Aliaksandr Kavalchuk

Wired Protocols

in Embedded Systems

Wired Protocols in Embedded Systems

A Comprehensive Guide to Wired Embedded Protocols

Aliaksandr Kavalchuk

This book is available at
https://leanpub.com/wiredembeddedprotocols

This version was published on 2025-02-18

* * * * *

* * * * *

Table of Contents

Table of Contents

Intro

Who Is This Book For?

This book is for anyone involved in or aspiring to work with embedded systems, particularly those who need a solid understanding of wired communication protocols. Whether you are a seasoned professional or just starting your journey in the world of embedded development, this book aims to provide the foundational knowledge and practical insights necessary to navigate the complexities of data communication.

- **Embedded System Engineers**

  For engineers designing embedded systems, a deep understanding of wired communication protocols is essential. From connecting sensors and actuators to interfacing with external modules and peripherals, wired protocols form the backbone of reliable and efficient data exchange in embedded applications. This book will equip you with the technical knowledge required to choose, implement, and troubleshoot protocols such as UART, SPI, I2C, and more.

- **Firmware Developers**

  Firmware developers often interact directly with communication protocols, writing code to initialize, configure, and manage data transmission. This book delves into the nuances of wired communication, helping firmware engineers write more robust and efficient code for protocol implementation.

- **Students and Academics**

  For students and educators in the fields of electronics, computer engineering, or related disciplines, this book serves as an

educational resource for learning and teaching wired communication protocols. It combines theoretical explanations with practical examples to bridge the gap between academic concepts and real-world applications..

- **Makers and Hobbyists**

  If you are a hobbyist or maker working on projects that involve microcontrollers, sensors, or other embedded components, understanding communication protocols can open up new possibilities. This book provides clear, concise explanations and practical examples, making it accessible for enthusiasts with varying levels of experience.

- **Project Managers and System Architects**

  For professionals responsible for overseeing embedded projects, understanding the capabilities and limitations of wired communication protocols is crucial for making informed decisions during system design and planning. This book offers a high-level overview alongside detailed technical discussions, ensuring you have the insights needed to guide your team effectively.

- **Anyone Curious About Embedded Systems**

  If you are simply curious about how embedded systems communicate or want to explore the fascinating world of wired protocols, this book is a great starting point. It’s designed to be approachable, with a gradual progression from basic concepts to advanced topics.

No matter your background or level of expertise, "Wired Embedded Protocols" is structured to guide you through the principles, implementations, and best practices of wired communication in embedded systems. By the end of this book, you will not only

understand the technical details but also gain the confidence to apply this knowledge in your projects and professional work.

Feedback

This book is complete, but as with any technical work, it may still contain errors or sections that could be clarified further. While I have made every effort to ensure accuracy and readability, English is not my native language, and some imperfections may remain.

If you encounter any mistakes or areas that are unclear, or if you have suggestions for improving the content, I would greatly appreciate your feedback.

You can reach me on Linkedin

Support

If you want to help me, you may provide any feedback or review using your favorite social network or blog.

Part I. Introduction

1. Introduction to Communication Protocols

Communication protocols are vital components in the world of embedded systems. They provide standardized methods for data transmission between various devices and systems, which is a key aspect of ensuring their interaction and compatibility. Embedded systems, whether in the automotive industry, medical equipment, or consumer electronics, heavily depend on effective communication to perform their functions. Without reliable communication protocols, these systems would not be able to exchange data effectively, which would significantly limit their functionality and applicability.

Wired communication protocols are a set of standardized rules for data transmission between different devices in embedded systems. These protocols define how data is encoded and transmitted through physical media, ensuring that the transmitted information is correctly interpreted by the receiver. Protocols define key aspects such as data transmission speed, error detection and correction methods, synchronization, and media access control.

- **Importance of Communication Protocols**: In embedded systems, where resources are limited and reliability requirements are high, the proper choice and use of communication protocols are of utmost importance. They enable efficient and reliable data transmission, minimizing errors and delays.
- **Key Functions**: The key functions of communication protocols include establishing connections, transmission synchronization, error handling, data flow control, and encryption. These functions ensure that data is transmitted accurately and securely.
- **Standardization**: Protocol standardization ensures compatibility between various devices and systems, which is critically important for integration and interaction within broader technological ecosystems.

1.1 Classification of Communication Protocols

Communication protocols, essential components in the world of embedded systems, can be classified based on several criteria, including the data transmission medium, access to the transmission medium, synchronization, and the number of data lines.

1.1.1 By Data Transmission Medium

- **Wired Protocols**: Use physical conductors, such as copper cables or fiber-optic cables, to transmit data. Examples include Ethernet, USB, CAN, and RS-232.

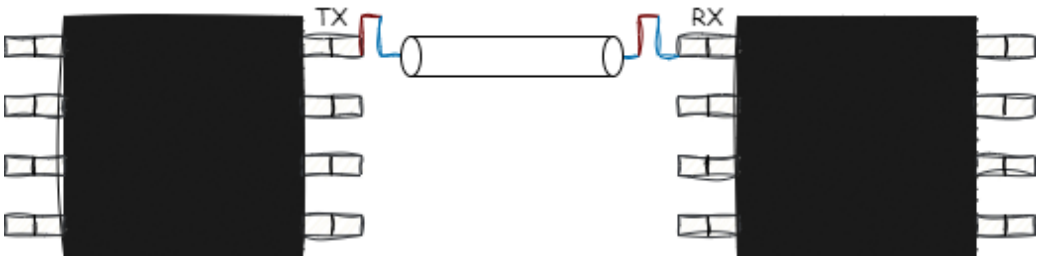

Figure 1.1 - Wired communication protocol

- **Wireless Protocols**: These protocols do not require wires for data transmission. Data can be transmitted using radio waves, infrared, optical, or laser signals. Examples of wireless protocols include Wi-Fi and Bluetooth.

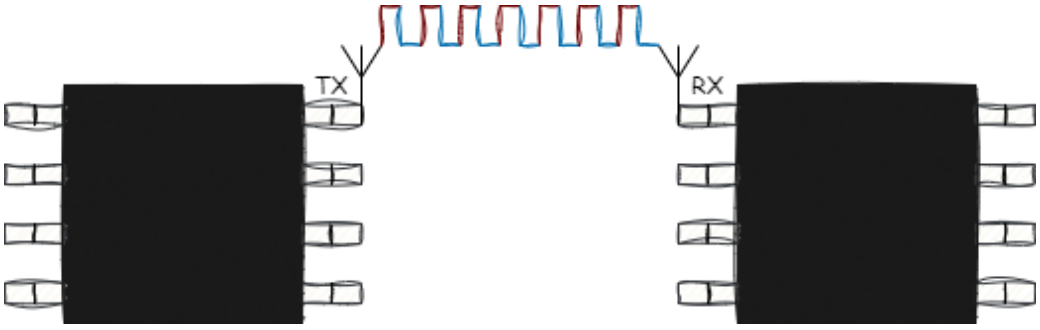

Figure 1.2 - Wireless communication protocol

1.1.2 By Ability for Simultaneous Transmission and Reception

- **Simplex Communication**: Allows data transmission in only one direction—either sending or receiving. A classic example of such communication is radio.

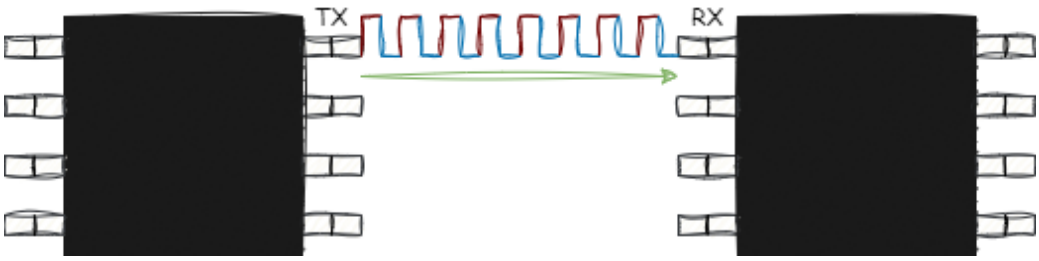

Figure 1.3 - Simplex communication protocol

- **Half-Duplex Communication**: Enables bidirectional data transmission, but not simultaneously. At any given moment, either data transmission (Figure 1.4A) or reception (Figure 1.4B) is possible. For instance, the RS-485 protocol is often used in half-duplex mode. In this mode, RS-485 allows data transfer in both directions but not at the same time. This approach is commonly used in industrial and commercial applications where reliable bidirectional communication is required over long distances, but simultaneous data transfer in both directions is unnecessary.

RS-485 protocol is commonly used in Half-Duplex mode, but can also support Full-Duplex communication when using four-wire configurations.

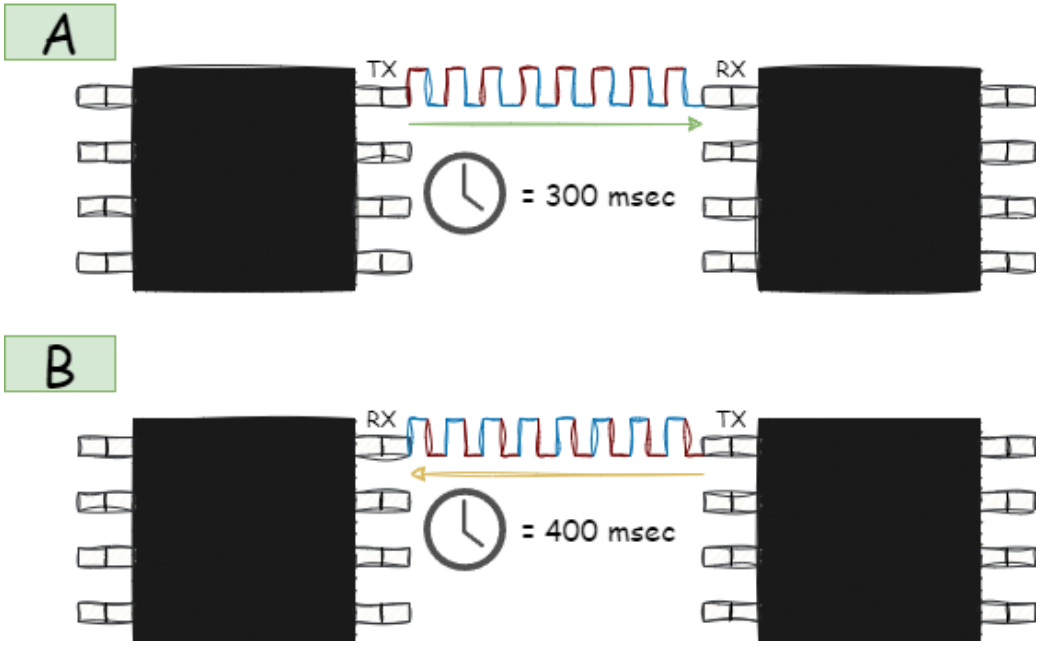

Figure 1.4 - Duplex communication protocol

- **Full-Duplex Communication**: Allows simultaneous bidirectional data transmission, meaning devices can send and receive data at the same time. This mode ensures more efficient communication compared to simplex and half-duplex communication, as it eliminates delays associated with switching between data transmission and reception. An example of full-duplex communication is the UART protocol.

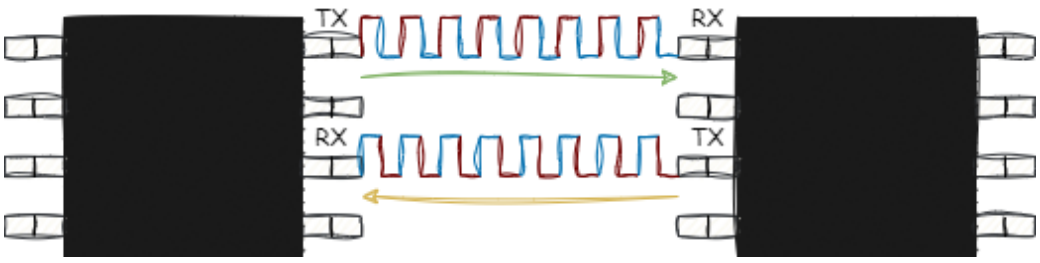

Figure 1.5 - Full-duplex communication protocol

It is important to understand that the classification of communication protocols based on features such as simplex, half-duplex, and full-duplex is often not strictly tied to the protocols themselves but rather to the communication method. For example, any higher-level protocol in this classification can easily operate in a lower-level mode. For instance, UART, traditionally considered a full-duplex protocol capable of bidirectional data transmission, can be configured to operate in simplex mode. In this context, an example could be Debug UART. This mode is often used for outputting debug information during the development and testing of embedded systems. In such a mode, the receive channel (RX) can even be completely disabled, and the system will use UART solely for data transmission (TX), such as for sending logs or error messages.

1.1.3 By Synchronization

- **Synchronous Protocols**: Require synchronization between the sender and receiver. Synchronization determines the moments when the data on the lines is considered valid, and at those moments, the data must be read by all receiving devices and set by all transmitting devices. The sync signal is transmitted over a separate line and represents a rectangular wave of a certain frequency, as shown in Figure 1.6.

Figure 1.6 - Sync communication protocol

The moments when the signal is valid can be determined at the rising edge (Figure 1.6A), falling edge (Figure 1.6B), or both (Figure 1.6C).

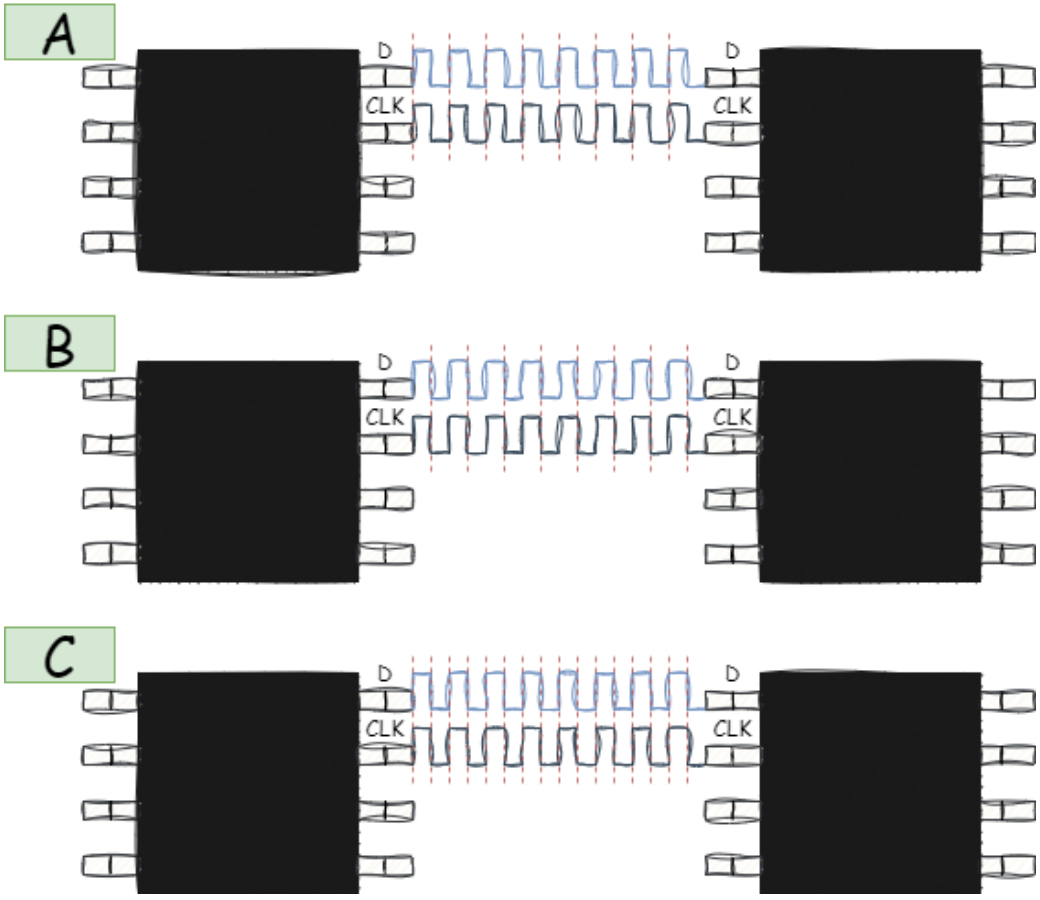

Figure 1.7 - Types of edge synchronizations: A - rising, B - falling, C - both.

- **Asynchronous Protocols**: Do not require strict time synchronization and, accordingly, do not have a separate synchronization line. A classic example of such a protocol is UART. It does not include a synchronization line, and stable data reception and transmission are achieved because all devices involved in the exchange must be pre-configured to the same transmission speed (baud rate). This ensures consistent interpretation of signals.

Figure 1.8 - Async communication protocol

1.1.4 By the Number of Bits Transmitted Simultaneously

- **Serial Protocols**: These protocols transmit only one bit of information per clock cycle. As seen in Figure 1.9, a serial protocol transmits 2 bytes of information in 16 clock cycles. Examples include USB and I2C.

Figure 1.9 - Serial communication protocol

- **Parallel Protocols**: Unlike serial protocols, this class of protocols can transmit more than one bit of information per clock cycle, usually by using multiple data lines, as shown in Figure 1.10. As seen, such a system can transmit 4 bits of information in a single clock cycle using lines D0, D1, D2, and D3.

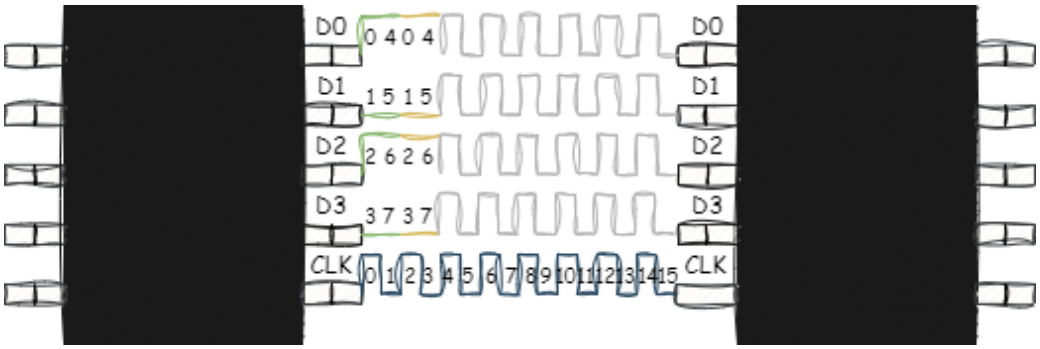

Figure 1.10 - Parallel communication protocol

In this book, we will focus on the analysis of wired protocols. We will examine their key characteristics, operating principles, and application areas, which will help better understand how to choose the appropriate protocol for specific tasks in the world of embedded systems.

1.2 Importance of Communication Protocols

To understand the importance of wired communication protocols, let us consider a very simplified example of a typical embedded device for cargo tracking and monitoring. Such a device is used to track and monitor valuable goods during transportation. It monitors the temperature, vibration, shocks, and GPS coordinates of the cargo, sending this data to a server.

The board for such a device, once again in a very simplified manner, might look approximately as follows:

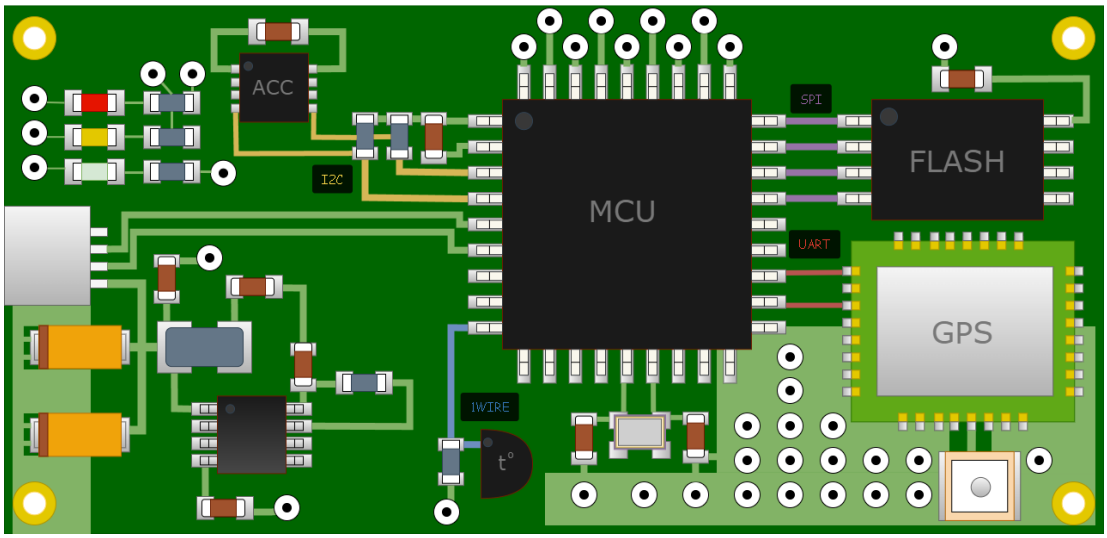

Figure 1.11 - Draft of GPS tracker and monitoring device

This board includes the following components and their corresponding protocols:

- **Temperature Sensor**, connected to the microcontroller via the 1-Wire protocol. This protocol allows data transmission using a single wire, ensuring efficient use of board resources.
- **GPS Module**, connected to the microcontroller via the UART protocol. UART is a widely used protocol for data exchange between devices, providing reliable serial communication.
- **External Flash Memory**, connected to the microcontroller via the SPI protocol. SPI supports high-speed data transfer, making it ideal for handling large volumes of data.
- **MEMS Accelerometer and Gyroscope**, connected to the microcontroller via the I2C protocol. I2C allows multiple low-speed devices to connect to a single bus with minimal wiring.

Each of these protocols has its own characteristics and is designed for different purposes. For example, 1-Wire is ideal for simple sensors with a small amount of data, while SPI and UART are better suited for devices requiring higher data transfer speeds.

Engineers working with embedded systems must understand the specific features of these protocols. Understanding the differences

between protocols, their advantages, and limitations helps developers design systems effectively, optimize their performance, and ensure reliable communication between components.

As we can see, even for a relatively simple device, such as a cargo monitoring system, it is necessary to use various wired protocols to ensure interaction between the microcontroller and external modules and sensors. This highlights how critically important knowledge of wired communication protocols is in the world of embedded systems.

In this book, I will aim to cover all aspects of wired data transmission protocols used in embedded systems as comprehensively as possible.

2. Basics

The main task of any protocol is to transfer data. In the world of digital electronics, data is represented in binary form, that is, as sequences of zeros and ones. These two symbols form the foundation for storing, processing, and transmitting all information in digital systems, with protocols serving as the key tools for exchanging these binary data, providing a structured and reliable method of transferring them between devices.

The concepts of a **logical one** and a **logical zero** play a central role in the architecture of digital electronics, creating the fundamental basis for all data operations. From simple logic gates that form the basic building blocks of electronic circuits to complex microprocessors and digital communication systems, these foundational concepts are employed everywhere.

Logical Zero typically represents the absence of voltage or a lower voltage level in an electronic circuit. In the context of the binary numeral system, a logical zero corresponds to the value "0".

Logical One represents the presence of voltage or a higher voltage level. In the binary numeral system, a logical one corresponds to the value "1".

However, in practice, these abstract concepts require specification. It is essential to clearly define which voltage levels correspond to logical zero and one and how these levels are transmitted in physical circuits. These aspects are fundamental for the design and utilization of any data transmission systems. In this chapter, we will delve into the details necessary for a deep understanding and effective work with wired data transmission protocols.

2.1 Voltage Levels

When we talk about logical one and logical zero, we often use relative terms: high or low level, presence or absence of a signal. However, in

real circuits, these levels are represented by specific voltage values. In this section, we will examine how voltage is used to encode logical levels in various digital circuit technologies.

In typical digital circuits, logical one and logical zero are encoded using different voltage levels:

- Logical one usually corresponds to the full supply voltage.
- Logical zero corresponds to zero voltage or a voltage close to zero.

Ideally, signals in digital circuits would always adopt only these two levels: maximum supply voltage for logical one and zero for logical zero. However, in practice, various factors, such as parasitic effects, voltage drops across transistors, and noise, lead to deviations from these ideal values.

Despite this, digital circuits can correctly interpret signals by using *voltage ranges* within which values are defined as logical one or logical zero. In real circuits, the levels of logical one and zero are not single specific values but rather ranges of voltage values, where a signal falling within these ranges will be interpreted as either logical one or logical zero:

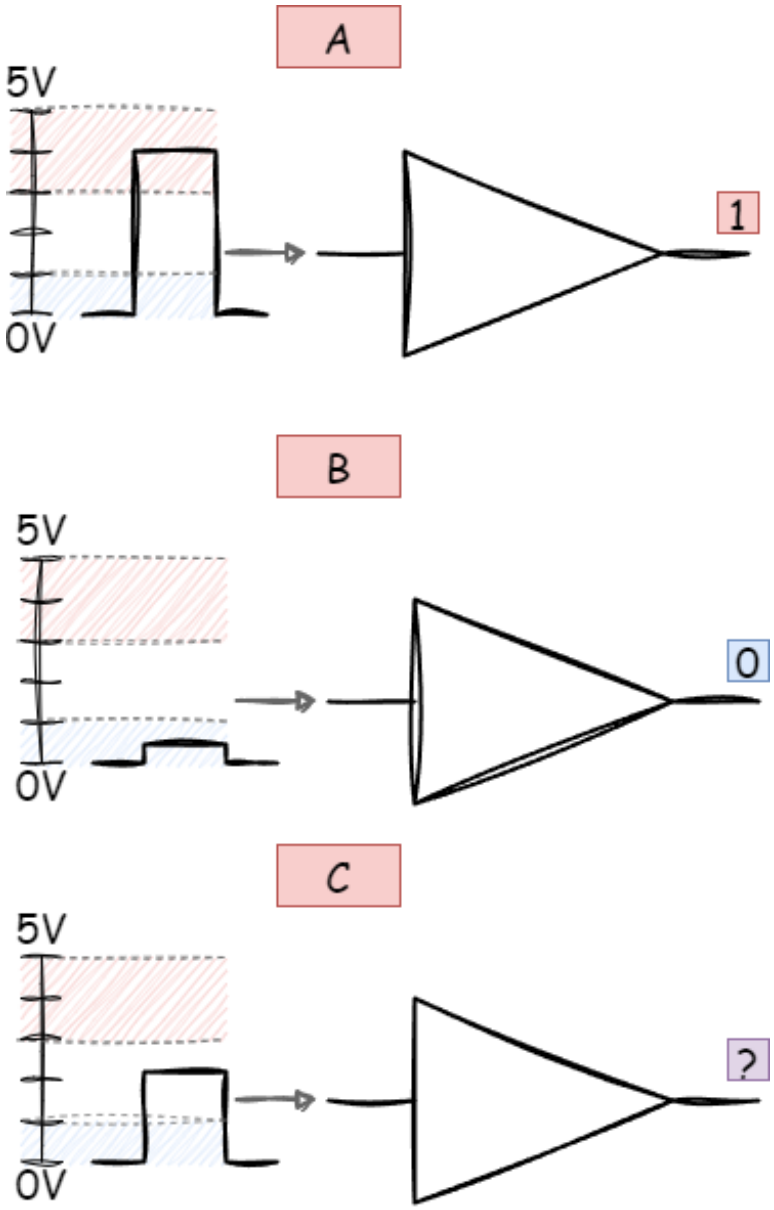

Figure 2.1 - Voltage ranges for interpreting 1 and 0

As you can see, the logical one and zero correspond to specific voltage ranges, marked in red for one and blue for zero. If the signal level falls within these ranges, the chip is guaranteed to correctly interpret the signal (sections A and B). However, there is a gap between these two ranges (section C), where the presence of a signal makes it impossible to interpret.

These voltage ranges are standardized based on the circuit technology used in the design of a particular chip. There are many such technologies: *ECL* (Emitter-Coupled Logic), *RTL* (Resistor-Transistor Logic), *DTL* (Diode-Transistor Logic), and so on. However, I will focus on two of the most well-known: *TTL* (Transistor-Transistor Logic) and *CMOS* (Complementary Metal-Oxide-Semiconductor).

Currently, CMOS is the dominant technology for manufacturing most modern integrated circuits, including processors, memory, and mobile devices.

2.1.1 TTL (Transistor-Transistor Logic)

TTL (Transistor-Transistor Logic) is a technology developed and widely adopted in the 1960s, which became the foundation for many digital devices due to its reliability and design simplicity. As the name suggests, all logic elements in this technology are implemented using bipolar transistors.

Let's look at an example of a two-input *NAND* gate constructed using TTL logic:

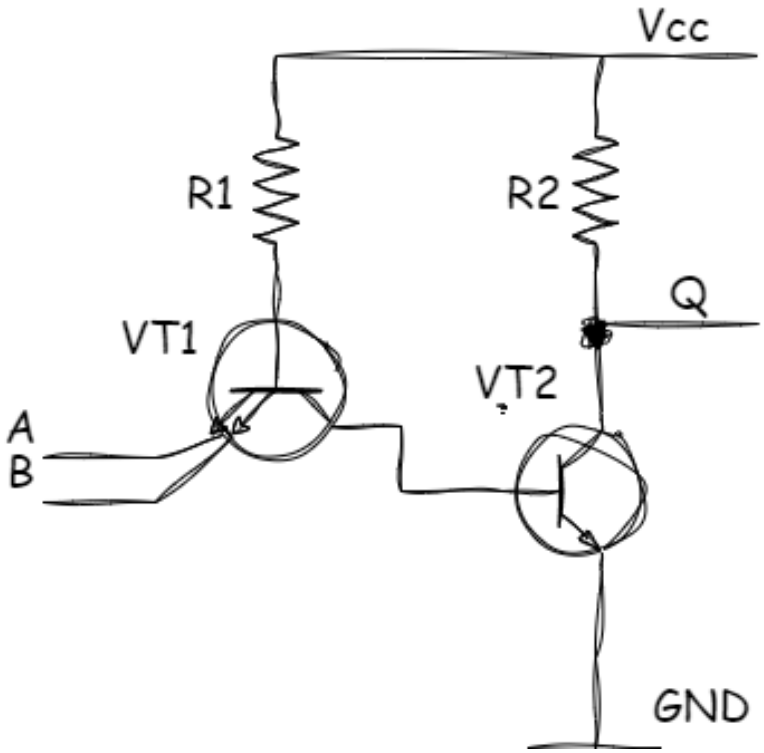

Figure 2.2 - Two-input TTL NAND gate

Components based on TTL logic are designed to operate with a standard supply voltage of 5 volts, with allowable variations within +/- 0.25 volts. Ideally, a signal corresponding to a high logical level would have a voltage exactly at 5 volts, and a signal representing a low logical level would be exactly 0 volts. However, in practice, TTL components can correctly interpret logical levels even when they deviate from these ideal values. Acceptable voltage ranges for input signals vary from 0 to 0.8 volts for a low level and from 2 to 5 volts for a high level. For output signals, manufacturers guarantee that, under certain load conditions, voltages will remain within 0 to 0.5 volts for a low level and 2.7 to 5 volts for a high level:

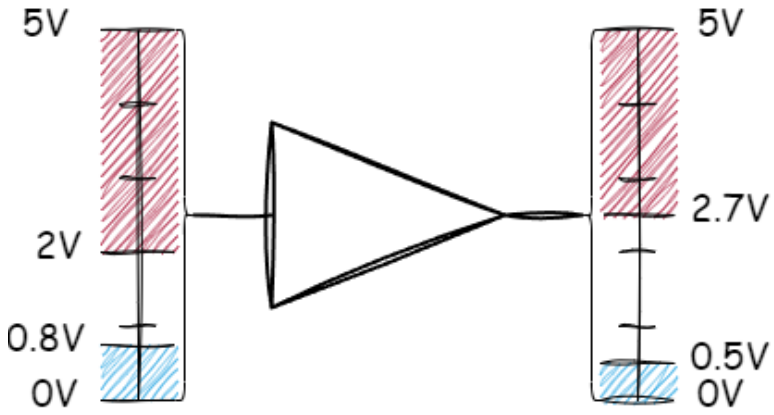

Figure 2.3 - Voltage ranges for TTL logic

When a signal with a voltage between 0.8 and 2 volts is applied to the input of a TTL element, no clear response from the circuit can be expected. Such a signal will be interpreted as undefined, and no manufacturer will provide guarantees as to which logical level it will be assigned.

You may notice that the allowable limits for output signal levels are stricter compared to those for input signals. This ensures that every digital signal transmitted from the output of one TTL element to the input of another corresponds to a voltage acceptable for the latter. This difference in tolerances for input and output signals is referred to as the *noise margin*. In TTL logic, the noise margin for the low logical level is the difference between 0.8 V and 0.5 V (i.e., 0.3 V), and for the high logical level, it is the difference between 2.7 V and 2 V (i.e., 0.7 V). In other words, the noise margin determines the maximum allowable noise or interference that can be superimposed on the output signal of a logic circuit before the receiving circuit begins to misinterpret it.

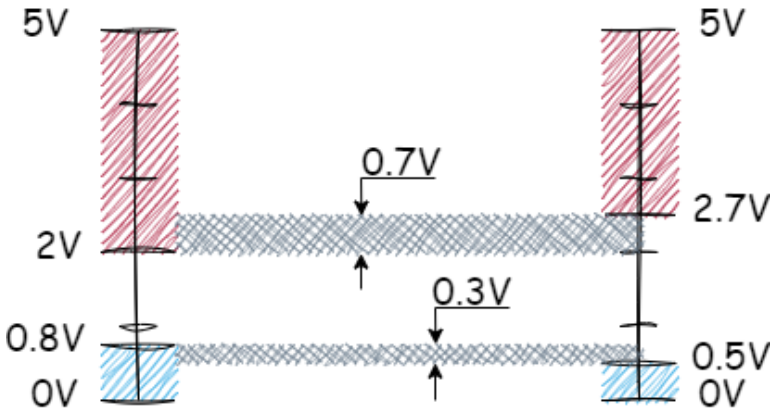

Figure 2.4 - Noise margin for TTL logic

If the allowable levels for input and output signals in TTL logic were the same, the system's noise immunity would be significantly reduced. Any slight change in the output signal caused by external noise or interference could lead to that signal being misinterpreted at the input of the next element. In such a situation, the system would become extremely vulnerable to external influences, potentially causing a chain reaction of errors in the logic circuits, rendering reliable system operation nearly impossible.

The noise margin creates a buffer between the levels at which signals are considered valid, ensuring that signals subjected to minor changes due to noise are still correctly interpreted by the receiver. This is critically important for ensuring the reliability and stability of digital devices, especially in environments with high levels of electromagnetic interference.

2.1.2 CMOS (Complementary Metal-Oxide-Semiconductor)

The CMOS technology, widely adopted since the 1970s, uses pairs of complementary field-effect transistors (MOSFETs), including *n*-channel and *p*-channel transistors, to create efficient logic circuits. This technology achieves high energy efficiency because significant power consumption occurs only during state transitions of the circuit, while in a static state, power consumption is extremely low. This characteristic

makes CMOS circuits ideal for portable devices where battery life is a critical concern.

Additionally, CMOS technology offers high noise immunity and the ability to operate within a wide range of supply voltages, extending its applications not only in portable electronics but also in various areas of digital technology, including computers, mobile phones, and consumer electronics. These features, combined with the capability to integrate a large number of transistors on a relatively small silicon substrate area, have made CMOS technology the dominant choice for manufacturing microprocessors, memory, and other integrated circuits.

The same two-input *NAND* gate implemented using CMOS logic would look as follows:

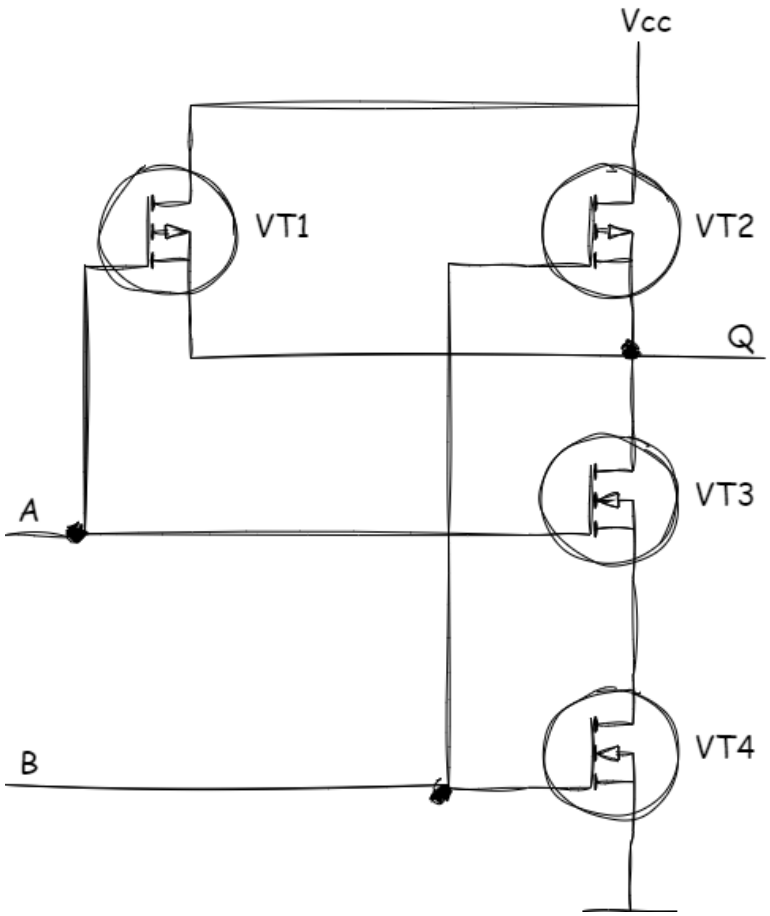

Figure 2.5 - Two-input CMOS NAND gate

Voltage levels for CMOS technology depend on the supply voltage, enabling these circuits to operate across a wide voltage range. This is one of the key advantages of CMOS, as it provides greater flexibility in designing electronic devices and systems. A typical range for CMOS supply voltages can vary from 1.8 V to 15 V.

Let us consider the voltage levels for CMOS logic with a supply voltage of 5V.

In the context of CMOS elements with a supply voltage of 5 volts, acceptable voltage levels for input signals range from 0 to 1.5 volts for a low logical level and from 3.5 to 5 volts for a high logical level. For output signals, manufacturers guarantee voltage levels between 0 and 0.05 volts for a logical zero and between 4.95 and 5 volts for a logical one:

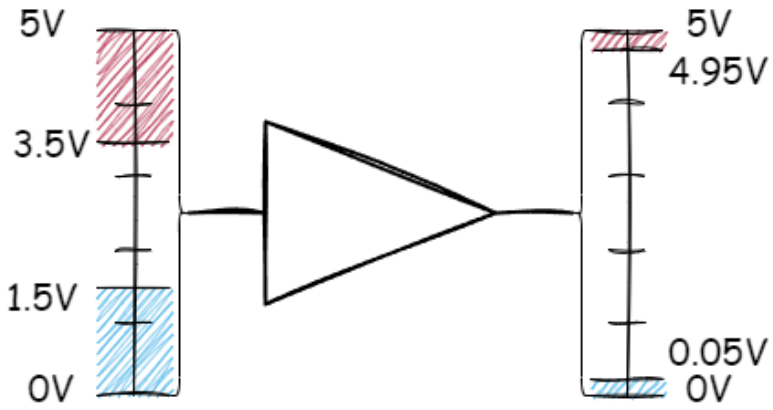

Figure 2.6 - Voltage ranges for CMOS logic

These voltage values illustrate that CMOS logic elements have significantly greater noise margins compared to TTL-based elements. The noise margin for CMOS is 1.45 volts for both logical zero and logical one, whereas for TTL, this margin reaches a maximum of 0.7 volts:

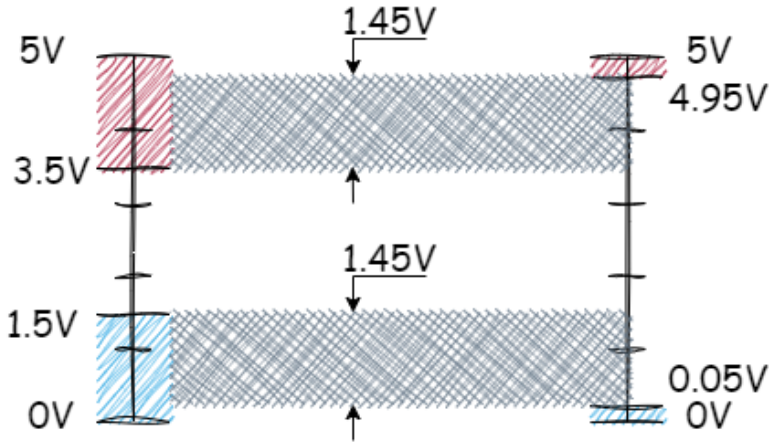

Figure 2.7 - Noise margin for CMOS logic

This means that CMOS circuits can tolerate more than double the amount of noise imposed on input signals without misinterpreting them as logical zeros or ones.

2.2 Pull-Up and Pull-Down Resistors

Pull-Up and Pull-Down resistors are fundamental components in embedded system design, used to ensure a stable logical level on the input or output of a microcontroller or other digital logic when the input/output may be inactive or floating.

A Pull-Up resistor is connected between the corresponding input/output and the power supply voltage (VCC) and Pull-Down is connected between the corresponding input/output and ground (GND). Its primary function is to "pull" the voltage level on the input/output to a high or low logical level when other active elements in the circuit are not driving that I/O.

What does it mean to "pull"? It means that in the idle state of the line—when no device is transmitting on it—a specific signal level will be established: either logical one or logical zero. This is a crucial concept because such pulling ensures that we can always reliably determine the current signal level on the communication line.

Pull-Up/Down resistors serve the following purposes:

1. **Eliminating Floating States**: Inputs on microcontrollers and other digital devices can "float" if they are not connected to a defined logical level. A "floating" input can randomly be interpreted as high or low due to electrical noise, leading to unpredictable behavior. Pull-Up/Down resistors ensure a stable logical state.

2. **Ensuring Correct Control Logic**: In certain circuits, for example, when a button or switch is used to close a circuit to ground, a Pull-Up/Down resistor ensures that the input remains high in the absence of activation and transitions to a low level only when the button is pressed.

3. **Reducing Noise**: Maintaining an input or output in a defined state reduces susceptibility to electrical noise.

2.2.1 Pull-Up Resistors

Pull-Up resistors are connected between the signal line and the positive power supply:

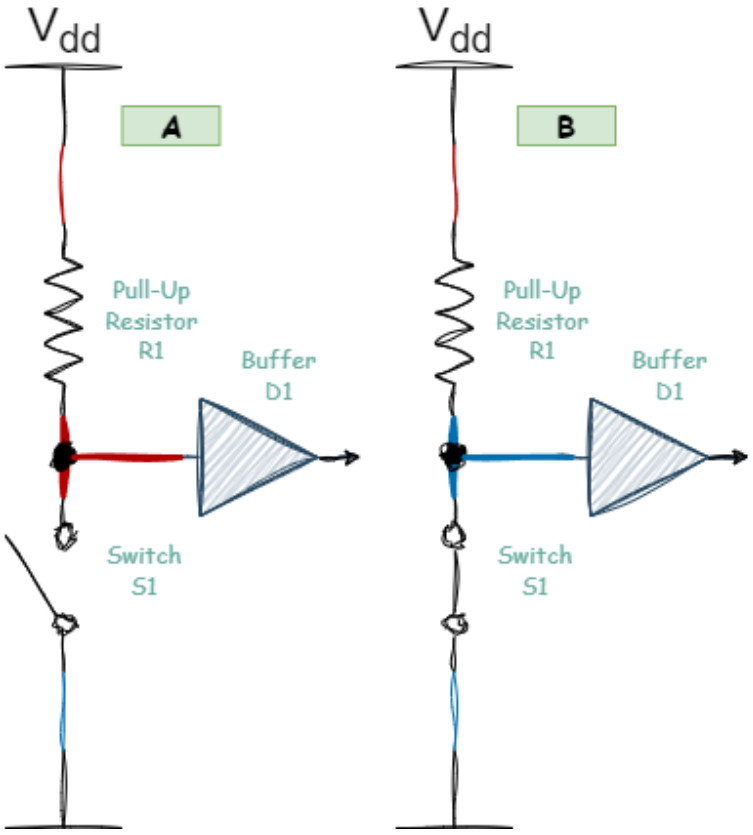

Figure 2.8 - Pull-Up Resistor

When the switch S1 is open (section A of Figure 2.8), the entire line is connected to the power supply through the Pull-Up resistor R1, and a high logical level is established on the line. If the switch S1 is closed (section B of Figure 2.8), it creates a direct connection to ground, and the logical level on the line changes to low.

Thus, this Pull-Up resistor ensures a high logical level on the line in an idle state. This type of resistor is crucial for our topic, as it is used in many protocols to pull signal lines up to the supply voltage. Without this resistor, the operation of these protocols cannot be organized. Examples of protocols that require the presence of a Pull-Up resistor include 1-Wire and I2C.

2.2.2 Pull-Down Resistors

Pull-Down resistors are connected between the signal line and ground:

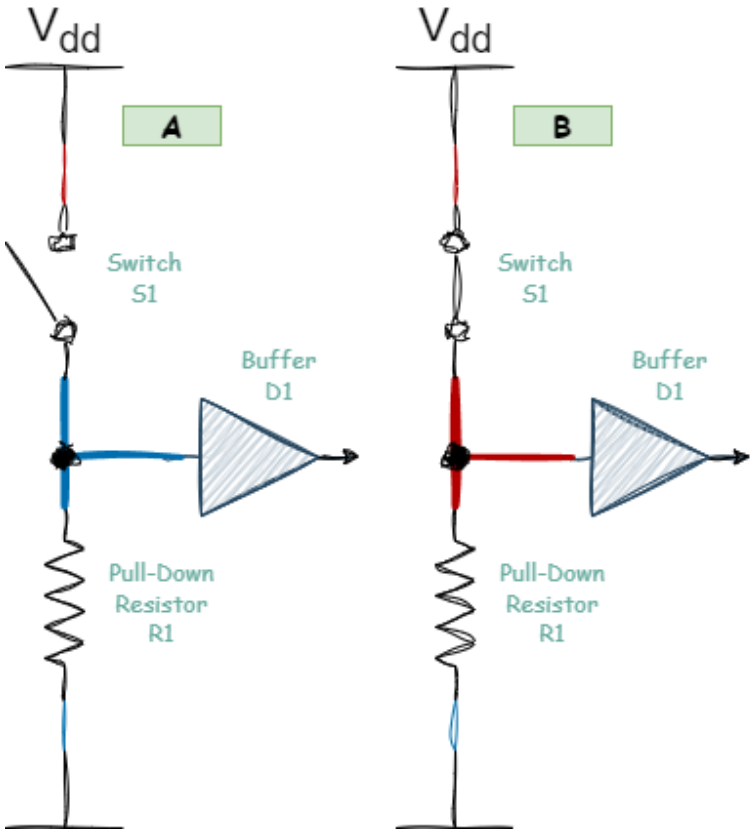

Figure 2.9 - Pull-Down Resistor

When the switch S1 is open (section A of Figure 2.9), the entire line is connected to ground through the Pull-Down resistor R1, and a low logical level is established on the line. If the switch S1 is closed (section B of Figure 2.9), it creates a direct connection to the power supply line, and the logical level on the line changes to high.

This type of resistor is not used in the protocols discussed in this book. However, it is frequently applied to ensure a stable logical zero level on the microcontroller pins. It is often used for all unused pins of a microcontroller.

2.2.3 Selecting the Resistor Value

The value of Pull-Up/Down resistors depends on the specific application. *They must be large enough not to impose a significant load on the*

power supply circuit, yet small enough to ensure rapid changes in logical levels.

- **Too low a resistance** increases current consumption, which is especially critical for battery-powered devices.
- **Too high a resistance** can slow down signal transitions, reduce the signal-to-noise ratio, and increase susceptibility to interference.

Let's take a closer look at the factors that influence the selection of the resistor value.

2.2.3.1 The Impact of Power Consumption on Resistor Selection

First, let's examine the phrase: *not to impose a significant load on the power supply circuit*. A current flows through the pull-up resistor, as it does through any circuit element, and this current is drawn from the power source connected to the device. According to Ohm's Law, the value of the pull-up resistor directly affects the current flowing through it, and consequently, the total current consumption of the entire circuit.

Ohm's Law states that the current (I) through a resistor is proportional to the voltage (V) across it and inversely proportional to its resistance (R):

$$I = \frac{V}{R} (2.1)$$

Let's compare the current consumption for three cases of pull-up resistor values: 1 kΩ, 4.7 kΩ, and 10 kΩ, with a supply voltage of 5V:

1. For a 1 kΩ resistor, the current value will be:

$$I = \frac{V}{R} = \frac{5}{1000(\Omega)} = 5mA$$

2. For a 4.7 kΩ resistor, the current value will be:

$$I = \frac{V}{R} = \frac{5}{4700(\Omega)} = 1mA$$

3. For a 10 kΩ resistor, the current value will be:

$$I = \frac{V}{R} = \frac{5}{10000(\Omega)} = 500\left(\mu A\right)$$

These values illustrate how current changes depending on the resistor's resistance. The difference between the extreme values is up to 10 times, which is critical for low-power devices.

A higher resistance value for the pull-up resistor can help save power. For example, in low-speed protocols, a larger pull-up resistor value can be chosen, thereby positively affecting the overall power consumption of the circuit.

Increasing the resistance of the resistor reduces the current flowing through the circuit, which decreases power consumption—an essential factor for battery-powered devices. However, by reducing the current through higher resistance, you also weaken the signal, and a weak signal:

- Increases the possibility of errors during data transmission due to the reduced signal amplitude relative to noise levels.
- Makes the line more susceptible to electromagnetic interference and parasitic capacitance.

Pull-up resistors are often chosen within the range of $1k\Omega - 10k\Omega$. This range provides a reasonable trade-off between power consumption, switching speed, and data transmission reliability.

2.2.3.2 The Effect of Signal Speed on Resistor Value Selection

Now let’s take a closer look at the phrase: *ensure rapid changes in logical levels*.

The fact is that the signal line and ground create what is known as a *parasitic capacitor* — a data transmission line inherently has capacitance, even if no physical capacitor is connected. This capacitance arises due to the parasitic capacitance of PCB traces, cables (in the case of external devices), IC pins, and other components of electrical circuits, all of which inherently possess some level of capacitance. These capacitances occur both between signal conductors themselves and between power lines, signal conductors, and everything else on the board.

When you add the resistance of the pull-up resistor to this capacitance, an RC circuit is formed:

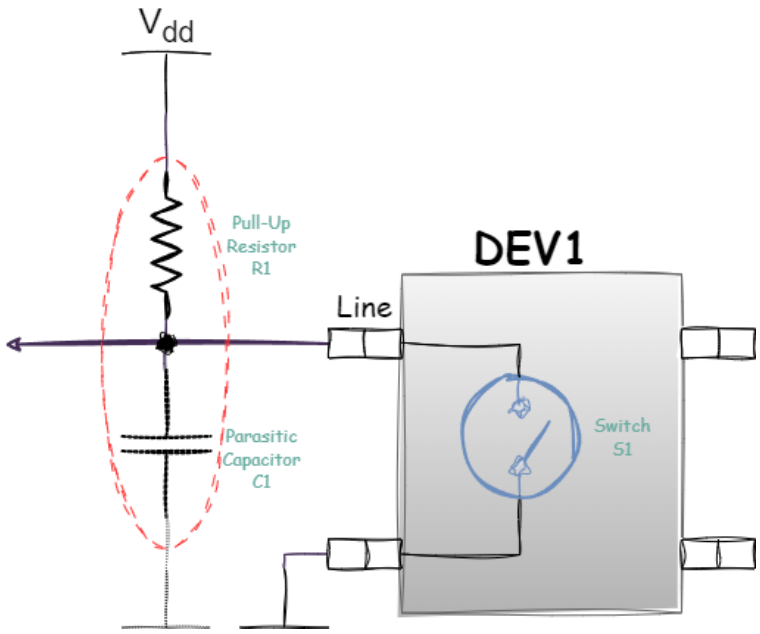

Figure 2.10 - RC circuit formed by a pull-up resistor and parasitic capacitance

The main characteristic of any RC circuit that affects the speed of signal changes is the *time constant of the circuit (RC constant)*.

The RC time constant of a circuit is defined by the following formula:

$$\tau = R \cdot C (2.2)$$

where:

- τ — the time constant of the circuit,
- R — resistance in ohms (Ω),
- C — capacitance in farads (F).

It is measured in seconds and represents the time required for the signal to reach approximately 63.2% of the full charge of capacitor in response to a step change in voltage across resistor from logical zero to logical one:

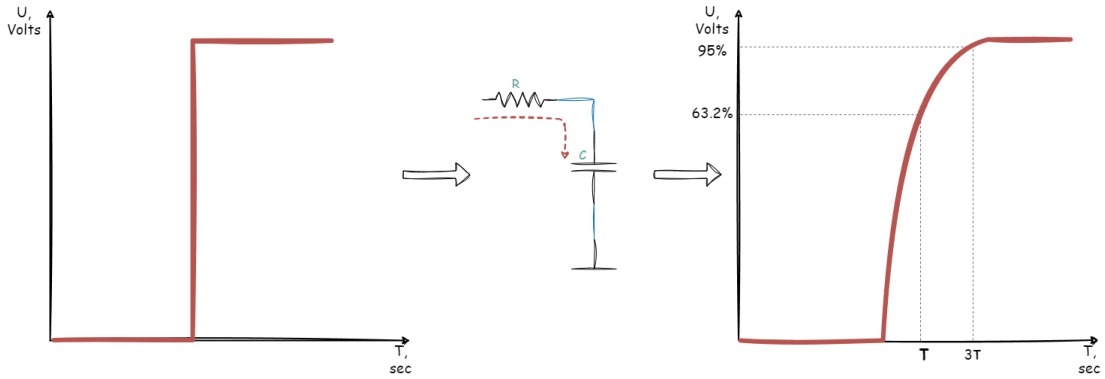

Figure 2.11 - The effect of the RC constant on capacitor charging

Or for the capacitor to discharge to 36.8% of its initial value in response to a step change in voltage across resistor from logical one to logical zero:

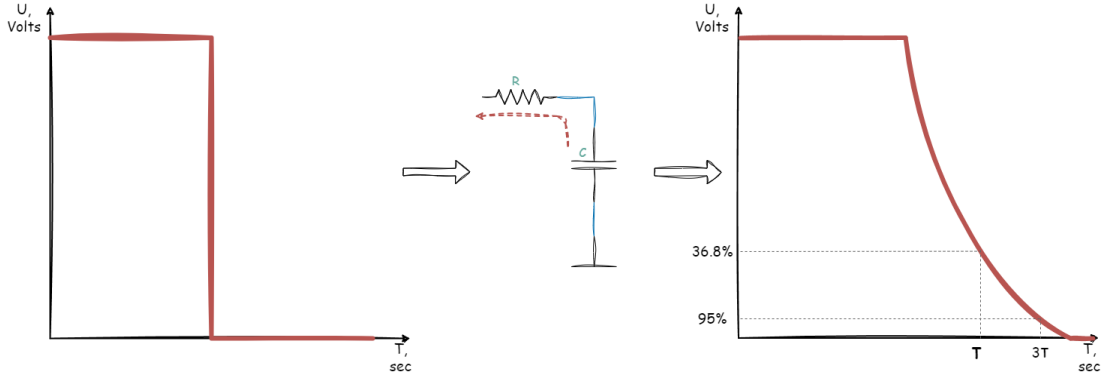

Figure 2.12 - The effect of the RC constant on capacitor discharging

The formula $\tau = R \cdot C$ is used for simplified circuit analysis. **The time constant (τ)** represents the time required for the voltage across the capacitor to reach approximately **63% of the full charge level** (V_{PULLUP}), and this formula is used as an approximation for rough calculations.

However, the charging process is described by the equation:

$$V(t) = V_{\mathrm{PULLUP}} \cdot (1 - e^{-\frac{t}{R \cdot C}}) (2.3)$$

To determine the time required for the voltage to reach a specific level $V(t)$ (different from 63% or 95%), this equation can be rearranged as:

$$t = -R \cdot C \cdot \ln\left(1 - \frac{V(t)}{V_{\mathrm{PULLUP}}}\right) (2.4)$$

This is a more precise expression that describes the **capacitor charging process**, based on the exponential relationship of voltage over time.

However, for our purposes—understanding the reason for delayed signal edges—the formula

$$\tau = R \cdot C$$

is sufficient.

From formula 2.2, it follows that the higher the resistor's resistance, the larger the value of τ, and thus the longer it takes for the signal to transition from one logical level to another. Consequently, the slower the protocol that can operate on this line.

Let's again compare the values of τ for the same three pull-up resistor values: 1 kΩ, 4.7 kΩ, and 10 kΩ, assuming a parasitic capacitance of 10 nF:

- For a 1 kΩ resistor:

$$\tau = R \cdot C = 1\ k\Omega \cdot 10\ nF = 0.00001\ s = 10\ \mu s (2.5)$$

- For a 4.7 kΩ resistor:

$$\tau = R \cdot C = 4.7\ k\Omega \cdot 10\ nF = 0.000047\ s = 47\ \mu s (2.6)$$

- For a 10 kΩ resistor:

$$\tau = R \cdot C = 10\ k\Omega \cdot 10\ nF = 0.0001\ s = 100\ \mu s (2.7)$$

This parameter significantly affects the shape of the signal we actually observe on the line. Let's explore how this happens in more detail.

Let's take our circuit with a pull-up resistor and a switch, but this time connect an oscilloscope to it:

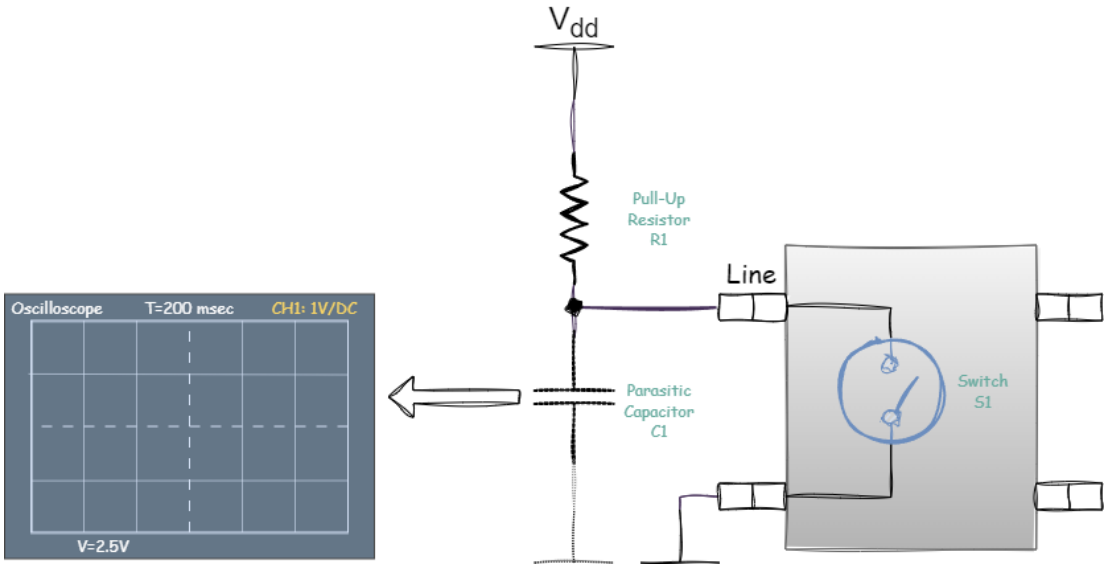

Figure 2.13 - Initial circuit for analyzing the effect of the RC constant on signal shape

We begin toggling the switch S1 at a fixed frequency. On the oscilloscope screen, we expect to see the following signal:

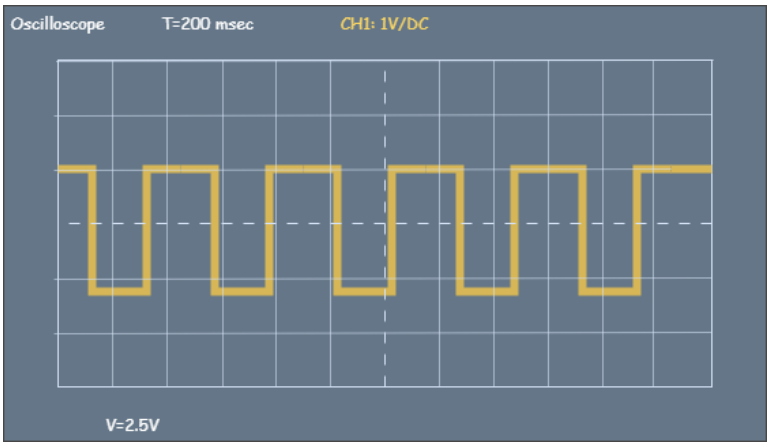

Figure 2.14 - Ideal signal shape

However, in reality, we might observe something like this:

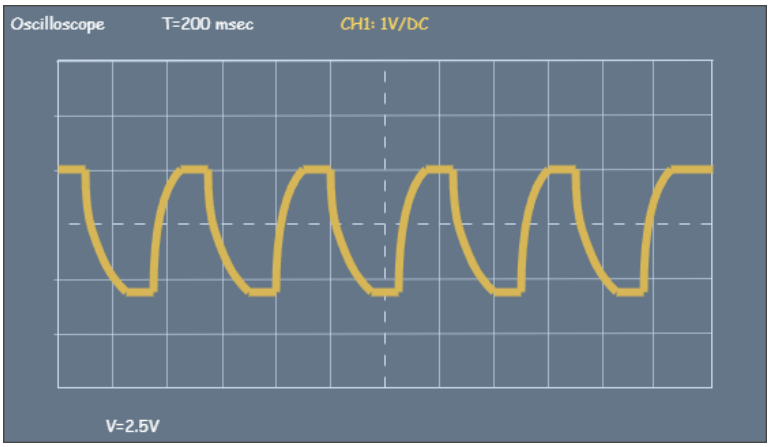

Figure 2.15 - Actual signal shape

This signal distortion occurs, as you might have guessed, due to the presence of parasitic capacitance and the formation of an RC circuit.

Let's examine this in detail using a single signal period as an example.

When the switch S1 is open, parasitic capacitance starts charging through the pull-up resistor from the power line. The charging time depends on the value of the RC constant. On the oscilloscope screen, this will appear as a distorted signal:

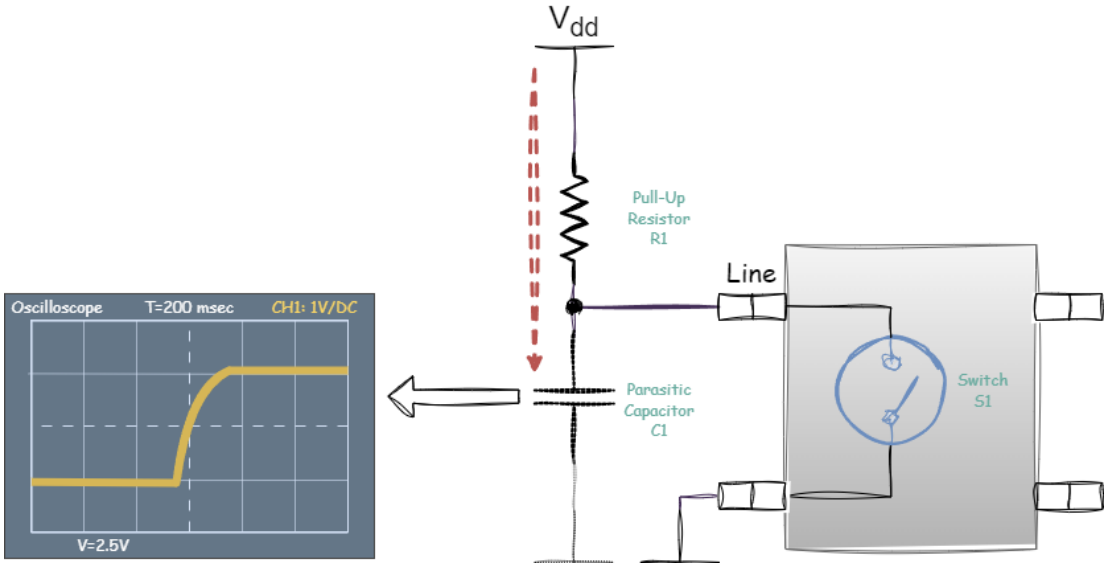

Figure 2.16 - Signal shape during the charging of parasitic capacitance

Now let's close the switch S1. The parasitic capacitor starts discharging through the switch S1. Since it discharges directly through the switch and not through the resistor, the discharge process is faster than in a classic RC circuit, as shown in Figure 2.12. This is why the falling edge appears smooth. When the capacitor is fully discharged, the line reaches a low logical level:

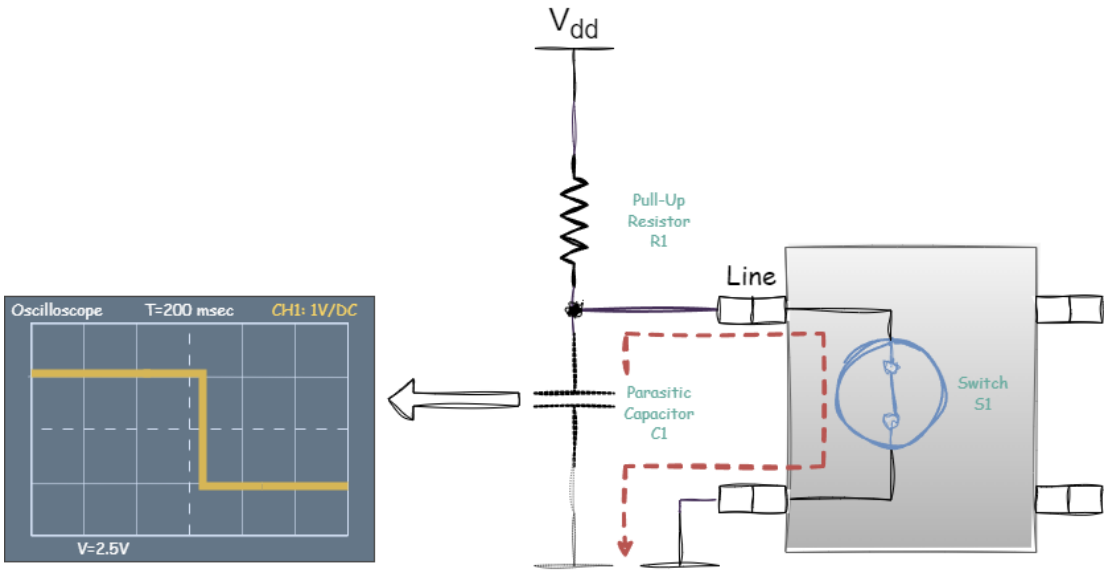

Figure 2.17 - Signal shape during the discharge of parasitic capacitance

This process repeats for every signal period, resulting in the following signal on the oscilloscope:

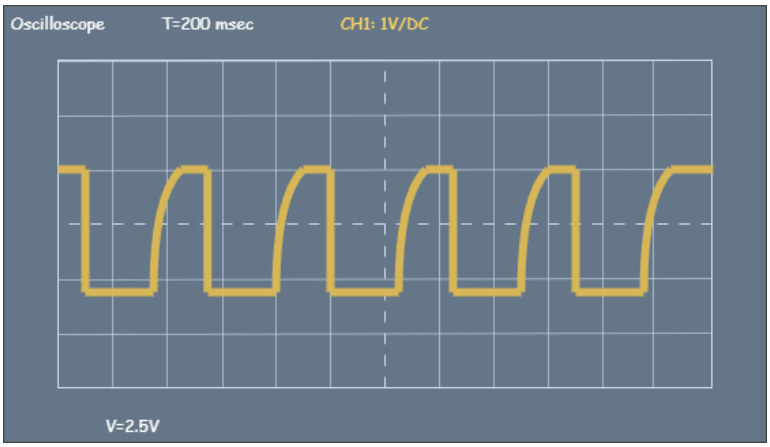

Figure 2.18 - Actual signal shape during capacitor charging/discharging

The greater the total parasitic capacitance, the more pronounced the distortion of the signal edges becomes. Since parasitic capacitance is often beyond our control, we must manage these distortions by adjusting the resistance of the pull-up resistor.

The main issue here is that parasitic capacitance and the RC circuit delay the signal edges, which severely limits the maximum frequency at which signals can be transmitted on the line. If this delay is too long, the signal may fail to reach the required voltage level for the IC to unambiguously interpret the input signal as a logical one. Let’s illustrate this using an example of an IC with CMOS voltage levels.

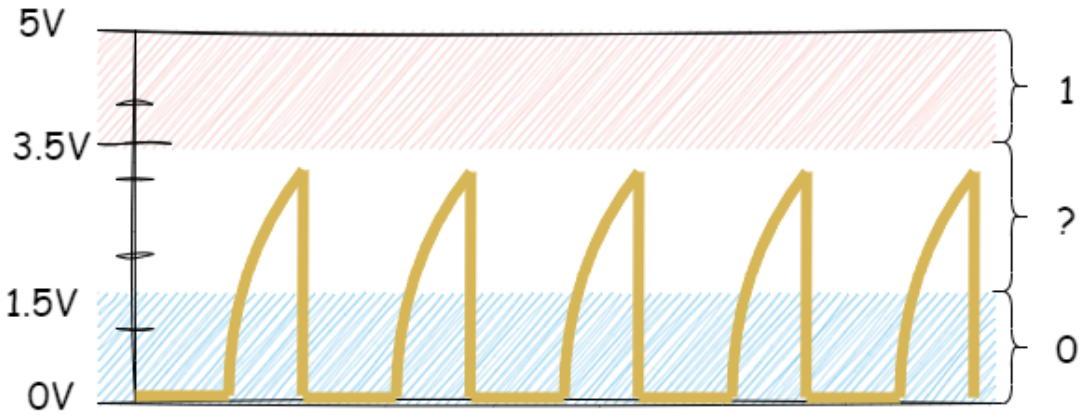

Figure 2.19 - Signal fails to reach the logical high level due to high parasitic capacitance

Due to high parasitic capacitance, the signal fails to reach the voltage level of 3.5V—the lower threshold for recognizing a logical high. One possible solution to this problem is to reduce the value of the RC constant by decreasing the resistance of the pull-up resistor.

The capacitance of the signal line is such an important parameter that it must be considered when implementing any protocol. In fact, almost all standards impose a specific limit on this value.

In summary, selecting the optimal resistor value requires balancing the desired signal switching speed and minimizing power consumption. For critical applications where switching speed is a priority, lower-resistance resistors are preferred. In applications where energy efficiency is more important, higher-resistance resistors can be used, provided that this does not compromise the functionality of the circuit.

2.3 Types of Output Stages

Modern microcontrollers, as well as other ICs, offer the ability to configure their pins to operate in one of two output modes: *Push-Pull* or *Open-Drain*. This configurability provides engineers with the flexibility to select the optimal mode for each specific case, considering factors such

as electrical characteristics, data exchange speed, and power consumption.

To ensure successful integration with various communication protocols connected to the microcontroller, it is essential to understand the features and operating principles of these modes.

2.3.1 Push-Pull Output Stage

The Push-Pull output stage consists of a pair of *complementary transistors* that work in harmony to **actively** control the voltage level on the output for both high and low states. One transistor is connected to the positive power supply, "pushing" the output signal to the high state, while the other transistor is connected to ground, "pulling" the output to the low state:

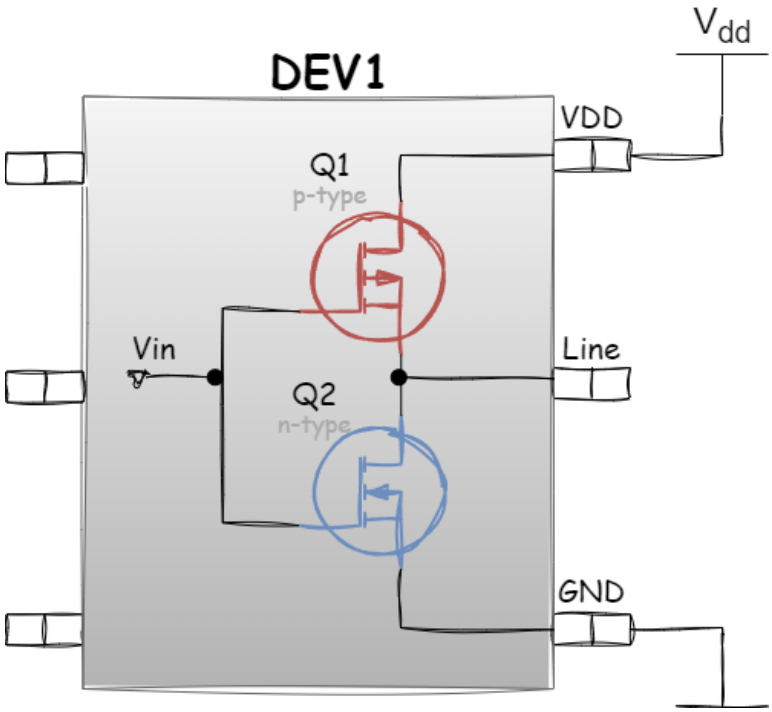

Figure 2.20 - Push-Pull Output Stage

The term **active** control, in the context of electronic circuits such as Push-Pull output stages, refers to the circuit's ability to dynamically and directly manage the voltage level at its output without relying on additional external components like pull-up or pull-down resistors. This means that the Push-Pull output stage itself dictates the line's voltage level, rather than any external components.

Complementary refers to transistors with opposite conductivity types. If one transistor in the pair is *n*-type, the other must be *p*-type. These transistors are controlled in opposition to each other: when one transistor is activated, it "pulls" the output to the power supply voltage level (Vcc), creating a high logical level. Simultaneously, the other transistor remains off, preventing current flow. To create a low logical level, the second transistor is activated, "pulling" the output to ground (GND), while the first transistor is turned off.

The logic behind the Push-Pull output stage is straightforward. If the input value is high, the *P*-channel transistor Q1 is off (does not conduct current), while the *N*-channel transistor Q2 is on (conducts current) — resulting in the output value being low valtage, thanks to a low-impedance connection to ground:

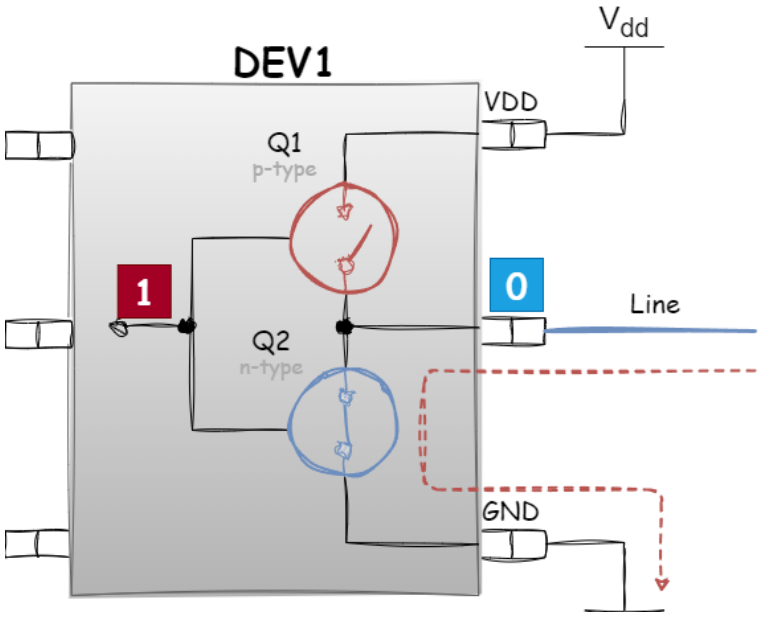

Figure 2.21 - Logical zero formation at the Push-Pull output stage

If the input value is low, the *P*-channel transistor Q1 is on (conducts current), while the *N*-channel transistor Q2 is off (does not conduct current) — resulting in the output value being *high voltage:

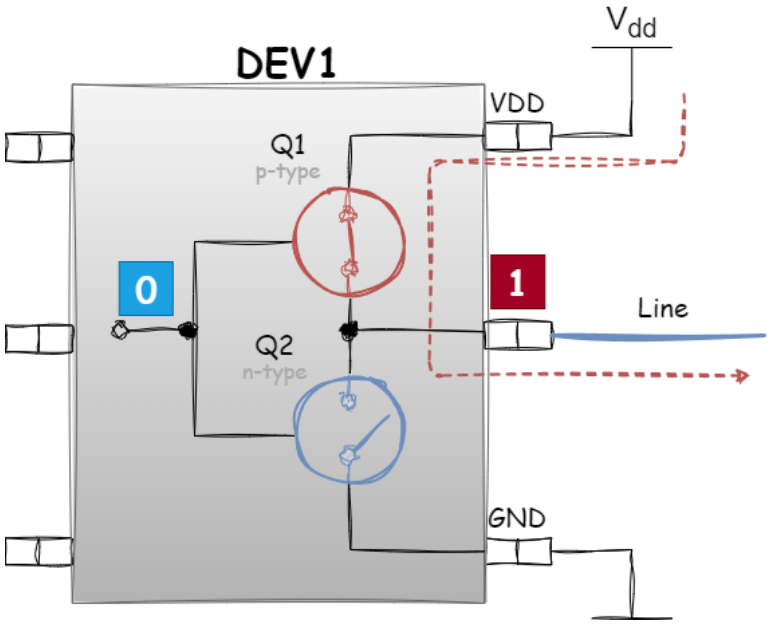

Figure 2.22 - Logical one formation at the Push-Pull output stage

Advantages:

1. **Fast Switching**: Due to active control of both levels, Push-Pull stages provide higher switching speeds compared to open-drain/collector configurations. Remember the discussion on parasitic capacitance? The speed at which any capacitance charges or discharges depends directly on the current supplied to it. In a Push-Pull design, active control of both voltage levels allows sufficient current to rapidly alter the charge of parasitic capacitance. This, in turn, increases the signal line's operating speed.

2. **Lower Power Consumption**: Since no external pull-up resistor is required to pull the signal to a high level, Push-Pull stages can be more energy-efficient in certain applications. The absence of a pull-up resistor eliminates quiescent current through the resistor, reducing overall power consumption.

Disadvantages:

1. **Risk of Short Circuit**: If, due to a design error or failure, both transistors are turned on simultaneously, an excessive current may flow

through them, potentially leading to rapid component failure and circuit damage. In a Push-Pull design, each device on the bus can actively drive the line, supplying both high and low levels. If two devices attempt to drive opposite signal levels simultaneously, this can result in a short circuit on the line.

Consider the two scenarios:

- **Short Circuit within the Push-Pull Stage**: This occurs if both transistors are turned on (conducting current) simultaneously. This can happen during a transition in gate control voltage when one transistor has not fully turned off and still conducts current, while the other has already turned on. This effectively shorts the power supply to ground, causing a short circuit:

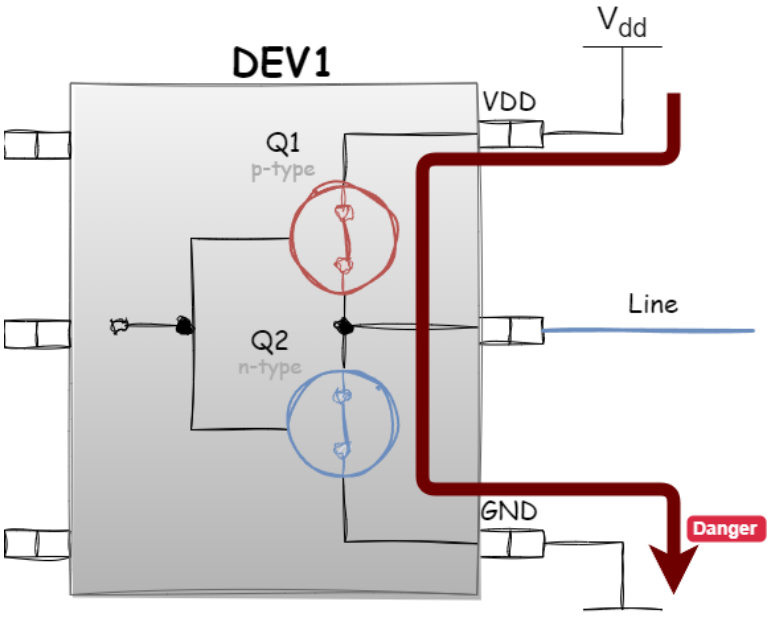

Figure 2.23 - Short circuit within the Push-Pull stage

- **Short Circuit across Multiple Push-Pull Stages**: This occurs if two devices are on the same line, and one device attempts to drive a logical one while the other tries to drive a logical zero simultaneously. Again, this results in the power supply being shorted to ground:

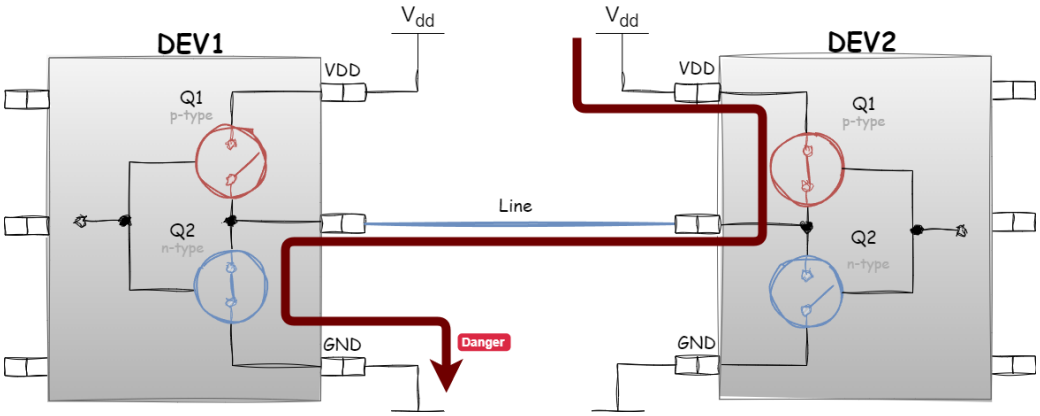

Figure 2.24 - Short circuit between multiple Push-Pull stages

2. **Complex Control**: Proper operation of a Push-Pull stage requires precise synchronization of control signals for both transistors to avoid simultaneous activation. This necessitates more sophisticated control mechanisms, complicating driver design.

2.3.2 Open-Drain Output Stage

The concept of an Open-Drain plays a key role in many digital circuits and embedded system protocols. This term refers to the way transistors are connected and operate in ICs, where the transistor's output is not directly connected to the power supply (it remains "open").

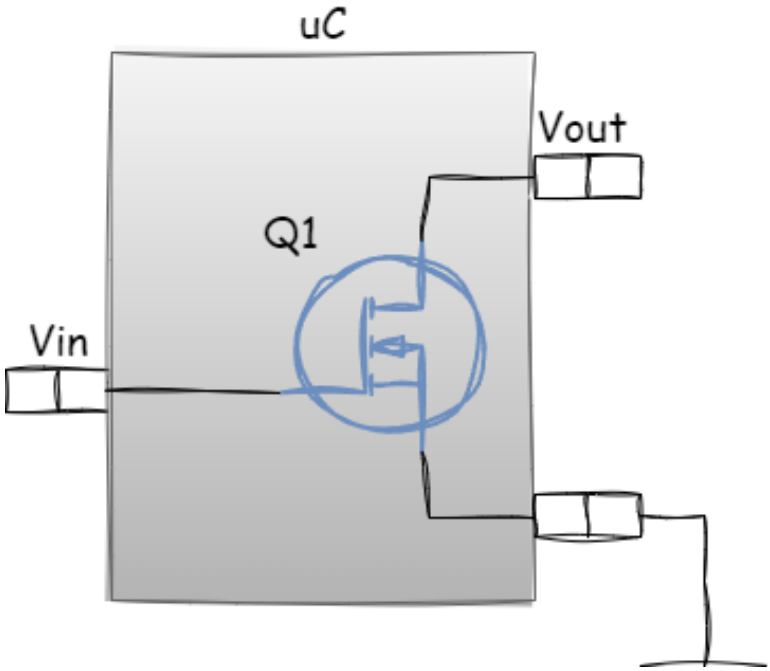

Figure 2.25 - Open-Drain Output Stage

For this circuit to operate, an external resistor is required to connect the transistor's output to the supply voltage:

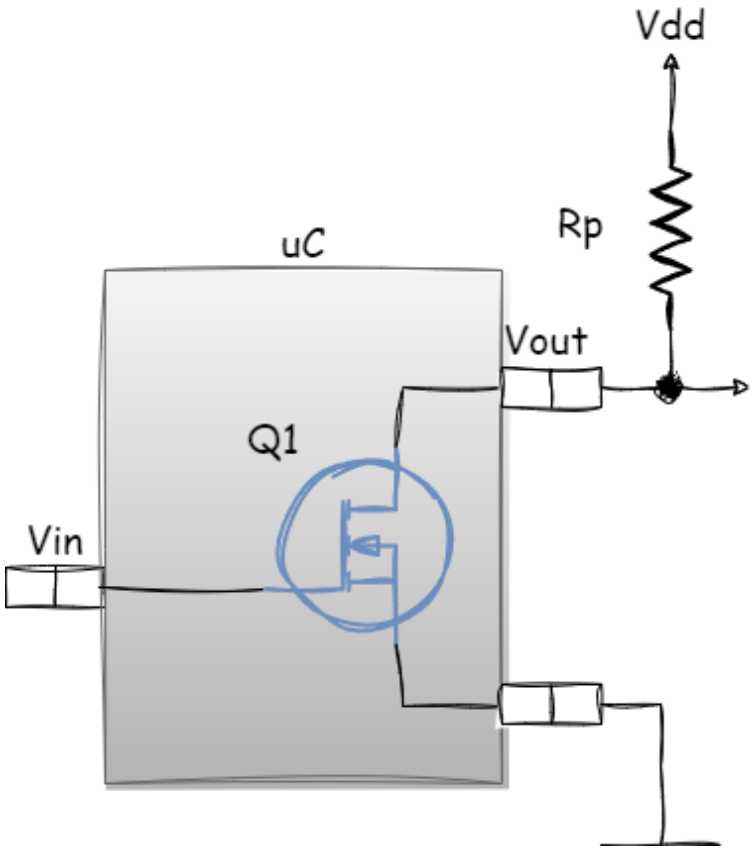

Figure 2.26 - Open-Drain Output Stage with an external resistor

When the transistor is off (does not conduct current), the output signal is formed by the pull-up resistor. This is fundamentally different from the Push-Pull configuration, where the signal is actively controlled entirely by the transistors:

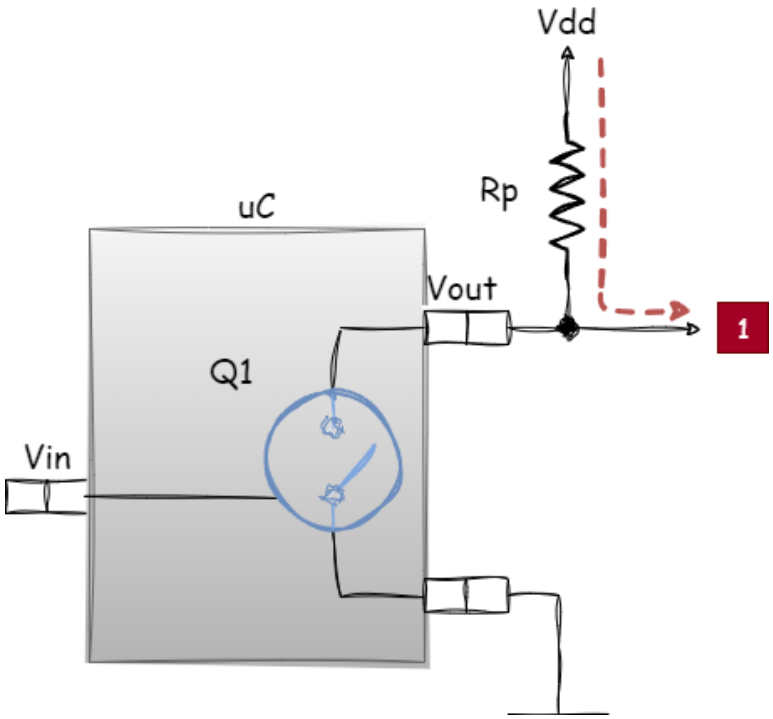

Figure 2.27 - Logical one formation at the Open-Drain output stage

When the transistor is on (conducts current), the output is connected to ground, and the output signal becomes low. No short circuits occur because the current flows through the resistor:

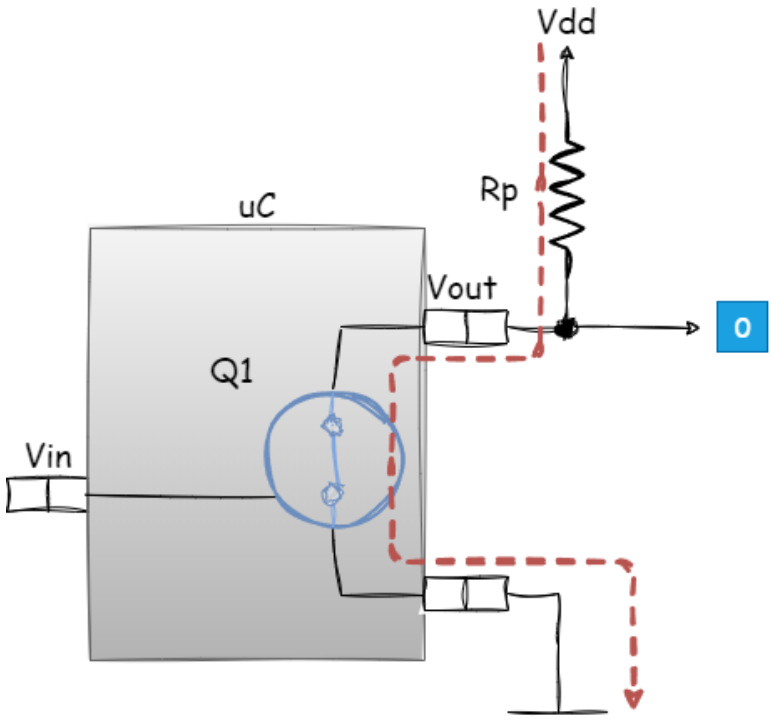

Figure 2.28 - Logical zero formation at the Open-Drain output stage

This approach to implementing a bus interface has several notable characteristics:

- The line will always remain high (logical one) unless one of the devices on the bus turns on its *N*-channel transistor to pull the logical level of the line down to zero. This connection type is known as *wired-AND* bus. More about this configuration will be discussed in the next section.
- Data transmission is effectively carried out only by *pulling the line down* to the value of logical zero, as the logical one is automatically set by the pull-up resistor.
- Only this configuration allows directly connecting two (or more) open-drain drivers to a single bus: the pull-up resistor ensures there is no short circuit between Vdd and GND.

2.4 Wired AND Connection

In wired embedded protocols, it is common to encounter situations where devices are connected to a shared line pulled up to the supply voltage through a pull-up resistor:

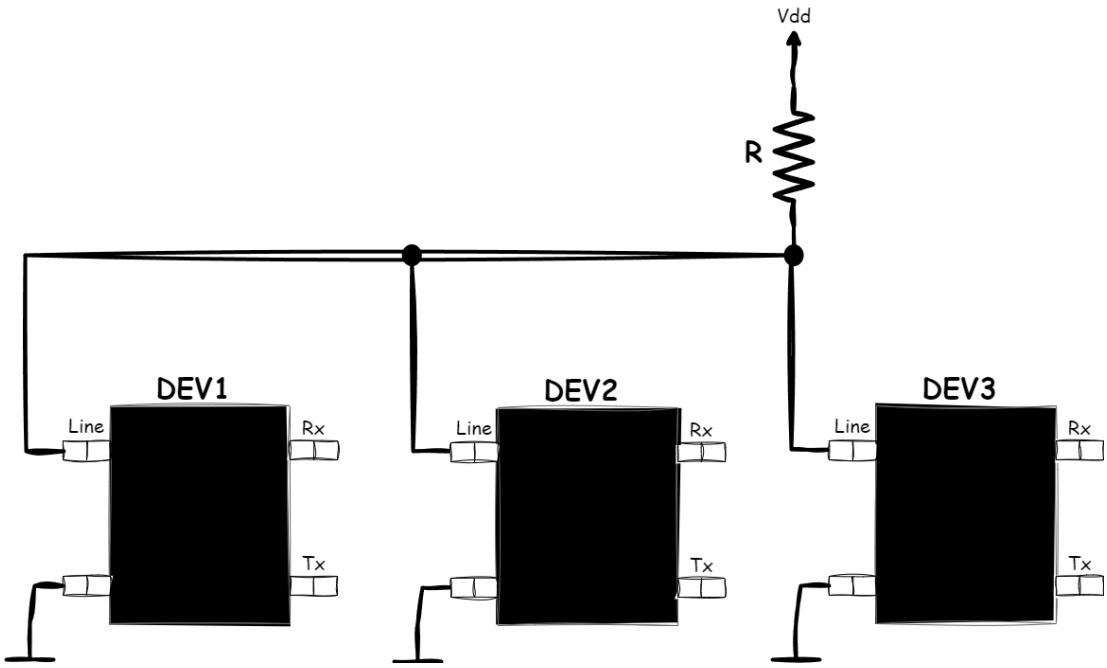

Figure 2.29 - Wired AND Connection

This type of connection is used in protocols such as 1-Wire, I2C, and I3C and is called *Wired AND* because, just like the result of a logical *AND* operation is zero if at least one operand is zero, in the Wired AND connection, *the line will be in a high voltage state if all devices on the line drive it to a high voltage, and in a low voltage state if at least one device drives it to a low voltage*.

The truth table for the diagram:

DEV1	DEV2	DEV3	BUS
0	0	0	0
0	0	1	0
0	1	1	0
0	1	0	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

This behavior is enabled by the fact that the output stages of all devices connected to the shared line are implemented as Open-Drain:

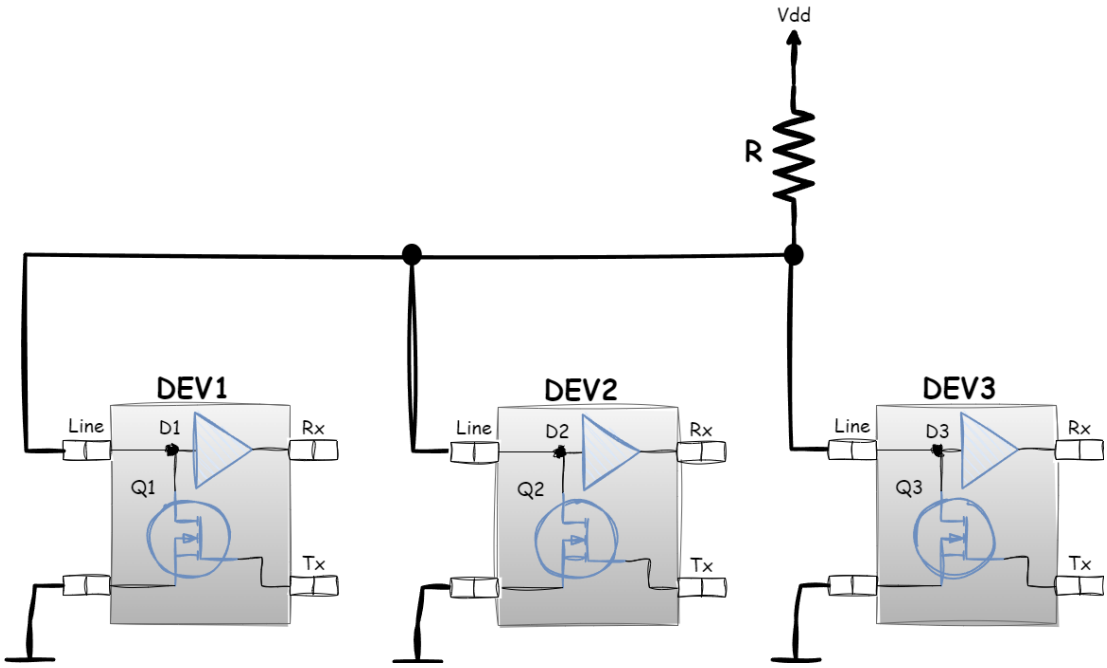

Figure 2.30 - Output stages of devices in a Wired AND connection

As you can see, all devices connected to the shared line use Open-Drain output stages: Q1, Q2, and Q3. Input buffers D1, D2, and D3 allow the devices to also read the line state.

Let's examine how the Wired AND connection works.

As we know from the section Open-Drain Output Stage, an Open-Drain stage actively drives only the low level, while the high level is set automatically by the pull-up resistor. Thus, when all output transistors of the devices connected to the bus are off, the entire line is pulled to the supply voltage by the pull-up resistor, resulting in a logical one state:

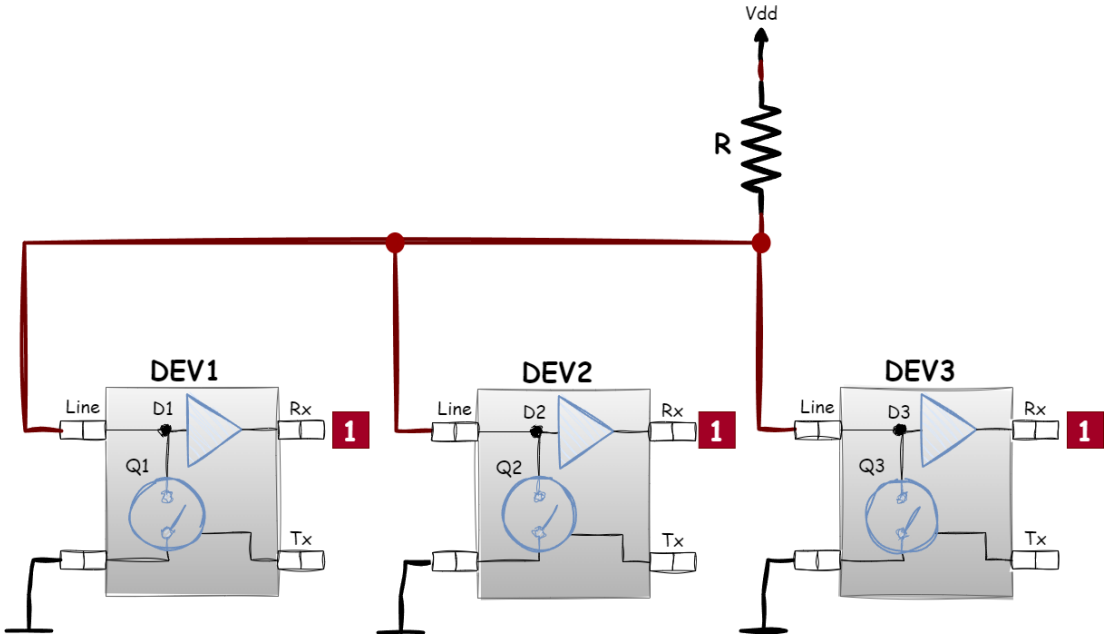

Figure 2.31 - Logical one formation in a Wired AND connection

Now let's see what happens if, for example, device DEV2 turns on its transistor Q2:

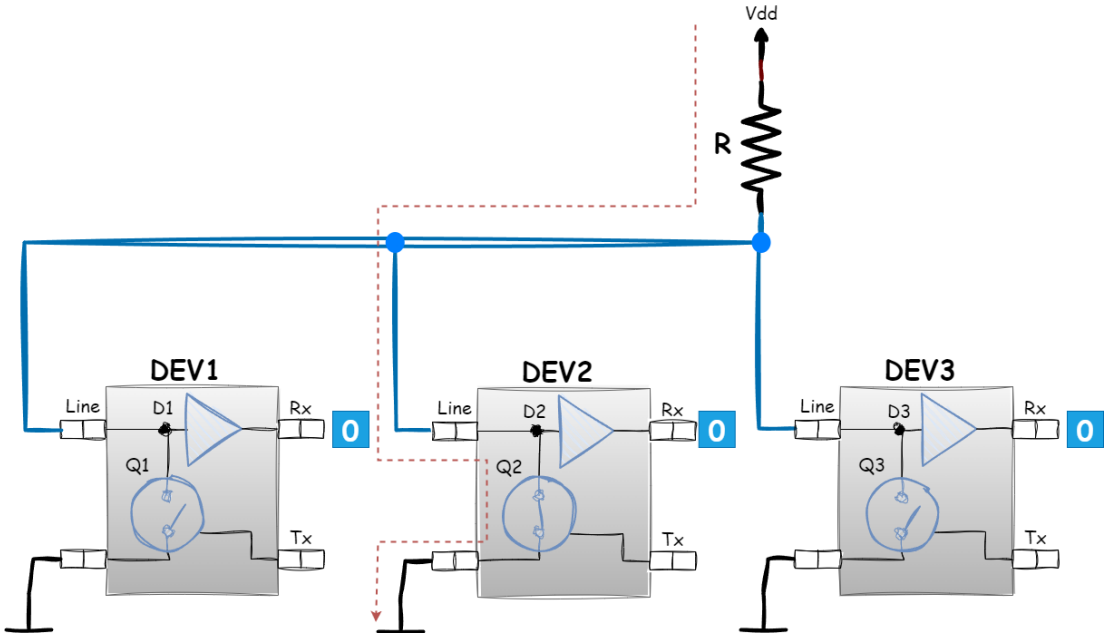

Figure 2.32 - Logical zero formation in a Wired AND connection

In this case, the entire line transitions to a low voltage state because device DEV2 effectively pulls the line to ground through its active transistor Q2. Notably, devices DEV1 and DEV3 do not change their states and effectively "want" to set a high voltage on the line. However, it only takes one device (DEV2 in this case) setting the line to a low voltage for the line to adopt a low voltage state. This is why the bit **0** is called *dominant*.

This feature of the Wired AND connection underpins fundamental functions in various protocols, such as the device discovery algorithm in the 1-Wire bus and bus arbitration in I2C, which will be discussed in their respective chapters. Moreover, thanks to the use of Open-Drain stages, this connection is safe for operation with multiple devices on the same line, regardless of whether any device is transmitting a logical one or zero. These unique characteristics make the Wired AND connection widely used in many wired communication protocols.

2.5 Conclusion

Communication protocols are the foundation of digital systems, enabling the transmission and processing of information in binary form. In this

chapter, we explored the key aspects underlying such protocols, from the concepts of logical one and logical zero to practical implementations, including various voltage levels, resistor usage, types of output stages, and device connection methods.

Understanding how data transmission works at the physical level is critical for designing and configuring wired protocols. Knowledge of which voltage levels correspond to logical values, how to select appropriate resistor values, and the characteristics of different output stages is essential for proper design and ensuring reliable data transmission.

Each of these factors can significantly impact system efficiency, noise immunity, and compatibility with various devices. Engineers and developers must consider these details when designing digital interfaces, enabling the creation of more robust and high-speed communication systems.

Further exploration of specific wired protocols, such as 1-Wire, UART, I2C, SPI, and others, will reveal even more details to consider when developing effective and reliable solutions for digital data transmission.

Part II. Intra-board Communication Protocols

3. 1-Wire

1-Wire is a serial communication protocol developed by Dallas Semiconductor that allows devices to communicate over a single data line, plus a ground connection (GND).

First introduced in the 1980s, the 1-Wire protocol was created to reduce wiring costs for peripherals and simplify PCB layout. It is ideal for applications where minimizing wiring and system complexity is essential.

A key feature of the 1-Wire protocol is that a single wire carries both data and power to connected devices, making it ideal for energy-efficient applications and devices with limited available I/O lines.

Due to its simplicity of connection, low power consumption, and reliability, 1-Wire is widely used in applications such as industrial systems, consumer electronics, and access control systems. The protocol is often used to connect devices such as:

- Temperature and humidity sensors (e.g., DS18B20)
- EEPROMs (e.g., DS2431)
- Identification and authentication chips (iButton)
- Real-Time Clock (RTC) (e.g., DS2417)

In this chapter, we will explore in detail the principles of 1-Wire and its architecture so that you can effectively use this protocol in your projects and designs.

This protocol can be used for both on-board device communication and remote sensors located up to several hundred meters away. However, this chapter focuses on scenarios where the master and all slaves are located on the same board.

3.1 Overview

The 1-Wire protocol does not have a dedicated clock line, but it is also **not** asynchronous in the typical sense. Instead, the master imposes strict timing through defined time slots, and each slave device responds within these precise intervals. This approach allows communication over a single data line without requiring independent, high-precision clocks on every device, as the master provides the necessary timing references.

1-Wire supports *half-duplex* operation. Devices on the bus can transmit and receive data, but simultaneous transmission and reception are not possible. At any given time, one device functions as the transmitter, while another functions as the receiver. Although every device on the bus can transmit data, only the master device can always initiate transactions.

The data transfer rate of 1-Wire is relatively slow. The protocol provides two modes of operation: **standard** mode (~15 kbps) and **overdrive** mode, supporting speeds up to 125 kbps. Devices that do not support the overdrive mode can operate on the same bus as those that do.

The protocol is based on a master-slave architecture with a clear division into master and slave devices. Dynamic role switching during protocol operation is not supported.

Despite its simplicity, 1-Wire is a highly versatile protocol. It incorporates many concepts that have applications in more complex protocols, making it valuable for both study and practical applications.

Although the protocol is referred to as 1 (one) Wire, it requires a ground connection and, in some cases, a separate power line for full functionality. There is a technique of powering the chips directly from the data line, which will be mentioned later.

3.2 Bus Connection

An example of device connection using the 1-Wire protocol is shown below:

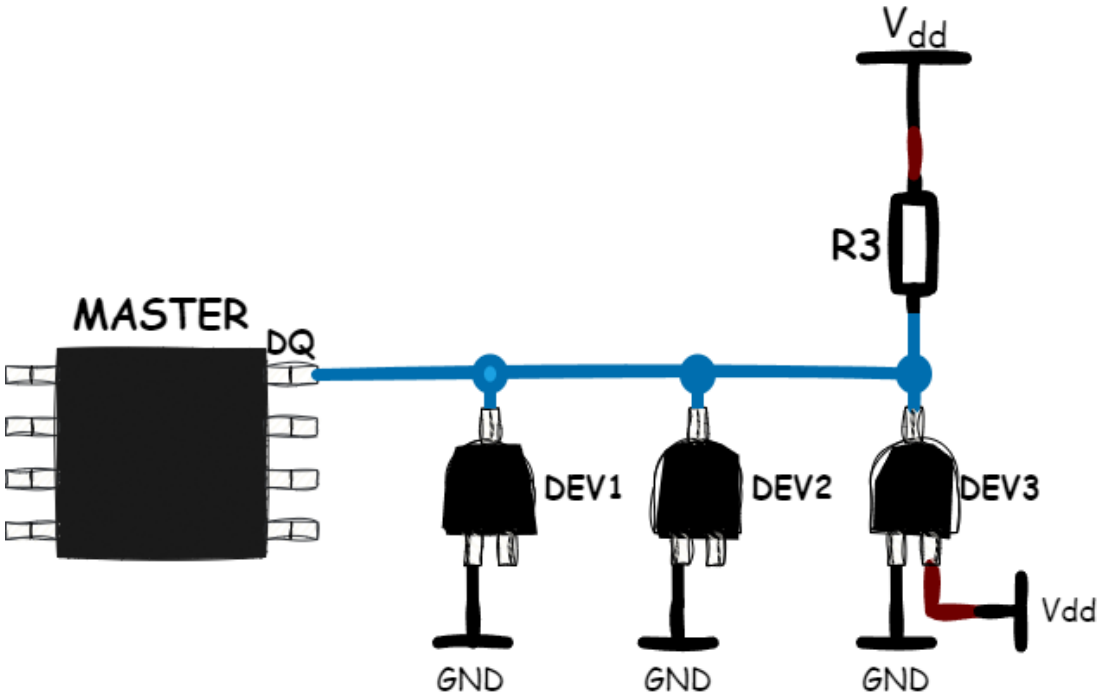

Figure 3.1 - A typical 1-Wire wiring diagram

This example shows a master and three slaves connected to the 1-Wire bus. The master initiates all communication on the bus. Even if a slave has data to send to the master, it must wait for the master to initiate the exchange.

You can also see that the whole bus is pulled up to the supply voltage using a pull-up resistor, which hints at the fact that the 1-Wire bus is schematically implemented using the *wired AND* scheme, which we discussed in the corresponding section of the Chapter 2 and in the context of the 1-Wire protocol we will talk about it in the Physical Layer section of this chapter as well.

If you look closely at the connected devices, you will notice that their connection methods are different: `DEV3` is connected to an external

power supply via a separate pin, while `DEV1` and `DEV2` appear to have no power connection, but in fact they are powered from the bus, this is one of the interesting features of the 1-Wire protocol called *parasitic powering*. This feature will be discussed in the Parasitic Power section of this chapter.

3.3 High-Level Data Transfer

Data on the 1-Wire bus is transferred bit by bit, byte by byte, starting with the least significant bit (LSB) of the least significant byte.

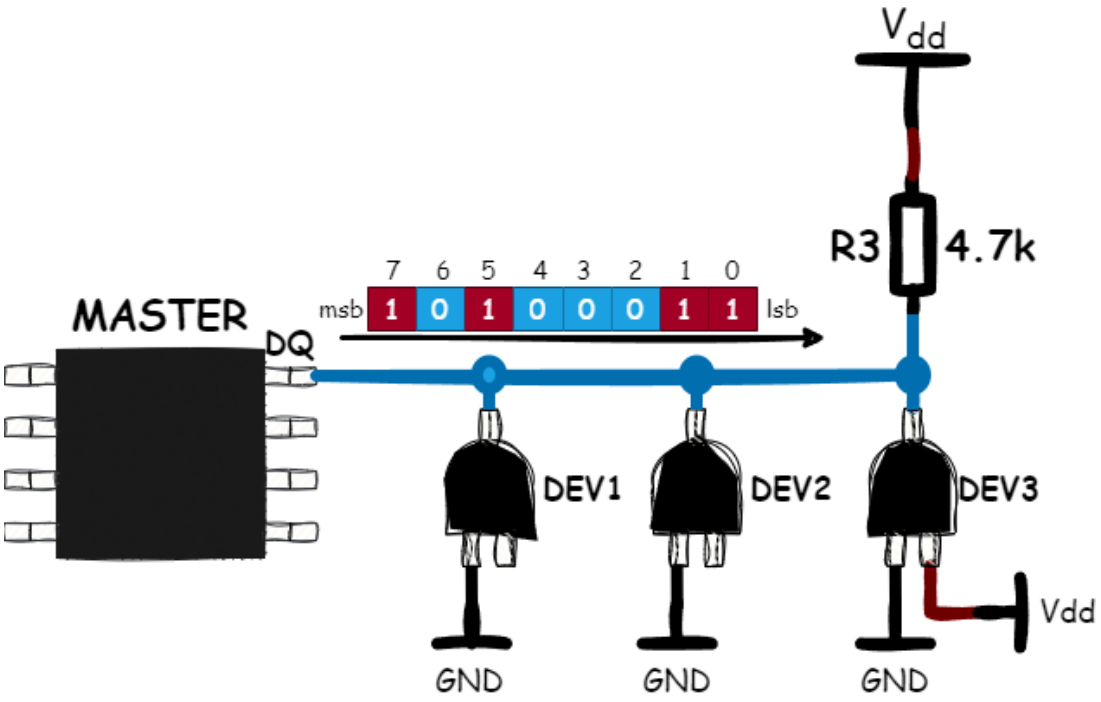

Figure 3.2 - Byte transmission format in 1-Wire

This is an important detail to consider if you are developing software for communication over the 1-Wire bus.

Most microcontrollers do not have dedicated hardware support for this protocol, unlike protocols such as UART, I2C, and SPI, which are typically hardware-supported.

As a programmer, you will need to ensure that data is transmitted in the correct order, starting with the least significant bit (LSB), and consider this order when reading data from the bus.

Although there is no hardware support for the 1-Wire protocol in microcontrollers, but it is not necessary to implement this protocol by software, because it can be emulated using the UART module, which will be described in detail in the Chapter 4 - UART.

3.3.2.1 Match ROM Command Example

In this example, three devices are connected to the bus: a master and two slaves. Therefore, the master must use the `Match ROM` command to address a specific slave.

The data exchange starts with a Reset Sequence. The master sends a `RESET` signal:

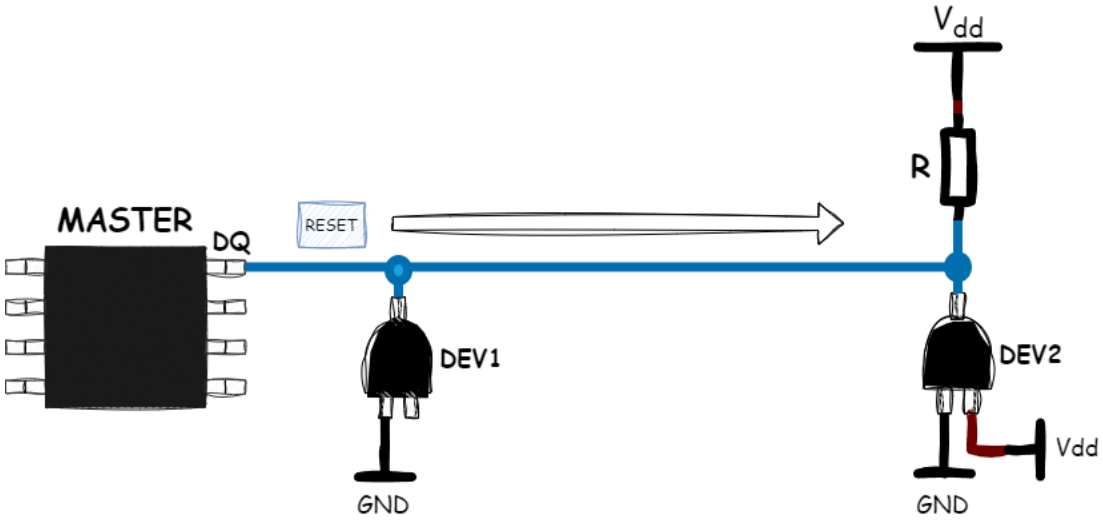

Figure 3.5. - RESET signal in the Match ROM command

After receiving this signal from the master, both slaves respond with the `PRESENCE` signal, thus confirming their presence on the bus and readiness for data exchange:

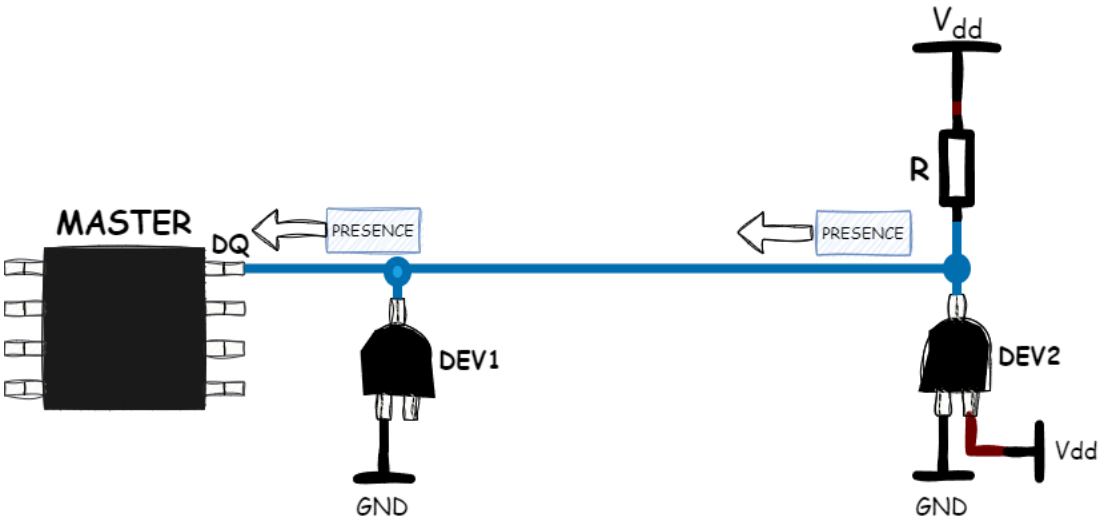

Figure 3.6 - PRESENCE signal in the Match ROM command

Upon receiving the PRESENCE signal, the master detects at least one device on the bus and sends the `Match ROM` command followed by the 8-byte ROM code of the target device:

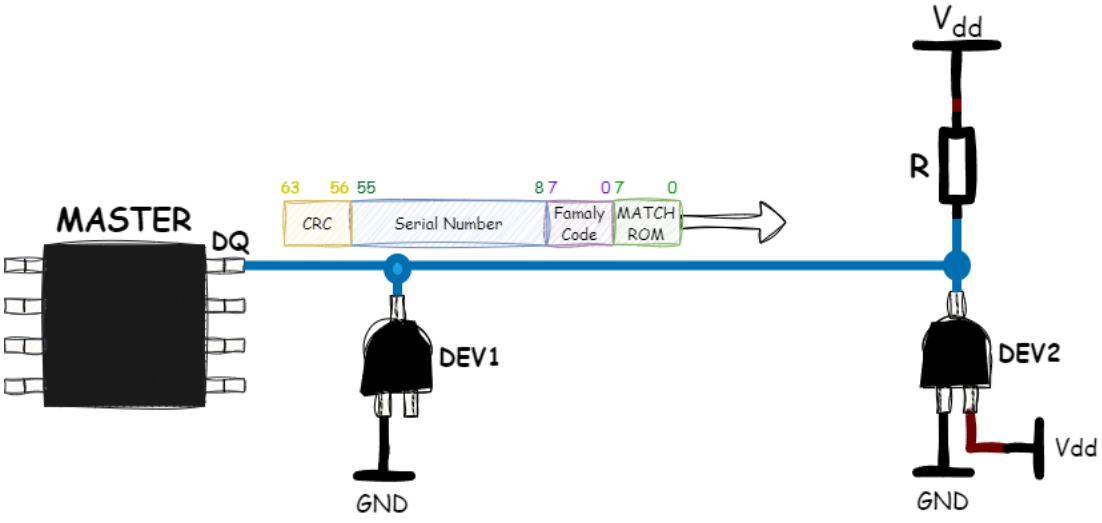

Figure 3.7. - Match ROM command transmission

The ROM code is transmitted starting with the least significant byte (**Family Code**).

All devices on the bus accept this transmission.

- `DEV1`, upon comparing the transmitted ROM code with its own, detects a mismatch and disconnects from the bus, terminating communication.
- `DEV2` recognizes its ROM code, remains active on the bus, and continues communication with the master:

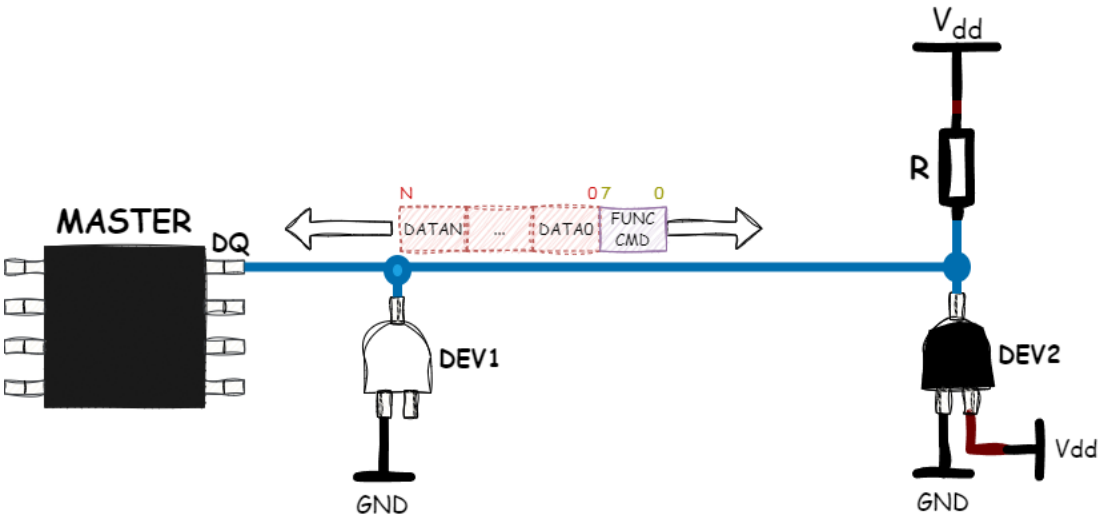

Figure 3.8. - Executing a functional command

Read the rest in the full version of the book.

3.4 Low-Level Data Transfer

At the low level, data transmission in the 1-Wire protocol is carried out by precisely controlling the timing intervals and the bus voltage state. The protocol defines precise timing parameters that determine the sequence of signal transmission and interpretation. For proper operation, devices must strictly adhere to the specified timings.

The timings are determined by the speed mode: **standard** or **overdrive**. In standard mode, the timings are longer to ensure reliable transmission,

especially on long lines. In overdrive mode, the timings are shorter, which allows higher exchange rates but requires more precise timing.

3.4.1 Time Slots

At the low level, data in the 1-Wire protocol is transmitted using **time slots** — the smallest time intervals within which a single bit of information is sent. These time-slots are divided into two types:

- *Synchronization* time slots (for establishing communication).
- *Data transfer* time slots (to write and read bits).

Synchronization time-slots:

- **RESET** time slot: a special reset signal to start a new transaction.
- **PRESENCE** time slot: in response to the RESET signal, the slave device generates a presence signal.

These two time-slots are conventionally combined into one called **Reset Sequence**, which marks the beginning of data exchange.

The data transmission time-slots are:

- **Write-1 Time Slo**t: Used by the master to transmit a '1' bit to the slave.
- **Write-0 Time Slot**: Used by the master to transmit a '0' bit to the slave.
- **Read-1 Time Slot**: Used by the master to read a '1' bit from the slave.
- **Read-0 Time Slot**: Used by the master to read a '0' bit from the slave.

3.4.1.1 Reset Sequence

Reset Sequence is a combination of two time slots: `RESET` and `PRESENCE` and looks as follows:

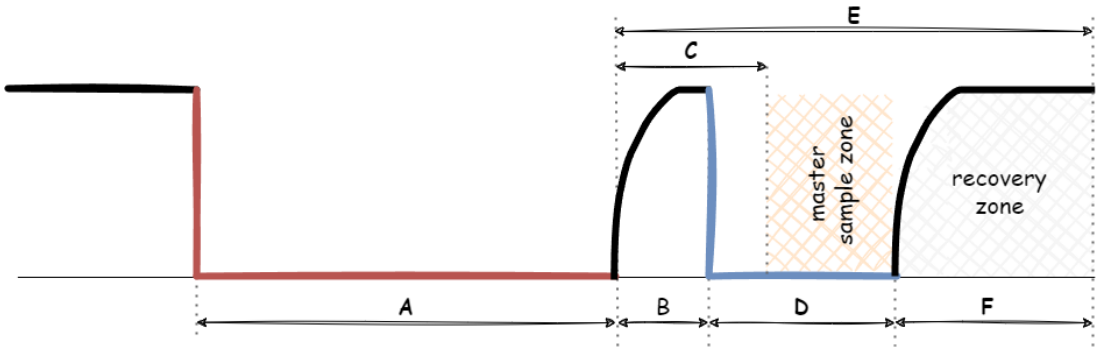

Figure 3.44. - Reset Sequence

In Figure 3.44 and following, the red color shows the section when the master sets the required level on the bus, the blue color shows the section when the slave sets the required level on the bus, and the black color shows the section when the level on the bus is set by the pull-up resistor.

It has the following timing characteristics:

Table 3.1 - The Reset Sequence time characteristics

Parameter	Standard Speed		Overdrive Speed	
	Min (µs)	Max (µs)	Min (µs)	Max (µs)
A	480	640	48	80
B	15	60	2	6
C	60	75	6	10
D	60	240	8	24
E	480	-(1)	48	-(1)
F	5(2)	-(1)	3(2)	-(1)

(1) - As you can see this time slot is not limited in its maximum length and it means that after the transmission of this time slot you can make any long pause in the transmission and this pause should not disturb this transmission in any way. In theory :).

(2) - This value is highly dependent on the line configuration and is specific for each individual device, this value is taken from the documentation of the specific device and the values given here are used for example only and are not standard.

3.5 Physical Layer

At the physical level, the 1-Wire bus follows the Wired AND principle. As explained in Chapter 2, the bus remains high if no device pulls it low, and it goes low if at least one device drives it low.

In practice, this is implemented using a pull-up resistor to tie the bus to the supply voltage. Each device can pull the line low via an open-drain output stage.

The data exchange and signal generation principles discussed below are common to all protocols following the Wired AND principle.

As mentioned earlier, the 1-Wire protocol serves as a foundation for other protocols, such as I2C, which also follows the wired AND principle. Understanding the data exchange principles in 1-Wire makes it easier to understand the I2C protocol.

3.5.1 Data Exchange Mechanism

Let's look at a simplified schematic of the hardware implementation of the 1-Wire interface:

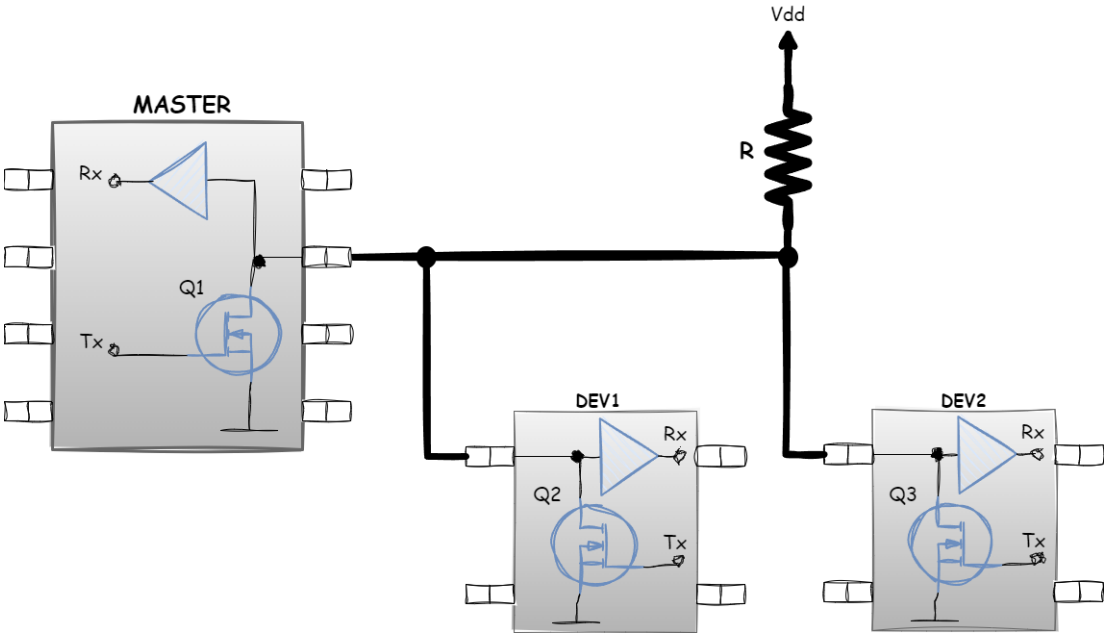

Figure 3.49. - A 1-Wire bus configuration

As shown in the figure, all three devices — `MASTER`, `DEV1`, and `DEV2` — are connected to a common data line through open-drain output stages, which drive the bus to the required logic levels, and input buffers for reading the current bus state.

The bus is pulled up to the supply voltage via the pull-up resistor R and remains at a high logic level when idle.

Assume that the master starts transmission by sending a `RESET` time slot on the bus.

3.6 Devices Search Algorithm (ROM)

Until now, we have assumed that the master knows the address (ROM) of each device on the bus or that there is only a single device on the bus. But what if multiple devices are connected, and their ROMs are unknown?

This is where the standard `Search ROM` command comes into play, which we have not considered until now, along with a special search algorithm, which we will examine in this section.

If the master does not know the ROMs of the devices connected to the bus, it is possible to detect them using the `Search ROM` command – all devices send the value of the low bit of their ROM to the bus; the *direct* value of the bit is sent first, and then the *inverted* value of the same bit.

Then the master can read the following sequences:

- `0b01` - if all slave devices have an LSB of `0`.
- `0b10` - if all slave devices have an LSB of `1`.
- `0b00` - if a conflict occurs, meaning some devices have an LSB of `1`, while others have `0`. It is this state that makes the master realize that more than one device is present on the bus.
- `0b11` - indicates that no slave devices are connected to the bus.

3.6.1 Search ROM Command Example

To illustrate how the `Search ROM` command works, consider the following bus configuration:

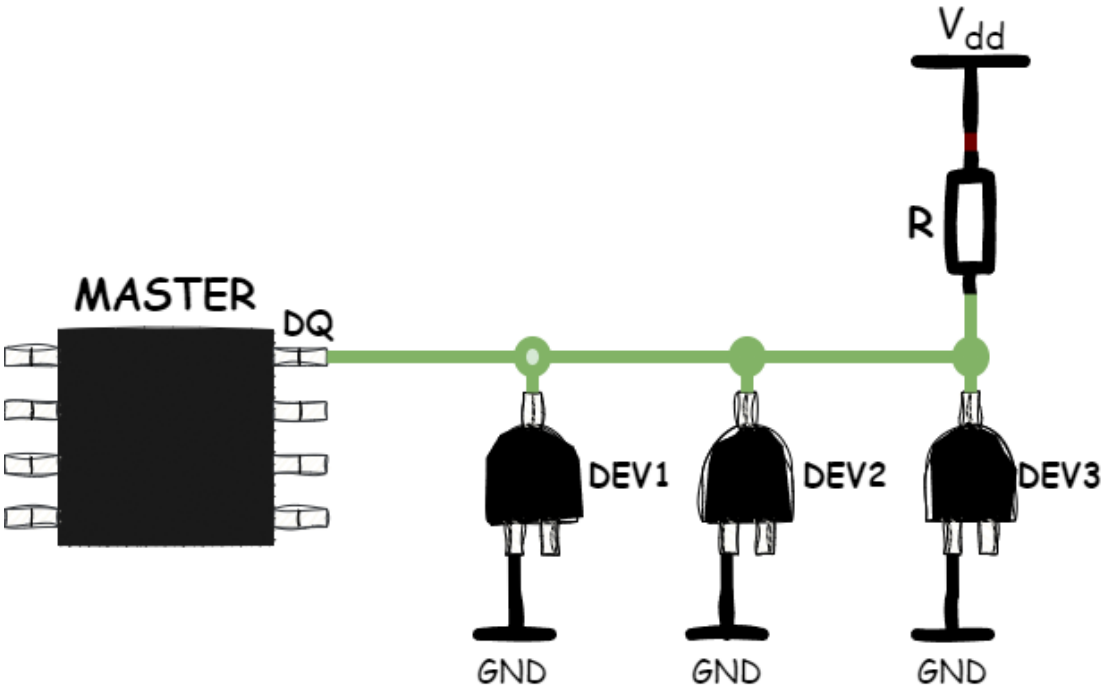

Figure 3.60 - 1-Wire bus setup for the Search ROM algorithm example

Assume the devices have the following ROM values:

- **DEV1:** `x...x10101010`
- **DEV2:** `x...x11001110`
- **DEV3:** `x...x00110011`

(Here, `x` represents the higher-order bits of the ROM code.)

Now, let's begin the `Search ROM` procedure:

After all these steps, the master will know how many devices are connected to the bus at the moment and the ROM of each of these devices.

The algorithm for searching devices in the 1-Wire protocol is similar to traversing a binary tree.

Each unique 64-bit ROM code of a device can be represented as a path in a binary tree, where each bit of the ROM code corresponds to one step to the left (bit = `0`) or to the right (bit = `1`).

The `Search ROM` algorithm essentially implements a *depth-first search (DFS)* traversal of this tree. The master device starts at the root (the least significant bit) and at each step decides which path (branch of the tree) to follow based on the slaves' responses.

As a result, the device scanning process can be visualized as follows:

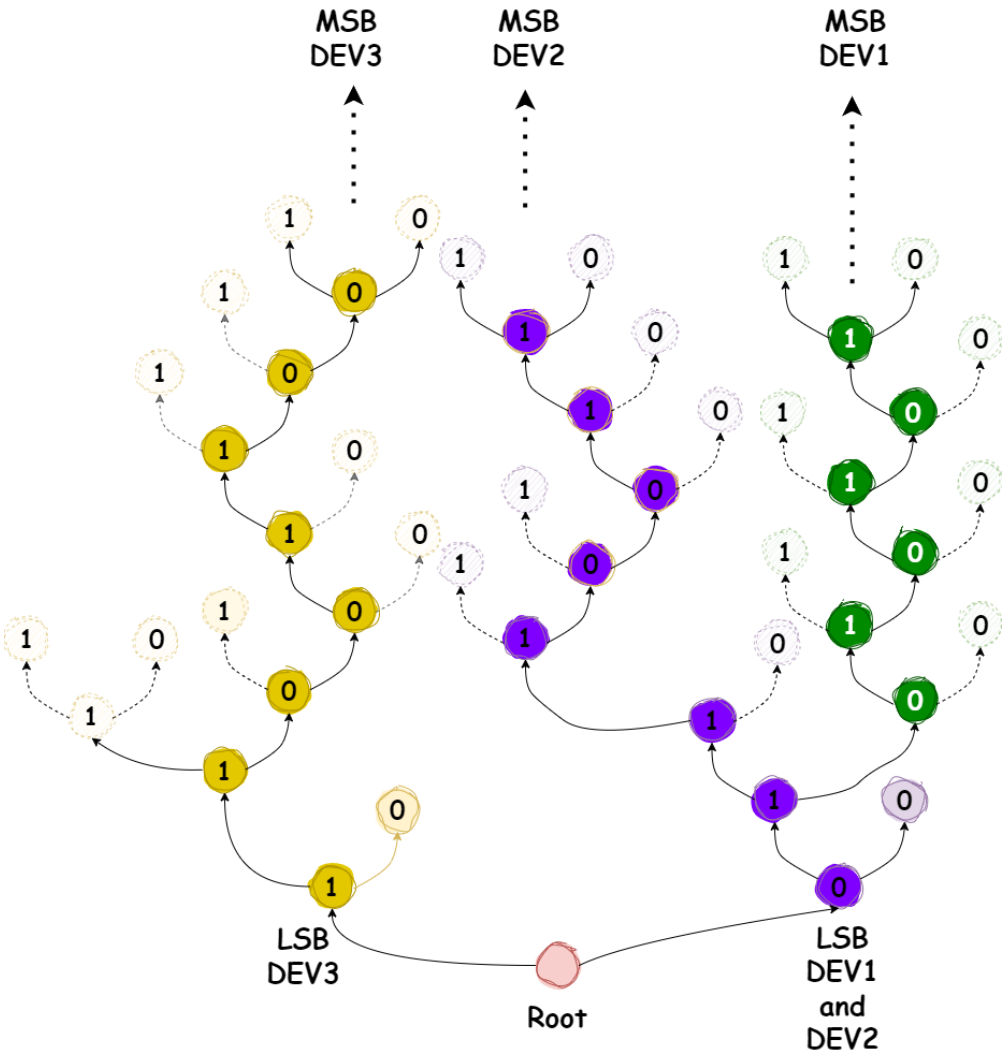

Figure 3.69 - Binary tree representation of the ROM search algorithm