FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Software for Nixie clock
Student:	Martina Bechyňová
Supervisor:	Ing. Matěj Bartík, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer engineering
Department:	Department of Digital Design
Validity:	until the end of summer semester 2021/2022

Instructions

Design, implement, and test a control software for an embedded system containing a microprocessor from the STM32WB series, which supports Bluetooth Low Energy (BLE) communication.

The software should:
- implement a clock and a calendar function, with the option to add additional features to the base program,
- support setting the clock, the calendar, and any other parameters via a mobile application, using the BLE interface,
- use the real-time clock (RTC) to keep time and date,
- operate the individual nixie tube segments to display the desired data.

Document the resulting code properly. The implementation can be done in any programming language. However, C/C++ is preferred.

Electronically approved by prof. Ing. Hana Kubátová, CSc. on 14 February 2021 in Prague.

Bachelor's thesis

SOFTWARE FOR NIXIE CLOCK

Martina Bechyňová

Faculty of Information Technology
Department of Digital Design
Supervisor: Ing. Matěj Bartík, Ph.D.
May 12, 2022

Citation of this thesis: Bechyňová Martina. *Software for Nixie clock.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Contents

List of Figures

List of Tables

I would like to express my utmost gratitude to my supervisor, Ing. Matěj Bartík, Ph.D., for his guidance and valuable advice given during the course of the thesis. I also would like to thank my family and friends for their moral support.

Declaration

In Prague on May 12, 2022 ....

Abstrakt

Cílem práce je vytvořit řídící software pro an míru vytvořený vestavný systém, který obsahuje mikrokontrolér z řady STM32WB a lze ovládat pomocí Bluetooth Low Energy. Práce se zabývá analýzou kladených požadavků na software a jeho následným vývojem. Práce klade důraz na kompatibilitu s komplementární BLE Android aplikací, která byla pro zakázkové zařízení vyvinuta, nezaobírá se však její implementací. Software byl napsán od základů, neboť nebyl nalezen žádný vhodný existující projekt, který by mohl být použit jako výchozí bod. Výsledkem práce je funkční řídící software, který lze využít na zmíněných zakázkových systémech nebo jako základ v podobných projektech.

Klíčová slova STM32WB, vestavný systém, řídící software, Bluetooth Low Energy, Generic Attribute Profile, Generic Access Profile, hodiny reálného času

Abstract

The aim of this thesis is to create control software for a custom-made embedded system using an STM32WB microcontroller unit that can be configured via Bluetooth Low Energy (BLE). The thesis concerns itself with analyzing the criteria placed on the software and the development of said software. The thesis also places emphasis on compatibility with the complementary BLE Android application that was developed for the custom device, but it does not cover the development of the application itself. The software is made from the ground up, because no suitable existing project was found that could be used as its basis. The result of the thesis is functional control software that can be used on the custom-made devices or as a basis for a similar project.

Keywords STM32WB, embedded system, control software, Bluetooth Low Energy, Generic Attribute Profile, Generic Access Profile, real-time clock

List of Abbreviations

BLE	Bluetooth Low Energy
DIY	Do It Yourself
DST	Daylight Saving Time
EEPROM	Electrically Erasable Programmable Read-Only Memory
GAP	Generic Access Profile
GATT	Generic Attribute Profile
GPIO	General-Purpose Input/Output
HAL	Hardware Abstraction Layer
HSE	High-Speed External (oscillator)
IoT	Internet of Things
IPCC	Inter-Processor Communication Controller
I^2C	Inter-Integrated Circuit
LED	Light-Emitting Diode
LL	Low-Layer
LSE	Low-Speed External (oscillator)
MCU	MicroController Unit
MSB	Most Significat Bit
PPCP	Peripheral Preferred Connection Parameters
ppm	parts per million
PWM	Pulse-Width Modulation
RF	Radio Frequency
RTC	Real-Time Clock
RX	Receive
SPI	Serial Peripheral Interface
TX	Transmit
UUID	Universal Unique Identifier or Universally Unique Identifier

Chapter 1

Introduction

A nixie tube (see Figure 1.1), also known as a cold cathode display, is an electronic device used to display numerals and symbols. The rare-gas-filled tube contains a wire-mesh anode and multiple cathodes, which are shaped like digits or other symbols. When voltage is applied to a cathode, the gas surrounding it lights up, illuminating the character. They were introduced to the market in 1955 and were at the height of their popularity in the late 1950s and 1960s. The tubes were often used as numeric displays in early technical equipment (such as voltmeters, multimeters, calculators (see Figure 1.2), and frequency counters) and early desktop calculators. The 1970s saw them being superseded by newer and cheaper technologies, most notably light-emitting diodes and vacuum fluorescent displays. However, the story of nixie tubes does not end there—ever since around the turn of the millennium, they have been slowly growing their niche community, for they have charmed many with their retro appearance. Nowadays, they can be usually seen in DIY or ready-made nixie clocks.

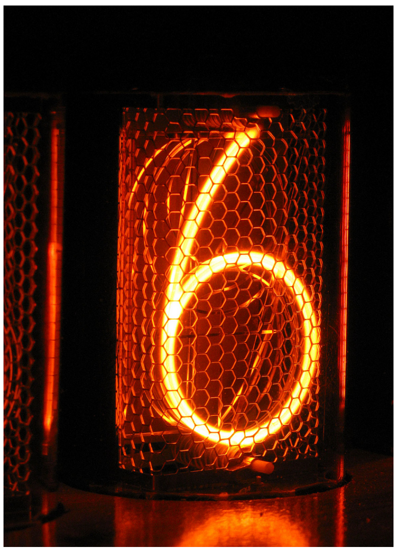


Figure 1.1 A Z566M nixie tube [1].

This thesis is complementary to the “Control application for nixie clock for mobile phones” bachelor thesis [2], whose goal was to create an Android application that can set up the custom-made device used in this thesis.

This thesis aims to design, implement, and test a control software for a custom embedded

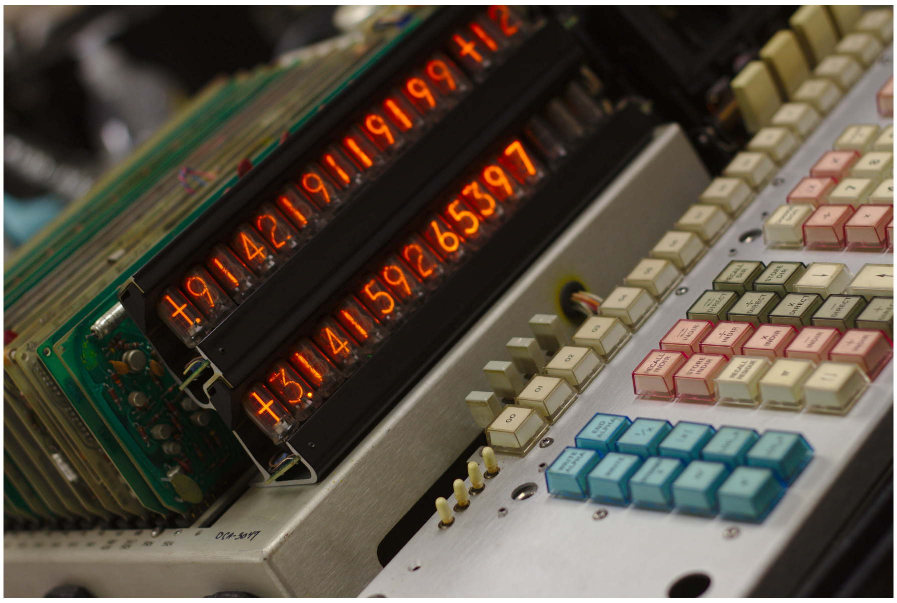


Figure 1.2 A Wang 700 Advanced Programmable Calculator [3].

system containing a microprocessor from the STM32WB series. One of the crucial requirements is that the device should display the set-up time and date. The program should also provide the extra features the mobile application contains (e.g., stopwatch and alarms). The solution might be expanded with additional functions, should they be deemed necessary. Another important criterion is that the software should allow setting up the device via the Bluetooth Low Energy interface. The implementation of the communication should be compatible with the Android application. Lastly, the software should use the real-time clock to keep time and date and show the desired data on the nixie tube display.

This thesis is not going to delve into the development of the mobile application itself, it only needs to ensure that the control software can communicate with the Android application correctly.

This thesis and the Android application thesis work in tandem to offer a nixie clock solution that uses BLE for configuration. Anyone can make use of the final product to create their own nixie clock, and they can choose to either adopt the solution as-is or use it as a foundation for their own project.

The thesis starts with the analysis, which explains the relevant terminology, describes the traits of the custom embedded systems, and highlights the important parts of the Android application thesis. It also pinpoints the software requirements in greater detail, talks about useful manufacturer resources, and identifies problems the software has to resolve. The implementation chapter details the development process, starting from the smaller projects used to find the correct configuration of the crucial elements and ending with the initial prototype of the control software, which was then gradually expanded with more features. The implementation chapter also includes a brief mention of the tools required during development. The testing chapter covers the tools and methods used for testing the software and presents the outcome of said tests. Finally, the last chapter evaluates and summarizes the results achieved.

Chapter 2
Analysis

The analysis aims to cover the research portion of the thesis. First, the analysis describes the hardware characteristics of the embedded system. After that, the analysis explains the terms used in this text, summarizes the state of the art, and details the constraints the mobile application places on the control software. Then the analysis analyzes any MCU (microcontroller unit) manufacturer resources that can be useful for implementation. Some computational problems encountered during development can be approached in more than one way—the final section of the analysis strives to compare the available algorithms to select the most optimal one.

2.1 Hardware Description

The custom-made embedded system (the schematic can be found in Appendix C, see Figure 2.2 and Appendix A for PCB photographs) uses an MCU from the STM32WB series. The device has a display consisting of six nixie tubes that can be used to show data to the user. Each tube can show the digits 0–9. The tubes require a high voltage (170 V) to properly illuminate the digits. The high voltage narrows the list of suitable candidates for the components that are exposed to it, as the components must be able to withstand it. [4]

The system uses two 32-channel serial-to-parallel converters to manipulate the display [4]. The converter (see Figure 2.1) is implemented as a 32-bit shift register, where each bit controls the value of a channel [5]. The channels are connected to individual nixie tube cathodes and turn the cathodes on/off [4]. The converters also offer the option to latch and blank the output. The MCU can communicate with the converters through SPI [4].

The system has four LEDs available to help indicate the type of data currently displayed on the nixie tubes [4]. Each LED can be controlled separately [4].

The system contains a buzzer that can be used to generate sounds. The system also contains an accelerometer that can be used to detect whether the user is shaking the device. The MCU can communicate with the accelerometer through I^2C, and the accelerometer also has access to two pins that it can use to indicate an interrupt event. Additionally, the system contains an EEPROM that serves as nonvolatile memory for the control software. The EEPROM uses the same I^2C bus as the accelerometer. [4]

The system has two external oscillators: one with a 32.768 Hz frequency (LSE), and the other with a 32 MHz frequency (HSE). The LSE is intended to clock the RTC, as the MCU only supports battery backup of the RTC when the LSE is used as the clock source for the RTC [6]. The main purpose of the HSE is to power the radio system [6]. The embedded system has a capacitor [4] that can be used to power the LSE and the RTC if a blackout occurs [6].

Due to chip shortages, some of the embedded systems have the HV5523 converter instead

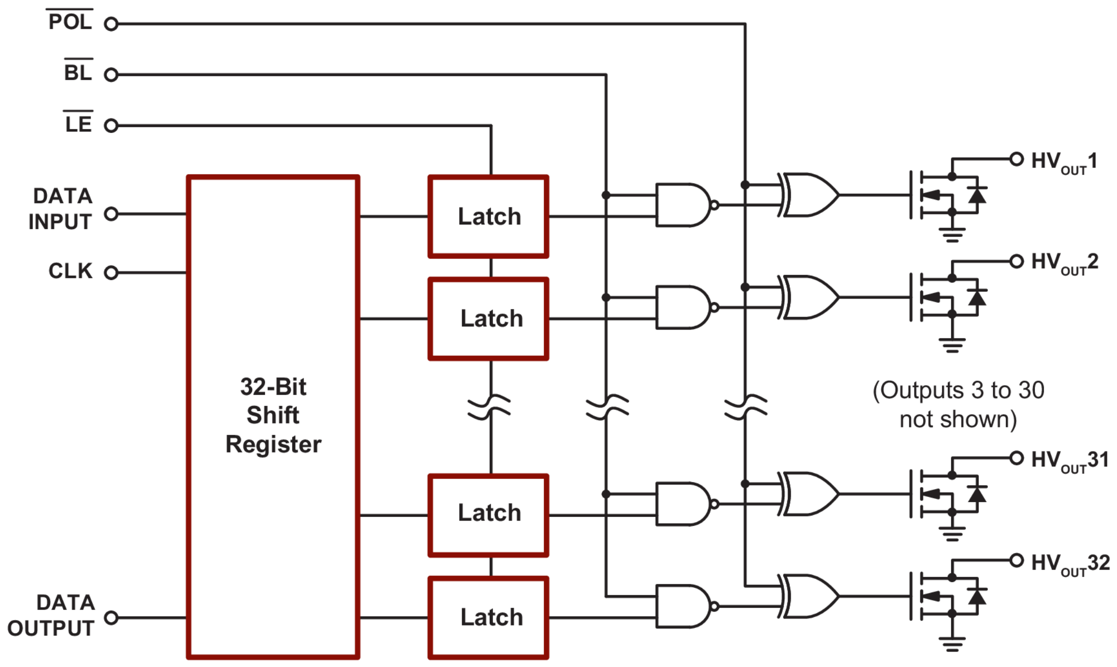


Figure 2.1 HV5623 serial converter block diagram [5].

of the HV5623 model and the ADXL345 accelerometer instead of the ADXL343 variant. While the accelerometers are interchangeable for the purposes of the control software, since they share their register design [7, 8], the converter change affects the software, as the HV5523 model maps shift register bits to channels in reverse order from the HV5623 model [5, 9]. To resolve the issue that the two converters have different output mappings, the software can use the EEPROM to store information about the converter type used on the system.

2.1.1 STM32WB

The STM32WB MCU series is manufactured by STMicroelectronics. The wireless MCUs consist of two cores: one Arm® Cortex®-M4 core that serves as the application processor and one Arm Cortex-M0+ core that serves as the network processor. The MCUs support BLE 5.2 and IEEE 802.15.4 wireless standards such as Zigbee and Thread. The architecture is optimized for real-time execution and allows flexible resource use and power management. See Appendix D for the block diagram of STM32WB55xx MCUs. [10]

The MCU series is divided into a few lines. The STM32WBx5 line offers multiple packages and memory sizes, and it provides users with enhanced performance and flexibility to address different levels of complexity. The STM32WBx0 value line focuses on the essentials and offers a feature-optimized, cost-effective solution. Lastly, the STM32WBxM line of modules offers the functionality of the STM32WBx5 line in a small LGA86 package with wide certification coverage. [10]

2.2 Bluetooth Low Energy

Bluetooth Low Energy (BLE) is a type of Bluetooth that focuses on energy efficiency, making it suitable for devices running on a battery. Unlike Bluetooth Classic, which is usually used for

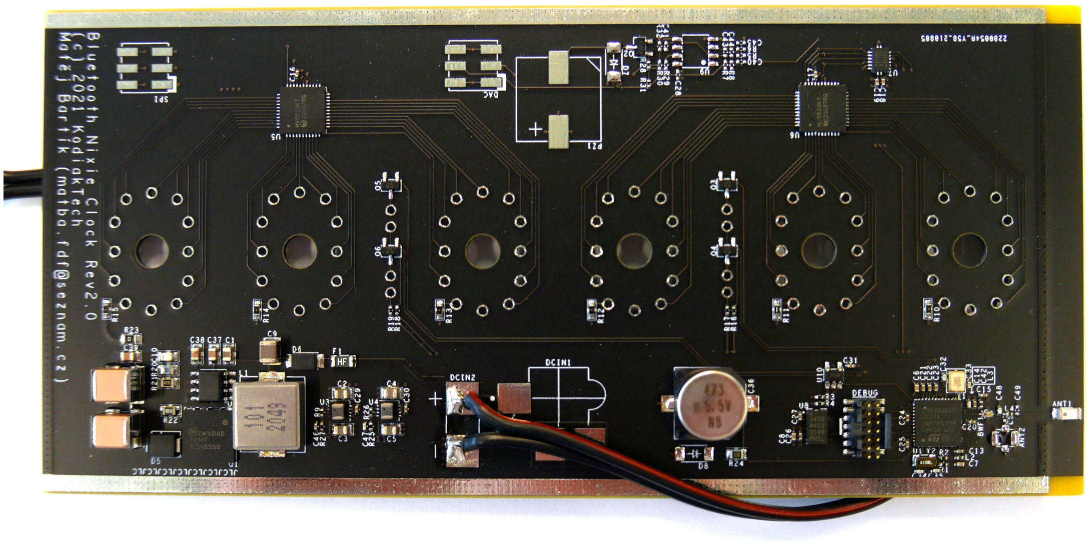

Bluetooth Nixie Clock Rev2.0
(c) 2021 KodiakTech
Matěj Bartík (matba.fdf@seznam.cz)

Figure 2.2 The PCB of the embedded system. (© Matěj Bartík)

audio streaming, BLE focuses on applications that work with small amounts of data and lower speeds (e.g., IoT). BLE can be used in smart or fitness devices, indoor locating, etc. [11]

The main advantages of BLE are its low energy consumption, free access to its specification documents, and the presence of BLE on most smartphones on the market. It is suitable for applications that transfer short bursts of data at low speed. However, this makes it inferior for video streaming and large data transfers. [11]

The technology used to be unsuitable for audio streaming as well [11], but with the introduction of LE Audio, BLE can now be used for this purpose, along with Bluetooth Classic [12].

At an abstract level, a BLE device can either be in advertising mode, which enables the device to send out data that can be discovered by others, or in connection mode. In advertising mode, the data transfer is one-directional, while in connection mode, both connected devices transmit data to each other. [13]

2.2.1 Generic Access Profile

The Generic Access Profile (GAP) defines how devices discover each other along with the advertising and scanning procedures, and how they connect to each other and maintain the connection afterward. This includes device roles, modes the devices operate in, connection parameters, advertising parameters, and security. [13]

GAP consists of two symmetrical pairs of roles. The first one is the Broadcaster and Observer pair. A Broadcaster advertises data but does not accept connections, and an Observer reads the advertisements and searches for other devices without initiating a connection. The second one is the Central and Peripheral pair. A Central device not only searches for other devices and reads their advertising data, but also has the option to connect to the advertisers. Analogically, a Peripheral sends out advertisements and allows connections to be formed. [13]

The Central device is usually more capable (faster CPU, larger memory/battery capacity…), while the Peripheral tends to be resource-constrained. By shifting most of the processing to the Central, the Peripheral can turn off the radio and enter sleep mode for longer, allowing it to conserve energy. A device can operate in multiple GAP roles at once, e.g., a smartphone can act as a Central in one connection and as a Peripheral in another. [11]

GAP modes are specific states a device can use to achieve a specific goal. They cover aspects like discoverability, connectability, and security. [13]

2.2.2 Advertisements

Advertisements are packets a device can send to broadcast data or indicate that it might want to be discovered and possibly connected to. They can indicate the advertiser's connectability and scannability status. When the advertiser is scannable, other devices can ask for additional advertising data in a scan request. The packets can also signal whether the advertisement is directed or not. Directed packets express the advertiser's desire to connect to a specific device. [14]

Advertisements can contain many types of information. The most common ones are the device name, transmit power, and GATT Service UUIDs. If the advertisement packets are not large enough to hold all the data the application wishes to expose, the program can utilize the scan response feature. Bluetooth 5.0 has introduced so-called advertising extensions—these mechanisms allow advertisements to transmit up to 8x more data. [14]

The device can choose how often it sends advertisements. The interval can range from 20 ms to 10.24 s in increments of 625 μs. The advertiser can choose any subset of the three (primary) advertising channels available. [14]

2.2.3 Connections

Connections are a mechanism that allows persistent and synchronized data exchange. They allow devices to achieve the desired power consumption level and data rate more easily by adjusting the connection parameters. [15]

After a connection is created, the Central device becomes the master, and the Peripheral device turns into the slave. The master is responsible for managing the connection and makes the final decision on the connection parameters. [15]

The process of establishing the connection starts with the Peripheral sending advertisements. This allows Central devices to discover the device and initiate a connection. Only a Central is capable of initiating a connection, a Peripheral merely advertises its ability to accept connections. A Central device can respond to advertisements with a connection request, after which the connection is considered to be created. After a set time period (the connection interval), the master sends a packet. When the slave receives the packet and responds with a packet of its own, the connection is considered established. [15]

The connection request packet contains important parameters that are needed to keep the connection synchronized and persistent. The connection interval defines how often connection events occur. A connection event starts with the master sending a data packet, and the slave responds with a data packet that contains any data that the slave needs to transmit to the master. If the slave does not have any data to send, it sends an empty packet. The slave latency defines the number of connection events the slave can skip. If the slave has no data to send back, it can ignore connection events, allowing it to sleep for longer. The supervision timeout defines the period of time since the last data exchange after which the connection is considered lost. Finally, the connection request packet contains the channel map and the channel hop increment. Each connection event is on a different RF channel. The channel map defines which channels can be used for the hopping mechanism. The hop increment helps to determine the next channel to hop to. [15]

2.2.4 Attribute Protocol

The Attribute Protocol (ATT) defines how a server exposes its data to a client and the structure of said data. Data is sorted into attributes, where attribute serves as a generic term for any data the server wishes to expose. [16]

An attribute consists of a 16-bit handle, an attribute type (also known as UUID), a value, and a set of permissions. UUIDs can be 16-bit (SIG-adopted) or 128-bit (custom). Values have a variable length and their format depends on the type. Finally, the permissions determine which operations can be done with an attribute (e.g., write, read, notify) and their security level. [16]

There are two roles within ATT. A server device exposes the data it contains and accepts commands from other peer devices. A client device interfaces with a server to read its exposed data or control its behavior. A device can be both a server and a client at the same time. [16]

ATT defines the following packet types [16]:

- **Commands**
  Packets sent by the client to the server. Commands do not require a response.
- **Requests**
  Packets sent by the client to the server. Unlike commands, requests require a response from the server.
- **Responses**
  Packets sent by the server that serve as a response to a request.
- **Notifications**
  Packets sent by the server to the client to indicate characteristic value changes. The server will send notifications only if the client has enabled them. Notifications do not require a response.
- **Indications**
  Indications share the same traits as notifications, except they require a response from the client.
- **Confirmations**
  Packets sent by the client that serve as a response to an indication.

2.2.5 Generic Attribute Profile

The Generic Attribute Profile defines the format of services and characteristics and also details the procedures to interface with these attributes. ATT is used as the underlying framework. [16]

A service is a grouping of related attributes that satisfy a specific functionality on the server. A service can contain two types of attributes: characteristics hold values, while non-characteristics help structure the service data. [16]

A characteristic represents a piece of information within the service. It also has additional attributes that help to define the value it holds. One such example is properties, which describe how a characteristic value can be used (e.g. read, write, notify). Descriptors hold additional information about the value, such as the user description or fields for notification subscription.

GATT uses the same roles as ATT. Instead of being set per device, the roles are determined by the transaction. Like in ATT, a device can act in both roles at once. [16]

2.3 Inter-Integrated Circuit

The Inter-Integrated Circuit (I^{2}C) bus consists of two bidirectional wires: serial data (SDA) and serial clock (SCL). These two lines carry information between the devices on the bus, all of which have a unique address. The protocol defines two pairs of roles for devices: transmitter/receiver and master/slave. A transmitter device sends data to the bus, while a receiver device receives data. A master device initiates a transfer, generates clock signals, and terminates a transfer. The device addressed by a master fulfills the slave role. [17]

Each transaction (see Figure 2.3) starts with a START condition and ends with a STOP condition. After the START condition, the slave address (7-bit or 10-bit long) is sent, followed by the data direction ($R/\overline{W}$) bit. The master can then send an unrestricted number of data bytes. When the transfer is finished, the master can end the transaction by sending the STOP condition, or it can transmit a repeated START condition if it wishes to address a different device. After each byte, the receiver uses the acknowledge bit to indicate whether it has received the data successfully, and another byte may be sent. Master devices may initiate a transfer only if the bus is not busy. [17]

Since the protocol allows more than one master to be connected to the bus, multiple masters might attempt to initiate a transfer at the same time. To resolve this issue, the protocol defines an arbitration procedure. During each bit, each master device checks if the SDA level matches the data it has sent. If a device tries to send a HIGH but detects a LOW on the SDA wire, it knows that it has lost the arbitration and will terminate the data transfer. The winning device completes the transaction, with no data loss occurring. [17]

The bus originally only supported up to 100 kbit/s operation (currently referred to as Standard-mode), however, the specification has added four more operating speed categories over time. The Fast-mode ($\leq$ 400 kbit/s speed), Fast-mode Plus ($\leq$ 1 Mbit/s speed), and High-speed mode ($\leq$ 3.4 Mbit/s speed) speed categories are downward-compatible, i.e., they allow devices to operate at lower bus speeds. However, the Ultra Fast-mode ($\leq$ 5 Mbit/s speed) is incompatible with the other variants, since it requires a unidirectional bus. [17]

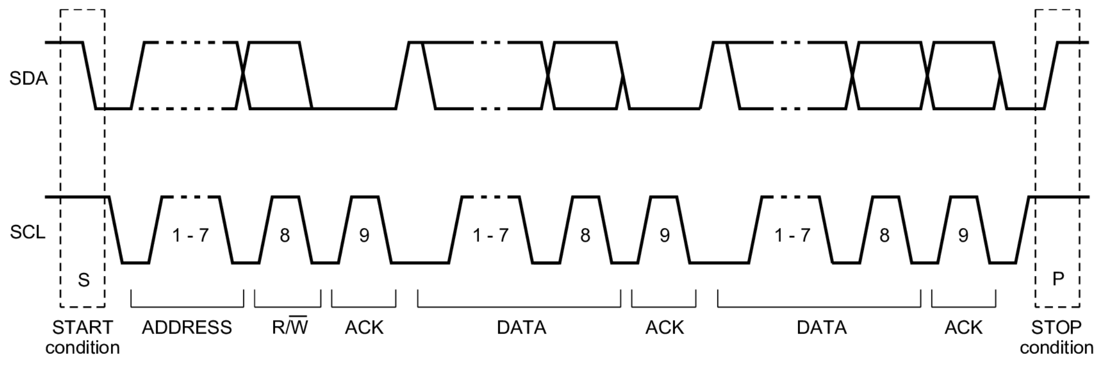


Figure 2.3 An example of a complete I^{2}C data transfer [17].

2.4 Serial Peripheral Interface

The Serial Peripheral Interface (SPI) bus allows duplex, synchronous, and serial communication between devices. It has four dedicated pins:

- **Slave Select ($\overline{\text{SS}}$)**
  The master can either use the $\overline{\text{SS}}$ pin to select which slave should communicate with it or for mode fault detection. The pin is configured as an output in slave selection mode and input in mode fault detection mode. The slave always treats the pin as an input.
- **Serial clock (SCK)**
  In master mode, the device uses this pin to control the data shift registers of the slave device. In slave mode, the SCK pin serves as the clock input.

- **Master out/slave in (MOSI)**
  In normal mode, the MOSI pin transmits data from the slave to the master. In bidirectional mode, the master uses the pin as serial data I/O (master out/master in, MOMI) and the slave does not use the pin.

- **Master in/slave out (MISO)**
  In normal mode, the MISO pin transmits data from the master to the slave. In bidirectional mode, the slave uses the pin as serial data I/O (slave out/slave in, SOSI) and the master does not use the pin.

During data transmission, the devices send and receive data simultaneously, using the serial clock to synchronize the exchange. The master initiates transmissions, determines the transmission speed, and uses the $\overline{\text{SS}}$ pin to select which slave to communicate with. The nonselected slaves ignore any bus activity. See Figure 2.4 for an example of a multislave SPI bus. The clock polarity (CPOL) and clock phase (CPHA) registers allow the SPI modules to operate in four different clock formats.If CPOL = 1, SCK is high in idle state, if CPOL = 0, SCK is low in idle state. If CPHA = 1, data sampling occurs at even edges of the SCK clock, if CPHA = 0, data sampling occurs at odd edges of the SCK clock. [18]

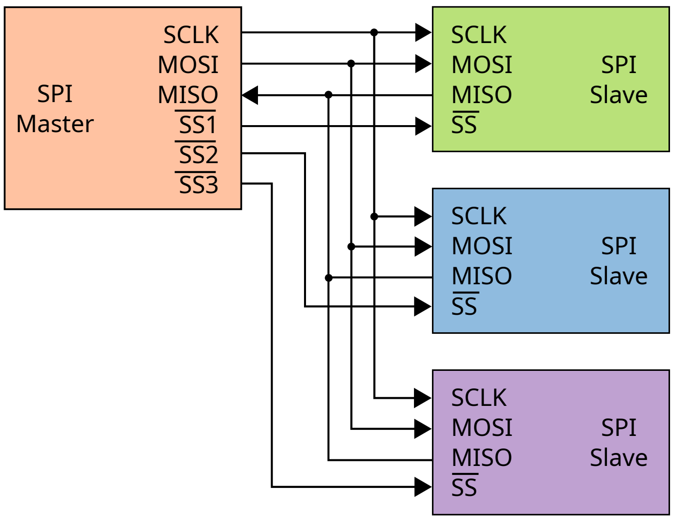


Figure 2.4 SPI bus with one master and three slave devices. The SCK pin is called SCLK in this image. [19]

2.5 Real-time Clock

A real-time clock is a specialized hardware counter for keeping time. It is capable of automatically making time- or date-specific adjustments, e.g., months having a different number of days or leap years. It might also implement additional useful features such as alarms, clock calibration, or battery backup. [20]

The MCU may have a built-in RTC (such is the case with most MCUs from the STM32WB series) [10], alternatively the RTC can be external, in which case it communicates with the MCU through a serial interface (e.g. I^2C, SPI) [21, 22].

2.6 State of the Art

2.6.1 Software in Ready-made Products

There is a notable selection of ready-made nixie clocks on the market. Differences between individual products can range from visual aspects (number and type of tubes used, overall design...) to more technical ones (technology used to set up the device, additional features...). [23, 24] showcase a few different types of clocks. Some sellers offer the option of buying an assembly kit [25]. Customers that already own enough nixie tubes can take advantage of the no-tube variants available [26].

Devices can belong under a specific brand or be unbranded [27]. Software in an unbranded product might be more likely to be of inferior quality. However, if the seller has made the source files and the hardware schematic public, this risk can be mitigated by checking the files.

Research by the thesis author suggests that both groups tend not to publish source codes and hardware designs. Although there are examples of sellers that disclose the materials [28, 29], they appear to be on the scarce side.

Reusing parts of the software in ready-made products could potentially save time during development. However, this advantage is overshadowed by the general issue of low portability of embedded applications. This is because such programs usually need to work directly with the hardware. No ready-made product that claims to use an STM32 microprocessor was found.

2.6.2 Software in Self-made Products

Some people have decided to create their nixie clock from scratch and have documented their efforts. This includes projects with published source code and hardware schematics [30, 31], both of which can serve as inspiration for others. Like with unbranded, ready-made products, it can be hard to evaluate how well-designed the device is without checking the resources. Using other self-made solutions also suffers from the low portability of embedded applications.

There is no project that uses an STM32WB MCU, only a work that uses an MCU from the STM32F1 series was discovered [32]. As this solution diverges in major aspects like not using BLE (or any other wireless technology) for device configuration, it is not suitable as a base for the control software.

2.7 Requirements

This section aims to formalize the requirements the resulting application should meet. The demands consist of the thesis assignment, the constraints required for compatibility with the NixieClock application, the additional functions the Android application implements, the initial draft of a more detailed assignment [33], and any further details the thesis author and the supervisor agree on. It is worth noting that the Google document serves as more of an outline and may not be strictly followed—if the aforementioned parties decide to settle for a solution that contradicts any points mentioned in the document, the agreement takes precedence.

2.7.1 Functional Requirements

Functional requirements, as the name implies, describe the functions and actions the application should be able to perform.

- **Clock and calendar**
  The application should show the time and date that has been set on the device.

- **Timekeeping**
  The application should keep time and date. It should utilize the RTC to meet this condition.
- **Automatic DST management**
  The application should automatically handle DST (daylight saving time) changes.
- **Backup power supply**
  The application should utilize the RTC backup power feature to power the clock when the main supply is off.
- **Alarms**
  The application should allow the user to configure up to 10 alarms. The time and on/off status can be changed separately. When an alarm is triggered, the digits on the display should start blinking, and the device should start ringing. These actions should end after a set period, or the user can end them early by shaking the device.
- **Stopwatch**
  The application should implement a stopwatch function. It should show the current stopwatch time on the display. The user should be able to start, pause, and reset the stopwatch at any time.
- **Timer**
  The application should allow the user to set a timer. It should show the time left on the display. The user should be able to start, pause, and reset the timer whenever desired. When the timer reaches zero, the application should notify the user by blinking the digits and buzzing. The notification should end after a set period, or the user can dismiss it by shaking the device.
- **Parameter configuration**
  The application should allow the user to change the following settings: LED blink interval, time display duration, date display duration, alarm ring duration, timer ring duration, and date format.
- **Nixie tube display operation**
  The application should use the nixie tube display to show any desired data.
- **LED operation**
  The application should use LEDs to indicate the type of data currently shown on the display.
- **Nixie tube detoxification**
  Nixie tubes can suffer from a phenomenon called cathode poisoning. When a cathode is lit up, it releases particles into its surroundings, which get attached to the other digits. If a cathode is not used for a longer period of time, these particles can accumulate into a thicker layer. This causes parts of the digit not to illuminate properly or go completely missing (see Figure 2.5). To prevent this, the application should periodically cycle through all cathodes to remove any buildup on their surface. [34]

2.7.2 Nonfunctional Requirements

Nonfunctional requirements define the attributes of the application and serve as constraints on the design. They may be concerned with qualities such as security, performance, or scalability.

- **Wireless communication technology**
  The application should use BLE to communicate with the device used to configure the embedded system.

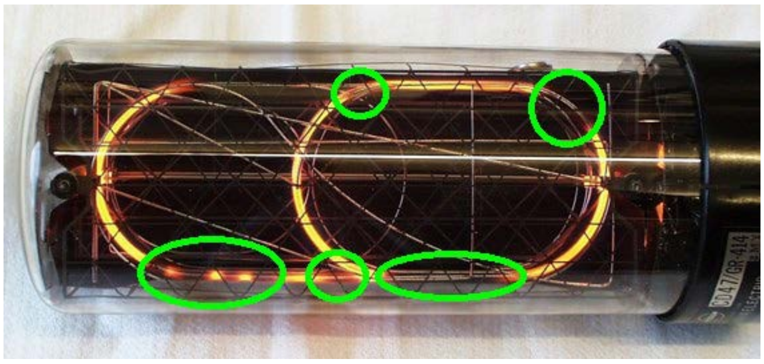


Figure 2.5 A CD47 nixie tube with cathode poisoning. The parts of the digit that are not illumating properly are highlighted in green. [35]

- **Compatibility with the NixieClock application**
  The application should be designed in a way that allows it to be set up by the NixieClock application. This requires the implementation of a GATT server that follows the same structure as the mobile application.

- **Portability**
  The STM32WB microcontroller series is composed of several product lines. As the lines are placed at different price points, the application should be as portable across the different lines as possible. This allows a decrease in the cost of the device.

2.8 Compatibility with NixieClock Android Application

One of the crucial requirements is for the control software to be compatible with the android application developed in the thesis “Control application for nixie clock for mobile phones” [2]. To meet this criterion, the software must follow the wireless communication design described in the corresponding section of the document. This section serves as a summary, the full description can be found in the original document.

The software should utilize the GAP and GATT profiles for BLE communication. The device should act as a GATT server and a Peripheral device [2].

The application should adhere to the GATT and GAP design as defined in [2]. The server should expose its data about the following functionalities: clock, alarms, stopwatch, timer, and settings [2]. A more detailed summary of the design can be found in Appendix B. The advertising packets should contain all service UUIDs and the connection interval [2].

2.8.1 Commentary on BLE Communication Design

The application shall adhere to the GATT server design defined by the Android application thesis. The original design does not interpret some edge cases, so they shall be covered here. If the LED blink interval is set to 0, the LEDs will remain on if the display is showing any data, otherwise they shall be turned off. If the time display duration is set to 0, the device shall not show the time when in clock mode. The same applies to date display duration. If both the date and time display duration are 0, the display shall be empty in clock mode. Turning off ringing for either the alarm or the timer, i.e. setting the respective duration to 0, shall also deactivate the display blink cue. Although the original design does not mention this explicitly, since the timer set time characteristic is write-only, the software design shall assume that any value written to the characteristic is single-use. If a new set time value is received while the timer is running, the timer shall restart with the new value. If the application receives invalid data on the GATT server, it shall rewrite them with valid values. After reviewing the source code for the Android application and the GATT server testing application in [2], it has been found that the Android application transmits data of custom characteristics in big-endian order.

The design mentions that the minimum and maximum connection interval in the peripheral preferred connection parameters characteristic should both have a value of 7.5 ms [2]. The characteristic has two more parameters: the slave latency and the connection supervision timeout multiplier [36]. Since the original design aims to maximize data transfer speed, the latency shall be set to 0. The multiplier value shall be set to 0xFFFF (no specific value requested).

The software cannot fulfill the proposed design of the advertising packet, as the MCUs currently do not support a packet size greater than 31 octets [37]. Although there is a scan response, which can increase the amount of data that can be advertised, available, the size of this packet must also not exceed 31 octets [37]. This is still not enough bytes to be able to advertise all service UUIDs. As a compromise, the application shall at least advertise the UUID of the main clock service, along with additional information.

The ST wireless library functions automatically insert certain data into the advertising packet when setting up the discoverability mode, depending on its type [37]. In addition, the application shall also include the complete local name, appearance, minimum and maximum slave connection interval value, and LE role in the advertising packet. The scan response shall consist of an incomplete list of 128-bit service UUIDs that contains the clock service UUID.

2.9 Manufacturer Resources

The MCU manufacturer offers a variety of additional resources that can be used for application development. The control software itself can mainly benefit from the supplied drivers/middleware and possibly example applications.

There are two categories of hardware drivers: Hardware Abstraction Layer (HAL) and low-layer (LL) drivers. The main features of HAL drivers are their high level of abstraction and portability. LL drivers offer a more optimized API at the cost of decreasing portability. The application can even use both, as long as certain conditions are met. [38]

Since the goal of the control application is to be portable across the STM32WB MCU series, it shall use HAL drivers whenever possible. It will only use LL drivers when the HAL variant does not offer the required functionality.

The manufacturer also offers a wireless interface that is shared across the whole MCU series [37]. For this reason, the control software shall use this interface for BLE manipulation.

2.10 Identified Problems

2.10.1 Alarm Sorting

The design of the GATT server offers 10 alarms that the user can use [2], but the RTC has only two alarms available [20]. Since there are not enough hardware alarms, the software will need to reschedule the alarm. To simplify the rescheduling process, the software shall keep a sorted array of active alarms.

As alarm changes are an asynchronous event from the user, the software shall update the event whenever any of the alarm data changes to keep the array ready for the next reschedule. Since the software changes only one alarm at a time, it adds a new record to a nearly sorted array, and therefore it may be able to take advantage of adaptive sorting algorithms. The small size of the array ($n = 10$) may also impact which algorithm should be chosen. According to [40], straight insertion sort should be a good candidate, as well as the new algorithm developed in the article. The newly proposed algorithm combines straight insertion sort and quickersort with

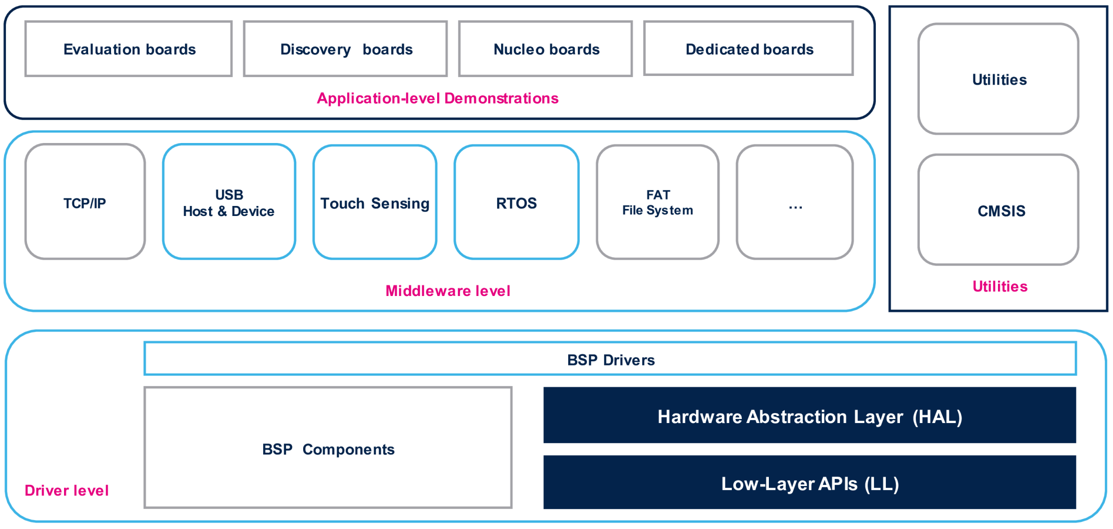


Figure 2.6 ST-supplied resources [39].

merging [40]. The new algorithm does not have a specific name and is only referred to as ”new sorting algorithm” [40]. As both algorithms have the same performance on very small and very nearly sorted arrays [40], the software shall use straight insertion sort, since the other algorithm requires more memory in the form of an additional array [40]. Should the number of alarms be increased in the future, the potential speed increase achieved by using the new algorithm might compensate for the additional memory it requires.

2.10.2 Determination of the Day of the Week

The Czech Republic observes DST: summer time begins at 2:00 on the last Sunday in March and ends at 3:00 on the last Sunday in October. Therefore, the software needs to know the current day of the week to automatically manage DST. However, the GATT Date Time characteristic, which is used to represent the current time and date [2], does not contain this information [41]. As such, the software has to calculate the day of the week from the date. The application can take advantage of the RTC having a day of the week field [6]: it can calculate the day of the week when a characteristic write event occurs on the clock Date Time characteristic and update the relevant field when setting up the RTC with the new time and date value. The RTC will then automatically update the day of the week when necessary [6]. It is worthy of note that not all countries observe DST, and even countries that do observe it might differ in when summer time begins and ends.

The Rata Die algorithm [42] works on a simple basis: after choosing an arbitrary starting date D, the algorithm keeps track of how many days have passed since D, which can be useful for calendrical calculations. One of them is the determination of the day of the week. Since the algorithm knows which day of the week D is, it can determine the day of the week of other dates by calculating how many days have passed since D to the date and then using $\bmod 7$ on the day count to get the day of the week.

Another method of determining the day of the week is using Zeller’s congruence. The formula goes as follows [43]:

$$h = (q + \left\lfloor \frac{26(m+1)}{10} \right\rfloor + K + \left\lfloor \frac{K}{4} \right\rfloor + \left\lfloor \frac{J}{4} \right\rfloor - 2J) \bmod 7 \tag{2.1}$$

The variables in the formula have the following meaning [43]:

q the day of the month

m the month; January and February are considered the 13th and 14th month of the previous year respectively

K tens and units of the year ($year \bmod 100$)

J year hundreds ($\lfloor \frac{year}{100} \rfloor$)

h the day of the week the date falls on; $0 = \text{Sat}$, $1 = \text{Sun}$, ...

The formula works on Gregorian dates [43]. Zeller has also supplied a version for Julian dates [43], but since the Date Time characteristic assumes a Gregorian date [41], only the Gregorian version is needed.

Since the application does not benefit from other uses of day counting besides the day of the week determination, the application shall use Zeller's congruence. The congruence should be easier to implement than Rata Die, as figuring out how many days have passed since D might be a more complex task due to uneven days in months and leap years. The congruence formula shall be used with a minor modification, since it is to be used in computer code. The expression in the parantheses may return a negative value for some dates (e.g., 2002-04-19), which may complicate the implementation, since many programming languages do not have a modulus operator, only a remainder one. The formula can be modified to an equivalent version that does not contain any subtractions using modular arithmetic. Since $(a+b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$ [44], $(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$ [44], and $-2 \equiv 5 \pmod 7$, the formula can be modified to:

$$h = (q + \left\lfloor \frac{26(m+1)}{10} \right\rfloor + K + \left\lfloor \frac{K}{4} \right\rfloor + \left\lfloor \frac{J}{4} \right\rfloor + 5J) \bmod 7 \tag{2.2}$$

2.11 Chapter Summary

First, the analysis described the characteristics of the target embedded device. The embedded system uses an STM32WB MCU, which is a dual-core MCU capable of wireless communication, including BLE. The display consists of six nixie tubes that can be controlled by a pair of serial-to-parallel converters. Additionally, the device has four LEDs, two external oscillators, a buzzer, an EEPROM, and an accelerometer. The system also implements a power backup system, allowing it to drive the LSE and the RTC during power outages. Due to chip shortages, some devices might use different component models.

The analysis then covered the relevant technology: BLE, I^2C, SPI, and RTC. After that, the analysis described the state of the art. As no existing solution that is similar enough to serve as a basis for the control software was found, the software will be made from scratch. The next topic was the requirements the control software should meet. The functional requirements described the expected functionalities of the software, and the non-functional requirements named the constraints the software should adhere to. The analysis then focused on what it means to be compatible with the Android application created in [2] and gave feedback on the wireless communication design. After that, the analysis covered the manufacturer resources that can be take advantage of during development. The application will use the HAL drivers whenever possible to maximize its portability. Lastly, the analysis identified the problems the control software has to resolve and suggested an appropriate algorithm for each issue. The software will use the Insertion Sort algorithm for sorting alarms and a slightly modified version of Zeller's congruence to calculate the day of the week from the date.

Chapter 3

Implementation

This chapter covers the implementation process. The first steps consisted of resolving crucial elements such as display and RTC manipulation. After that, a minimalistic initial prototype of the control software was created with the aim of successfully combining the previous steps together. The prototype was then gradually expanded with the desired functionalities.

3.1 Initial Functional Testing of the Device

The first step in the implementation process was to check whether the device was operational. To begin with, the electrical characteristics of the circuit needed to be inspected. Measuring the PCB revealed that its components are supplied with the desired voltage, and no short circuits were detected.

After that, an MCU programming test had to be performed. LED blinking was an ideal task for this purpose, as it was simple to implement and could be reused in the final application. After consulting the schematic, an initial draft of GPIO configuration was created and used to generate the initialization code. In the main program loop, the application turned all the LEDs on, waited for a certain time period (e.g., 1 s), then turned all the LEDs off and waited once again.

When the program was done, it had to be flashed into the device with a programmer. One of the options available is the STLINK-V3MINI programmer (see Figure 3.1), which was the programmer of choice for this thesis. After the program was successfully uploaded, it was found that while the LEDs blinked in the correct interval, the light they emitted was notably weaker than it should have been. After rechecking the initialization settings, it was discovered that the issue was with setting the pins to open-drain mode, which resulted in them not producing sufficient voltage to drive the LEDs properly. After the GPIO mode was changed to push-pull, the LEDs lit up properly. See Figure 3.2 for the difference between open-drain and push-pull outputs.

3.2 Display Manipulation

The next step was to find out how to manipulate the nixie tube display correctly. The first subtask was to configure the SPI peripheral, as it is used to communicate with the display. Certain settings had to be chosen based on the converter (e.g., CPOL, CPHA), while some were up to the developer to choose (e.g., bit order). After analyzing the waveform diagram of the converters [9, 5], CPOL was set to 0 and CPHA to 1. The SPI data transfer size can be set to 4–16 bits [6]. 16-bit size minimizes the number of transfers needed to rewrite the

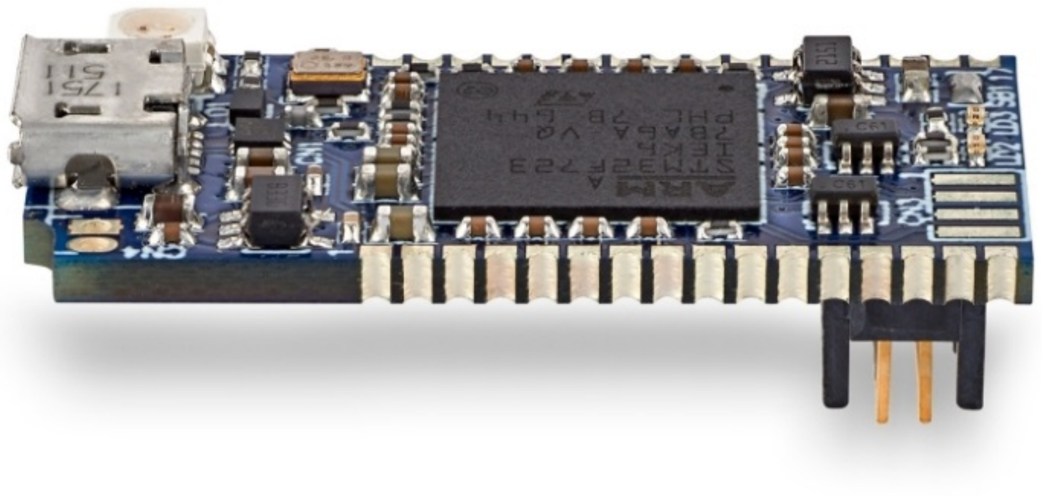


Figure 3.1 The STLINK-V3MINI programmer [45].

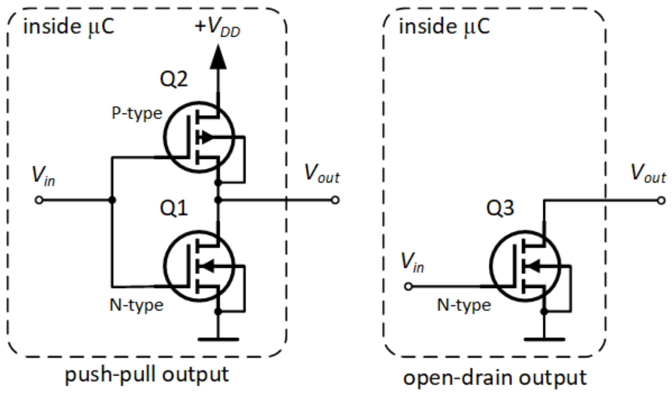


Figure 3.2 Push-pull vs open-drain output [46].

display and divides the bit size of the display data without a remainder, making it the optimal pick. Since the shift register output of the first converter is connected to the shift register input of the second converter, the two can be thought of as a single 64-bit register. As such, the application can construct the display data in a single 64-bit variable. For the same reason, the SPI was configured to send data in MSB order, allowing the software variable to better reflect the hardware implementation (the lowest bit of the variable corresponds to the lowest bit of the ”64-bit” shift register, . . .). The application also had to manipulate the latch and the blank pin appropriately. Display blanking was not necessary at the time, but latching was used to prevent visual noise when changing display data.

To test the configuration, a simple program was made: it slowly shifted a bit through all the shift register positions, allowing the developer to check whether the appropriate digit lit up. Running the application showed that the digits were not being cycled through correctly: not only did they light up in the wrong order, but sometimes two different digits were active at the same time. To better determine the underlying issue, a logic analyzer was used to check the signals sent to the display. Checking the signals revealed that the actual data sent to the display did not match the data supplied to the HAL (Hardware Abstraction Layer API) SPI function. To find out whether the problem was with the circuit or the MCU, an alternative version of the application was created: instead of using the dedicated SPI peripheral, the signals were generated using bit-banging. The bit-banging variant worked as intended, narrowing the potential culprit to either the SPI peripheral or the software. To determine whether the SPI peripheral was working correctly, one more variant of the application was created using LL (Low-Layer API) instead of HAL. The LL version also worked correctly, which meant that the initial version not working correctly was related to the usage of HAL. After further experimenting, it was discovered that the HAL version was not working correctly because the HAL SPI transmit function did not receive the data in the correct format. The data parameter is defined as uint8_t* [38], so the application originally split the data into a byte array before passing it to the function. However, it was discovered that the data format depends on the transfer bit size. E.g., for 16-bit transfers, the function behaves correctly if it is passed an uint16_t array, while it malfunctions if it passed an uint8_t array. The situation is reversed for 8-bit transfers: passing an uint8_t array results in the appropriate behavior, but passing an uint16_t array breaks the function.

The final part of the SPI configuration tests was to determine whether it was better to set the latch pin as a regular GPIO pin or as the SPI $\overline{\text{SS}}$ pin. After examining the behavior of both, the regular GPIO pin solution was selected. The issue with the SPI variant was that it unlatched the display after each data transfer, which is undesirable, because transfers only transmit 4–16 bits at a time [6], making it impossible to rewrite the shift registers in a single transfer.

With the SPI configuration completed, it was time to implement proper display manipulation. To light up a specific digit on a specific tube, the application needs to know the bit position that is responsible for manipulating that digit on that tube. The two display converters have 64 channels in total, while the display only has 60 digits, which means that some channels do not control any digit. The application can take advantage of the sequence in which the digits are ordered to account for the four unused channels. The bits corresponding to the digits of a specific tube are always in succession, and the order of the digits within the bit range of a tube is always the same (Table 3.1 and Table 3.2 illustrate this pattern). As such, the position offset can be split into two numbers: the offset of the first of the 10 bits corresponding to a tube, and the offset of the specific digit from the first tube bit.

To test whether the offsets were calculated correctly, a simple program was made: it cycled through the digits and showed that digit on all tubes at once (i.e., light up digit 1 on all tubes, light up digit 2 on all tubes, . . .). The results showed that the solution was mostly functional, but some offsets were not counted correctly. After replacing the faulty values, the solution worked as expected.

	OPT1	OPT2	OPT3	OPT4	OPT5	OPT6
Digit 1	2	12	22	34	44	54
Digit 2	3	13	23	35	45	55
Digit 3	4	14	24	36	46	56
Digit 4	5	15	25	37	47	57
Digit 5	6	16	26	38	48	58
Digit 6	7	17	27	39	49	59
Digit 7	8	18	28	40	50	60
Digit 8	9	19	29	41	51	61
Digit 9	10	20	30	42	52	62
Digit 0	11	21	31	43	53	63

Table 3.1 Bit offsets for the display digits on devices using the HV5623 converter.

	OPT1	OPT2	OPT3	OPT4	OPT5	OPT6
Digit 1	29	19	9	61	51	41
Digit 2	28	18	8	60	50	40
Digit 3	27	17	7	59	49	39
Digit 4	26	16	6	58	48	38
Digit 5	25	15	5	57	47	37
Digit 6	24	14	4	56	46	36
Digit 7	23	13	3	55	45	35
Digit 8	22	12	2	54	44	34
Digit 9	21	11	1	53	43	33
Digit 0	20	10	0	52	42	32

Table 3.2 Bit offsets for the display digits on devices using the HV5523 converter.

3.3 RTC Manipulation

One of the crucial requirements is that the device keeps time and date accurately using the RTC. The LSE is the only RTC clock source that can be supplied by the power backup mechanism [6]. Since this feature will be essential for the RTC power backup functionality, the LSE was chosen to clock the RTC. The asynchronous (PREDIV_A) and synchronous (PREDIV_S) prescalers must have the appropriate values for the RTC to keep time properly; specifically, they must fulfill the following formula: $\frac{f_{RTCCLK}}{(PREDIV\_A+1)(PREDIV\_S+1)} = 1$ [6]. If both prescalers are necessary, the configuration should aim for the highest possible PREDIV_A value to minimize power consumption [6]. Taking the aforementioned points and the LSE frequency (32.678 Hz) in consideration, PREDIV_A is set to 127 and PREDIV_S to 255. The current aim was merely to assess if the RTC kept time correctly, so the time and date registers were kept at the default values.

To test the RTC configuration, a program was made that would print the new time and date value whenever the RTC counter got incremented. To keep the prints synchronized with the RTC, the application can make use of the RTC wake-up flag and its corresponding interrupts. The 1 Hz wake-up frequency was chosen, as it allows the application to immediately react to the RTC counter updating, while keeping the number of interrupt events at the minimum value needed. After verifying that the application prints new values at the correct interval (i.e., every second), the program was left running for a few hours so that the print output could be checked for any inconsistencies in RTC counter incrementation. The output did not indicate any problems, and the RTC appeared to work correctly.

3.4 Running an Example BLE Application

Another important criterion is that the control software is capable of communicating via BLE. One of the manufacturer-provided example applications was generated to test radio communication. However, the program was not working correctly. After consulting STForums, multiple issues were found. The first problem was that the author of the thesis did not generate some of the necessary interrupts, resulting in the application crashing. For BLE communication to work, the application needs to implement the HSEM (hardware semaphore) interrupt, the IPCC (inter-processor communication controller) RX and TX interrupts, and the RTC wake-up interrupt. The second problem was that the author of the thesis did not correctly configure the RF wake-up clock source. RF wake-up can either be clocked by the prescaled HSE or the LSE. The chosen clock source must be reflected in the `CFG_BLE_LSE_SOURCE` macro in the `app_conf.h` file. Lastly, it was discovered that the secondary MCU core (Cortex-M0 based) must be flashed with an appropriate wireless stack for BLE applications to work properly (see Appendix E.1). After the aforementioned issues were resolved, the example application started working as expected.

3.5 Building the First Prototype

After resolving the previous steps, it was time to slowly start building the control software. The goal of the first prototype was to show the current time and to implement the GATT server. This iteration did not yet need to react to the user changing data in the server; the aim of this prototype was just to test whether the server data can be read and set.

The initialization generation tool is capable of generating most of the BLE source code, but it is not suitable in this case, as the tool can only generate up to five services with up to five characteristics, which is not enough to meet the server design. To save time, the control software will reuse the BLE source files from the GATT server emulator developed in the Android application thesis [2]. The source files can be found in the `STM32_WPAN/App` folder. Most of the files were left as-is, excluding some minor refactoring. The main exceptions are the `custom_app` files: these are meant to implement the BLE application and will therefore undergo extensive changes over the course of the implementation. The manufacturer also offers additional utilities for BLE applications: the sequencer implements a simple background scheduling function, while the timer server provides virtual timers for the application to use [47]. The application makes use of the sequencer, as it would have to implement a similar functionality otherwise. However, the timer server will not be used if possible, as the dedicated timers should offer better precision.

Since the application will grow larger as more functionalities are implemented, an application controller structure was added to store any data needed for the program to behave correctly. To show the current time, the application has to implement a date/time update task and a display update task. The date/time update task updates any relevant application data, update the clock GATT service, and schedule a display update. To keep the date/time update task synchronized with RTC counter updates, the application uses the RTC wake-up interrupt to schedule said task. The display update task has to rewrite the display data according to the state of the application. In this iteration, the display was always in clock mode, and the clock state was set to time display.

The application also needs to react to data changes in the GATT server. When the server software receives a write command, it creates an application notification containing the relevant information such as the characteristic being changed, the data itself, or data length. The application then processes said notification: it validates the data if necessary and updates the corresponding application data.

Running the application revealed that the software did not update the display correctly. The application did not work properly because the timer server uses the RTC wake-up flag to measure time, placing some constraints on the application even when the server is not running any timers. Firstly, the timer server does not support the 1 Hz wake-up frequency option, which was what the application was using. The server only supports the $\frac{RTCCLK}{2}$, $\frac{RTCCLK}{4}$, $\frac{RTCCLK}{8}$ and $\frac{RTCCLK}{16}$ wake-up frequencies. Secondly, the timer server automatically turns off the RTC wake-up flag functionality if there is no active timer running on the server. To fix the problem, the application switched to the $\frac{RTCCLK}{16}$ wake-up frequency, which invokes the least extra interrupts out of the viable options, and created a dummy timer on the timer server. The `CFG_RTC_WUCKSEL_DIVIDER` macro in `app_conf.h` must match the selected wake-up frequency. To test whether the GATT server and the notifications work correctly, the application printed out the received data whenever it received a notification. The tests revealed that the notifications and the GATT server behave as intended.

3.6 Showing the Date, Date Format

The next step was to switch from only displaying the time to displaying both the time and the date. The switching was done with a timer: the timer can be configured to measure the time period the specific data type should be displayed for, and when said period elapses, the timer can change the clock state and schedule a display update. TIM2 was chosen to manipulate the display, as it has the biggest autoreload register (ARR) out of the available timers [6], allowing for more precise time measurements. To measure the desired period, the application takes advantage of ARR being large enough to hold the time/date display period by setting ARR to the respective period and the prescaler to a millisecond's worth of timer clock ticks (all time periods in the GATT settings service are in ms). To show the date in the correct format, the application checks which format should be used and adjusts the display data calculation accordingly. Specifically, the application changes the tube positions of the day of the month, the month, and the year to fit the selected date format. Running the application revealed that the application switched between showing the time and the date in correct intervals, the display data matched the RTC counter, and that the software showed the date in the selected format. Additionally, the application reacted to changes in the respective GATT characteristics appropriately.

3.7 LED Blinking

To avoid using the TIM1 timer, which is the only timer with access to the buzzer pin, and the TIM16/TIM17 timers, which are not present on all STM32WB lines, the application uses one of the low-power timers, specifically LPTIM1. However, even when at the smallest timer frequency, the low-power timers do not have large enough registers to measure the larger LED blink interval values in a single run. To remedy the issue, the application calculates how many timer ticks correspond to the desired time period and then keeps track of how many ticks are left until the period elapses. When the counter hits zero, the application changes the LED state and schedules the LED update task, which turns the LEDs off/on according to the LED state and the data being displayed. To make it easier to distinguish between the time and date in clock mode, the application only turns on the lower two LEDs when showing the date. If LED blinking is turned off, the LEDs are left on if the display is showing any data, otherwise they are turned off. Running the application showed that LED blinking worked correctly, and changing the interval value in the GATT server resulted in the application updating the timer to work with the new value.

3.8 RTC Power Backup

At start-up, the application should check whether the power backup system RTC was kept running while the device was powered off. The application checks whether both the RTC and the LSE are running (LSEON, LSERDY and RTCON bits in the RCC_BDCR register [6]) at start-up and skips their initialization if they are, allowing the device to retain the configured time and date. Testing the application revealed that on some devices, the program would restart the RTC despite being charged for extended periods of time. It was discovered that the MCU has a dedicated register for activating power backup (bits VBE and VBRS of the PWR_CR4 register [6]) that the application did not enable. After the shortcoming was fixed, the backup mechanism worked correctly.

3.9 GAP and Advertising Adjustments

There were some adjustments left to do on BLE communication: the Peripheral Preferred Connection Parameters (PPCP) characteristic, the scan response and the advertising data had not been configured properly yet. PPCP initialization should be done in the GAP initialization function (`Ble_Hci_Gap_Gatt_Init` function in `app_ble.c`), as it is a GAP characteristic. The function now updates the characteristic data with the values defined in Section 2.8.1. Advertising data can be set up using the function responsible for activating discoverable mode [37]. The scan response can be set up in a similar manner—the wireless interface offers a function that can fill the scan response with the desired data [37]. In the `Adv_Request` function in `app_ble.c`, which is in charge of starting the advertising process, the application sets both the scan response and advertising data. After the aforementioned changes were added, the PPCP, the scan response and the advertising packets contained the desired data.

3.10 Alarms

The program uses one of the RTC alarms for the user alarms. To make the alarm ring at the desired time, the application sets the time registers accordingly and takes advantage of the alarm masks to ignore the date registers. When the RTC alarm time and the current time match, the alarm raises a flag, allowing the application to react accordingly. When an alarm gets triggered, the application should notify the user by buzzing and flashing the display. It also needs to reschedule the RTC alarm to the next user alarm.

The application uses two arrays to store alarm data. The first one stores the active alarms, where a record consists of the alarm time and a counter of how many alarms have been set up for said time. The second one is a copy of GATT server data, which is needed to update the active alarms array correctly (e.g., when the time on an active alarm is changed, the application needs to remove the previous value from active alarms first).

Two timers are required to implement the buzzing: the first one is in charge of generating the PWM signal and the second one measures the ringing period. Since TIM1 is the only timer with access to the buzzer pin, it has to be used for PWM generation. The timer generates a 2300 Hz signal, because the buzzer performs at its highest sound level at said PWM frequency [48]. That means that only LPTIM2 remains for measuring the ringing period. Using LPTIM2 has the same problem encountered in LED blinking—since the timer cannot measure larger periods of time in one run, the application has to keep track of the elapsed timer ticks here as well. The application should also flash the display during alarm notifications. If LED blinking is turned on, the flashing is synchronized with the LEDs. If the blinking is turned off, the application reuses the LED timer to flash the display for the duration of the buzzing. If a new alarm trigger event occurs while the ringing is still on, the new event terminates the previous ringing early and restarts it.

After running the application, it was discovered that the alarms did not trigger notifications successfully. To simplify the process of debugging, a different, simpler program was made to test the RTC alarm configuration. Experimenting with the testing program revealed that the RTC alarms were set up correctly, but adding BLE code prevents the RTC alarm interrupt from triggering. The issue was encountered in all of the tested example BLE applications. As the root cause was not found, a workaround was implemented: since the hardware interrupt flags work properly, the date/time update task can check the flag and call the interrupt function manually if needed. Adding the workaround resulted in the application working as intended.

To allow the user to deactivate the ringing early without having to use the Android application, the application uses an accelerometer to detect movement. The user can then move the device to indicate that the ringing should be turned off. The accelerometer can detect (in)activity in general, and it can also be configured to react to more complex types of movement such as single/double-tap. The application does not need to concern itself with how the user moves the device, it merely needs the information that the device is being moved. As such, the accelerometer was configured to react to activity on any of its axes. When the activity threshold is crossed on any axis, the accelerometer generates an interrupt. Setting the two accelerometer pins in interrupt mode allows the application to react to the user shaking the device with minimal delay.

Testing the accelerometer configuration in a separate program showed that the EXTI interrupts also do not work with BLE applications. To remedy the issue in the control software, the date/time update task also checks the interrupt pins and calls the corresponding interrupt if needed. It was also found that the device only called the interrupt on the first activity event and ignored any subsequent ones. When an interrupt gets triggered on the accelerometer, the accelerometer latches the interrupt pin output until certain registers are read. Since the application did not read any of said registers, the pin output would not return to the idle state, so the test application was unable to detect any successive activity events. The program was changed to read one of the registers after an interrupt gets triggered, which resulted in the desired behavior. As such, the configuration was applied to the control software, where no further issues were discovered.

3.11 Tube Detoxification

After analyzing the equivalent functionality in other nixie clocks (see Section 2.6), it was observed that the detoxification routine is often done on the frequent side. On the other hand, the routine should not take too long, as it might interfere with the user trying to use the device. As a compromise between the two, the application sets the second RTC alarm to trigger every hour, after which it begins the detoxification procedure: in cca 30 seconds, it cycles through all the digits to remove any buildup on the cathodes. Since the device cannot show the time/date and detoxify the tubes at once, the application reuses the TIM2 timer to change the digit every 500 ms. Detoxification places no particular constrains on how to configure the PSC and ARR registers, so the application just sets PSC to 0 and ARR to 500 ms's worth of ticks. Instead of staying on a single digit for longer periods of time, the application lights up a digit multiple times for shorter periods, as the latter option makes it easier to indicate that the behavior is intentional and that the device has not malfunctioned. The second RTC alarm suffers from the same interrupt issues as the first one, and the same workaround is used here. Running the application revealed no problems with the detoxification functionality.

3.12 Daylight Saving Time Management

As mentioned in Section 2.10.2, the Czech Republic observes DST, so it was decided that the application should automatically manage the DST changes. Automatic DST is more comfortable for the user, as they do not have to remember to update the clock manually. As DST observation differs across the globe, this section of the application might have to be adjusted for the country the clock will be used in. The application uses the dedicated RTC DST registers to change the time. The ADD1H and SUB1H are write-only registers that add/subtract one hour from the current time respectively [6]. The advantage of using them is that they do not require the RTC to enter initialization mode to be used, allowing the application to change the time without having to stop the RTC [6]. Additionally, the RTC contains a bit that the application can use to store DST status [6]. The program takes advantage of the hourly RTC alarm already used for tube detoxification and also performs a DST check.

The April–September month range always falls under summer time; likewise, the November–February range can fall under standard time. March and October require further inspection, as they can belong in either category. If it is not the last week of the month, the DST change has not yet happened. In the last week of the month, the application has to check whether the current day is Sunday and whether the hour matches the times when the change should occur. When both conditions are met, the application changes the DST accordingly. Additionally, the program has to account for the possibility that the DST change occurred while the device was off. To determine whether that is the case, the application checks whether the next Sunday overflows into the next month or not. If it does, the DST change has already happened, and the application needs to change the RTC time accordingly.

Testing the application revealed that the DST change occurred at the designated time, but the application also kept adding/subtracting an hour at every hour mark after the DST change. It was discovered that the thesis author misunderstood how the DST registers should be used; after adjusting the application logic to reflect the intended usage, the application worked as expected.

3.13 Stopwatch

Since there is only one display, the application reuses TIM2, which was only used to switch between displaying the time and the date so far, for the stopwatch. Using TIM2 also reduces the number of timers the application requires to run. When in stopwatch mode, the timer sets PSC to 0 and ARR to one second's worth of timer clock ticks. Minimizing the PSC value allows the application to store the subsecond count when the stopwatch gets paused. The current prescaler count cannot be accessed [6], the count will be lost if the device has to switch to a different mode. Setting PSC to 0 results in clock ticks immediately propagating to the CNT register, whose value can be read and saved for later use. When unpausing the stopwatch, the application can set CNT to the stored value, preserving the subsecond value. After each second, the timer updates the stopwatch time and schedules a display update. For simplicity, the application keeps showing stopwatch data as long as the stopwatch is not turned off. When the stopwatch is turned off, the application resets the time to 0 and returns to clock mode.

The application shows the time in the HH:MM:SS format, and days are displayed in the equivalent number of hours (e.g., 2 days's worth of seconds shows up as 48:00:00). Due to the size of the display, the application only shows the hour tens and units.

Since the stopwatch counts up, the time variable will eventually overflow. The stopwatch time is in seconds, and the data type of the GATT characteristic and its corresponding variable is uint32, so the time will overflow in cca 136 years. The author of the thesis has evaluated that the chance of the user running the stopwatch for such a period of time is minimal, so the application does not perform any special actions in the event of an overflow.

Testing the application revealed that the stopwatch was mostly functional, except that the counter instantly jumped to 1 s instead of starting at 0 s. The skip was caused by the application not clearing the timer update interrupt flag, which is raised during timer initialization. When the timer was started, the flag immediately triggered the update interrupt, which is responsible for incrementing the counter. Clearing the flag before starting the timer resulted in the application acting correctly.

3.14 Timer

In principle, the timer is very similar to the stopwatch. The main differences are that the timer is counting down instead of up and that it should start buzzing and flashing the display when the timer is done. The timer also uses the TIM2 timer to update the counter every second, and the timer registers are configured in the same manner. The application stores the CNT register value for the timer as well, to preserve the subsecond count. Since the stopwatch and the timer functions share the same MCU timer, the application has to define an arbitration process. The application allocates the MCU timer on a first-come-first-serve basis: if one of the two is currently using the MCU timer (i.e., the status of the functionality is "turned on"), application does not allow the other to be started. If the application is in either the timer or stopwatch mode, but the counter is paused, the application allows the other functionality to be started. If both are paused, the application shows the most recently started of the two on the display. For a more consistent behavior, the application checks the status of the other function when either of them is being turned off, and switches to showing the other function if it is in paused mode. As such, the application only shows the clock if both the stopwatch and the timer are turned off (paused does not count as turned off in this context).

The application also uses the same MCU timer for both alarm and timer ringing. If an alarm or the timer gets triggered, any currently running buzzing gets ended early, and the buzzing is started anew, with the ringing duration depending on the ringing source (alarms and timers can be configured for different ringing durations). Testing the application revealed no issues with the timer function.

3.15 EEPROM Manipulation

The preliminary step to managing the EEPROM is evaluating what information to store in it. It has been decided that EEPROM should store the following:

- **Display converter type**
  Storing the converter type allows the application to adjust the calculation of the display SPI data accordingly, without having to recompile the program to match the used converter.

- **Year hundreds**
  The RTC can only store year tens and units [6]. Setting 20 as the default year hundreds value can cover the situation where the user synchronizes the device with the current time reasonably well, but the user might wish to set it to a time that falls outside of the current year hundreds. Storing the year hundreds in the EEPROM allows the application to retain the correct year hundreds value even after shutting down.

- **Data in the GATT settings service**
  Storing the settings allows the application to recover the user-set values after power outages, which is more comfortable for the user.

The aforementioned data should not change often, and storing them brings merit to the application. However, the application does not store the following data in the EEPROM:

- **Alarm times and statuses**

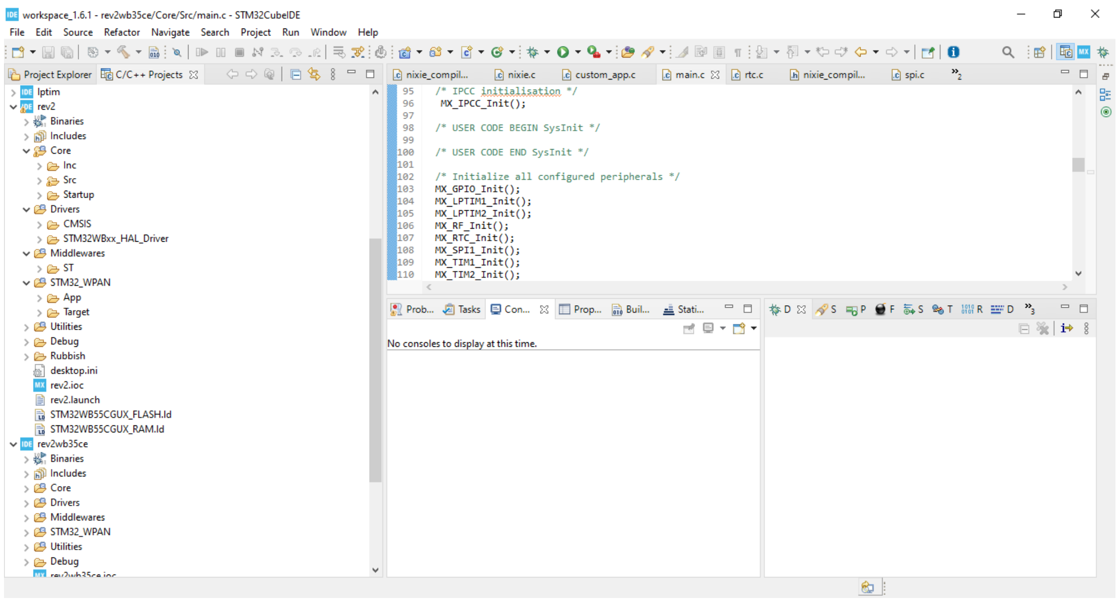
- **Stopwatch time and status**

- **Timer time and status**

The values of the above characteristics are expected to change frequently, so storing them in the EEPROM would negatively impact its lifespan. The converter type gets written in the EEPROM during its initialization only and is treated as read-only afterward. The remaining data are also initialized to the default values used in the application, but they may also need to be updated when the corresponding GATT characteristic changes. To avoid wearing down the EEPROM, the application does not write in the EEPROM if the old value matches the new value. To simplify the application design, the EEPROM stores multibyte data in the same endianness as the GATT server.

The initial version of the EEPROM read/write functions was kept simple: it read from and wrote to the EEPROM byte by byte. Testing the functions revealed that the write function failed when it attempted to write multiple bytes in succession. The EEPROM does not acknowledge the master while it is processing a write request [49]. When the application attempted to write the second byte, the EEPROM did not acknowledge I^2C transmission. It appears that the HAL I^2C function used does not reattempt the transaction, so the function returned a fail status after not getting acknowledged by the EEPROM. Since the write function needs to check the return status for other possible I^2C errors, it was edited to wait until the EEPROM write cycle elapses after each transaction, thus eliminating the EEPROM being busy processing a write request as a possible cause of the error. Adding the delay loop fixed the problems with the EEPROM write function.

After testing the basic version, a more elegant solution was made. The EEPROM allows sequential reads: as long as the MCU keeps responding with ACK (acknowledge), the EEPROM increments the address counter and sends the corresponding byte [49]. Sequential reads do not place any data size limits [49], so the EEPROM read function can just read the desired data in a single transaction. The EEPROM also supports writing multiple bytes at once, but the data size of the transaction must not exceed the page size, as the EEPROM would start rewriting the previously transmitted data [49]. As such, the EEPROM write function has to split the data into multiple transactions if the data size exceeds the page size. Testing the improved version revealed no problems.



```
  /* IPCC initialisation */
   MX_IPCC_Init();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_LPTIM1_Init();
  MX_LPTIM2_Init();
  MX_RF_Init();
  MX_RTC_Init();
  MX_SPI1_Init();
  MX_TIM1_Init();
  MX_TIM2_Init();
```

No consoles to display at this time.

Figure 3.3 STM32CubeIDE (screenshot done by the author of the thesis).

3.16 Development Tools

STMicroelectronics has created a free development environment called the STM32Cube Ecosystem, which will be used to develop the control software. The ecosystem offers four main tools: [50]

- **STM32CubeMX**
  A tool that allows developers to easily generate peripheral initialization code for any of the STMicroelectronics MCUs.
- **STM32CubeIDE**
  An open-source IDE (see Figure 3.3) that includes compilation reporting and advanced debugging features, and it also integrates other tools from the ecosystem (e.g., STM32CubeMX).
- **STM32CubeProgrammer**
  A programming tool that provides an easy-to-use environment for reading, writing, and verifying devices / external memories.
- **STM32CubeMonitor**
  A family of monitoring tools that help developers fine-tune the behavior and performance of their applications in real time.

3.17 Chapter Summary

Before starting any development, the electrical characteristics of the device were checked, and a programming test of the MCU was performed. After confirming that the device was operational, several separate applications were made to test the implementation of the core features of the control software. Specifically, the applications tested display and RTC manipulation and BLE communication. To manipulate the display properly, the application has to use the SPI peripheral correctly and has to be capable of calculating the display data based on which digits the application desires to light up on the nixie tubes. With the RTC properly configured, the time/date gets updated correctly, and the wake-up flag of the peripheral allows the application to update its data in synchrony with the RTC clock source. BLE communication was verified by running one of the example BLE applications supplied by the manufacturer.

The individual applications were then merged into an initial prototype of the control software, which only showed the current RTC time. It was found that the timer server utility that is automatically included in BLE applications places some constraints on RTC configuration, so the RTC settings had to be adjusted accordingly. The software was then gradually expanded with the rest of the functionalities. To switch between showing the time and the date, the software uses the TIM2 timer. Showing the date in the desired format entails storing which format should be used and adjusting the display data calculation accordingly. LED blinking is managed by the LPTIM1 timer. To retain RTC data after power outages, the application checks if the power backup mechanism kept the RTC and the LSE running and skips their initialization power backup was successful. Then, some minor adjustments to the advertisement packet data and the PPCP characteristic were made to fulfill the BLE communication design as mentioned in Section 2.8.1.

The application uses an RTC alarm to implement the user alarm functionality. Since the user can set up more than one alarm, the application has to reschedule the RTC alarm after it gets triggered to ensure the desired behavior. To notify the user that an alarm was triggered, the application starts ringing for a set time period. To implement the ringing, the application uses the TIM1 timer to generate the PWM signal for the buzzer and the LPTIM2 timer to ensure that the ringing ends on its own after the maximum ringing period has elapsed. Additionally, the application flashes the display while the the device is ringing. The user can end the ringing early by moving the device, which generates an accelerometer interrupt. The application uses a different RTC alarm to detoxify the nixie tubes every hour. Since the display the time/date while it is being detoxified, the application reuses the TIM2 timer to cycle through the digits. The application also uses the hourly RTC alarm originally configured for tube detoxification to check for DST changes. If the RTC is not adjusted for DST correctly, the application uses the dedicated RTC DST registers to switch to the appropriate time. Both the stopwatch and the timer use the TIM2 timer to increment/decrement their counter every second respectively, since the display can only show one type of information at once. The timer turns on ringing when the countdown is done. The timer ringing works on the same basis as alarm ringing, except the user can set up different durations for the two. The application also stores certain data on the EEPROM: some are crucial for the application to behave correctly, while some offer QoL changes to the user. The chapter ends with a brief mention of the tools used for development.

Chapter 4
Testing

This chapter covers the testing methodology and the tools used. It then reports any problems discovered during the tests and describes how they were resolved. It also provides feedback on the Android application developed in [2].

4.1 Methodology

Some basic testing was already done during the implementation phase to check for any obvious issues. The bugs discovered during the implementation were covered in the corresponding chapter.

The fact that the correct behavior of the application depends on the MCU's register values makes it difficult to automate tests. As such, the testing was done manually. The tests aimed to cover the base and edge cases of the implemented functionalities.

4.2 Testing Tools

The main tool for testing BLE communication was the Android application created in [2], as it is meant to be the application of choice for the user. A generic BLE scanner application was also used to double-check the GATT server data and to check the aspects the Android application does not show (e.g., the PPCP characteristic and advertisement packet data). The former has to be edited slightly and recompiled to be usable, as it only filters the Nucleo board used during the testing of the Android application. In the `ScannerViewModel.kt` file, line 71 needs to be changed to a more generic address prefix. After observing the default addresses on a few different STM32WB MCUs, the common prefix is 0x00:80:E1, which is the ST company ID according to [47]. Said document also mentions an ID for the WB series, but the number was not encountered in any of the addresses, so using just the company ID prefix is recommended. Assuming that the application should filter addresses starting with 0x00:80:E1, the prefix string on line 71 should be edited to "00:80:E1".

STM32CubeIDE and STM32CubeProgrammer were used on the embedded system itself. The former was used for its debugging capabilities and the latter for its ability to read / write to the peripheral registers.

4.3 Manual Testing

4.3.1 Clock

The clock functionality was tested by setting the GATT characteristic to a variety of different date time values and observing if the RTC data were changed accordingly. After uploading a new value, the device was left running for some time to check for timekeeping issues. It was also verified whether the application rejects invalid date time values. Testing the clock functionality revealed that the application did not update some of its controller data, which resulted in incorrect behavior when the year hundreds increased. The bug was fixed by updating the controller data used to check RTC year overflow.

4.3.2 Alarms, Ringing, Manual Ringing Deactivation

The alarm functionality was tested by setting the characteristics to a variety of times and evaluating whether the application sorted the alarms correctly and whether it rescheduled to the appropriate alarm. The alarms were turned on in sequences that required the application to insert new times (in)to the beginning, the middle, and the end of the array to verify the sorting algorithm. It was also checked whether the application reacted to the time changing on an already active alarm appropriately and whether it rejects invalid time and status values. Lastly, the tests also checked if the application handled duplicate times as intended and if the alarms work correctly regardless of how many and which of the alarms are turned on. The tests showed that the application did not reschedule correctly after the last alarm of the day was triggered if at least one alarm was turned off. The problem was caused by doing boundary checks against the maximum capacity of the array instead of the current number of valid records. The bug was fixed by checking against the current array size.

Ringing tests consisted of verifying that the ringing stayed active for the desired period of time when the timer or one of the alarms was triggered. It was checked if the application set up the correct duration based on the ringing source. They also checked if the display flashing behaves as expected. The tests showed that the application got stuck in timer mode if the timer ringing duration was 0. The bug was fixed by immediately returning the device to clock mode when the timer ends and the duration is set to 0.

Accelerometer tests were done by checking if moving the device ended the alarm/timer ringing early. They also checked if moving the device when it is not currently ringing results in any unexpected behavior. The tests were passed successfully.

4.3.3 Stopwatch, Timer

The stopwatch functionality tests consisted of checking if the application reacts to stopwatch status changes appropriately and that the application increments the counter correctly. They also verified if the application rejects invalid status values. The tests revealed that the application did not consider the display state when pausing / turning off the stopwatch, which could result in inappropriate behavior. The bug was fixed by checking the display state in the relevant functions. The timer functionality was tested similarly to the stopwatch, as the two are not very different implementation-wise. The tests checked if the application reacts to status changes, decrements the counter correctly, and rejects invalid status values. The tests additionally tried setting the GATT set time characteristic to a variety of values to see if the application behaves appropriately. The tests showed that the application did not account for the possibility of starting the timer with the counter set to 0, resulting in an underflow. The problem was fixed by adding a counter value check when starting the timer. They also revealed that the application did not consider the display state when pausing / turning off the timer, just like in the stopwatch functions. The issue was fixed in an analogous manner here, as well.

4.3.4 Tube Detoxification

The tube detoxification functionality was tested by leaving the device on for extended periods of time and verifying that the detoxification occurs every hour for the expected period. The test was passed successfully.

4.3.5 MCU Timer Sharing

Since the stopwatch, timer, and tube detoxification functionalities share the same MCU timer, it was also verified if the timer is allocated between the three appropriately. The tests checked if the application did not allow the timer to be turned on if the stopwatch was running and vice versa, but allowed the other functionality to be started if the current one is paused. They also verified whether detoxification did not interfere with the other two functions if one of them was running. Lastly, the tests checked if turning off the timer/stopwatch made the application switch to showing the other functionality's data if it was currently paused. The tests revealed that the application did not account for the current display state when starting detoxification, which resulted in the timer/stopwatch malfunctioning if it was currently running. The issue was fixed by checking the display state and only starting detoxification if neither of the two is currently running.

4.3.6 RTC Power Backup

RTC power backup was tested by first letting the device charge for a bit and then leaving it turned off for a few hours. The device was then turned on to check whether the RTC counter held the expected value. The tests were passed successfully.

4.3.7 Daylight Saving Time Management

DST management tests consisted of setting the RTC time to a variety of times that were close to a change in DST state. Then it was checked if the RTC time was adjusted correctly when it reached the corresponding DST change mark. The tests also verified whether the application updated the time correctly in a situation where the DST had changed while the device was off. The tests showed that the application did not handle the situation in which the device was turned off and on after DST change from summer to standard time. The summer-to-standard change shifts the hours back by one, making it tricky to evaluate whether the clock is in the expected DST state when the DST check is done at device start-up. After careful consideration, the issue was fixed in the following way: if the current RTC time is 02:00–02:59 and it is the last Sunday in October, the application checks the recorded DST state. If the RTC store operation bit claims that the time was adjusted to standard time, the application assumes that the time was changed correctly. This behavior will result in the DST not changing at start-up if the device was powered off before the change to summer time occurred and the current time is 02:00–02:59 and the date falls on the last Sunday in October. For such a scenario to happen, the capacitor would have to sustain the RTC and the LSE for cca 7 months, but the personal experience of the author of the thesis is that the power backup mechanism does not last for such a long period of time. As such, the probability of such scenario occurring should be minimal. If it does happen, the RTC will hold the wrong time until the DST change takes place.

4.3.8 Display

Time/date display duration tests were done by changing the corresponding GATT characteristics and observing if the application reacted appropriately. The date format tests were also included in this set, as it is closely related to displaying the date. It was evaluated if the date format changes according to the GATT server data and if the application rejects any invalid format values. The tests were passed successfully.

4.3.9 LEDs

LED blinking tests consisted of changing the value and verifying that the blinking interval was updated appropriately. They also checked if the LEDs state gets managed correctly if LED blinking was turned off. The tests revealed that the LEDs were not in the expected state if both the time and display duration were set to 0, the blinking interval was 0, and the display was showing the stopwatch/timer time. The LEDs were supposed to temporarily turn on while the device was in stopwatch/timer mode and then turn off when the device reentered clock mode, but the LEDs remained in the off state even in stopwatch/timer time. The issue was fixed by scheduling a LED update when starting the stopwatch/timer.

4.4 Portability Testing

The development of the application was done on a single embedded system that contains an STM32WB55CGx MCU. Some of the other assembled devices use an STM32WB35CEx MCU instead, allowing some verification of the portability requirement. The STM32CubeIDE version used during the thesis does not support changing the target MCU after creating the project, so a new project was created with the same STM32CubeMX configuration settings (see Appendix E for more detailed steps).

Due to time constraints, only some basic tests were done on the ported version of the application. It was found that the CPOL = 0, CPHA = 1 SPI configuration did not work on the WB35CE system, the settings had to be changed to CPOL = 1, CPHA = 0 for the display to work correctly. However, the WB55CG system behaves correctly with either configuration. As such, the value of CPOL and CPHA might have to be adjusted to fit the particular system.

During the portability test of the control software, it was discovered that the page write variant of the EEPROM write function did not work correctly. The bug was not detected earlier because of presumably improper basic testing during the implementation and because the author of the thesis forgot to perform an additional round of EEPROM tests in the prior stages of testing. For lack of time, the EEPROM write function was reverted to the simpler byte write version and retested.

4.5 RTC Desynchronization

Leaving the RTC to run for extended periods of time showed that the RTC may desynchronize notably due to the ± 20 ppm frequency accuracy of the LSE. To remedy the issue, the application can use the smooth calibration feature of the RTC to adjust the frequency. The CALP and CALM registers allow for digital calibration with a resolution of cca 0.954 ppm with a range from -487.1 ppm to +488.5 ppm [6]. The calibration feature has three different calibration periods available: 8 s, 16 s and 32 s [6]. The 32 s period was chosen as it offers the most accurate frequency adjustment out of the three options.

The first attempted solution was that the device would measure the accuracy and set up the calibration registers accordingly at start-up. The application ran two timers at once: one of the LPTIM timers was run for a time period that was equal to the calibration period (i.e., 32 s). TIM2 was used as a counter. When the LPTIM timer was done, it stopped TIM2. The application then compared the value of the CNT register with the ideal value (i.e. the value the register would have if the LSE had a 32.768 Hz frequency exactly) to calculate the mismatch between the ideal and the measured frequency and set up the calibration registers accordingly. However, testing the calibration revealed that the measured frequency was notably different from the empirically observed frequency. In reality, the device desynchronized by cca 2 s (i.e., the LSE had a cca +20 ppm frequency accuracy), but the calculated imprecision was larger by a noteworthy margin (most measurements yielded values close to or higher than the maximum ppm value the RTC can calibrate). The reason for the measurements being largely imprecise was not found, so the idea was abandoned.

Since the first solution did not provide the desired results, an alternate solution was made: the GATT server was expanded with an additional service that the mobile application can use to set the CALP and CALM registers. The mobile application can measure the frequency imprecision and then calibrate the RTC accordingly. The service was designed as follows:

- Service UUID: 000075e0-f908-44b0-81b9-3450ca663f46
- CALP characteristic
  - UUID: 000075e1-f908-44b0-81b9-3450ca663f46
  - Data type: uint8. The CALP register is a single bit, so only the lowest bit of the characteristic is used. Any unused bits should be set to 0.
  - Properties: read, write
- CALM characteristic
  - UUID: 000075e2-f908-44b0-81b9-3450ca663f46
  - Data type: uint16. The CALM register is 10 bits long, so only the lowest 10 bits of the characteristic are used. Any unused bits should be set to 0. The characteristic uses the same endianness as the rest of the GATT server.
  - Properties: read, write

After the appropriate changes were made to the application, the solution was tested for any deficiencies. An error was found in the function responsible for setting the CALP register according to the characteristic data, which was promptly fixed. Checking the reaction of the Android application to the new GATT server structure revealed that having an additional service does not interfere with the rest of the application.

4.6 Feedback on the NixieClock Application

When setting the date manually, the application does not send the correct month value to the clock, it sends a value that is smaller by 1 instead. The application also does not write new values of the duration/interval characteristics in the settings service correctly. The application reads the value of the characteristics correctly, and when the user attempts to change the value of any of the characteristics, it claims that the characteristic has been updated with the new value. In truth, the write operation does not happen, which can be checked by using a different BLE scanner application. When the user switches from the settings screen to a different one and then back, the NixieClock application correctly shows that the characteristic still has the old value. The stopwatch status button does not update correctly after changing the value. The user has to switch to a different screen and back to force an update. The timer screen has a similar issue.

The application also does not seem to account for the possibility of the user connecting to the clock via a second BLE application at the same time. If the user changes any GATT server data that do not have notifications and the application is currently located on the screen corresponding to said GATT data, the NixieClock application does not reflect said changes until the user switches to a different screen and back. It is expected that the application would behave similarly if two different Android devices were connected to the clock at once. However, this hypothesis was not tested, because the control software does not support multiple connections, as such behavior would be undesirable. There was no mention found about how the application should behave in such situations in the thesis, so it cannot be said with certainty whether this behavior is intentional.

According to [2], the Android application does not show the local device name. However, the application always displayed the correct name while it was used during the implementation and testing of the control software.

It should be noted that since the application had to be edited slightly and recompiled, any aforementioned bugs might have been caused by not building the source code correctly. Should the implementation of the Android application be revisited in the future, it might be worthwhile to check the issues listed above.

4.7 Chapter Summary

Some basic testing was already done during the implementation to check for any major issues. The tests were done manually, because the correct application behavior is dependent on the MCU register values. A slightly modified version of the NixieClock application and a generic BLE scanner were used to verify BLE communication, while tools from the STM32Cube ecosystem were used to check for inconsistencies in the register or software data.

The chapter described how the functionalities were tested and reported any found issues, all of which were fixed afterward. The software was also ported to a system with an MCU different from the one used during development. The basic tests of the ported version revealed that the SPI CPOL and CPHA values might have to be adjusted depending on how the configuration behaves on the specific system. The portability test also revealed an issue that had gone unnoticed in the previous stages of the testing, which was then addressed. It was also found that the frequency error of the LSE could result in RTC desynchronization at significant rates. The attempts to have the MCU measure and calibrate the frequency yielded unsatisfactory results, so the GATT server was expanded with a new service instead. The Android application can use the service to set the RTC smooth calibration registers as needed. Lastly, the chapter provided feedback on the NixieClock application.

Chapter 5
Conclusion

The goal of this thesis, which was to create control software for a custom nixie clock embedded system containing an STM32WB MCU (see Figure 5.1 and Appendix F for photographs of a fully-assembled device), was met. The thesis includes an analysis of the state of the art. No suitable solution was found that could be used as the basis for the software, so it was built from the ground up. To be compatible with the complementary Android application, the software follows the same BLE communication design. The only exception is the advertising data format, which could not be met because of driver/firmware limitations.

The software was developed by implementing smaller subparts and integrating them into the final product. Choosing such a development method meant that component testing could be done during the implementation. After finishing the implementation, a final round of testing was performed. The tests revealed a notable number of problems, such as incorrect edge case handling or faulty boundary checks. A detailed description of the problems found and their fixes can be found in the testing chapter.

The software can be improved in various ways, such as adding more features or optimizing its power consumption. The software can also be ported to different MCUs or reused in similar projects. Any changes to the BLE communication design should be reflected in the Android application.

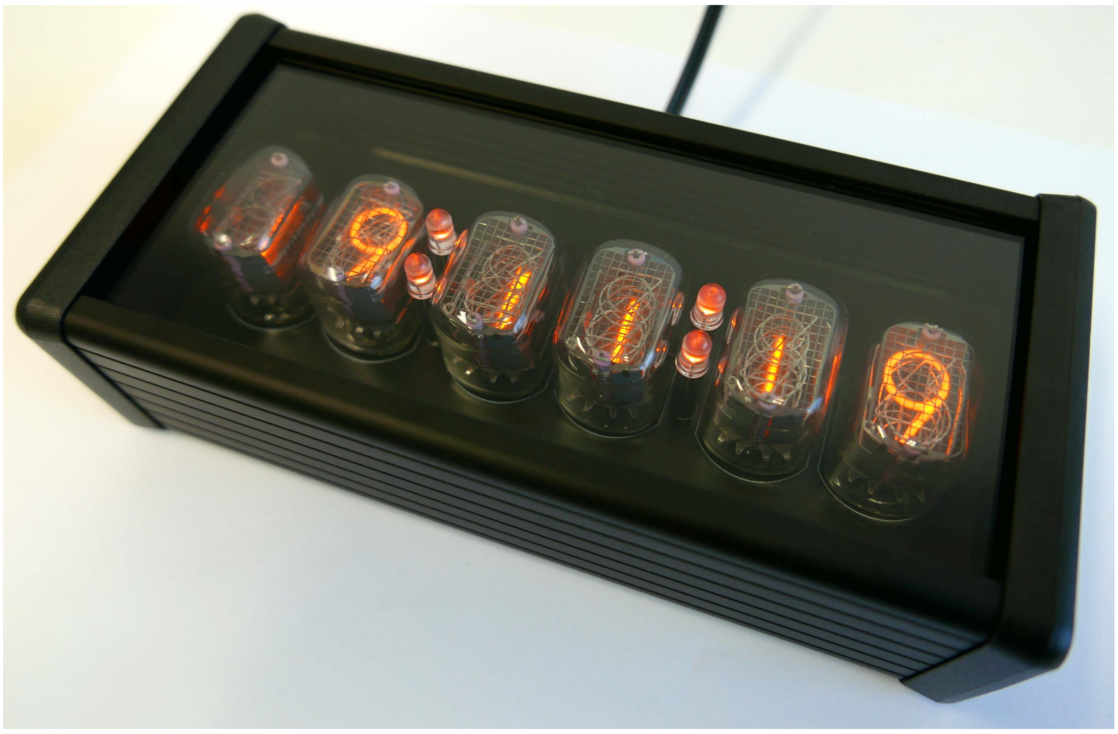


Figure 5.1 A fully-assembled nixie clock device. (© Matěj Bartík)

Appendix A

Additional Photographs of the PCB Board

This appendix consists of additional close-up photographs of the PCB board.

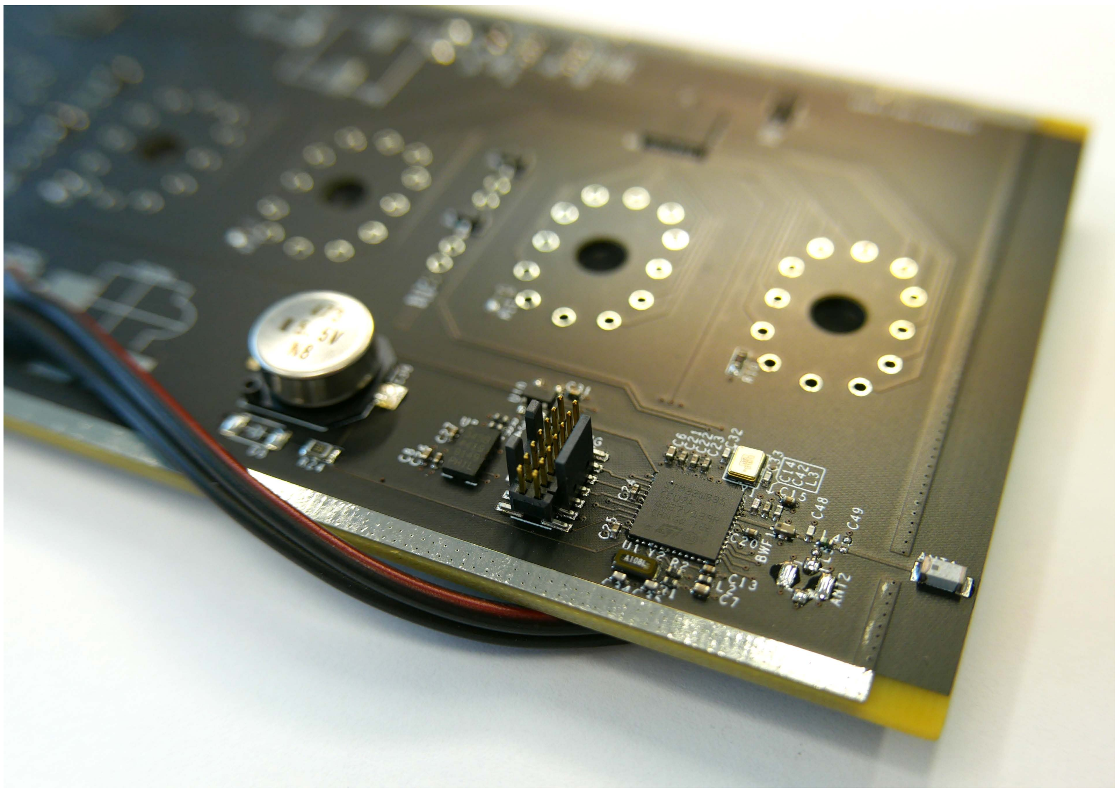


Figure A.1 A close-up image of the MCU segment of the PCB (© Matěj Bartík).

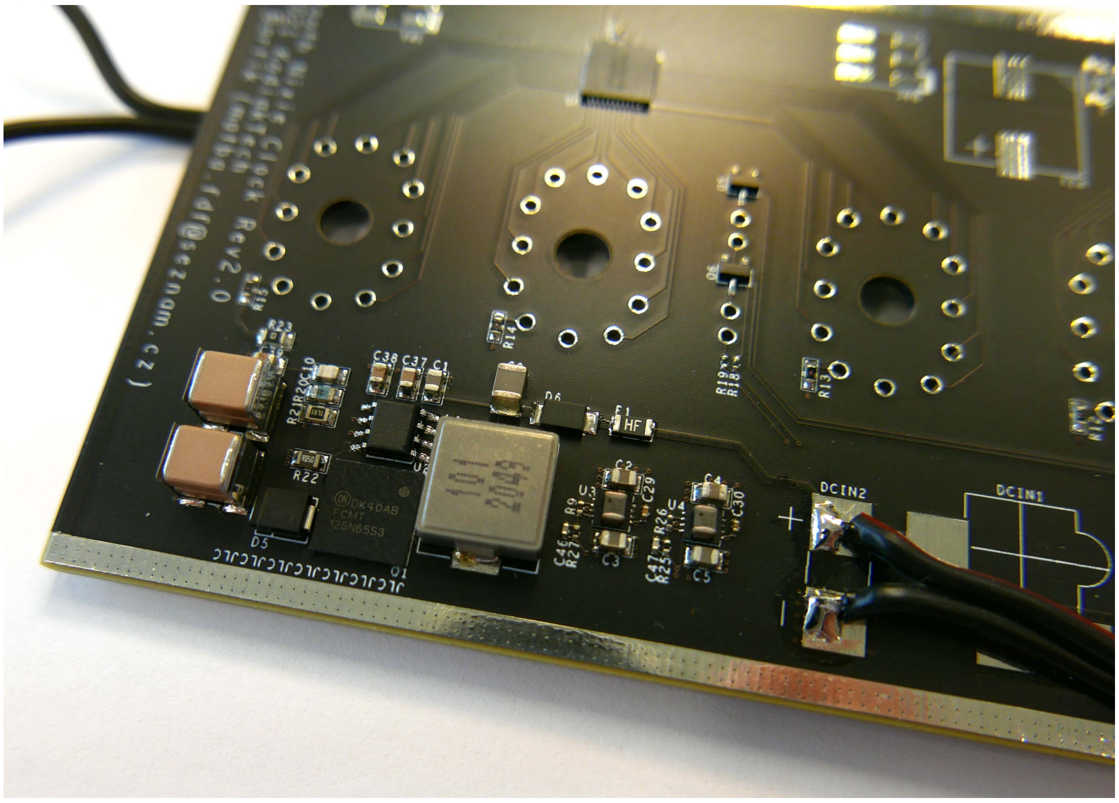


Figure A.2 A close-up image of the power supply segment of the PCB (© Matěj Bartík).

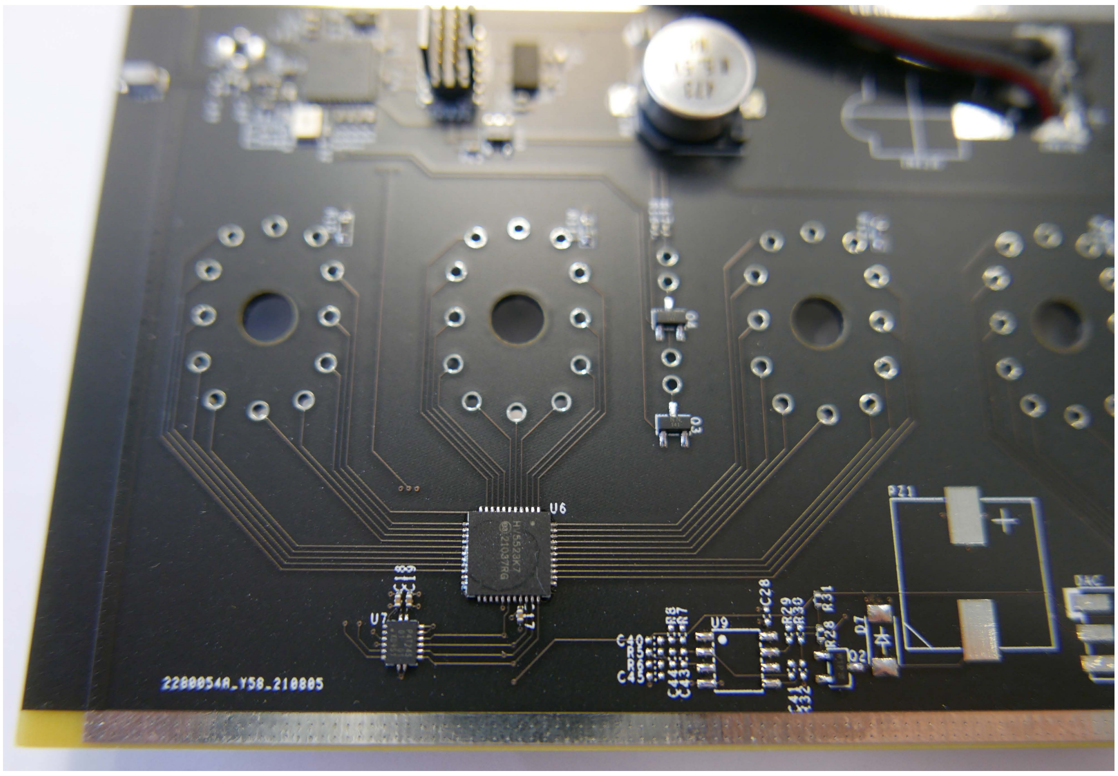


Figure A.3 A close-up image of one of the serial-to-parallel converters that manipulate the nixie tube display. (© Matěj Bartík).

Appendix B

GATT and GAP Design

This appendix serves as a concise summary of GATT and GAP design defined in [2].

The GATT server should have the following structure:

- Clock service
  - Service UUID: 00007500-f908-44b0-81b9-3450ca663f46
  - Date Time characteristic
    - UUID: 0x2A08
    - Data type: Date Time [41]
    - Properties: read, write, notify
- Alarm service (10×)
  - Service UUID: 000075X0-f908-44b0-81b9-3450ca663f46, where X corresponds to the alarm number in hexadecimal notation. The alarms are numbered 1–10 (i.e., 0x1–0xA).
  - Time characteristic
    - UUID: 0x2A08
    - Data type: Date Time [41]
    - Properties: read, write
  - Alarm status characteristic
    - UUID: 00007511-f908-44b0-81b9-3450ca663f46
    - Data type: uint8
    - Properties: read, write
- Stopwatch service
  - Service UUID: 0000075b0-f908-44b0-81b9-3450ca663f46
  - Stopwatch current time characteristic
    - UUID: 000075b1-f908-44b0-81b9-3450ca663f46
    - Data type: uint32
    - Properties: read, notify
  - Stopwatch status characteristic
    - UUID: 000075b2-f908-44b0-81b9-3450ca663f46
    - Data type: uint8

    * Properties: read, write

- Timer service
  - Service UUID: 0000075c0-f908-44b0-81b9-3450ca663f46
  - Timer current time characteristic
    * UUID: 000075c1-f908-44b0-81b9-3450ca663f46
    * Data type: uint32
    * Properties: read, notify
  - Timer set time characteristic
    * UUID: 000075c2-f908-44b0-81b9-3450ca663f46
    * Data type: uint32
    * Properties: write
  - Timer status characteristic
    * UUID: 000075c3-f908-44b0-81b9-3450ca663f46
    * Data type: uint8
    * Properties: read, write, notify
- Settings service
  - Service UUID: 0000075d0-f908-44b0-81b9-3450ca663f46
  - LED blink interval characteristic
    * UUID: 000075d1-f908-44b0-81b9-3450ca663f46
    * Data type: uint32
    * Properties: read, write
  - Time display duration
    * UUID: 000075d2-f908-44b0-81b9-3450ca663f46
    * Data type: uint32
    * Properties: read, write
  - Date display duration
    * UUID: 000075d3-f908-44b0-81b9-3450ca663f46
    * Data type: uint32
    * Properties: read, write
  - Alarm ringing duration
    * UUID: 000075d4-f908-44b0-81b9-3450ca663f46
    * Data type: uint32
    * Properties: read, write
  - Timer ringing duration
    * UUID: 000075d5-f908-44b0-81b9-3450ca663f46
    * Data type: uint32
    * Properties: read, write
  - Date format
    * UUID: 000075d6-f908-44b0-81b9-3450ca663f46
    * Data type: uint8
    * Properties: read, write

The GATT server also contains a dedicated service for the GAP profile. The application should set up the characteristics as:

- Device name characteristic
  - Properties: read
  - Value: the design does not request a specific name
- Appearance characteristic
  - Properties: read
  - Value: 256 (Generic Clock category)

Appendix C
Embedded System Schematic

This appendix shows the schematic of the target embedded system.

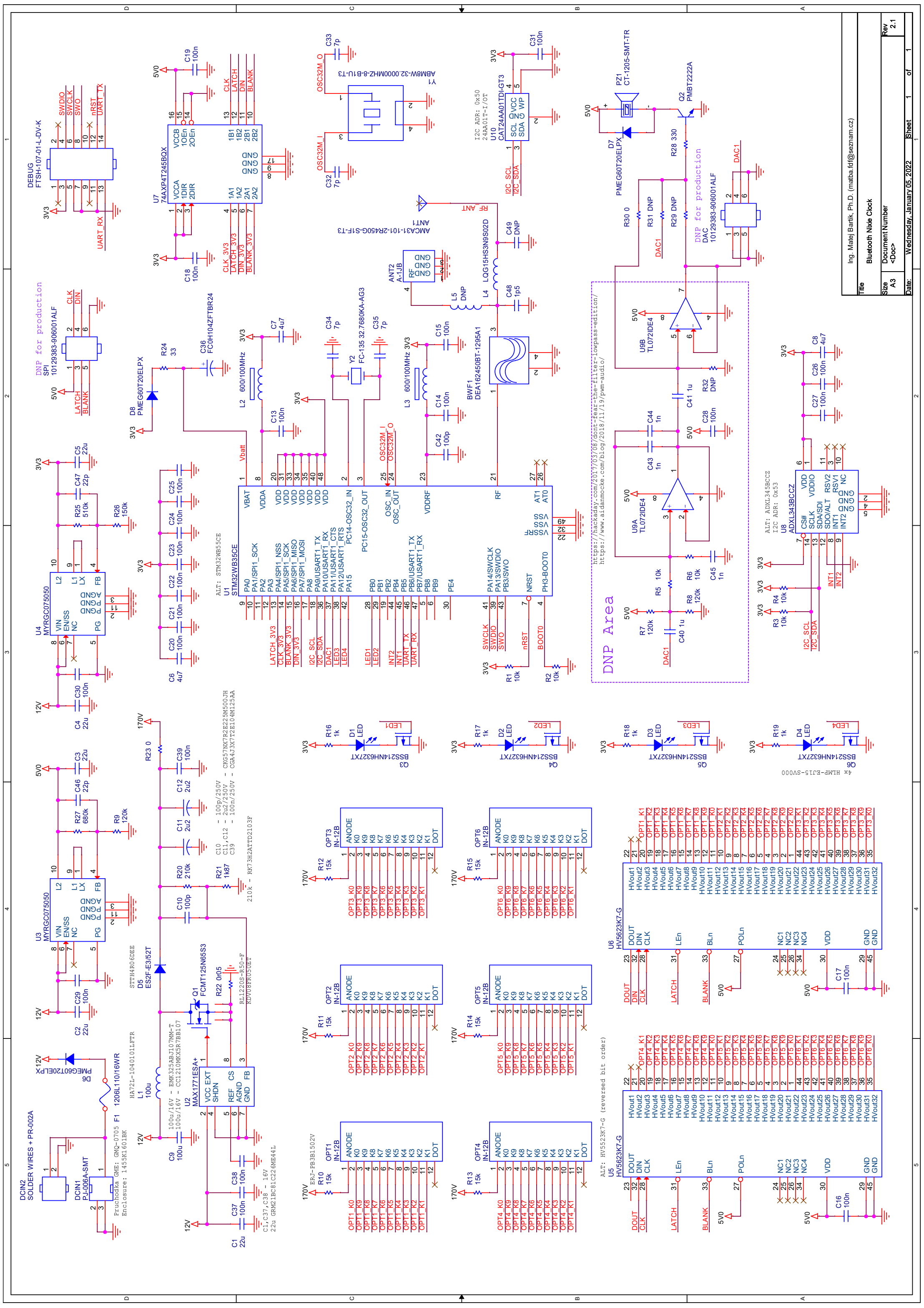


Figure C.1 Nixie clock schematic [4].

Appendix D
STM32WB55xx Block Diagram

This appendix shows the block diagram of the STM32WB55xx MCUs, which is the MCU on the device that was used during development.

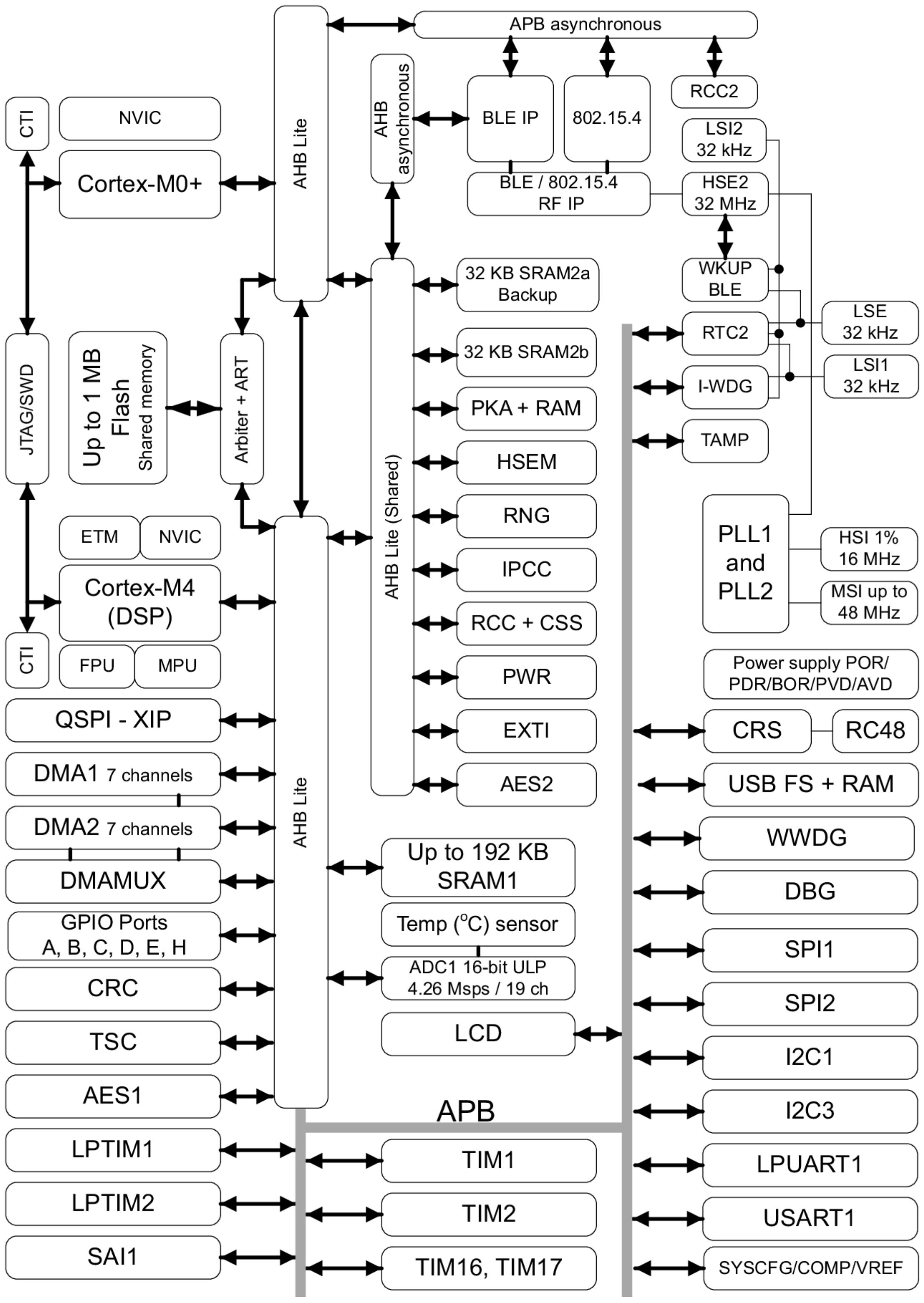


Figure D.1 STM32WB55xx block diagram [20].

Appendix E
Installation Guide

This appendix covers the steps necessary for running the software on a new device.

E.1 Wireless Stack Installation

For the application to function properly, the MCU radio core must be flashed with the appropriate wireless stack [51, zip file]. The STM32CubeWB package [51] contains the available wireless binaries, along with the installation instructions. The manufacturer offers multiple BLE stacks to choose from, which differ in how many functionalities they implement and, by extent, in the size of the binary [51, zip file]. To avoid unnecessary reflashing, the secondary core was flashed with the full-stack version, which can be used on any STM32WB MCU [51, zip file].

E.2 Existing Projects

An STM32CubeIDE project was created for the STM32WB55CGx and STM32WB35CEx MCU lines, as they were needed during development. If the target device uses either of the MCUs, the corresponding project can be used to flash the software on the MCU. As mentioned in Section 4.4, it was discovered that SPI CPOL and CPHA might need to be changed, depending on the device. If the initialization code is regenerated with CubeMX, some adjustments will have to be done to the project. The specific steps can be found below.

E.3 New Ports

If there is no existing project for the target MCU, a new one has to be created. The CubeMX configuration from one of the existing projects has to be copied into the new project so the initialization code can be generated. The IDE does not currently offer an elegant solution to migrating projects to different MCUs. As such, the settings might have to be ported manually.

E.3.1 CubeMX Configuration

The following list names the configuration used in the main development project. Some of the settings may be changed or optimized, as the time constraints did not always allow more thorough experiments with the settings.

E.3.1.1 Pinout & Configuration Tab

- GPIO
  - Set the LEDx, LATCH, and BLANK as GPIO output, with output level = ”Low”, GPIO mode = ”Output Push-Pull”, GPIO Pull-up/Pull-down = ”No pull-up and no pull-down”, and maximum output speed = ”Low”.
  - Set DIN as SPI1_MOSI and CLK as SPI1_SCK, with GPIO mode = ”Alternate Function Push Pull”, GPIO Pull-up/Pull-down = ”No pull-up and no pull-down”, and maximum output speed = ”Low”.
  - Set I2C_SDA as I2C1_SDA and I2C_SCL as I2C1_SCL, with GPIO mode = ”Alternate Function Open Drain”, GPIO Pull-up/Pull-down = ”No pull-up and no pull-down”, and maximum output speed = ”Low”.
  - Set DAC1 as TIM1_CH4, with GPIO mode = ”Alternate Function Push Pull”, GPIO Pull-up/Pull-down = ”No pull-up and no pull-down”, and maximum output speed = ”Low”.
  - Set INT1 and INT2 as TIM1_CH4 as GPIO EXTI5/4 respectively, with GPIO mode = ”External Interrupt Mode with Rising edge trigger detection” and GPIO Pull-up/Pull-down = ”Pull-down”.
- HSEM
  - Activate HSEM. The interrupt should be on by default, activate it if that is not the case.
- IPCC
  - Activate IPCC and the RX/TX interrupts.
- RCC
  - Activate HSE and LSE as ”Crystal/Ceramic oscillator”. The other settings can be left as default.
- LPTIM1
  - Set mode to = ”Counts internal clock events”, clock prescaler to ”Prescaler Div 128”, update mode = ”Update Immediate”, and trigger source = ”Software Trigger”. Turn on LPTIM1 global interrupt.
- LPTIM2
  - Set mode to = ”Counts internal clock events”, clock prescaler to ”Prescaler Div 128”, update mode = ”Update Immediate”, and trigger source = ”Software Trigger”. Turn on LPTIM2 global interrupt.
- RTC
  - Activate clock source, calendar, internal alarm A/B and internal wake-up.
  - Set clock format = ”Hourformat 24”. The asynchronous prescaler should be set to 127 and the synchronous one to 255. In BLE applications, the values of the prescalers are set by the `CFG_RTC_ASYNCH_PRESCALER` and the `CFG_RTC_SYNCH_PRESCALER` macros in `app_conf.h`.
  - Calendar time/date and alarm A/B values can be left as default, the application will update them as needed.
  - Set the wake-up clock as RTCCLK / 16 and the counter as 0. the clock value has to match the `CFG_RTC_WUCKSEL_DIVIDER` macro in `app_conf.h`.
  - Activate the wake-up and alarm interrupts.

- TIM1
  - Set clock source as internal clock and channel 4 as ”PWM Generation CH4”.
  - Counter settings are managed by the application as needed. The application does not use trigger output, break and dead time management, and clear input settings.
  - Set PWM generation mode to ”PWM mode 1”, enable output compare preload, disable fast mode, set CH polarity as high and CH idle as low. The pulse is managed by the application as necessary.
  - Enable TIM1 update interrupt.
- TIM2
  - Set clock source as internal clock. Counter settings are managed by the application as needed, trigger output settings are not used.
  - Activate the TIM2 global interrupt.
- I2C1
  - Set I2C1 to I2C mode. Disable custom timing, set speed mode to standard and speed frequency to 100 KHz. Digital and analog filter settings can be left as default.
  - Slave settings are not used.
  - Enable the I2C1 error and event interrupts.
- RF
  - Activate RF1.
- SPI1
  - Set SPI mode to ”Transmit Only Master”, disable the hardware NSS signal.
  - Set the frame format to Motorola, data size to 16 bits, and MSB order. Set the prescaler to 16. Low baud rate is not recommended, as they might result in occasional visual noise. The baud rate should not exceed 16 Mbit/s [5, 9]. As mentioned in Section 4.4, the CPOL/CPHA settings might have to be adjusted to the specific system. According to [5, 9], CPOL should be set to ”Low” and CPHA to ”2 Edge”. Disable CRC calculation, set NSS signal type to software.
  - Enable the SPI1 global interrupt.
- STM32_WPAN
  - Turn on BLE.
  - In ”BLE Application and services”, set wireless stack to full, application type to server profile. The IDE does not properly include the required drivers if none of the server modes are enabled, so the custom template mode should be activated. The other settings can be ignored.
- SEQUENCER
  - The sequencer should be automatically activated when BLE is turned on.
- TINY_LPM
  - TINY_LPM should be automatically activated when BLE is turned on.

E.3.1.2 Clock Configuration Tab

- Set HSE frequency to 32 MHz and LSE frequency to 32.768 Hz.
- Set HSE_SYS as system clock source, LSE as RTC clock source. Set PCLK1 as I2C1, LPTIM1, and LPTIM2 clock source. Set LSE as RFWKUP source. RFWKUP must match the `CFG_BLE_LSE_SOURCE` macro in `app_conf.h`. SMPS can be left to the default value. Prescalers can be left as /1.

E.3.1.3 Project Manager Tab

- No changes are necessary in the Projects subtab.
- In the code generation subtab, choose ”Copy only the necessary library files”, ”Generate peripheral initialization as a pair of '.c/.h' files per peripheral”, and ”Keep user code when re-generating”.
- In the advanced settings subtab, choose HAL for all peripherals. Make sure ”Generate Code” is ticked for all peripheral. Enable ”Register CallBack” for I2C, LPTIM, RTC, SPI, TIM.

E.3.1.4 Tools Tab

No changes are required in this tab.

E.3.2 Adjusting the Generated Code

The CubeMX version used in STM32CubeIDE v.1.7.0 is unable to generate some aspects of the initialization code as needed. The RTC power backup feature needs to be manually added. In BLE applications, the registered RTC wake-up callback is not executed because the BLE application redefines the function called in the interrupt, which does not take the callback into consideration. Lastly, CubeMX STM32_WPAN generation is not capable of generating all of the GATT-related settings needed. If the shortcomings are addressed in future versions of the tools, it is recommended that the initialization is done via the generator tool. For STM32_WPAN specifically, the tools must be able to generate all the services and characteristics and to set up the advertisement / scan response data and the PPCP to the desired values.

E.3.2.1 RTC Power Backup

In the `main.c` file, add a flag that is going to be used to indicate whether a backup domain restart is needed. In the `SystemClock_Config` function, set the flag according to LSE and RTC status. If both are running, skip LSE and RTC initialization. After adjusting the code, `main.c` should look similar to:

```c
...
/* USER CODE BEGIN Includes */
#include "stm32wbxx_ll_rcc.h"
/* USER CODE END Includes */
...
/* USER CODE BEGIN PV */
bool bkup_domain_restart_needed = true;
/* USER CODE END PV */
...
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
  RCC_PeriphCLKInitTypeDef PeriphClkInitStruct = {0};

#if (NIXIE_FORCE_BACKUP_DOMAIN_RESET == 0)
  if (LL_RCC_LSE_IsEnabled() && LL_RCC_LSE_IsReady()
      && LL_RCC_IsEnabledRTC())
  {
    bkup_domain_restart_needed = false;
  }
#endif

  /** Configure LSE Drive Capability
  */
  HAL_PWR_EnableBkUpAccess();
  if (bkup_domain_restart_needed)
  {
    __HAL_RCC_LSEDRIVE_CONFIG(RCC_LSEDRIVE_LOW);
  }
  /** Configure the main internal regulator output voltage
  */
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI
                              |RCC_OSCILLATORTYPE_HSE
                              |RCC_OSCILLATORTYPE_LSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.LSEState = RCC_LSE_ON;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
  if(!bkup_domain_restart_needed)
  {
    RCC_OscInitStruct.OscillatorType &= ~RCC_OSCILLATORTYPE_LSE;
  }
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  if(!bkup_domain_restart_needed)
  {
    RCC_OscInitStruct.OscillatorType |= RCC_OSCILLATORTYPE_LSE;
  }

  /** Configure the SYSCLKSource, HCLK, PCLK1 and PCLK2 clocks dividers
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK4|RCC_CLOCKTYPE_HCLK2
                              |RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSE;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.AHBCLK2Divider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.AHBCLK4Divider = RCC_SYSCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
  {
    Error_Handler();
  }
  /** Initializes the peripherals clocks
  */
  PeriphClkInitStruct.PeriphClockSelection = RCC_PERIPHCLK_SMPS
                              |RCC_PERIPHCLK_RFWAKEUP
                              |RCC_PERIPHCLK_RTC|RCC_PERIPHCLK_USART1
                              |RCC_PERIPHCLK_LPTIM1|RCC_PERIPHCLK_LPTIM2
                              |RCC_PERIPHCLK_I2C1;
  PeriphClkInitStruct.Usart1ClockSelection = RCC_USART1CLKSOURCE_PCLK2;
  PeriphClkInitStruct.I2c1ClockSelection = RCC_I2C1CLKSOURCE_PCLK1;
  PeriphClkInitStruct.Lptim1ClockSelection = RCC_LPTIM1CLKSOURCE_PCLK1;
  PeriphClkInitStruct.Lptim2ClockSelection = RCC_LPTIM2CLKSOURCE_PCLK;
  PeriphClkInitStruct.RTCClockSelection = RCC_RTCCLKSOURCE_LSE;
  PeriphClkInitStruct.RFWakeUpClockSelection = RCC_RFWKPCLKSOURCE_LSE;
  PeriphClkInitStruct.SmpsClockSelection = RCC_SMPSCLKSOURCE_HSI;
  PeriphClkInitStruct.SmpsDivSelection = RCC_SMPSCLKDIV_RANGE1;
  if(!bkup_domain_restart_needed)
  {
    PeriphClkInitStruct.PeriphClockSelection &= ~RCC_PERIPHCLK_RTC;
  }
  if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  if(!bkup_domain_restart_needed)
  {
    PeriphClkInitStruct.PeriphClockSelection |= RCC_PERIPHCLK_RTC;
  }
  /* USER CODE BEGIN Smps */
  /* USER CODE END Smps */
}
...
```

In the `main.h` file, declare the flag as extern:

```
...
/* USER CODE BEGIN Includes */
#include <stdbool.h>
/* USER CODE END Includes */
...
/* USER CODE BEGIN ET */
extern bool bkup_domain_restart_needed;
/* USER CODE END ET */
...
```

In the `rtc.c` file, only call the RTC initialization functions if a backup domain restart is necessary:

```c
...
/* USER CODE BEGIN 0 */
#include "main.h"
/* USER CODE END 0 */
...
void MX_RTC_Init(void)
{
  /* USER CODE BEGIN RTC_Init 0 */
  /* USER CODE END RTC_Init 0 */

  RTC_TimeTypeDef sTime = {0};
  RTC_DateTypeDef sDate = {0};
  RTC_AlarmTypeDef sAlarm = {0};

  /* USER CODE BEGIN RTC_Init 1 */
  /* USER CODE END RTC_Init 1 */

  /** Initialize RTC Only
  */
  hrtc.Instance = RTC;
  hrtc.Init.HourFormat = RTC_HOURFORMAT_24;
  hrtc.Init.AsynchPrediv = CFG_RTC_ASYNCH_PRESCALER;
  hrtc.Init.SynchPrediv = CFG_RTC_SYNCH_PRESCALER;
  hrtc.Init.OutPut = RTC_OUTPUT_DISABLE;
  hrtc.Init.OutPutPolarity = RTC_OUTPUT_POLARITY_HIGH;
  hrtc.Init.OutPutType = RTC_OUTPUT_TYPE_OPENDRAIN;
  hrtc.Init.OutPutRemap = RTC_OUTPUT_REMAP_NONE;
  if (bkup_domain_restart_needed && HAL_RTC_Init(&hrtc) != HAL_OK)
  {
    Error_Handler();
  }

  /* USER CODE BEGIN Check_RTC_BKUP */
  /* USER CODE END Check_RTC_BKUP */

  /** Initialize RTC and set the Time and Date
  */
  sTime.Hours = 0x0;
  sTime.Minutes = 0x0;
  sTime.Seconds = 0x0;
  sTime.SubSeconds = 0x0;
  sTime.DayLightSaving = RTC_DAYLIGHTSAVING_NONE;
  sTime.StoreOperation = RTC_STOREOPERATION_RESET;
  if (bkup_domain_restart_needed
      && HAL_RTC_SetTime(&hrtc, &sTime, RTC_FORMAT_BCD) != HAL_OK)
  {
    Error_Handler();
  }
  sDate.WeekDay = RTC_WEEKDAY_MONDAY;
  sDate.Month = RTC_MONTH_JANUARY;
  sDate.Date = 0x1;
  sDate.Year = 0x0;
  if (bkup_domain_restart_needed
      && (HAL_RTC_SetDate(&hrtc, &sDate, RTC_FORMAT_BCD) != HAL_OK))
  {
    Error_Handler();
  }
  /** Enable the Alarm A
  */
  sAlarm.AlarmTime.Hours = 0x0;
  sAlarm.AlarmTime.Minutes = 0x0;
  sAlarm.AlarmTime.Seconds = 0x0;
  sAlarm.AlarmTime.SubSeconds = 0x0;
  sAlarm.AlarmTime.DayLightSaving = RTC_DAYLIGHTSAVING_NONE;
  sAlarm.AlarmTime.StoreOperation = RTC_STOREOPERATION_RESET;
  sAlarm.AlarmMask = RTC_ALARMMASK_NONE;
  sAlarm.AlarmSubSecondMask = RTC_ALARMSUBSECONDMASK_ALL;
  sAlarm.AlarmDateWeekDaySel = RTC_ALARMDATEWEEKDAYSEL_DATE;
  sAlarm.AlarmDateWeekDay = 0x1;
  sAlarm.Alarm = RTC_ALARM_A;
  if (bkup_domain_restart_needed
      && HAL_RTC_SetAlarm_IT(&hrtc, &sAlarm, RTC_FORMAT_BCD) != HAL_OK)
  {
    Error_Handler();
  }
  /** Enable the Alarm B
  */
  sAlarm.Alarm = RTC_ALARM_B;
  if (bkup_domain_restart_needed
      && HAL_RTC_SetAlarm_IT(&hrtc, &sAlarm, RTC_FORMAT_BCD) != HAL_OK)
  {
    Error_Handler();
  }
  /** Enable the WakeUp
  */
  if (bkup_domain_restart_needed
      && HAL_RTCEx_SetWakeUpTimer_IT(&hrtc, 0, RTC_WAKEUPCLOCK_RTCCLK_DIV16)
        != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN RTC_Init 2 */
  /* USER CODE END RTC_Init 2 */
}
```

E.3.2.2 RTC Wake-up Interrupt

The RTC wake-up interrupt must be adjusted to schedule a date time update. After making the changes, `stm32wbxx_it.c` should look like:

```
...
/* USER CODE BEGIN Includes */
#include "stm32_seq.h"
#include "app_conf.h"
/* USER CODE END Includes */
...
void RTC_WKUP_IRQHandler(void)
{
  /* USER CODE BEGIN RTC_WKUP_IRQn 0 */
  UTIL_SEQ_SetTask(1 << CFG_TASK_NIXIE_UPDATE_DATETIME_ID, CFG_SCH_PRIO_0);
  /* USER CODE END RTC_WKUP_IRQn 0 */
  HAL_RTCEx_WakeUpTimerIRQHandler(&hrtc);
  /* USER CODE BEGIN RTC_WKUP_IRQn 1 */
  /* USER CODE END RTC_WKUP_IRQn 1 */
}
...
```

E.3.2.3 STM32_WPAN

To resolve `STM32_WPAN` generation issues, copy the contents of the `template/STM32_WPAN` folder into the `STM32_WPAN` folder in the IDE project.

E.3.3 Adding Helper Files

To add the helper files, copy the contents of `template/Core/Inc` and `template/Core/Src` into the respective folders in the IDE project.

Appendix F

Additional Photographs of the Clock

This appendix contains additional photographs of a fully-assembled clock device.

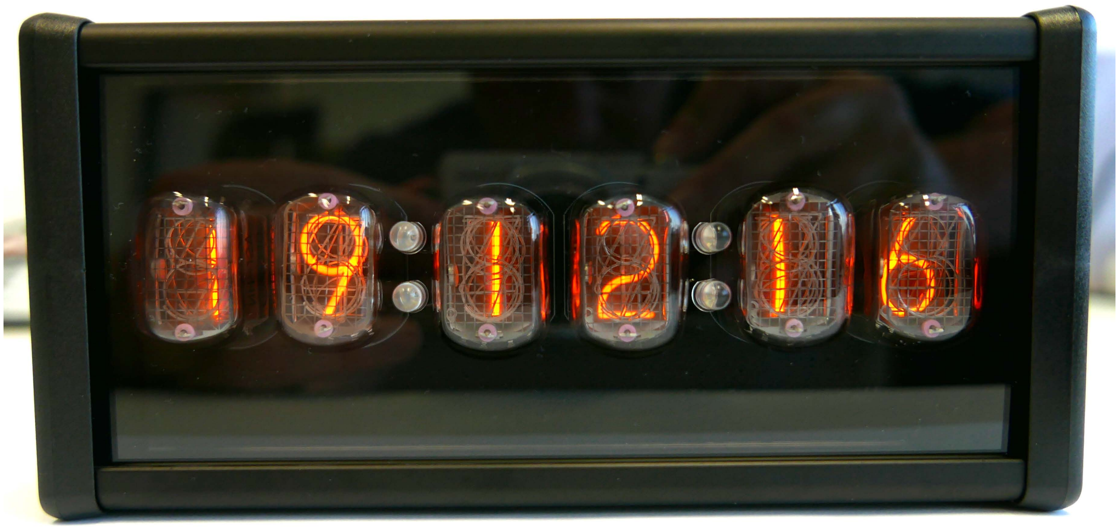


Figure F.1 A front view of the clock. The clock is showing the current time. The LEDs blinking was in the off state at the time of taking the photograph. (© Matěj Bartík)

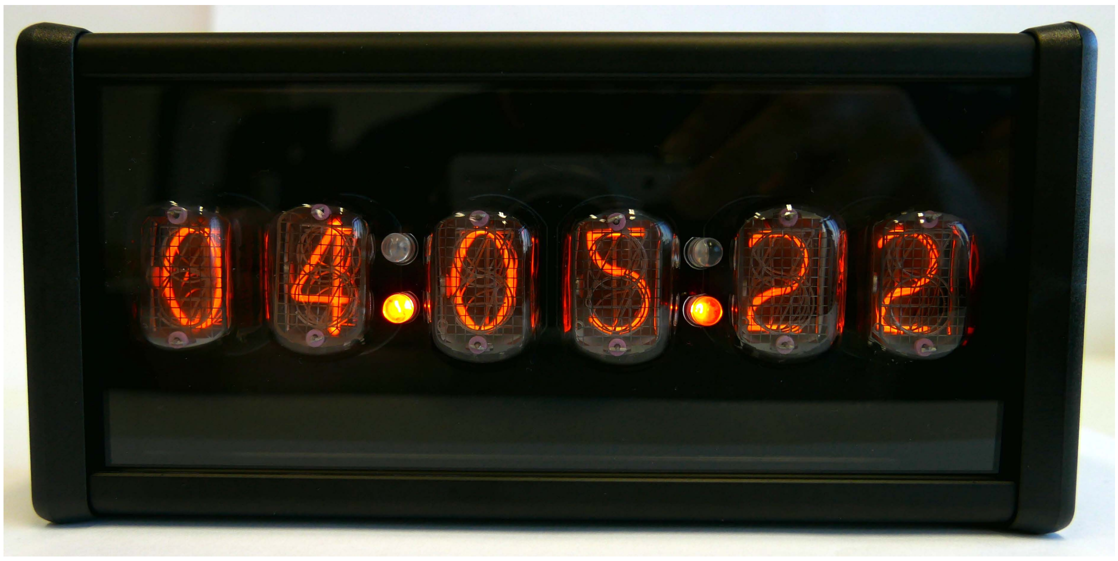


Figure F.2 A front view of the clock. The clock is showing the current date. Only the lower two LEDs are turned on to make it easier to differentiate between the time and the date. (© Matěj Bartík)

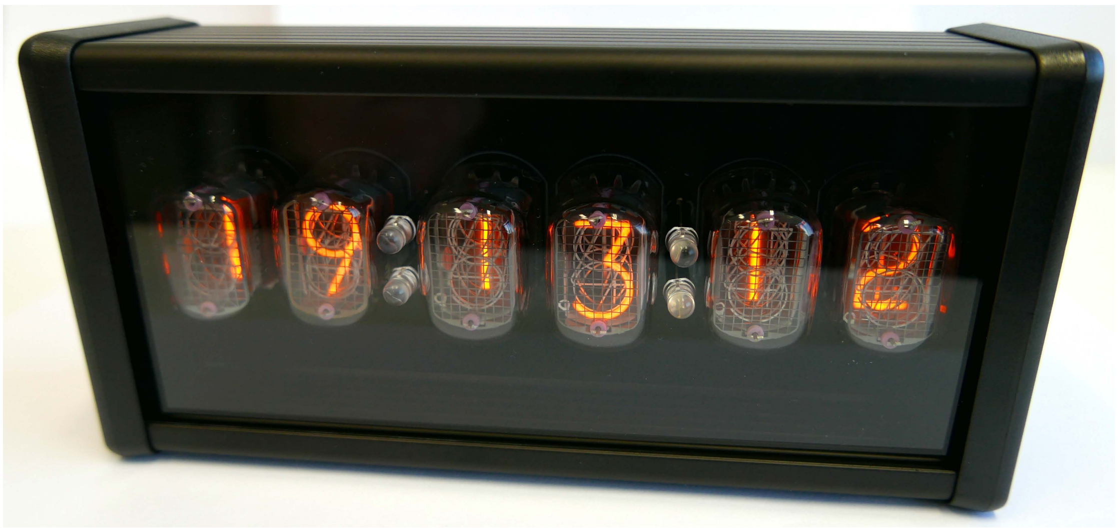


Figure F.3 A front view of the clock from a slightly higher angle. (© Matěj Bartík)

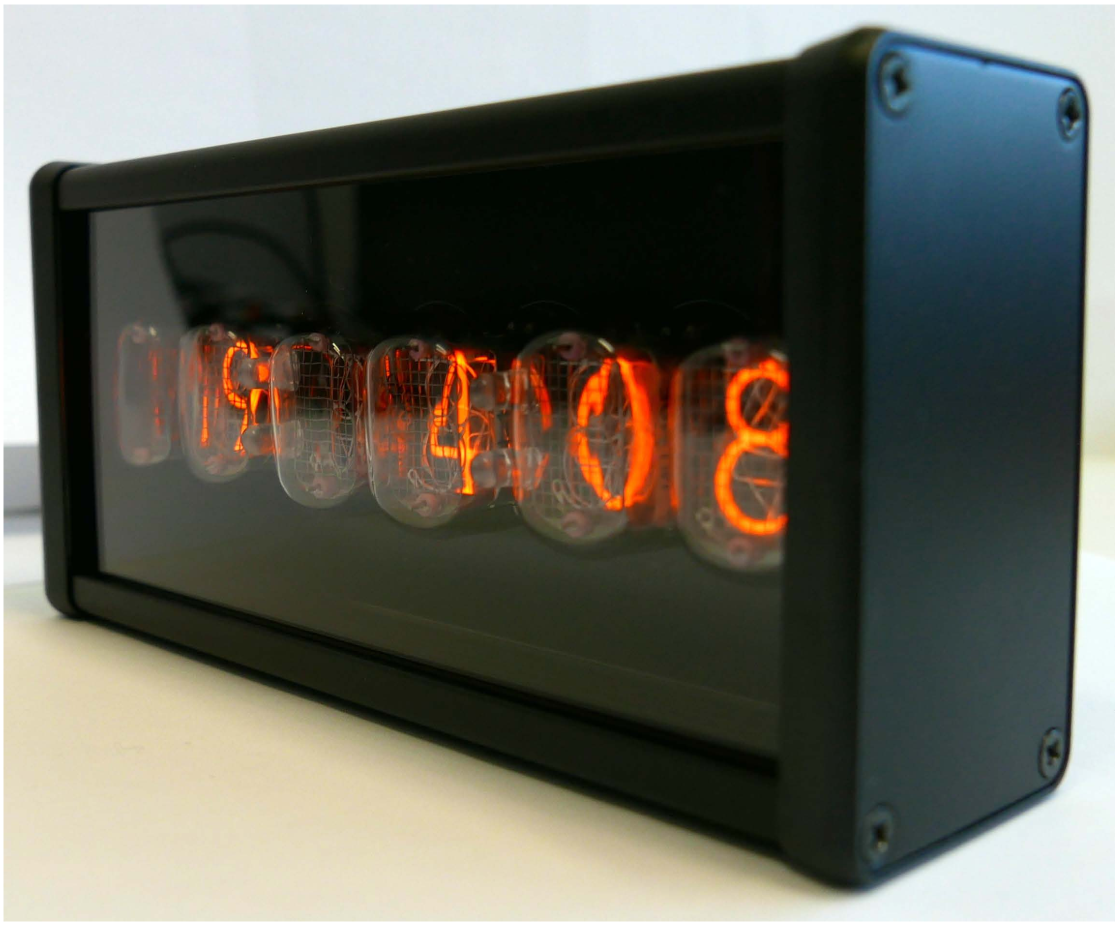


Figure F.4 A side view of the clock. (© Matěj Bartík)

Figure F.5 A back view of the clock. (© Matěj Bartík)

Figure F.6 A top view of the clock. (© Matěj Bartík)

Bibliography

1. HELLBUS. *Z566M digit 6* [online]. 2009-09 [visited on 2022-04-16]. Available from: `https://commons.wikimedia.org/wiki/File:Z566M_digit_6.jpg`.
2. VERNER, David. *Mobilní aplikace pro ovládání digitronových hodin*. Prague, 2021. Available also from: `https://dspace.cvut.cz/handle/10467/95573`. Bachelor thesis. Czech Technical University in Prague, Faculty of Information Technology.
3. ZUIJLEKOM, Dennis van. *Wang 700 Advanced Programmable Calculator (9220477911)* [online]. 2013-07-05 [visited on 2022-04-16]. Available from: `https://commons.wikimedia.org/wiki/File:Wang_700_Advanced_Programmable_Calculator_(9220477911).png`.
4. BARTÍK, Matěj. *Bluetooth Nixie Clock: Rev 2.1*. 2022.
5. MICROCHIP TECHNOLOGY INC. *HV5623 32-Channel Serial-to-Parallel Converter with Open-Drain Outputs* [online]. 2019-05-14 [visited on 2022-04-19]. Available from: `https://ww1.microchip.com/downloads/en/DeviceDoc/HV5623-32-Channel-Serial-to-Parallel-Converter-with-Open-Drain-Outputs-Data-Sheet-20005702A.pdf`.
6. STMICROELECTRONICS. *Multiprotocol wireless 32-bit MCU Arm®-based Cortex®-M4 with FPU, Bluetooth® Low-Energy and 802.15.4 radio solution - RM0434 Reference manual* [online]. 2021-04-20 [visited on 2021-06-18]. Available from: `https://www.st.com/content/ccc/resource/technical/document/reference_manual/group0/83/cf/94/7a/35/a9/43/58/DM00318631/files/DM00318631.pdf/jcr:content/translations/en.DM00318631.pdf`.
7. ANALOG DEVICES, INC. *ADXL343 (Rev. 0) Data Sheet: 3-Axis, ±2 g/±4 g/±8 g/±16 g Digital MEMS Accelerometer* [online]. 2012 [visited on 2022-04-19]. Available from: `https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL343.pdf`.
8. ANALOG DEVICES, INC. *ADXL345 Data Sheet: 3-Axis, ±2 g/±4 g/±8 g/±16 g Digital Accelerometer* [online]. 2015 [visited on 2022-04-19]. Available from: `https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf`.
9. MICROCHIP TECHNOLOGY INC. *HV5523 32-Channel Serial-to-Parallel Converter with Open-Drain Outputs* [online]. 2019-05-14 [visited on 2022-04-19]. Available from: `https://ww1.microchip.com/downloads/en/DeviceDoc/HV5523-32-Channel-Serial-to-Parallel-Converter-with-Open-Drain-Outputs-Data-Sheet-20005700A.pdf`.
10. STMICROELECTRONICS. *STM32WB - Bluetooth, Wireless Microcontrollers (MCU) - STMicroelectronics* [online]. 2022 [visited on 2022-04-19]. Available from: `https://www.st.com/en/microcontrollers-microprocessors/stm32wb-series.html`.
11. ELLISYS S.A. *Ellisys Bluetooth Video 1: Intro to Bluetooth Low Energy* [online]. 2018-04-09 [visited on 2022-01-25]. Available from: `https://www.youtube.com/watch?v=eZGixQzBo7Y`.

12. BLUETOOTH SIG, INC. *LE Audio — Bluetooth® Technology Website* [online]. 2020-01-06 [visited on 2022-01-25]. Available from: `https://www.bluetooth.com/learn-about-bluetooth/recent-enhancements/le-audio/`.
13. ELLISYS S.A. *Ellisys Bluetooth Video 2: Generic Access Profile* [online]. 2018-04-09 [visited on 2022-01-25]. Available from: `https://www.youtube.com/watch?v=8OfOwD8f2VI`.
14. ELLISYS S.A. *Ellisys Bluetooth Video 3: Advertisements* [online]. 2018-04-18 [visited on 2022-01-25]. Available from: `https://www.youtube.com/watch?v=be9ct7OKI7`.
15. ELLISYS S.A. *Ellisys Bluetooth Video 4: Connections* [online]. 2018-05-08 [visited on 2022-04-16]. Available from: `https://www.youtube.com/watch?v=YmMDy8qYX_c`.
16. ELLISYS S.A. *Ellisys Bluetooth Video 5: Generic Attribute Profile (GATT)* [online]. 2018-06-05 [visited on 2022-04-16]. Available from: `https://www.youtube.com/watch?v=eHqtiCMe4NA`.
17. NXP SEMICONDUCTORS N.V. *I^2C-bus specification and user manual* [online]. 2021-10-01 [visited on 2022-03-14]. Available from: `https://www.nxp.com/docs/en/user-guide/UM10204.pdf`.
18. MOTOROLA, INC. *SPI Block Guide V04.01* [online]. 2004-07-14 [visited on 2022-03-04]. Available from: `https://www.nxp.com/files-static/microcontrollers/doc/ref_manual/S12SPIV4.pdf`.
19. CBURNETT. *SPI three slaves* [online]. 2006-12-19 [visited on 2022-03-05]. Available from: `https://commons.wikimedia.org/wiki/File:SPI_three_slaves.svg`.
20. STMICROELECTRONICS. *Datasheet - STM32WB55xx STM32WB35xx - Multiprotocol wireless 32-bit MCU Arm®-based Cortex®-M4 with FPU, Bluetooth® 5.2 and 802.15.4 radio solution* [online]. 2021-04-17 [visited on 2021-08-03]. Available from: `https://www.st.com/resource/en/datasheet/stm32wb55cg.pdf`.
21. STMICROELECTRONICS. *M41T81 - Serial real-time clock (RTC) with alarm - STMicroelectronics* [online]. 2022 [visited on 2022-04-19]. Available from: `https://www.st.com/en/clocks-and-timers/m41t81.html`.
22. STMICROELECTRONICS. *M41T93 - Serial SPI bus real-time clock (RTC) with battery switchover - STMicroelectronics* [online]. 2022 [visited on 2022-04-19]. Available from: `https://www.st.com/en/clocks-and-timers/m41t93.html`.
23. GLOW TUBE RESEARCH S.R.O. *Products* [online] [visited on 2022-01-24]. Available from: `https://www.daliborfarny.com/shop/`.
24. NIXIE SHOP. *Nixie Shop — Buy Nixie Clocks Online* [online]. 2021 [visited on 2022-01-24]. Available from: `https://nixieshop.com/store.html`.
25. 3DSIMO S.R.O. *Nixie clock (kit) — 3dsimo.com* [online] [visited on 2022-01-24]. Available from: `https://eshop.3dsimo.com/products/nixie-clock`.
26. 3DSIMO S.R.O. *Nixie clock (kit without Nixie tubes) — 3dsimo.com* [online] [visited on 2022-01-24]. Available from: `https://eshop.3dsimo.com/products/nixie-clock-kit-without-nixie-tubes`.
27. Y-0009. *Nixie Clock 4 x IN-12 With Tube RGB Backlight Assembled 12/24 format USA store — eBay* [online] [visited on 2022-01-24]. Available from: `https://www.ebay.com/itm/184362056151`.
28. FOMIN, Aleksej. *GRA & AFCH Nixie Tubes Clock on NCM105, NCM107, NCM109* [online]. 2014 [visited on 2022-01-23]. Available from: `https://github.com/afch/NixieClock`.
29. GLOWTIME NIXIE CLOCKS LTD. *Nixie Clock IN-14 Kit (no tubes) Arduino Open Source — eBay* [online] [visited on 2022-01-24]. Available from: `https://www.ebay.com/itm/132023669741`.

30. GREATSCOTTLAB. *Make Your Own Retro Nixie Clock With an RTC! : 7 Steps (with Pictures) - Instructables* [online]. 2019-12-16 [visited on 2022-01-24]. Available from: `https://www.instructables.com/Make-Your-Own-Retro-Nixie-Clock-With-an-RTC/`.

31. MCER12. *GitHub - mcer12/Nick-ESP8266: Nick is a series of different Nixie clocks based on ESP8266.* [Online]. 2020 [visited on 2022-01-24]. Available from: `https://github.com/mcer12/Nick-ESP8266`.

32. TYSCH. *GitHub - tysch/STM32-Nixie: GPS-disciplined STM32-based alarm clock* [online]. 2017 [visited on 2022-01-24]. Available from: `https://github.com/tysch/STM32-Nixie`.

33. BARTÍK, Matěj. *Digitronové hodiny (Nixie clock) - specifikace pro účely BP* [online]. 2020 [visited on 2022-01-24]. Available from: `https://docs.google.com/document/d/1xetzb_nKhZtPHte_wYI1smfKd7ueFHZZWfDHvDGd1Fs/edit?usp=sharing`.

34. DALIBORFARNY.COM. *Nixie Tube Health - Zen Nixie Clock - 1* [online]. 2022 [visited on 2022-04-23]. Available from: `https://docs.daliborfarny.com/zen-nixie-clock/1/en/topic/nixie-tube-health`.

35. WÄCHTER, Dieter. *pois-01* [online] [visited on 2022-04-20]. Available from: `http://www.tube-tester.com/sites/nixie/different/cathode%20poisoning/cathode-poisoning.htm`.

36. BLUETOOTH SIG, INC. *Bluetooth Core Specification: Rev 5.3* [online]. 2021-07-12 [visited on 2021-09-16]. Available from: `https://www.bluetooth.com/specifications/specs/core-specification-5-3/`.

37. STMICROELECTRONICS. *STM32WB Bluetooth® Low Energy (BLE) wireless interface - AN5270 Application note* [online]. 2021-01 [visited on 2021-06-22]. Available from: `https://www.st.com/content/ccc/resource/technical/document/application_note/group1/25/56/53/7b/a0/17/47/c8/DM00571230/files/DM00571230.pdf/jcr:content/translations/en.DM00571230.pdf`.

38. STMICROELECTRONICS. *Description of STM32WB HAL and low-layer drivers - UM2442 User manual* [online]. 2020-03 [visited on 2021-04-24]. Available from: `https://www.st.com/content/ccc/resource/technical/document/user_manual/group1/4b/81/e9/cf/40/24/4f/1a/DM00524025/files/DM00524025.pdf/jcr:content/translations/en.DM00524025.pdf`.

39. STMICROELECTRONICS. *STM32Cube: embedded software quality assurance* [online] [visited on 2022-04-23]. Available from: `https://www.st.com/content/ccc/resource/sales_and_marketing/presentation/product_presentation/b4/36/4a/82/9d/49/4b/1c/stm32cube_HAL_qualification.pdf/files/stm32cube_HAL_qualification.pdf/jcr:content/translations/en.stm32cube_HAL_qualification.pdf`.

40. COOK, Curtis R.; KIM, Do Jin. Best Sorting Algorithm for Nearly Sorted Lists. *Commun. ACM.* 1980, vol. 23, no. 11, pp. 620–624. ISSN 0001-0782. Available from DOI: `10.1145/359024.359026`.

41. BLUETOOTH SIG, INC. *GATT Specification Supplement: v6* [online]. 2022-03-22 [visited on 2021-04-10]. Available from: `https://www.bluetooth.com/specifications/specs/gatt-specification-supplement-6/`.

42. REINGOLD, Edward M.; DERSHOWITZ, Nachum. *Calendrical Calculations: The Ultimate Edition.* 4th ed. Cambridge University Press, 2018. ISBN 9781107415058. Available from DOI: `10.1017/9781107415058`.

43. ZELLER, Christian. Kalender-Formeln. *Acta Mathematica.* 1887, vol. 9, pp. 131–136. Available from DOI: `10.1007/BF02406733`.

44. JONES, Gareth A.; JONES, J. Mary. *Elementary Number Theory.* 1st ed. Springer London, 1998. ISBN 978-1-4471-0613-5. ISSN 2197-4144. Available from DOI: `10.1007/978-1-4471-0613-5`.

45. STMICROELECTRONICS. *STLINK-V3 compact in-circuit debugger and programmer for STM32* [online] [visited on 2022-04-30]. Available from: `https://www.st.com/en/development-tools/stlink-v3mini.html`.
46. TVZ MECHATRONICS TEAM. *Digital inputs and outputs* [online]. 2018-10-11 [visited on 2022-05-05]. Available from: `https://os.mbed.com/teams/TVZ-Mechatronics-Team/wiki/Digital-inputs-and-outputs`.
47. STMICROELECTRONICS. *Building wireless applications with STM32WB Series microcontrollers - AN5289 Application note* [online]. 2021-04-20 [visited on 2021-04-24]. Available from: `https://www.st.com/content/ccc/resource/technical/document/application_note/group1/43/ea/2f/dc/10/a3/46/e6/DM00598033/files/DM00598033.pdf/jcr:content/translations/en.DM00598033.pdf`.
48. CUI DEVICES. *CMT-1271-88-SMT-TR Datasheet - Audio Transducers | Buzzers | CUI Devices* [online]. 2020-01-22 [visited on 2022-05-10]. Available from: `https://www.cuidevices.com/product/resource/cmt-1271-88-smt-tr.pdf`.
49. SEMICONDUCTOR COMPONENTS INDUSTRIES, LLC. *CAT24AA01, CAT24AA02 - EEPROM Serial 1/2-Kb I2C* [online]. 2018-04 [visited on 2022-05-10]. Available from: `https://www.onsemi.com/pdf/datasheet/cat24aa01-d.pdf`.
50. STMICROELECTRONICS. *STM32Cube - Discover the STM32Cube Ecosystem - STMicroelectronics* [online] [visited on 2022-05-07]. Available from: `https://www.st.com/content/st_com/en/ecosystems/stm32cube-ecosystem.html`.
51. STMICROELECTRONICS. *STM32CubeWB - STMicroelectronics* [online] [visited on 2022-05-07]. Available from: `https://www.st.com/en/embedded-software/stm32cubewb.html`.

Contents of the Enclosed Storage Medium

```
 readme.txt ........................ a brief summary of the contents
└── src
    ├── impl ........................ the implementation source files
    │   ├── template ........ the helper source files to be included in IDE projects
    │   ├── legacy .......... the original IDE project used during development
    │   └── new ............. new, cleaned-up IDE projects
    └── thesis ............ the source files of the thesis text in LaTeX format
└── text ............................ the thesis text
    └── thesis.pdf .................. the thesis text in PDF format
```