 4. TFT screen and joystick

E. MICROCONTROLLER FIRMWARE – vMotorControl

As this task's name suggests, this task represents the central control node of the system. It integrates all user inputs and manages the data flow to different modules. It monitors multiple event sources: joystick on the development board, the touch sensitive display and the data from the desktop Wi-Fi application. When the user presses any button of the joystick, the corresponding GPIO interrupt handler sets the value of a structure that indicated the type of press (up, down, left, right and click/OK). After the user selects the desired parameter they want to change, they should confirm it by clicking OK. Then the user is prompted to modify the selected parameter by clicking up or down for increasing or decreasing the selected parameter. When the user is happy with the changes they've made, the changes can be confirmed by clicking OK. Only then are the confirmed parameters forwarded to other tasks to be used by them and the machine returns to its default state – waiting for user input.

The process is exactly the same for modifying the parameters via the touch screen except instead of clicking physical buttons the user is touching the resistive touchscreen. Coordinates that are collected and filtered in vTFTtask are sent to this task using a notification as this was the most memory efficient approach. A set of coordinate borders was made to define which button was being pressed according to the received coordinates.

It's worth mentioning that this task notifies the user in which state they're in by writing a string on the screen which can be problematic because the TFT task is using the screen for most of the operational time of the system. This issue was solved by adding a mutex which this task has to "take" before writing the text on the display for the sake of thread safety. This task's state machine is shown in Fig. 5.

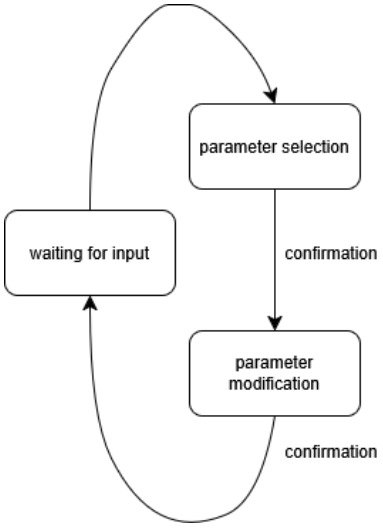

Fig. 5. vMotorControl state machine visualized

F. MICROCONTROLLER FIRMWARE – vWifiTask

Like vMotorControl task, this one is also developed as a state machine, visualized in Fig. 6. The microcontroller communicates with the WiFi ESP click board using the microcontroller's UART module to send AT commands to the Wi-Fi board and receive its replies. AT commands are a

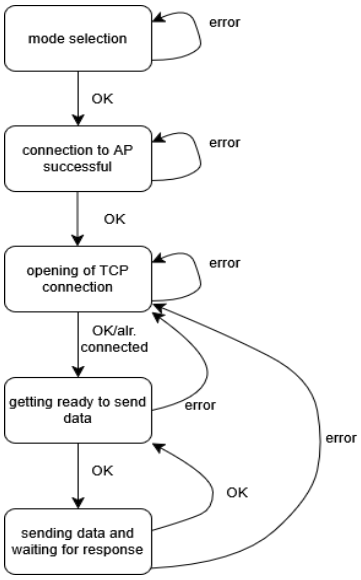

Fig.6. vWiFiTask state machine visualized

command set used for communication and control of devices such as the WiFi ESP click board.

The state machine starts from the CWMODE state, in which a command is sent to set the Wi-Fi board's operating mode to act as a client. If the operating mode setup is successful the state machine goes to the next state which is CWJAP. The AT+CWJAP command is used to connect to the access point. The access point is defined by the router's name and password. If the connection is successful, the TCP connection is ready to be started. In the following state, CIPSTART, a TCP connection is opened at the address and port that need to be specified in both the PC application and the microcontroller's firmware. The last two states manage the data transmission itself. In the CIPSEND state, the AT+CIPSEND command is sent along with the length of the data it plans to send, waiting for a signal from the Wi-Fi module that it is ready to receive. In the SEND state, a data packet is sent with the PID parameters, the current speed and the setpoint values. If an error occurs in any part of the sending the state will be set to CIPSTART.

G. Python Wi-Fi desktop application

The application constantly receives data from the microcontroller via Wi-Fi and updates the graphical interface in real time. The interface is shown in Fig. 7. For developing the app, PyQt5 Python library was used. Communication is two-way between the controller and the application. When the application is started, the graphical user interface Is initialized using the init_ui method. This creates all widgets, configures layouts, and establishes a structure with a left panel for user input and textual display of active parameters and right panel for displaying the speed and setpoint in the graph.

After the GUI is shown to the user, the application's server thread starts. The server thread is in charge of starting a server on a specific port and address to which the controller needs to connect. After successful initialization the server waits for connections.

After the TCP connection is successfully established, the server emits the connection_status_changed signal in the app which updates the display on the status bar of the application's graphical interface. Establishing a connection enables mutual communication between the microcontroller and the application. In the main thread meant for the graphical interface, the update_timer checksfor new data with an interval of 100 ms by calling the check_for_updates method.

If changes in data are detected the new data is updated in the interface. If the user wants to change the parameters, they enter them in the bottom left fields of the window and press the blue button or simply press enter on their keyboard. The input fields are then cleared and the parameters are periodically sent to the controller until it confirms that it has received the data. The user also has an option to activate the autotuning algorithm via the button in the application. The principle of sending a request for autotuning is the same as for sending the parameters.

The application has a robust error recovery system built into it. If three seconds pass without receiving any data from the controller, the connection is automatically restarted. This allows for a quick reconnection in the event of a sudden loss of connection.

IV. DISCUSSION

What sets this project apart from similar ones is its IoT character built into the architecture itself. Instead of an isolated system with local control, the motor has become a connected device that can be controlled from any area covered by the Wi-Fi access point.

The Wi-Fi functionality gives the user the flexibility to relocate the controlled motor anywhere inside the area covered by the Wi-Fi signal without additional cost that comes with rewiring.

The lack of wires also means that there is no risk of them being damaged, resulting in unstable connection. It is worth noting that Wi-Fi can be sensitive to various interferences such as other wireless networks, other 2.4 GHz frequency devices and high voltage devices that generate electromagnetic interference. Reinforced concrete walls and

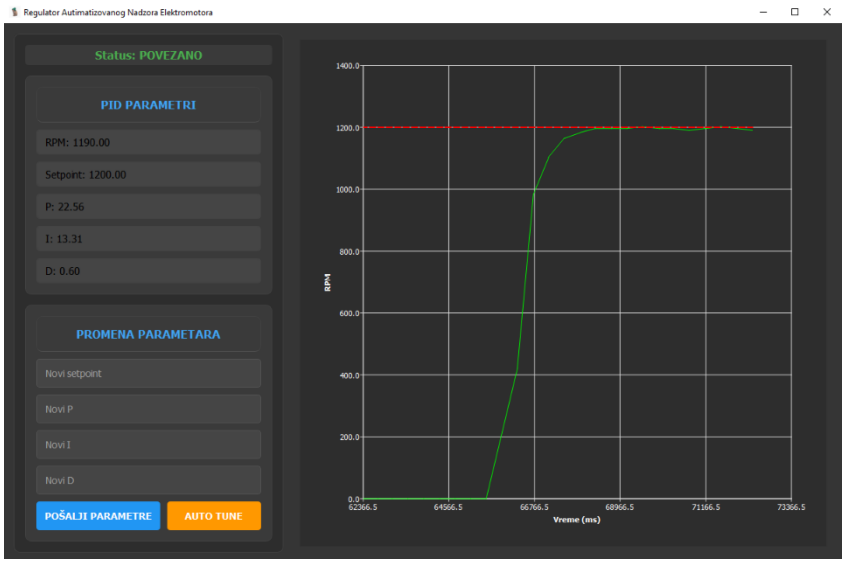

Fig.7. Wi-Fi application window showing a step response with autotuned parameters

other physical constructions can also play part in interfering with Wi-Fi signals.

Wired connections are immune to most of these interferences. Wi-Fi should be used where its flexibility outweighs the risk of short and occasional connection loss.

V. CONCLUSION

This paper shows how Wi-Fi communication can improve the classic control system in a way that preserves local autonomy. The microcontroller manages the regulation independently, while Wi-Fi serves as a bridge to a remote user, without creating dependance on the internet connection thanks to local interfaces. The system is robust, with automatic connection recovery ensuring maximum stability and real time control and supervision. A block diagram of the whole system is shown in Fig 8.

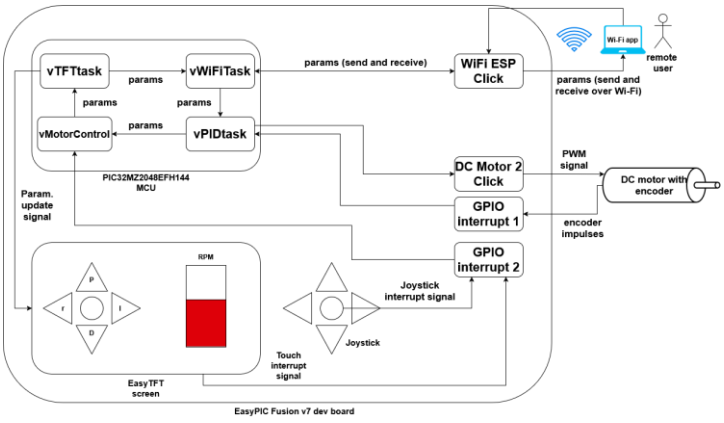

Fig.8. System block diagram

Unlike commercial controllers such as SmartPID, which offer a plug-and-play solution but limit access to internal code, our solution provides full control over the PID algorithm, autotuning, and communication protocol.

Packet loss is minimized by TCP's reliable delivery and retransmission. Under regular operating conditions, the communication link remains stable and control performance is not affected. If connection loss exceeds 3 seconds, the system automatically starts the reconnecting procedure.

Compared to manual tuning (P = 10, I = 15, D = 1), the autotuned PID (P = 22.56, I = 13.31, D = 0.60) both achieve zero overshoot. However, autotuning significantly improves rise time reducing it from 3.5 s to 2.2 s. Steady-state error is in both cases equivalent to the encoder's resolution error.

Further development could include data encryption, multi-user access with different privilege levels (e.g., monitoring only vs. parameter adjustment), and long-distance internet connectivity to align with Industry 4.0 concepts [2][3][4].

REFERENCES

[1] R. Barry, "Mastering the FreeRTOS Real Time Kernel – A Hands-On Tutorial Guide," Real Time Engineers Ltd., 2023.

[2] M. Urekar, J. Djordjevic-Kozarov, I. Gutai, S. Mirković, M. Subotin, Đ. Novaković, "Uloga metrologije u tranziciji industrijske proizvodnje na koncept Industrije 4.0 u Srbiji", 64th national conference ETRAN 2020, Novi Sad, Serbia, 2020.

[3] T. Samad, "Control Systems and the Internet of Things [Technical Activities]," in IEEE Control Systems, vol. 36, no. 1, pp. 13-16, Feb. 2016.

[4] S. N. Swamy and S. R. Kota, "An Empirical Study on System Level Aspects of Internet of Things (IoT)," in *IEEE Access*, vol. 8, pp. 188082-188134, 2020.