BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

VISUAL PROGRAMMING TOOLKIT BASED ON MICROPYTHON FOR ESP32 PLATFORM

NÁSTROJ PRO VIZUÁLNÍ PROGRAMOVÁNÍ PLATFORMY ESP32 V JAZYCE MICROPYTHON

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. DANIEL PAUL**
AUTOR PRÁCE

SUPERVISOR **Ing. VÁCLAV ŠIMEK**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

Institut:	Department of Computer Systems (DCSY)	155806
Student:	**Paul Daniel, Bc.**	
Programme:	Information Technology and Artificial Intelligence	
Specialization:	Information Systems and Databases	
Title:	**Visual Programming Toolkit Based on MicroPython for ESP32 Platform**	
Category:	Web applications	
Academic year:	2023/24	

Assignment:

1. Study the principles of the so-called visual programming concept and the existing software tools used for that purpose.
2. Explore the contemporary technology and approaches suitable for implementing web-based applications. Prepare a short review on that topic.
3. Make yourself acquainted with the domain of embedded systems with a special emphasis given to the ESP32 platform, its hardware features, and means of programming.
4. Outline the well-structured architecture of a visual programming toolkit tailored for the ESP32 platform with the capability to generate a MicroPython code.
5. Develop a proof-of-concept example to showcase the feasibility and viability of the proposed architecture from point 5) of the assignment.
6. Implement the proposed concept into a fully-fledged visual programming toolkit. Try to pay attention to the modular structure of the code.
7. Rigorously test the toolkit in diverse scenarios to validate its functionality, robustness, and performance under various conditions.
8. Evaluate the functionality of the complete tool. Assess the achieved results and try to propose further directions for the development.

Literature:
- According to the instructions of the supervisor.

Requirements for the semestral defence:
- Fulfillment of points 1 to 5 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Šimek Václav, Ing.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2023
Submission deadline:	17.5.2024
Approval date:	30.10.2023

Abstract

This thesis presents the development of a Visual Programming Toolkit designed for programming the ESP32 platform using MicroPython. The toolkit leverages the intuitive nature of the visual programming paradigm to simplify the process of programming microcontrollers for users without extensive coding experience. The core of the toolkit is built on a web-based interface that utilizes the ReactFlow library to enable drag-and-drop functionality with flow-based diagrams, allowing users to assemble code through visual blocks that represent MicroPython commands and structures. Furthermore, it covers the backend hosted on the ESP32 device itself, allowing communication with the frontend client.

Abstrakt

Táto diplomová práca predstavuje vývoj Vizuálneho Programovacieho Nástroja určeného na programovanie platformy ESP32 pomocou MicroPythonu. Nástroj využíva intuitívnu povahu paradigmy vizuálneho programovania na zjednodušenie procesu programovania mikrokontrolérov pre používateľov bez rozsiahlych skúseností s programovaním. Jadro programovacieho nástroju je postavené na webovom rozhraní, ktoré využíva knižnicu ReactFlow na umožnenie funkcie "tahaj a pusť" s diagramami založenými na toku, čo umožňuje používateľom zostavovať kód prostredníctvom vizuálnych blokov, ktoré predstavujú príkazy a štruktúry jazyka MicroPython. Ďalej pokrýva backend umiestnený na samotnom zariadení ESP32, ktorý umožňuje komunikáciu s frontendovým klientom.

Keywords

Visual programming, ESP32, MicroPython, Flow-Based programming, Hardware programming, Web-Based applications, React

Klíčová slova

Vizuálne programovanie, ESP32, MicroPython, Programovanie založené na toku, Programovanie Hardvéru, Webové aplikácie, React

Reference

PAUL, Daniel. *Visual Programming Toolkit Based on MicroPython for ESP32 Platform.* Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Václav Šimek

Rozšířený abstrakt

V dnešnom svete, ktorý nás na každom kroku obklopuje modernými technológiami, je potrebné držať krok s dianím okolo nás. Všade, kam sa pozrieme, sa nachádzajú snímače, kamery, mikrofóny a ďalšie periférne zariadenia, ktoré sú prostredníctvom riešení internetu vecí (IoT) neoddeliteľnou súčasťou nášho života. Táto prepojená sieť zariadení mení spôsob, akým komunikujeme so svetom, a ponúka efektivitu, konektivitu a pohodlie. Snímače sú prítomné v našich smartfónoch, nositeľných zariadeniach, domácich spotrebičoch alebo dokonca v mestskej architektúre a výrazne nám uľahčujú život tým, že šetria drahocenný čas, zhromažďujú a analyzujú údaje a prenášajú ich tam, kde sú potrebné. To sú dôvody, prečo by malo byť porozumenie a programovanie mikrokontrolérov dostupnejšie pre ľudí, ktorí sa v týchto úlohách dobre nevyznajú.

Preto sa táto diplomová práca zaoberá vývojom nástroja pre vizuálne programovanie, ktorý je určený pre platformu ESP32 a využíva jazyk MicroPython. Cieľom práce je poskytnúť nástroj, ktorý uľahčí programovanie mikrokontrolérov aj používateľom bez rozsiahlych skúseností s kódovaním. Vizuálne programovanie umožňuje používateľom vytvárať programy prostredníctvom intuitívneho rozhrania, ktoré používa metódu "tahaj a pusť" na manipuláciu s vizuálnymi blokmi reprezentujúcimi príkazy a štruktúry jazyka MicroPython. Hlavnou zložkou nástroja je webové rozhranie založené na knižnici ReactFlow, ktoré podporuje funkcionality ako sú "tahaj a pusť" a vytváranie diagramov založených na toku čo umožňuje užívateľom ľahko zostaviť kód.

Práca sa podrobne venuje integrácii nevyhnutných knižníc MicroPythonu ako je machine modul obsahujúci GPIO, SPI, I2C, ADC a Timer a asynchrónnemu spracovaniu operácií. Nechýba ani podpora riadiacich štruktúr, premenných, práca s kolekciami ako sú zoznam, množina, ntica a slovník, vykonávanie matematických, logických operácií a operácií na sekvenciách, ako je dĺžka sekvencie, usporiadanie, alebo počet výskytu prvku. Ďalším významným prínosom je podpora integrácie periférnych zariadení, čo robí tento nástroj všestranným prostriedkom pre vývoj interakcií s hardvérom. Nástroj taktiež podporuje komunikačné protokoly ako MQTT a HTTP, pre možnosť prijímania a odosielania informácií. V neposlednom rade nástroj podporuje aj konštukcií, ako je pridanie časového razítka, naformátovanie reťazca, alebo aj vlastný Python kód. Významná časť vývoja bola venovaná jednoduchosti použitia, škálovateľnosti a schopnosti nástroja efektívne zvládať rôzne programovacie scenáre. Nástroj umožňuje komunikáciu s backendom na základe počítačového komunikačného protokolu WebSocket, prostredníctvom ktorého vie užívateľ vyexportovať program na ESP zariadenie a nadviazať s ním spojenie pre spätný výpis odozvy. Na výpis odozvy v nástroji slúži konzola, ktorá zobrazuje informácie zo zariadenia. Nástroj taktiež umožňuje ukladať a načítať klientskú reprezentáciu programu pre prenositeľnosť a znovupoužiteľnosť.

Testovanie nástroja ukázalo jeho efektívnosť v zjednodušení programovania pre vstavané systémy, čím sa stáva cenným zdrojom pre začiatočníkov, ktorí sa chcú v tejto oblasti vzdelávať a hobby programátorov. Práca tiež natieňuje možné budúce vylepšenia na rozšírenie jeho schopností, ako sú podpora väčšieho množstva periférií, podpora viac možností komunikácie a napríklad možnosť zapúzdrenia časti toku pre jeho jednoduchšie použitie. Tento výskum prispieva do oblasti poskytovaním praktického riešenia, ktoré premošťuje medzeru medzi komplexnými programovacími konceptami a prístupnosťou technológií pre užívateľov.

Hoci je textové programovanie omnoho bežnejšia metóda programovania, ktorá ponúka väčšiu kontrolu a slobodu pri písaní inštrukcií, ktoré sa majú vykonať, vizuálny nástroj umožňuje prekonať bariéru bežných programátorov, ktorú mikrokontroléry môžu predstavovať.

Visual Programming Toolkit Based on MicroPython for ESP32 Platform

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Václav Šimek. The supplementary information was provided by Sergei Silnov. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Daniel Paul
May 14, 2024

Acknowledgements

I would like to express my profound gratitude to my supervisor Ing. Václav Šimek, for his academic mentorship and insightful, constructive, and valuable feedback. I am also deeply thankful to Sergei Silnov for his technical expertise and his generous, encouraging spirit. Lastly, my heartfelt appreciation goes to my family and friends for their unwavering support.

Contents

Chapter 1

Introduction

1.1 Motivation

In today's world, which surrounds us at every step with modern technology, it is necessary to keep up with what is happening around us. Everywhere we look there are sensors, cameras, microphones, and other peripherals that are integral parts of our lives via Internet of Things (IoT) solutions. This interconnected web of devices is transforming the way we interact with the world, offering efficiency, connectivity, and convenience. Sensors are present in our smartphones, wearables, home appliances, and even in urban architecture, making our lives much easier by saving precious time, collecting and analyzing data, and transmitting it to where it is needed. These are the reasons, why understanding and programming microcontrollers should be more available for people, who are not well acquainted with this task. The motivation behind this thesis is to create a lightweight, understandable entry point for people, who want to get into programming microcontrollers. The visual aspect of the toolkit serves the purpose of helping ease the reasoning and visualization of information. My motivation for this thesis is rooted in a belief, that the beauty of programming lies within the creativity of a person behind the computer and the enjoyment of creating and learning new things while avoiding the pitfalls of frustration and continuous disappointment.

1.2 What will this thesis discuss

Firstly, in chapter 2, this thesis will discuss what is visual programming paradigm, its properties, why it is useful, compare visual programming with traditional text-based programming, and compare a few types of visual programming languages. Secondly, the chapter 3 will look into modern technologies and approaches for web-based applications, comparing single-page and multi-page applications, different JavaScript frameworks and how they implement the key concepts of modern web application development. Thirdly, chapter 4 will look at Espressif chips, MicroPython, and some of its notable libraries for IoT development and means of programming the ESP32 platform. Consequently, in chapter 5 this thesis will propose a solution for implementing a web-based toolkit for programming the ESP32 platform in MicroPython, define technologies used, and clarify the design. In chapter 6 this thesis will discuss technical aspects and details of the implementation. Chapter 7 will outline what problems were faced during implementation, what solutions were used for these obstacles, and in what ways was the toolkit tested. Lastly, chapter 8 will summarize the results of the thesis and propose possible extensions.

Chapter 2

Brief Survey of Visual Programming Approaches

This chapter serves as an overview of visual programming concepts. Firstly, it will explore their key characteristics and how they facilitate the creation of software in comparison to text-based programming languages. Lastly, it will cover specific examples of visual programming languages, offering a detailed description of their approach.

2.1 Visual Programming Paradigm

Visual programming is an intuitive method of developing software applications without the need for traditional coding. It involves a graphical representation of the interaction between various components, enabling developers to efficiently and expediently create sophisticated programs. Visual programming languages (VPLs) consist of systems in which icons, symbols, charts, and forms are used to specify a program [36]. These visual elements are used as a more intuitive representation of programming structures and allow users to implement their ideas in a more graphical and accessible manner.

2.2 Comparison of Visual Programming and Traditional Programming

The traditional approach to writing software consists of writing code in a text-based programming language. The main stumbling block is the necessity of a strong understanding of syntax, semantics, and concepts of programming, including algorithms and data structures. Visual programming, on the other hand, emphasizes the usage of visual elements such as flowcharts, icons, and symbols in typically drag-and-drop interfaces, which are in general more user-friendly [56]. Table 2.1 shows the comparison and the main differences between visual programming and traditional programming.

Aspect	**Visual Programming**	**Traditional Programming**
Coding Approach	Utilizes graphical elements, drag-and-drop interfaces to design software	Writing text-based code in programming language
Suitability for Beginners	Beginner-friendly, suitable for people without a coding background	Requires a strong understanding of programming concepts and syntax
Education Focus	Widely used in education to teach programming concepts	Less commonly used for education purposes due to text-based complexity
Development Speed	Accelerates development for simple applications and automation tasks	Offers more control and flexibility, but slower for simple tasks
Complexity	Ideal for simple to moderately complex applications	Suited for complex software development, requiring advanced coding skills
Control vs. Ease of Use	Emphasizes ease of use and visual design over fine-grained control	Offers fine-grained control, but comes with a steeper learning curve
Use Cases	Suitable for prototyping, automation, user-friendly interfaces	System-level software, complex applications, performance-critical software

Table 2.1: Comparison of Visual and Traditional Programming. Obtained at [56].

2.3 Elements of Visual Programming Languages

Icons are designed to represent specific actions within a program, for example, mathematical operations like addition or subtraction, or they can correspond to a control flow statement such as the if-else condition.

Symbols denote specific data types or variables within a program.

Charts are used to visualize the flow and structure of a program, commonly taking the form of flowcharts, where arrows are used to represent program logic and control flow.

Forms in VPLs are graphical user interfaces (GUIs) that allow users to input data into their programs. They are generally represented as input fields, buttons, checkboxes, or other interactive elements.

2.4 Cognitive Effects of Visual Languages

The graphic nature of visual languages enhances pattern recognition and spatial reasoning, both of which are crucial in visual programming. This is due to reduced syntax complexity as graphical elements eliminate the need for precise syntax rules. This can be encouraging for beginners who find traditional programming intimidating or non-intuitive. In addition, VPLs operate at higher levels of abstraction, allowing users to work with a more conceptual representation of code, which can help them focus on the logic and structure of the program instead of being slowed down by low-level details.

2.4.1 Direct manipulation

Direct manipulation allows users to execute actions by directly interacting with visually displayed objects instead of having to describe the action. For instance, marking a file for deletion in a graphical user interface can often be done by dragging the file's icon into an icon depicting a trash can. This executes the command **(move <file> trash-folder)**. With direct manipulation, users can „depict“ the operations they desire; without it, they would have to describe the command to an interpreter which then translates the command into the actual action [5].

2.4.2 Visualization of information

A graph of a function makes its characteristics more explicit than a table of values of the same function, even though informationally, they are equivalent. Printed textual descriptions are frequently illustrated with pictures that serve to exemplify the ideas contained in the text, to provide different representations of the same information, or to complement what the text describes. In these cases, visual views of descriptive representations allow the viewer to discover relations that are hidden in textual representations [5].

2.4.3 Diagrammatic representation and reasoning

Many visual relations and properties are implicit in propositional representations and require extensive reasoning to be deduced, whereas visual representations such as diagrams display them explicitly at all times. These visual representations facilitate situated reasoning because they make serendipitous inferences, inferences by recognition, predictions by mental visualization, and cueing prior knowledge possible. This makes such representations especially suitable for reasoning by mental simulation [5].

2.5 Model of Visual Language

Considering visual languages usable by both computers and humans, there are three objects of interest to any theoretical or practical investigation of such languages. A *Computational system*, a *cognitive system*, and the *visual language* itself. The language is embodied by the visual representations used for communication. This visual representation encodes and conveys information that appears in the interface, the visual display. Communication requires comprehension, inference, and feedback. On the computational side, it requires parsing, interpretation or compilation, and program execution. Due to these reasons, the success of a visual language in this model rests on two criteria: its *computational tractability* (feasibility of solving a problem using an algorithm within a reasonable amount of time) and *cognitive effectiveness* (how well a representation method aligns with human cognitive processes). The model illustrated in Figure 2.1 provides a top-level understanding of the use of visual languages for human-computer interaction [5].

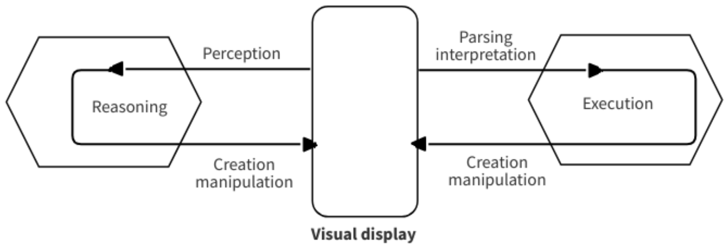


Figure 2.1: Model of visual language.

2.6 Spatial Relations

Spatial relations refer to relationships or interactions between spatial objects in a given space. These relations describe how objects are positioned oriented or connected to each other in a geometric space.

2.6.1 Objects and topology

The definition of basic geometric objects usually relies on topology which is itself a basis for defining relationships between objects. In the current context, the usual concepts of point-set topology with open and closed sets are assumed. The *interior* of a set λ_i (denoted by λ_i°) is the union of all open sets in λ_i. The *closure* of λ_i (denoted by $\overline{\lambda_i}$) is the intersection of all closed sets containing λ_i. The *complement* of λ_i (denoted by λ^{-1}) with respect to the embedding space $\mathbb{R}^n$ is the set of all points of $\mathbb{R}^n$ not contained in λ_i. The *boundary* of λ_i (denoted by $\partial\lambda_i$) is the intersection of the closure of λ_i and the closure of the complement of λ_i. It follows from these definitions that $\partial\lambda_i$, λ_i°, and $(\lambda_i^{-1})^\circ$ are mutually exclusive and $\partial\lambda_i \cup \lambda_i^\circ \cup (\lambda_i^{-1})^\circ$ is $\mathbb{R}^n$. [5]

The following restrictions apply to every pair of sets:

1. λ_i, λ_j be n-dimensional and $\lambda_i, \lambda_j \subset \mathbb{R}^n$
2. $\lambda_i, \lambda_j \neq \emptyset$
3. All boundaries, interiors, and complements are connected
4. $\lambda_i = \overline{\lambda_i^\circ}$ and $\lambda_j = \overline{\lambda_j^\circ}$

2.6.2 Spatial relations model

One formalized, systematic way to define and reason about spatial relations is Egenhofer's model. His approach distinguishes eight mutually exclusive relations. The 9-intersection is defined as a matrix shown in Formula 2.1 [5].

$$I_n(\lambda_i, \lambda_j) = \begin{pmatrix} \partial\lambda_i \cap \partial\lambda_j & \partial\lambda_i \cap \lambda_j^\circ & \partial\lambda_i \cap \overline{\lambda_j} \\ \lambda_i^\circ \cap \partial\lambda_j & \lambda_i^\circ \cap \lambda_j^\circ & \lambda_i^\circ \cap \overline{\lambda_j} \\ \overline{\lambda_i} \cap \partial\lambda_j & \overline{\lambda_i} \cap \lambda_j^\circ & \overline{\lambda_i} \cap \overline{\lambda_j} \end{pmatrix} \tag{2.1}$$

With this definition, the eight cases (*disjoint, meet, overlap, equal, covers/coveredBy, contains/inside*) can be easily characterized by the distinction between empty and non-empty intersections. For instance, the *contains* relation shown in Formula 2.2 is specified by the 9-intersection as follows [5].

$$I_5(\lambda_i, \lambda_j) = \begin{pmatrix} \emptyset & \emptyset & \neg\emptyset \\ \neg\emptyset & \neg\emptyset & \neg\emptyset \\ \emptyset & \emptyset & \neg\emptyset \end{pmatrix} \tag{2.2}$$

These eight cases of spatial relations mentioned above are called *primitive spatial relations.* Their description is shown in Table 2.2 and visual representation is shown in Figure 2.2.

Spatial relation	**Description**
Disjoint	Objects have no point in common
Touches	Objects have at least one point in common, but no part of one object is completely inside the other
Intersecting	Objects share some but not all points
Containing/Inside	One object contains another if all points of the second object are part of the first object, but interiors do not overlap
Covering/Covered_by	One object covers another if the interiors overlap and the exterior of covering is within the exterior of the covered object
Equal	Objects occupy the same space

Table 2.2: Description of primitive spatial relations.

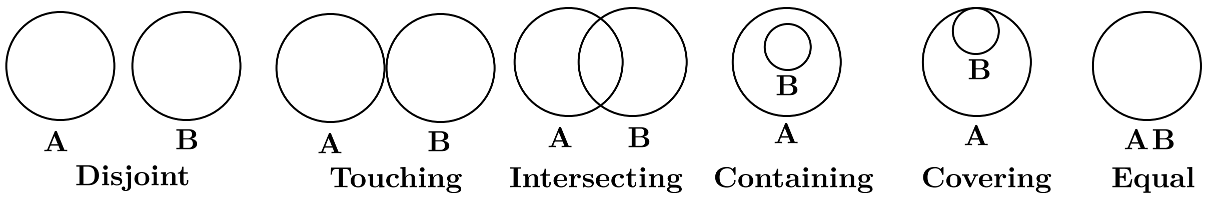


Figure 2.2: Primitive Relations Between A and B.

Primitive spatial relations can be used to build higher-level spatial relations. These include *directly_contains, pointing_to, starting_from, linked_with and part_of.* Their Description of mentioned higher level relations is shown in Table 2.3 and visual representation is shown in Figure 2.3 [5].

Spatial relation	**Description**
Directly_contains	A subset of the containing/inside relation
Linked_with	Connectivity of any two-dimensional object touching a line or arrow that connects, possibly through a chain, to another two-dimensional object.
Starting_from\Pointing_to	The direction of line segments (applies only to arrows)
Part_of	Describes partonomy of objects

Table 2.3: Description of higher-level spatial relations.

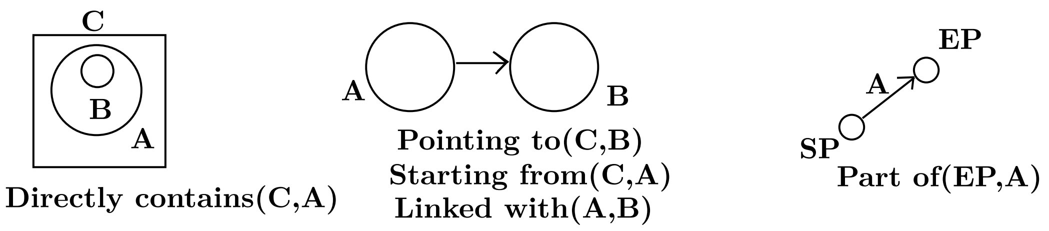


Figure 2.3: Visual representation of higher-level spatial relations

2.7 Types of Visual Programming Languages

There is no clear way to define a type of visual programming language as most of them include multiple paradigms. The most common are Block-based Programming, Flowcharts, State Machines.

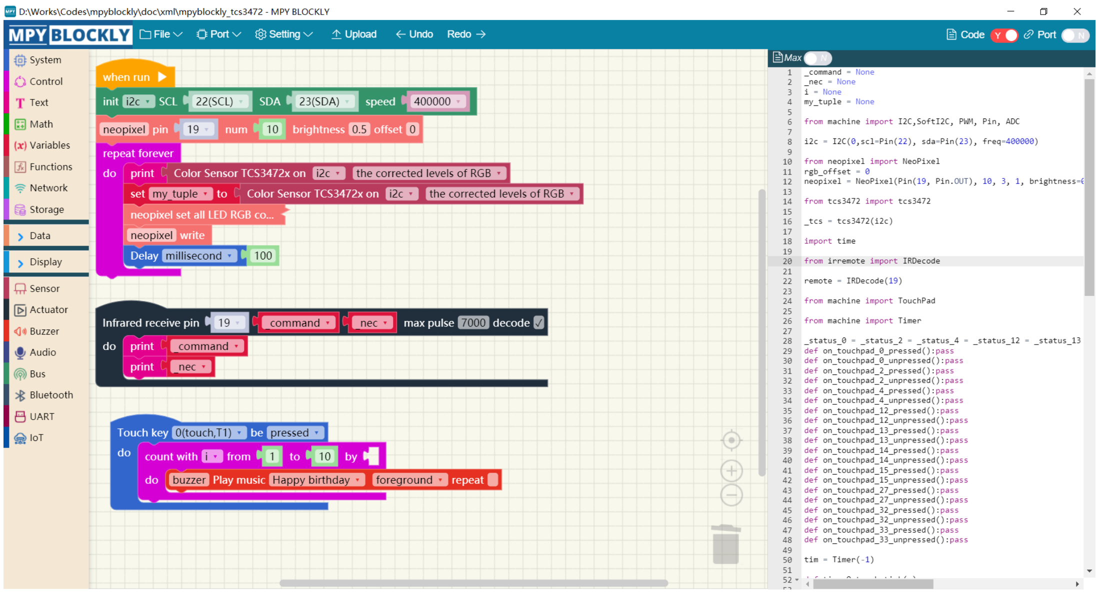
2.7.1 Block-based programming

Block-based programming refers to a programming language and integrated development environment that separates executable actions into modular portions called blocks. These blocks are generally represented with icons that can be clicked and dragged to reorder them. Editable fields, like drop-down menus, allow developers to provide further input. The graphical representation of the code can demonstrate the process to new users who may be unfamiliar with programming. This approach can be easier to learn than traditional text-based programming based on languages, like Python or Java. However, block coding is far more limited than text languages, which require more specific instructions from the programmer to complete actions [26].

MPY Blockly

MPY Blockly is a free visual programming tool for MicroPython on ESP32. By stacking colored blocks on top of each other a control program can be rapidly generated. This simple click'n'drag programming method allows developers to rapidly develop control sequences for real life microcontroller projects. MPY Blockly also supports standard 'text' programming for those who prefer to use a text editor for programming [35].

It implements blocks for conditions and loops, mathematical functions, variables, string operations, functions, network communication, storage, sensors, actuators, buses, audio, Bluetooth, NeoPixel LEDs, and many more [35]. An example of a program defined in MPYBlockly can be seen in Figure 2.4.



```
when run
init i2c SCL 22(SCL) SDA 23(SDA) speed 400000
neopixel pin 19 num 10 brightness 0.5 offset 0
repeat forever
do  print Color Sensor TCS3472x on i2c the corrected levels of RGB
    set my_tuple to Color Sensor TCS3472x on i2c the corrected levels of RGB
    neopixel set all LED RGB co...
    neopixel write
    Delay millisecond 100

Infrared receive pin 19 _command _nec max pulse 7000 decode ✓
do  print _command
    print _nec

Touch key 0(touch,T1) be pressed
do  count with i from 1 to 10 by
    do  buzzer Play music Happy birthday foreground repeat
```

```
_command = None
_nec = None
i = None
my_tuple = None

from machine import I2C,SoftI2C, PWM, Pin, ADC

i2c = I2C(0,scl=Pin(22), sda=Pin(23), freq=400000)

from neopixel import NeoPixel
rgb_offset = 0
neopixel = NeoPixel(Pin(19, Pin.OUT), 10, 3, 1, brightness=

from tcs3472 import tcs3472

_tcs = tcs3472(i2c)

import time

from irremote import IRDecode

remote = IRDecode(19)

from machine import TouchPad

from machine import Timer

_status_0 = _status_2 = _status_4 = _status_12 = _status_13
def on_touchpad_0_pressed():pass
def on_touchpad_0_unpressed():pass
def on_touchpad_2_pressed():pass
def on_touchpad_2_unpressed():pass
def on_touchpad_4_pressed():pass
def on_touchpad_4_unpressed():pass
def on_touchpad_12_pressed():pass
def on_touchpad_12_unpressed():pass
def on_touchpad_13_pressed():pass
def on_touchpad_13_unpressed():pass
def on_touchpad_14_pressed():pass
def on_touchpad_14_unpressed():pass
def on_touchpad_15_pressed():pass
def on_touchpad_15_unpressed():pass
def on_touchpad_27_pressed():pass
def on_touchpad_27_unpressed():pass
def on_touchpad_32_pressed():pass
def on_touchpad_32_unpressed():pass
def on_touchpad_33_pressed():pass
def on_touchpad_33_unpressed():pass

tim = Timer(-1)
```

Figure 2.4: Example of a program in MPYBlockly. Obtained at [35].

2.7.2 Flowcharts

A flowchart is a diagram that depicts a process, system, or computer algorithm. They are widely used in multiple fields to document, study, plan, improve, and communicate often complex processes in clear, easy-to-understand diagrams. Flowcharts use rectangles, ovals, diamonds, and potentially numerous other shapes to define the type of step, along with connecting arrows to define flow and sequence [31].

The most common flowchart symbols are depicted in Figure 2.5.

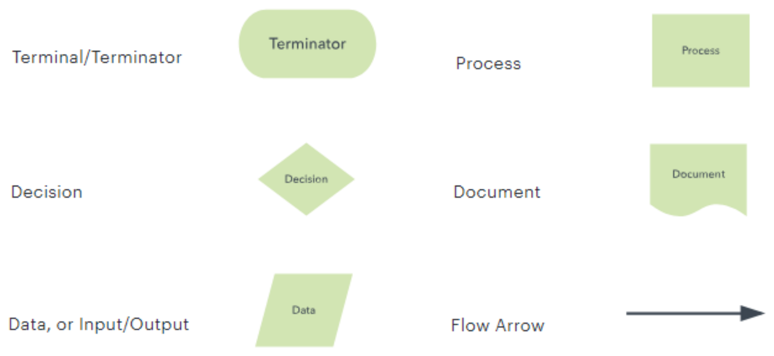


Figure 2.5: Visual representation of common flowchart symbols. Obtained at [31].

Flowcode

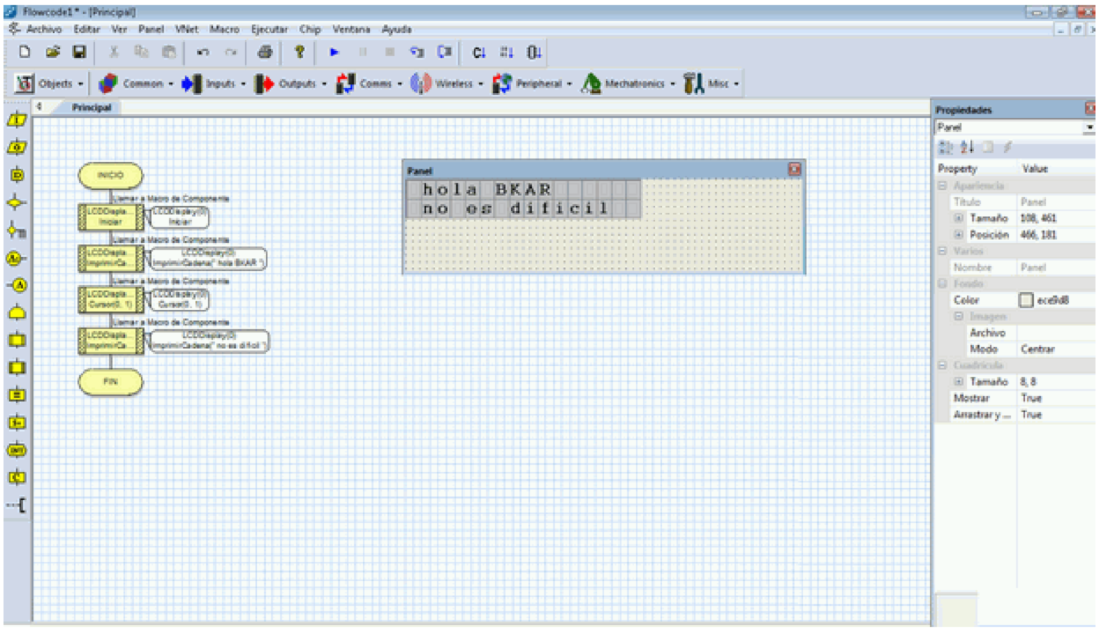
Flowcode is an advanced graphical programming IDE for electronic and electromechanical system development. It enables the user to quickly and easily develop complex electronic and electromechanical systems for control and measurement based on microcontrollers [43]. Flowcode supports Arduino, Espressif ESP32, Raspberry Pi, ARM family, and Microchip PIC devices [30]. Creating a program with flowcode involves connecting components into a flowchart, which is then compiled to C code, afterward to assembler, and flashed to a chip.

Icons represent the control flow of the flowchart. Flowcode common icons include Input, Output, Delay, Decision (if statement), Switch, Loop, C code or Interrupt icons.

Macro represents a subroutine that can be executed upon a certain event or in the control flow. Every macro has a name, description, parameters, variables, constants, and return type.

Functions are built-in routines designed to help with operations such as mathematical operations or string manipulation.

A component is essentially a library controlled by a Flowcode program extending the flowcode interface. It can be anything that is self-constrained like an electronic device, a measuring instrument, or a library of macros. Some built-in components include analog input, keypad, analog output, audio output, LED, graphical library for displays, sensors for audio, force, light, magnetic interaction, air sensors, storage, communication interfaces, networking, etc. An example of a flowcode flowchart can be seen in Figure 2.6.


Flowcode1* - [Principal]
Archivo Editar Ver Panel VNet Macro Ejecutar Chip Ventana Ayuda
Objects Common Inputs Outputs Comms Wireless Peripheral Mechatronics Misc
Principal
INICIO
Llamar a Macro de Componente
LCDDispla... Iniciar
LCDDisplay(0) Iniciar
LCDDispla... ImprimirCa...
LCDDisplay(0) ImprimirCadena(" hola BKAR ")
LCDDispla... Cursor(0, 1)
LCDDisplay(0) Cursor(0, 1)
LCDDisplay(0) ImprimirCadena(" no es dificil ")
FIN
Panel
hola BKAR
no es dificil
Propiedades

Panel

Property	Value
Apariencia	
Titulo	Panel
Tamaño	108, 461
Posición	466, 181
Varios	
Nombre	Panel
Fondo	
Color	ece9d8
Imagen	
Archivo	
Modo	Centrar
Cuadrícula	
Tamaño	8, 8
Mostrar	True
Arrastrar y ...	True

Figure 2.6: Example of flowcode flowchart showing information on the display from the thermometer. Obtained at [25].

2.7.3 State machines

A state machine is a behavior model. It consists of a finite number of states and is therefore also called a finite-state machine (FSM). Based on the current state and a given input the machine performs state transitions and produces outputs. The basic building blocks of a state machine are states and transitions. A state is a situation of a system depending on previous inputs and causes a reaction on following inputs. One state is marked as the initial state; this is where the execution of the machine starts. A state transition defines for which input a state is changed from one to another. Depending on the state machine type, states and/or transitions produce outputs [29].

Moore machines are characterized by their states and transitions. These states can generate outputs, and most importantly, the output is exclusively influenced only by the present state, without any dependence on the input. An example of Moore machine is shown in Figure 2.7.

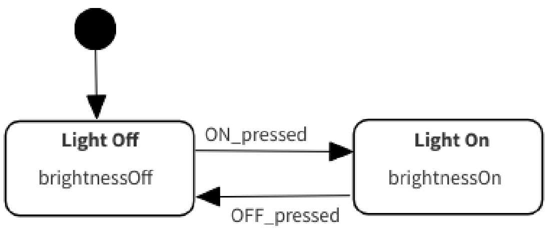


Figure 2.7: Example of Moore machine describing the light switch.

Mealy machines consist of states and transitions, where states are producing outputs. Unlike Moore machines, the output in Mealy machines is dependent on the current state and the input. An example of the Mealy machine is shown in Figure 2.8.

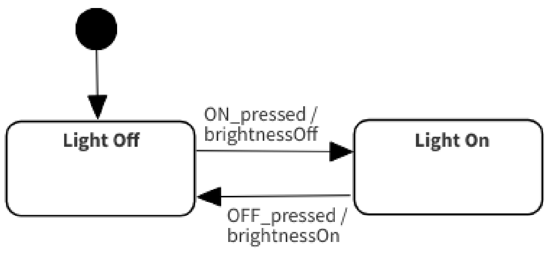


Figure 2.8: Example of Mealy machine describing the light switch.

Stateflow

Stateflow is a software that provides a graphical language that includes state transition diagrams, flow charts, state transition tables, and truth tables. Developers can use Stateflow to describe how MATLAB algorithms and Simulink models react to input signals, events, and time-based conditions. Stateflow lets model combinatorial and sequential decision logic that can be simulated as a block within a Simulink model or executed as an object in MATLAB [28].

Stateflow chart is a graphical representation of a finite state machine consisting of states, transitions, and data. *Symbols* are used, to add new data, events, or messages to the chart canvas. After a state is added to the canvas its properties need to be defined, such as name and state action. Two states can be connected with a transition that needs to meet some defined condition [54]. An example of a stateflow chart can be seen in Figure 2.9. There are two ways to use Stateflow charts. The first one involves designing logic by using state charts and executing that logic as MATLAB programs. The second one is using the state Stateflow chart as blocks in Simulink models.

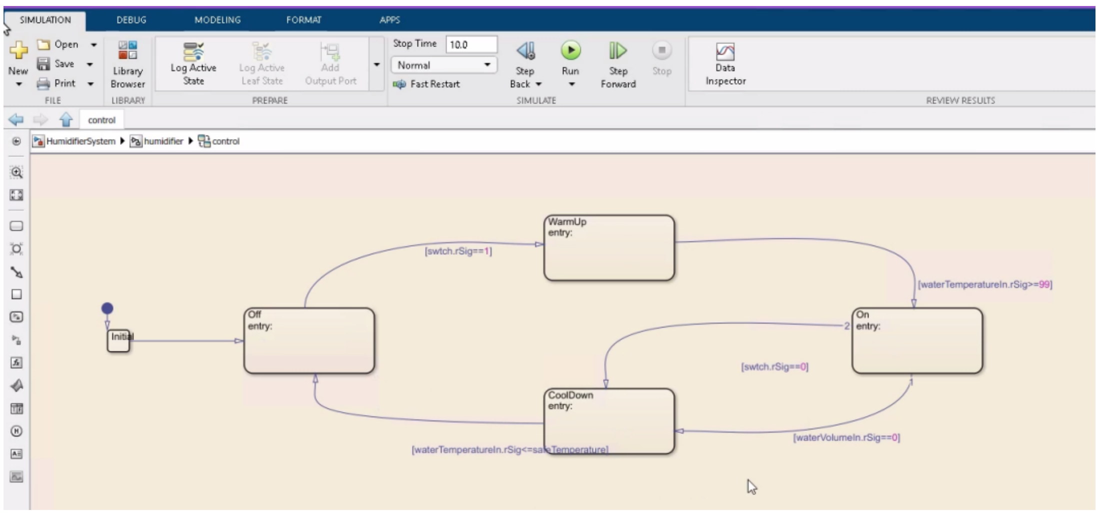


Figure 2.9: Example of Stateflow chart controlling water temperature. Obtained at [55].

Chapter 3

Modern Technology for Web-Based Applications

This chapter serves as an overview of contemporary technologies and approaches used in development of web-based applications. It explores commonly used paradigms for web development and the differences between different modern JavaScript frameworks.

3.1 Single-page vs. Multi-page Applications

One of the important things to note when developing a web-based application is the significance of layout, as it affects the overall user experience of users. This is why it is critical to understand the concept of navigation, so the user is not confused while working with the application. Beyond the visual structure, the quantity of pages within the application is also a crucial factor. The most common approaches for navigation are **Single-page** and **Multi-page** application [27].

3.1.1 Single-page application (SPA)

Adopting a single-page design for web applications requires keeping in mind simplicity and cleanliness. Using this approach, the home page serves as the only container for all information, contributing to a unified and straightforward user experience. The main idea lies within the minimalistic structure and cohesive interface, allowing users to access information easily without the need to navigate through multiple pages [27].

Advantages

- Easy navigation and clean layout
- Mobile friendly
- Easy to build and update

Disadvantages

- Limited space
- Hard to share or track

3.1.2 Multi-page application (MPA)

The multi-page design revolves around having a home page serving as a hub for general information with a navigation bar leading to multiple internal pages. Unlike SPA, which loads a single HTML file and updates the content dynamically, MPA is based on loading different HTML files and the page reloads when navigating to the content [27].

Advantages

- Scalability
- Better search engine optimization
- Usability
- Navigation

Disadvantages

- Maintenance
- Less user-friendly on mobile
- More distracting

3.2 Javascript Frameworks

Javascript is known for its large, rich, and dynamic ecosystem of frameworks and libraries. The key classes of these systems are frameworks for building the frontend of web-based systems, such as React or Svelte. These frameworks are built with the intention of implementing modern, dynamic, and responsive web interfaces, also called single-page applications [23].

Key concepts of javascript frontend frameworks include:

Component-Based architecture - UI is broken into modular, reusable components.
State management - handling user interactions and dynamic updates.
Virtual DOM - lightweight of real DOM, optimizing the rendering process.
Data Binding - synchronization between the application's data model and the UI.
Tooling and Development Workflow - development tools and build systems.

This section will discuss various modern JavaScript framework approaches to these key concepts and cover some libraries suitable for implementing visual applications.

3.2.1 React

React is a library developed by Facebook to help developers build user interfaces as a tree of pieces called components. A component in the context of React is a mixture of HTML and JavaScript that captures the logic required to display a small section of a larger UI. Multiple of these are combined to build a complex web application [4].

JSX

JSX is an XML-like syntax extension to ECMAScript without any defined semantics. It is intended to be used by various preprocessors (transpilers) to transform these tokens into standard ECMAScript. The purpose of JSX is to specify concise and familiar syntax for defining tree structures with attributes [32].

React components and props

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML. React components have two variations, **Class components** and **Function components** [57].

Class components inherit from **React.Component**, which gives them access to React.Component's functions. They also need to implement **render()** method, returning HTML.

The function component also returns HTML and behaves the same way as the Class component, but requires much less code. Components can be passed as **props**, which stands for properties. Props are similar to function arguments and they can be sent to components as attributes.

React state

React components have built-in state object. The state object stores property values that belong to the component. Changes in the state are triggered in response to user input. When the state object changes, the component re-renders [58].

React hooks

Hooks allow to use of different React features from components. There are different types of built-in hooks or users can define custom ones [45].

State hook lets a component remember information like user input. Such hook is often used to update the variable and it is defined with **useState()** method.

Context hook lets a component receive information from distant parent components without passing it as props. It reads and subscribes to a context. This is especially useful when passing information from the app's top-level component to the deepest ones. Context hook is defined with **useContext()** method.

Ref hook lets a component hold some information that is not used for rendering, like DOM node or timeout ID. Unlike the state, updating a ref hook does not re-render a component. They are useful when working with non-React systems such as built-in browser API. Ref hook is defined with **useRef()** method.

Effect hook lets a component connect to and synchronize with external systems, usch as network, browser DOM, animations, widgets, and other non-React code. Effect hook is defined with **useEffect()** method.

Other hooks include **performance hooks** for optimizing re-rendering performance, **resource hooks** for reading values from resources such as promises or contexts, or a developer can define their own hooks.

3.2.2 ReactFlow

ReactFlow is a fully-fledged library designed for the React framework, providing developers with a tool for creating interactive and dynamic flow-based diagrams [59]. Its core consists of creating nodes, placing them on the canvas, and interconnecting them with other nodes. The nodes can be manipulated, moved around, connected to other nodes, or deleted and they can have modified properties based on the user's actions. Furthermore, ReactFlow's modular design and customizable components allow developers to adjust the appearance and data contained in the node and the edges. The library provides many features such as drag-and-drop capabilities, styling options, sub-flows, and additionally the canvas with flow minimap, zooming, and panning out of the box. All the details of the library are contained in well-written API documentation. All of these features make it a well-tailored choice for implementation of a visual programming toolkit.

Node

Node is a React component encapsulated in a ReactFlow wrapper that allows the component to provide functionality like selecting and dragging. The wrapper also provides properties like position and data. The data property of a node is important because it provides developers with a way to create their custom nodes with specific functionality. A ReactFlow node can contain several *handles*. A handle is a point representing a connection from/to a node and can be either target (input) or source (output). An example node containing source and target handles and text input can be seen in Figure 3.1.

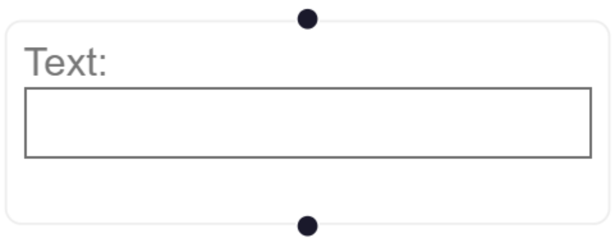


Figure 3.1: An example of ReactFlow custom node with text input.

Edge

Edge is a React component enclosed in a ReactFlow wrapper similar to node, that allows an edge to connect to node handles and renders it as a curve. It contains information about the IDs of nodes that are connected by this edge and it also contains properties like labels and data allowing it to implement tailored functionality. Edges can also be styled to further clearly imply the purpose of the connection between the nodes. An example of an edge connecting to nodes is depicted in Figure 3.2.

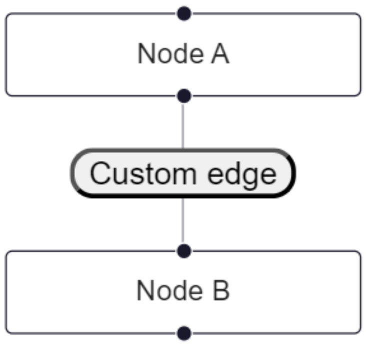


Figure 3.2: An example of ReactFlow custom edge with a button connecting two nodes.

Subflows

Subflows in ReactFlow represent a flow inside a node. A flow is encapsulated in the parent node, effectively creating a hierarchy of interconnected child nodes. The primary goal of utilizing subflows is to create a container for a set of interconnected child nodes, forming a single unit within the larger flow diagram. An example of a subflow can be seen in Figure 3.3.

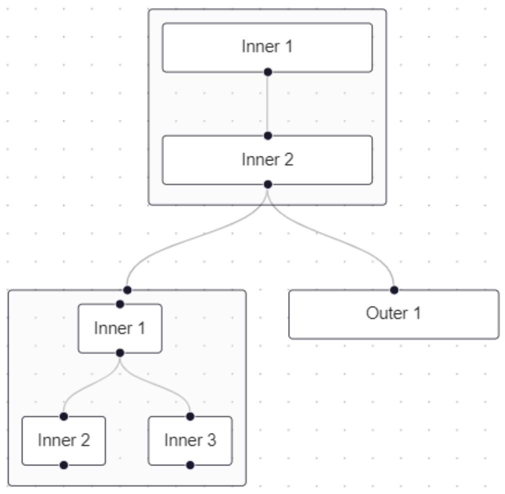


Figure 3.3: An example of ReactFlow subflows.

3.2.3 Zustand

A small, fast, and scalable barebones state management solution. It leverages React Hooks, such as useReducer and useState, to provide an imperative or functional approach to state updates, reducing boilerplate code and making it developer-friendly [37].

Zustand serves as an alternative to more complex state management solutions like Redux. Developers can use it to efficiently manage the state in React applications while keeping their codebase clean and concise.

3.2.4 Svelte

Svelte provides a different approach to building web applications than some other frameworks. While frameworks like React and Vue do the bulk of their work in the user's browser while the application is running, Svelte shifts the work into a compile step that happens only when the application is built. The compiler generates optimal JavaScript code without any runtime overhead [34].

Svelte components and props

Components are building blocks of Svelte applications. They are written as a superset of HTML. Svelte uses **export** keyword to mark a variable declaration as a property, which means it becomes accessible to parent components. It is possible to specify a default value of the prop variable in case the parent element does not provide any.

Runes

Runes are symbols that influence the Svelte compiler. These symbols offer developers a more straightforward and explicit way to manage reactive variables in their code [1].

State rune marks values as reactive. This approach aims to make the reactivity more explicit and predictable.

Derived rune ensures, that derived values are recalculated lazily and kept in sync. This approach aims to make the framework more type-safe and maintain consistent behavior across component logic.

Effect rune offers an alternative to handle side effects in response to reactive changes. It is designed to replace lifecycle methods like *onMount*.

3.2.5 SvelteFlow

SvelteFlow is a Svelte implementation of ReactFlow library. It provides tools to create interactive and dynamic flowcharts using nodes and edges. Its main downside is that it does not support all the features that ReactFlow provides such as custom edges or plugins as it is at the time of writing of this thesis in alpha version [60].

Chapter 4

Using Python on ESP32 Platform

This chapter will cover the AIoT devices and their variants developed by the company Espressif Systems (Czech) s.r.o. [53]. Additionally, it will examine the integration of MicroPython, clarifying its implications for embedded systems development.

4.1 Espressif Microcontrollers

Espressif microcontrollers are a series of low-cost, low-power system-on-a-chip (SoC) microcontrollers developed by Espressif Systems [52]. They are known to be running under ultra-low power consumption, supporting various communication protocols like Wi-Fi and Bluetooth via their SPI/SDIO or I2C/UART interfaces, along with GPIO pins for an ability to connect a variety of peripherals, making them useful devices for IoT integration or smart home infrastructure.

4.1.1 ESP32 specification

ESP32 is a low-cost system on a chip used in the development of IoT solutions. Full specification can be found in ESP32-Datasheet [51]. A picture of the ESP32 microcontroller can be seen in Figure 4.1 and its functional block diagram is shown in Figure 4.2.

Some notable features include:

- Wi-Fi (2.4 GHz), up to 150 Mbps
- Bluetooth v4.2 and Bluetooth Low Energy (BLE)
- Xtensa single/dual-core 32-bit LX6 microprocessor(s)
- 520 KB SRAM
- 34 programmable GPIOs
- Two 8-bit DAC and one 12-bit ADC
- Four SPI, two I2C, two I2S, three UART interfaces
- One host (SD/eMMC/SDIO), One slave (SDIO/SPI)

Figure 4.1: ESP32 development board. Obtained at [46].

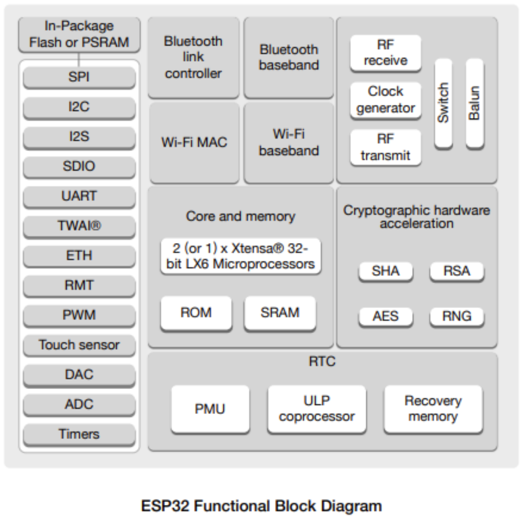


Figure 4.2: ESP32 functional block diagram. Obtained at [51].

4.1.2 ESP32-S3 specification

ESP32-S3 is a low-power MCU-based system on a chip with higher performance than its predecessor ESP32. Full specification of the microcontroller can be found in ESP32S3-Datasheet [50]. A picture of the ESP32S3 microcontroller can be seen in Figure 4.3 and its functional block diagram is shown in Figure 4.4.

Some notable features include:

- Wi-Fi (2.4 GHz)
- Bluetooth v5.0, BLE and Bluetooth Mesh
- Xtensa® dual-core 32-bit LX7 microprocessor
- 512 KB SRAM
- 45 programmable GPIOs
- Two 12-bit SAR ADCs
- Three UART, two I2C, two I2S interfaces
- LCD interface
- One temperature sensor
- 14 touch sensing IOs
- One USB Serial/JTAG controller
- One SD/MMC host controller with 2 slots

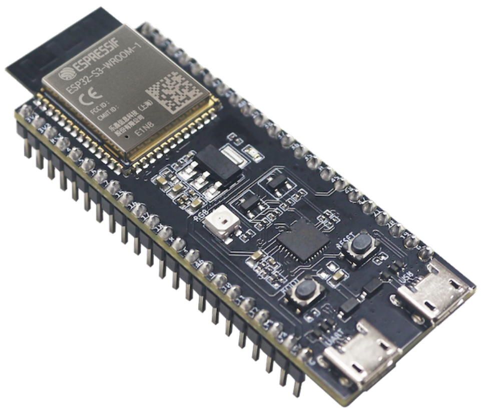


Figure 4.3: ESP32S3 development board. Obtained at [47].

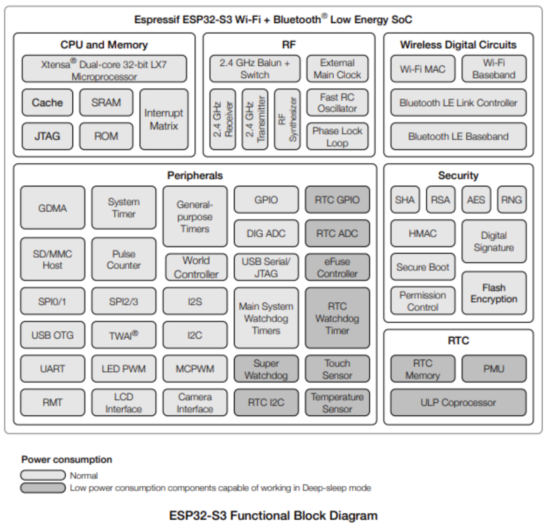


Figure 4.4: ESP32S3 functional block diagram. Obtained at [50].

4.2 Means of Programming the ESP32 Platform

4.2.1 ESP-IDF

ESP-IDF is Espressif's official IoT Development Framework for the ESP32, ESP32-S and ESP32-C series of SoCs. It provides a self-sufficient SDK for any generic application development on these platforms, using programming languages such as C and C++. [48] To develop with ESP-IDF framework it is required to install a toolchain to compile code for ESP32 chips, build tools such as CMake and Ninja to build a full application, and install all ESP-IDF dependencies [49]. The workflow of ESP-IDF is shown in Figure 4.5.

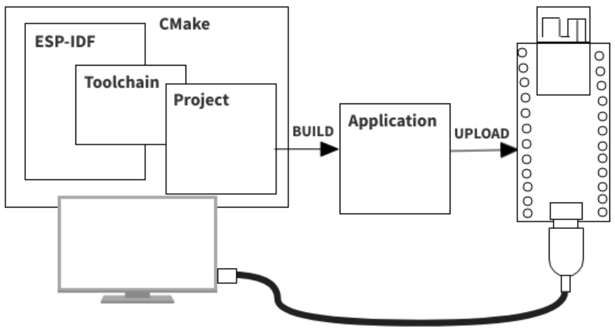


Figure 4.5: ESP-IDF workflow.

4.2.2 Rust

Rust programming language provides dependencies for programming Espressif chips. There are two approaches that can be chosen for development: The **std** library, also called as Standard Library, or the **core** library, also called no_std (bare metal development) [38]. The rust official ESP-related crates (dependencies) and build tools are managed by the organization **esp-rs** [41]. In order to flash an ESP project written in Rust, the utilization of **espflash** project is required. [42]

Standard Library

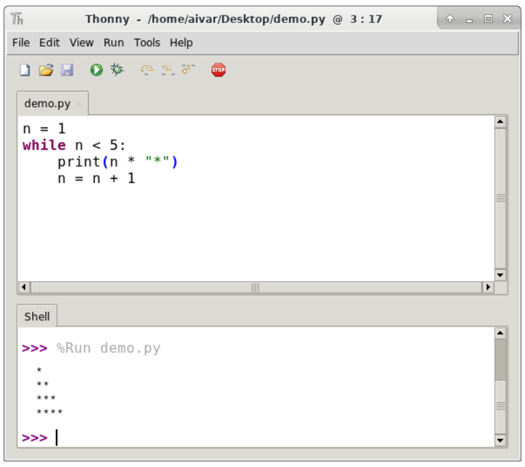
Espressif provides the previously mentioned C-based development framework ESP-IDF. It provides a **newlib** environment with enough functionality to build the Rust **standard** library on top of it. The std crate provides a rich set of functionality that can be used to build applications quickly and efficiently, without worrying, too much, about low-level details [39].

Core Library

No_std rust uses the Rust **core** library for the bare-metal approach. As this library is part of the Rust standard library, a no_std crate can be compiled in a std environment. Bare metal allows more customization and fine-grained control over the behavior of an application. Another benefit of this approach is a small memory footprint and direct hardware control, such as low-level device drivers or access to specialized hardware features [40].

4.2.3 MicroPython

MicroPython can easily run on the ESP32 platform. In order to install MicroPython on a particular ESP32 microcontroller, it is required to download the MicroPython binary file specifically designed for that ESP32 model and then flash the firmware using the **esptool** software [22]. It is also recommended to erase the entire flash from the device before doing this procedure [7]. Afterward, a MicroPython code can be run from REPL or by writing a MicroPython program using, for example, **Thonny IDE** shown in Figure 4.6 [2]. Additionally, the Python files can be transferred to the ESP device with the use of **mpremote**. It is a command line tool used to interact with serial ports. It can be used to transfer files to the ESP device or run the Python files [9].


Thonny - /home/aivar/Desktop/demo.py @ 3 : 17
File Edit View Run Tools Help
demo.py
Shell

```
n = 1
while n < 5:
    print(n * "*")
    n = n + 1
```

```
>>> %Run demo.py
 *
 **
 ***
 ****
>>> 
```

Figure 4.6: Thonny IDE. Obtained at [3].

4.3 MicroPython

4.3.1 Overview

MicroPython is an open-source, compact, and effective implementation of the Python 3 programming language, designed to run on microcontrollers and in limited contexts. It is a full Python compiler and runtime that runs on the bare-metal, with an interactive prompt (the REPL) to execute commands immediately, along with the ability to run and import scripts from the built-in filesystem. However, it comes only with a limited subset of the Python standard library. MicroPython also includes advanced capabilities, including an interactive prompt, arbitrary precision integers, closures, list comprehension, generators, exception handling, and more [24]. Initially, MicroPython was created for specially designed microcontroller Pyboard, but the community has created implementations for ESP32 microcontrollers.

4.3.2 Libraries in MicroPython

MicroPython provides built-in modules that mirror the functionality of Python standard library (os, time) as well as MicroPython-specific modules (e.g. Bluetooth, machine). Most Python standard library modules implement a subset of the functionality of the equivalent in Python module due to RAM constraints [10].

Some notable Python standard libraries and micro-libraries include:

array - Arrays of numeric data
asyncio - Asynchronous I/O scheduler
builtins - Builtin functions and exceptions
collection - Collection and container types
gc - Control of the garbage collector
hashlib - Hashing algorithms
json - JSON encoding and decoding
math - Mathematical functions
os - Basic operating system services
random - Pseudo-random number generator
select - Wait for events on a set of streams
ssl - SSL/TLS module
sys - System specific functions
time - Time related functions
_thread - Multithreading support

Some notable MicroPython libraries include:

btree - Simple BTree database
cryptolib - Cryptographic ciphers
machine - Functions related to the hardware
micropython - Control of MicroPython internals
neopixel - Control of WS2812/NeoPixel LEDs
network - Network configuration

And finally some libraries specific to ESP32 include:

esp32 - functionality specific to ESP32 (RMT, ULP co-processor...)
espnow - support for ESP-NOW wireless protocol

4.3.3 Microdot

Microdot is a minimalistic Python web framework inspired by Flask. Given its size, it can run on systems with limited resources, such as microcontrollers. Both standard Python (CPython) and MicroPython are supported [33]. Microdot framework is used to run an HTTP server, providing decorators for defining routes to endpoints, defining HTTP methods, and handling requests and responses.

4.3.4 Asyncio

Asyncio is a CPython library designed to write concurrent code using async/await syntax (its MicroPython equivalent is called uasyncio). This allows to control the execution of Python concurrent coroutines, performs network IO and interprocess communication, control subprocesses, distribute tasks via queues, **create and manage event loops**, **handle OS signals** and **bridge callback-based libraries and code**. [11]

Coroutines

At the heart of async IO are coroutines. In Asyncio, a coroutine is a specialized version of a Python generator function. It can suspend its execution before reaching *return* statement, and it can indirectly pass control to another coroutine for some time. The syntax *async def* introduces either a **native coroutine** or an **asynchronous generator**. The asynchronous generator-based coroutine is defined with decorator *@asyncio.coroutine* and by *yield from func()* syntax, meanwhile, native coroutine uses the keyword *await.* Furthermore, when await is used, it hands over control of the function back to the event loop, essentially pausing the execution of the surrounding coroutine. [44]

Chained Coroutines

It is an async IO design pattern which is composed of chained coroutines, as a coroutine object is awaitable, so another coroutine can *await* it. This allows to break programs into smaller, more managable, reusable coroutines [44].

Queue

The asyncio package provides queue classes designed to facilitate asynchronous communication and coordination between different parts of a program. Their primary use is intended for coordinating multiple producers and consumers without direct association. Unlike the chaining approach, individual producers add items to the queue at varying times. Consumers without signals pull items from the queue as they are added. This design eliminates the need for direct connections between consumers and producers, which are asynchronous, and the queue acts as a communication channel between them [44].

Event loop

Event loop is a mechanism used to monitor coroutines, take feedback on what's idle, and look for things that can be executed in the meantime. It is capable of reactivating an idle coroutine upon the fulfillment of the awaited condition, effectively resuming the execution of coroutine when the required resource or event becomes accessible. Management of the event loop is handled by *asyncio.run()* function, which handles getting the event loop, running tasks until they are all marked as complete and then closing the event loop. The important thing to note is that coroutines are most effective when associated with an event loop, because an event loop serves as a central coordinator, enabling efficient handling of multiple asynchronous operations and maximizing program responsiveness by scheduling tasks [44].

Tasks

Scheduling the execution of a coroutine on the event loop creates a task. This is achieved by calling *asyncio.create_task(coroutine)* followed by *asyncio.run()*. The problem with this pattern is that if the coroutine is not awaited within the asyncio.run() callback, it may finish before the callback itself signals that it is complete [44].

4.3.5 Machine

Machine is a MicroPython module containing specific functions related to the hardware on a particular board. Most functions allow for direct and unrestricted access to hardware blocks on a system, like CPU, timers, buses, etc. All callbacks used by functions and class methods in machine module should be considered as executing in an interrupt context for both physical devices with IDs >= 0 and „virtual devices“ with negative IDs like -1 [12].

Class Pin - Control of I/O pins

A **pin object** is used to control I/O pins (also known as GPIO). Pin objects are associated with physical pins that can drive an output voltage and read an input voltage. The pin object can be controlled with multiple class functions, namely *Pin.on()* to set pin to „1“ output level, *Pin.off()* to set it to „0“ output level or *Pin.irq()* to configure interrupt handler to be called when the trigger source of the pin is active. Possible triggers for triggering interrupts are *Pin.IRQ_FALLING* - on falling edge, *Pin.IRQ_RISING* - on rising edge, *Pin.IRQ_Low_LEVEL* - on low level, *Pin.IRQ_HIGH_LEVEL* - on high level. You can also set *wake* parameter when setting interrupt handling which defines the power mode in which interrupt can wake up systems. Possible options are *machine.IDLE*, *machine.SLEEP* or *machine.DEEPSLEEP* [13].

Class Signal

The **signal class** is an extension of the Pin class. Unlike the Pin class, the signal allows the pin to have an asserted or deasserted state, and it introduces the concept of logical inversion, enabling the signal to be active-low or not based on the configuration. Signal should be used for controlling simple on/off devices like LEDs, multi-segment indicators, relays, buzzers, or read simple binary sensors. A signal constructor takes a *pin object* as an argument and *invert* - if True, the signal will be inverted (active low) [14].

Class ADC

The **ADC class** provides an interface to analog-to-digital converters and represents a single endpoint that can sample a continuous voltage to a discretized value. The ADC constructor takes a pin object as an argument. Raw analog value can be read by the *read_u16()* function, and value in microvolts can be read by the *read_uv()* function. [15]

Class PWM

The *pulse with modulation class* provides a way to generate and control pulse-width modulated signals on pin, allowing users to adjust the frequency and duty cycle of the signal, making it useful for tasks like controlling motor speed or LED brightness [16].

Class UART

The *UART class* provides a UART/USART duplex serial communication protocol. The unit of communication is a character-wide 8 or 9 bits. It can have set baudrate, parity bits and number of stop bits. For reading the number of bytes from the UART bus the *UART.read(nbytes)* function is defined. Similarly, in order to write the buffer of bytes to bus the *UART.write(buf)* function is defined. Lastly, for creating a callback to be triggered when data is received on the UART a *UART.irq()* function can be used [17].

Class SPI

SPI is a synchronous serial protocol that is driven by a controller. At the physical level, a bus consists of 3 lines: SCK, MOSI, MISO. Multiple devices can share the same bus. Each device should have a separate, 4th signal, CS (Chip Select), to select a particular device on a bus with which communication takes place. Management of a CS signal should happen in user code (via machine.Pin class). In order to perform read/write operations, it is required to initialize a machine.PIN as a chip select, set it to desired value to select a peripheral, and read a number of bytes using the *SPI.read(nbytes)* function or write the bytes contained in the buffer using *SPI.write(buf)* [18].

Class I2C

I2C is a two-wire protocol for communicating between devices. At the physical level, it consists of two wires: SCL and SDA, the clock and data lines, respectively. To write the bytes from buffer to the peripheral specified by address, the *I2C.writeto(addr, buf)* is used. The function check that an ACK is received after each transfered byte and stops transmitting the remaining bytes if NACK is received. To read the number of bytes from the peripheral specified by address the function *I2C.readfrom(addr, nbytes)* is used [19].

Class Timer

Hardware timers deal with timing of periods and events. Timers are perhaps the most flexible and heterogeneous kind of hardware in MCUs and SoCs, differently greatly from model to model. The timer class defines an object executing a callback with a given period, either once or periodically [20].

Class SDCard

SD cards are one of the most common small-form factor removable storage media. They come in a variety of physical form factors. MMC cards are similar to removable storage devices, while eMMC devices are electrically similar storage devices designed to be embedded into other systems. All three forms share a common protocol for communication with their host system, and high-level support looks the same for them all. In machine module they are implemented in a class machine.SDcard. The ESP32 provides two channels of SD/MMC hardware and also supports access to SD cards through either of the two SPI ports that are generally available to the user. As a result the slot argument can take a value between 0 and 3, inclusive. Slots 0 and 1 use the built-in SD/MMC hardware, while slots 2 and 3 use the SPI ports. Slot 0 supports 1, 4 or 8-bit wide access while slot 1 supports 1 or 4-bit access; the SPI slots only support 1-bit access [21].

Chapter 5

Design of Visual Programming Toolkit

This chapter describes used technologies for the implementation of the visual toolkit for programming the ESP32 platform in MicroPython. It covers the underlying architecture and explores approaches considered for its implementation.

5.1 Used Technologies

Prior to diving into the architecture, it is necessary to define the underlying technologies used for the implementation. The frontend will utilize the **ReactFlow library** for defining flow-based flowcharts and **Zustand** library for state management of the flowchart. To implement the backend on the ESP32 platform, several MicroPython libraries are necessary. The **Microdot** library will establish a communication channel between the React application and MicroPython backend through a lightweight HTTP server. **Asyncio** will manage coroutines, handling the logic of nodes implemented in the visual toolkit. Finally, the **Machine** library will be used to handle peripheral control and manipulation. While additional libraries will contribute to the complete implementation, these are the most notable ones required for seamless operation of the backend.

5.2 Architecture

Implementing a web-based visual toolkit that will provide the functionality to build a flow that will represent a MicroPython program for the ESP32 platform is a challenging task and requires a well-defined architecture. My approach is to first implement a frontend based on a flow-based interactive dynamic environment with a drag-and-drop interface and configurable nodes and edges for the representation of a MicroPython program. The second step is to choose a data interchange format and design a well-defined structure for serialization of the MicroPython program defined by nodes and edges. The last step is to deserialize it and make it runnable by MicroPython interpreter on ESP32 platform.

5.2.1 Visual toolkit

The interface of the visual toolkit must be simple and easy to comprehend by a user. For this purpose, I have designed an interface that will contain a sidebar containing an input field for the IP address of an ESP32 device, a section with drawer components containing drag-and-drop nodes that a user will be able to place on the canvas, and an export flow button, which will send the serialized version of the flow to the ESP32 device. Each drawer will have a descriptive name for the nodes that are contained inside, e.g., the control drawer will contain control flow nodes such as conditional statements. At the bottom of the page will be a console component, which will have an informative purpose and will either inform the user about errors, such as empty configuration fields for some nodes or display feedback information from the ESP32 itself. The rest of the web interface will be filled with the canvas containing the visual flowchart itself.

Node

Each node in a visual toolkit will be a custom React component consisting of its unique identifier, type and its own properties defined by input fields inside its boundaries and handles. These properties will be either HTML input fields or selection boxes for options. Every node will need to be an exclusive React component, and it will need to be treated differently. For example, some nodes, such as conditional statement node will contain different number of source handles in order to redirect the flow of the program based on the output of the condition. Each change in the input field will trigger a callback which sets the new state of the node according to user input. After each change of the node state the node will be synchronized with the whole flow.

To break it down even further, presume a design of a Timer node. In MicroPython, the Timer class needs to define the timer *period* in milliseconds, a *mode*, which can take in values either Timer.PERIODIC for periodic repetition or Timer.ONE_SHOT for singular execution and a *callback*. The Timer node will contain a number type input field for defining period in milliseconds, select input field containing choices „Periodic“ and „Once“ representing their equivalents in MicroPython implementation and the callback will be the execution of the connected node to the output of the Timer node.

Edge

Edges will consist of a unique identifier a node that they connect to and a type labeling for example branching result. Their main purpose will be to represent connections between nodes and they will describe the flow of the program. They will be connecting to the handles of nodes.

Loops

Loops will be represented by a node which will define a variable and iterate over an iterable with respective iterable values being available in the variable. The body output handle of the loop will be colored blue for clear distinguishability.

Conditions

Conditional statements will be implemented as a node with two operands and one operation. This node will have two outputs, one will represent the True outcome of the condition and the latter will represent false. These outputs will be distinguishable by the color of the output handle.

Data

The data node will be represented by its value and data type, which will be one of the primitive types. Other types of data like objects will be the direct output of the specific nodes.

Variables

The variable node will be a special object which will contain a certain value specified by the input handle. Variables will be treated in a unique way as the flowchart will need to track what variables are available in order to use them as inputs in other nodes. This will be achieved, by modification of the global state of variables via the Zustand library.

Custom Python Code

There is an idea of implementing the custom Python code node, which would be used to specify the internal logic of the node with MicroPython code itself by the user. This approach is debatable, as it is not in the spirit of the visual programming language, but allows users to integrate some functionality, that might not be possible in the visual programming toolkit. It will be represented as a node, which would take one input parameter called *code*, and if the function returns any value it would need to be assigned to the variable called _output. This would allow to use *exec()* function in MicroPython to interpret the text as a MicroPython code which takes as parameters local variables. These variables would be used as a communication channel between the backend internal representation and the custom code itself.

5.2.2 Integration

It is crucial to define a means of communication between the web application and the ESP32 microcontroller. A flow-based representation will be serialized into a list of JSON objects, which will be sent via WebSocket connection to the microcontroller which will save the file into its internal storage as a flow.json file for repeatable builds. Afterward, the microcontroller will restart and deserialize the JSON object into the MicroPython program. A JSON object representing a node with edges will consist of a template shown on Listing 5.1.

```
{
  "id": "Node identifier",
  "type": "Type of the node",
  "outputs": [
    {
        "id": "Edge identifier",
        "type": "Type of the edge",
        "target_id": "Target node identifier"
    }
  ]
}
```

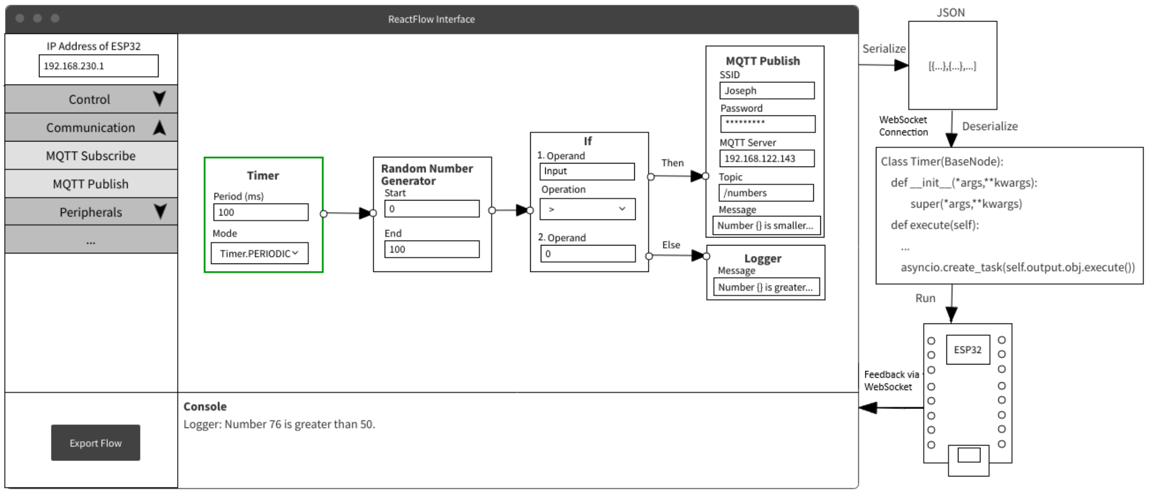
Listing 5.1: Example JSON Object.

5.2.3 Backend

The backend will be running on the ESP32 microcontroller itself in MicroPython at it will consist of an HTTP server upgrading the HTTP connection to WebSocket, receiving a JSON representation of flow from the frontend side, and loading it into internal representation. The internal representation will consist of classes that encapsulate the black-box behavior of the nodes. These classes will be the fundamental part of the execution logic and they will contain specific properties which will be used to alter their MicroPython functionality and execute function which will perform the instructions that are equivalent in the MicroPython and trigger the execute function of the next object in the flow with the output from the current object. They will be initialized by deconstructing the JSON input from the frontend application and saved to a dictionary, where id of the node will be the key and the value will be to instance of the object. After initialization, the edges will be resolved to Output classes and set to the respective objects representing nodes. Subsequently, starting nodes will be executed as starting coroutines. After the internal logic of the starting node is executed, the output will be sent to another node and this procedure will happen to all the connected nodes in a flow effectively creating a chain of coroutines.

5.2.4 Summary

In summary, the workflow consists of a frontend interface, where user defines IP address of his ESP32 microcontroller, drag-and-drops nodes on the canvas, sets their properties, connects them together, and defines starting nodes. Afterward, the flowchart can be exported onto the ESP32 device via WebSocket request, where it is saved in a JSON format in the microcontroller storage for reproducible runs, the JSON is deconstructed into the MicroPython classes, where each class represents a specific implementation of the internal logic of the node. Afterward, the Output objects are initialized and put into the node objects as properties. Finally, the flow is executed by coroutines, which are initiated from the starting nodes and the program is executed as a linked list of chained asynchronous coroutines. The architecture of a whole workflow can be seen in Figure 5.1.



Console
Logger: Number 76 is greater than 50.

```
Class Timer(BaseNode):
  def __init__(*args,**kwargs):
      super(*args,**kwargs)
  def execute(self):
    ...
    asyncio.create_task(self.output.obj.execute())
```

Figure 5.1: An architecture of the development workflow.

Chapter 6

Technical Aspects of the Implementation

This chapter describes the implementation details of the proposed architecture. It will cover elements of the ReactFlow frontend, available nodes in the visual programming toolkit and backend execution as well as communication via WebSockets.

6.1 Elements of ReactFlow Frontend

6.1.1 Canvas

ReactFlow component essentially represents a canvas. It is an interactive dynamic environment containing all the information and handling the interaction between nodes and edges. The library itself contains control elements allowing manipulation of zoom and also provides a minimap. The component needs to be provided with properties *nodes*, *edges* which are state variables representing the actual nodes and edges on the canvas, callback functions, which handle changes of nodes, edges, and drag-and-drop functionality, and a custom dictionary containing *nodeTypes* of custom nodes. To clarify, **nodeTypes** is a special keyword property that takes in a dictionary where the key is the text representation of the *type* of the node and the value is the reference to the custom React component object representing the node. Figure 6.1 illustrates how the ReactFlow canvas looks.

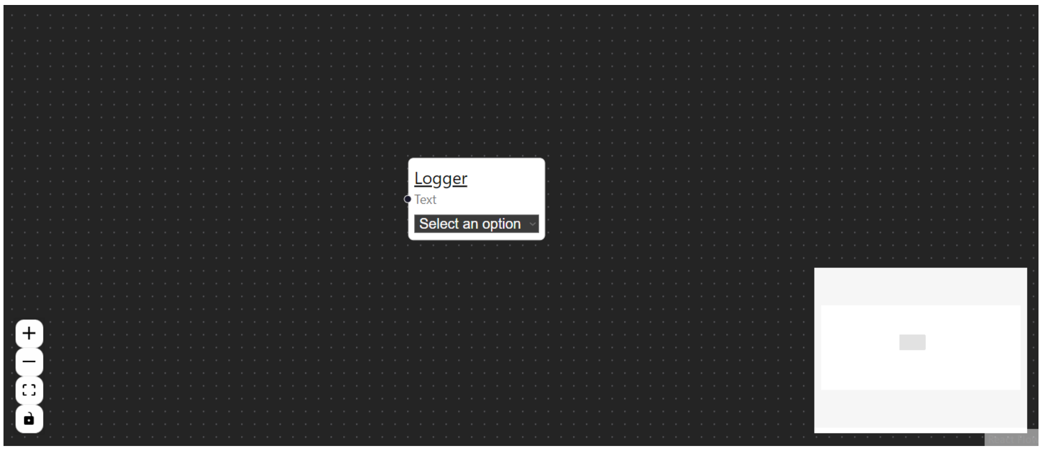


Figure 6.1: Visualisation of ReactFlow canvas with a node.

6.1.2 Sidebar

The sidebar is a custom React component consisting of drawers and node templates. Each drawer is an expandable and retractable section with a descriptive name containing nodes that resemble its description. It consists of *name*, *expanded* flag, and *nodes* which is a list of dictionaries containing the *name* and the *type* of the node. Nodes are HTML div elements that a user can drag-and-drop by simply clicking on the name of the node and dragging the mouse cursor on the canvas. Additionally, it includes a field for the IP Address of the ESP Device in case the user wants to connect to the backend.

Menu

The sidebar also contains a collapsible menu for operations to export flow to the ESP device, load the flow from the file, save the flow to the file (see 6.1.11), clear the logs and display debug information to the console component (see 6.1.12). The Figure 6.2 depicts the sidebar of the toolkit and Figure 6.3 shows the sidebar menu.

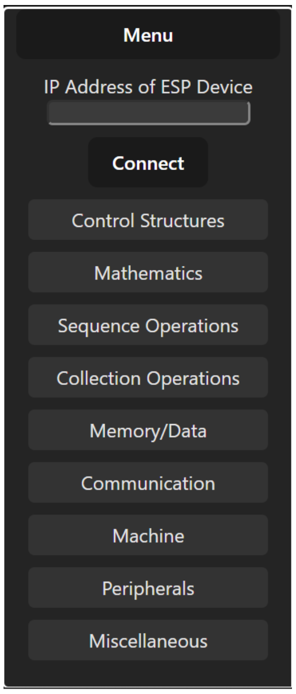


Figure 6.2: Illustration of a Sidebar component.

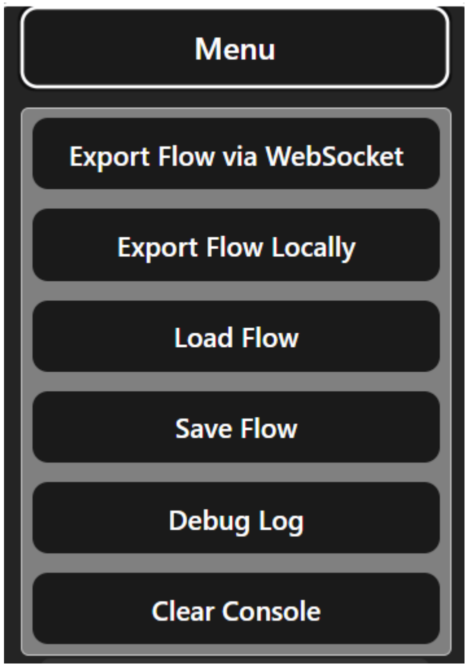


Figure 6.3: Illustration of Sidebar Menu.

6.1.3 Drag-and-drop functionality

When a user clicks on the node it initiates a callback event that sets the global state of the drag event to the type of the node. In case the node is dragged and dropped into the interface of the canvas another callback function is called which gets the type of the node from the global state of the drag event, gets the current position of the mouse, and initializes the node object by creating a new identifier, setting the default properties and getting the node component constructor from the nodeTypes dictionary by using the value from the global state of the drag event (the type of the node) as a key and updates the global application state with the new node.

6.1.4 Custom nodes

Each node has a unique design in terms of HTML composition, but what they all have in common is the properties dictionary which uses the state hook to keep the integrity of the application intact. This dictionary holds the information about the data that the node contains and displays it to the user. This data is set by the user by filling out the input fields or selecting an option from HTML select elements. Each input field has attached *onChangeEvent* handler which updates the state of the node properties. After this change is applied, *useEffect* hook is triggered, which in turn updates the global state (see 6.1.7) of the node in the entire application.

6.1.5 Selection field

With the aim of making custom nodes more modular and less repetitive (following DRY principle - don't repeat yourself), I have created a common React component called **Select Field**. This component is used as a property input field and provides the ability for a component property value to be set up with different options. These options include **input**, which represents the input value passed by the predecessor node output, **variable** selection, which requires a **Define Variable** node connected as a predecessor somewhere in the flow, and **data**, which represents a type and value pair with type being a list of types passed by the parent React component as a prop. This approach allows to easily add a dynamic input to any property, which can acquire the value from different sources, that a user specifies.

Values of Data

Understanding the actual format of the data in the Selection Field can be unclear. Since the input field type for data is text, it's essential to clearly define the format used by each data type. The description for each data type is shown in description 6.1.5.

String: Any text written will be interpreted as string (Text)
Integer: Any integer written will be interpreted as integer (42)
Float: Any floating point number will be interpreted as float (3.14)
Complex: Any text containing number, operation, and number followed by „j“ will be interpreted as a complex number (5+3j)
Boolean: Any Python expression convertible to boolean will be interpreted as boolean (True, False, 1, 0, None...)
List: Any text surrounded by brackets and having the same number of opening and closing brackets while having comma-separated values will be interpreted as a list ([1,2,„Abc“])
Dictionary: Any text surrounded by braces and having the same number of opening and closing braces while containing keys and values separated by a colon ({1:2, „a“:„b“, „l“: [1,2,3]}
Set: Any text surrounded by braces while having comma-separated values will be interpreted as set ({1,2,3})
Tuple: Any text surrounded by parentheses while having comma-separated values will be interpreted as a tuple ((1,2,3,„a“))
None: Any text will be interpreted as *None*
Range: Any text surrounded by parentheses with two integers inside separated by a comma will be interpreted as a range ((1,100))

6.1.6 Update hook

To reduce code duplication I created a custom hook called **useComponent**. Almost every custom node implements this hook as it updates the internal state of the node (properties). It also encapsulates the useEffect hook which takes care of updating the global state. Lastly, this hook returns the state of properties and a callback that handles the change of a property.

6.1.7 Managing the global state

Although the ReactFlow component instance encapsulates the whole workflow with the data of the nodes, it is essential to have some kind of global state management of the entire application in order to have access to the information anywhere, easily and effectively. For this purpose, Zustand Library has been chosen. It has been briefly covered in 3.2.3. The library provides **store object**, which holds information about all the nodes and edges present on the canvas and implements functions for handling the changes related to nodes and edges, like changes of their properties, addition and deletion of nodes and edges, and changes to their position. Additionally, it is used to store information about defined variables and provides utility functions such as depth-first search for finding whether some type of node is present in the previously connected nodes and returns the requested property of nodes, which match the requested type. Lastly, it implements a function that inserts a property to a node indicating that it is a starter node - a node that starts the flow.

6.1.8 Context menu

To interact with nodes a user can drag them on the canvas with a mouse, and fill out the input fields to set their properties, but there needs to be an option how to conveniently do other operations. For this purpose, the context menu has been created. It is activated by right-clicking the node and offers options to toggle the node as a *starter node*, *delete the node*, or *duplicate* it. Setting the node as a starter will mark it with a green border to visually emphasize the property and this will be reflected on the backend as the property of the node is also set. Deleting it simply removes the node from the canvas while synchronizing the global state and duplicating it will create a copy of the node with all the properties a user has defined. This can be handy, because some nodes require a lot of setup steps and there may arise a situation, where the node needs to be reused later in the flow, with the same, or very similar configuration of properties. The context menu is shown in Figure 6.4

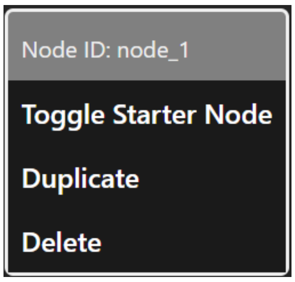


Figure 6.4: Representation of context menu in the visual toolkit.

6.1.9 Removal of the edges

It is essential to allow users to simply remove edges when they want to make changes to the flow, otherwise, the source/target node would have to be deleted and recreated again while making the correct connection. Removing of the edge can be done by simply clicking on either end of the edge and dragging it on the canvas.

6.1.10 Export

After the flow is built, it can be exported by clicking on the export button. Afterward, a callback is called, which accesses the store object containing all the nodes and edges that make connections between them and subsequently calls a serializing function, effectively creating a JSON object that is sent by the javascript fetch method to the ESP32 device. This serialization function is specially designed, to make the most compact representation of the flow, so the file only contains data, that the backend needs. Each node will be sent with ID, properties, and outputs representing the IDs of connected nodes and the source handle name, which represents the type of edge (this is especially important when dealing with nodes, that have two outputs like for loop or if condition, to differentiate the outputs).

6.1.11 Save and Load

It is essential to have some way to save the frontend flow as a file for the purposes of permanently storing the program, reusability, sharing, collaboration, or primitive version control. This is done by keeping a reference to the ReactFlow component instance and serializing it as a Blob object into JSON format, which fundamentally contains all the information about the nodes and edges - their properties, position, etc. Additionally, the current state of the identifier number is saved to have information from which number the application needs to identify the new nodes correctly, when the flow program is loaded and new nodes are added.

In order to load the JSON and recreate the state of the application, the data must be deserialized correctly and loaded into the ReactFlow instance. This is partially being done by the ReactFlow library itself. The global state of the application is recreated as the loaded nodes are instantiated because when any custom node is created, it automatically fills in the data into the global store by its useEffect hook which depends on the properties of the React component.

6.1.12 Console

Receiving feedback from the backend is crucial for effective monitoring of events happening on the backend. In this context, the console serves as an essential tool. It provides notifications to the user regarding the successful export of the flow, confirms the backend's readiness to send feedback messages to the frontend, and displays messages sent from the Logger block.

6.2 Nodes for Visual Programming Flow

There is a variety of nodes present in the application. This section will discuss each available node in more detail in order to get a better understanding of what each individual node actually does and what is possible to create with the current state of the application.

6.2.1 Control structures

The execution of the program is directed by the connections made between nodes, but sometimes there is a need to manage the control flow explicitly by evaluating a certain condition or executing a set of nodes in a loop. For this purpose two nodes were created - If Statement and For Loop. For detailed description refer to C.1.

If statement serves as a branching node. It evaluates a condition and based on whether the result is true or false continues to execute the flow from that point of connection.

For loop iterates over the provided iterable and executes the body with provided iterable elements.

Figure 6.5 illustrates the visual representation of the If statement and For loop nodes in the visual toolkit.

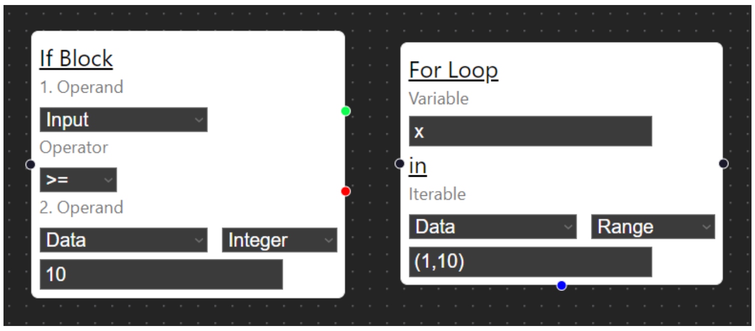


Figure 6.5: Representation of Control Structure nodes.

6.2.2 Mathematics

Mathematical/logical operations are often used in programming to calculate the values of expressions. This subsection covers nodes responsible for doing mathematical operations on non-sequence inputs - Absolute value and Operation. For detailed description refer to C.2.

Absolute value takes a parameter and returns its absolute value.

Operation in essence returns calculated mathematical or logical expression based on used operands and operator.

Figure 6.6 portrays the visual representation of Absolute value and Operation nodes in the visual toolkit.

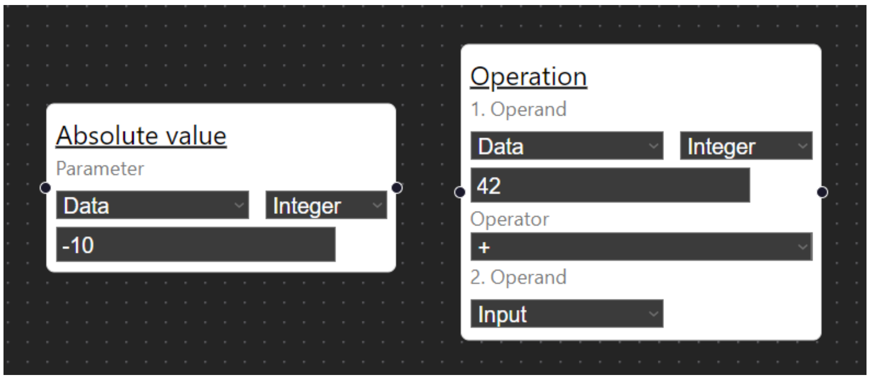


Figure 6.6: Representation of Mathematics nodes.

6.2.3 Sequence operations

Working with sequences often requires operations to manipulate the data efficiently. To aid this process, specific sequence operation nodes have been developed serving a distinct purpose - Length, Sorted, Sum, and Count. For a detailed description refer to C.3.

Length calculates the length of an iterable.

Sorted sorts all elements of an iterable.

Sum adds up all the elements of an iterable.

Count counts all occurrences of an element in an iterable.

In Figure 6.7 are all mentioned sequence operations with some example values they could contain.

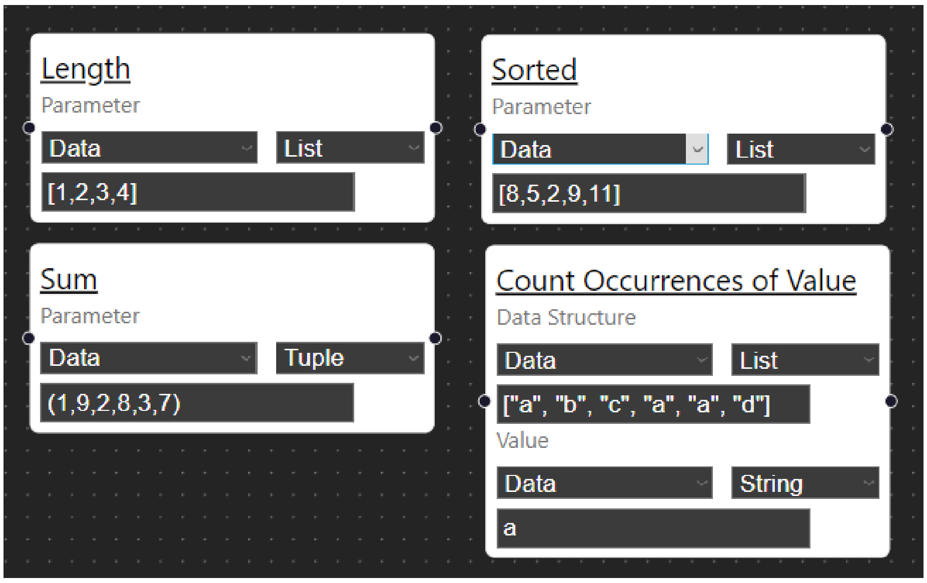


Figure 6.7: Representation of Sequence Operations nodes.

6.2.4 Collection operations

Manipulating and managing collections is a common task in programming, especially with data structures like dictionaries and lists. To make this process effective, specific nodes have been created, each addressing a unique requirement - Set in/Get from/Remove from Dictionary, Get Dictionary Items/Values/Keys, Get/Set/Remove value on index and Append to list. For detailed description refer to C.4.

Set Dictionary Value sets a value for some key in the specified dictionary.

Get Dictionary Value gets a value of some key in the specified dictionary.

Remove by Dictionary Key removes a key-value pair specified by a key in a dictionary.

Get Dictionary Values gets values of specified dictionary and returns them in a list.

Get Dictionary Keys gets the keys of the specified dictionary and returns them in a list.

Get Dictionary Items gets items of specified dictionary and returns them in a list.

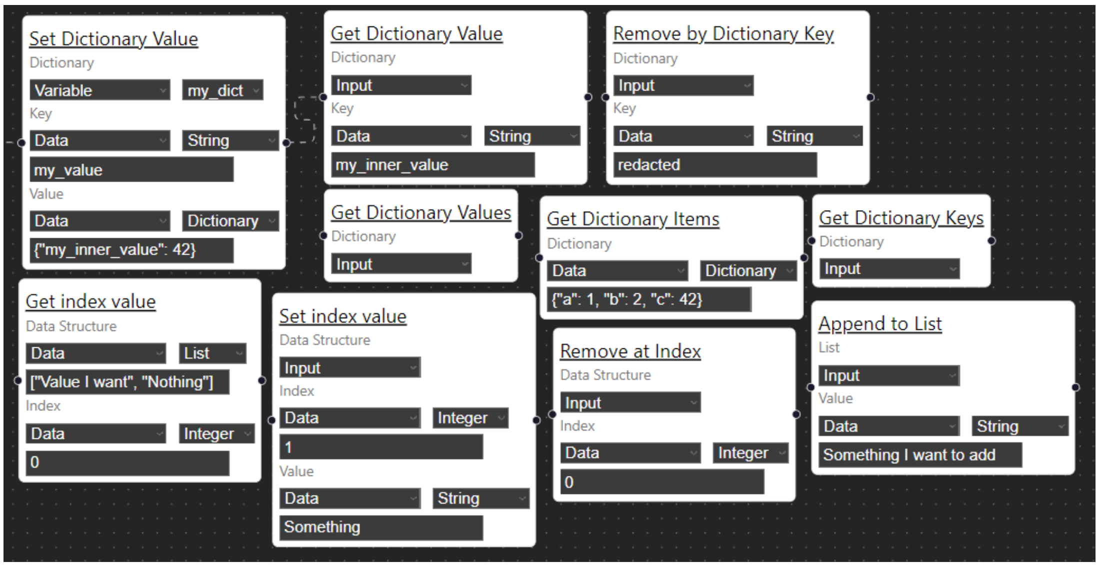
Get Index Value gets a value of a data structure specified by index.

Set Index Value sets a value of a data structure specified by index.

Remove Index Value removes a value of a data structure specified by index.

Append to List Appends a specified value to a list.

Figure 6.8 illustrates collection operation nodes with some example values.


Set Dictionary Value
Dictionary: Variable | my_dict
Key: Data | String | my_value
Value: Data | Dictionary | {"my_inner_value": 42}

Get Dictionary Value
Dictionary: Input
Key: Data | String | my_inner_value

Remove by Dictionary Key
Dictionary: Input
Key: Data | String | redacted

Get Dictionary Values
Dictionary: Input

Get Dictionary Items
Dictionary: Data | Dictionary | {"a": 1, "b": 2, "c": 42}

Get Dictionary Keys
Dictionary: Input

Get index value
Data Structure: Data | List | ["Value I want", "Nothing"]
Index: Data | Integer | 0

Set index value
Data Structure: Input
Index: Data | Integer | 1
Value: Data | String | Something

Remove at Index
Data Structure: Input
Index: Data | Integer | 0

Append to List
List: Input
Value: Data | String | Something I want to add

Figure 6.8: Representation of Collection Operations nodes.

6.2.5 Memory/Data

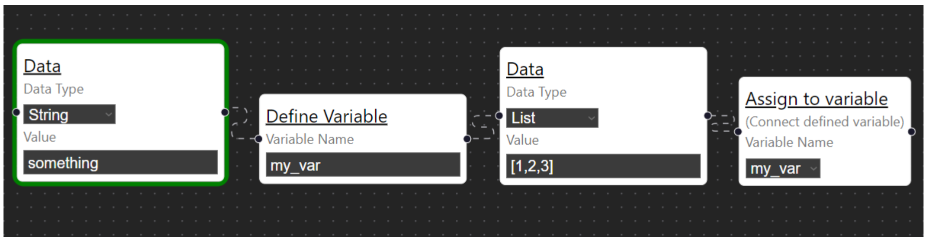
It is essential to have the option to store values in variables with an aim to use them later in the program or define certain values of data types ourselves. For this purpose, I designed nodes - Define Variable, Assign Variable, and Data. For detailed description refer to C.5.

Define variable defines a variable with a specified identifier. This identifier can contain any value that is representable in string format, but the validation functionality will automatically strip the value of any spaces. This node expects input as variables in Python cannot be declared, only defined.

Assign variable assigns an input to a selected variable.

Data node creates a constant value of specified type.

Each defined variable name needs to be unique, as each node needs to know which specific *Define variable* node has been chosen when a variable has been selected in the Selection Field, due to the integrity of data represented on the backend. In this context, the redefinition of variables or assigning values is handled by **Assign To Variable** node. Representation of Data/Memory nodes can be seen in Figure 6.9.


Data
Data Type: String
Value: something

Define Variable
Variable Name: my_var

Data
Data Type: List
Value: [1,2,3]

Assign to variable
(Connect defined variable)
Variable Name: my_var

Figure 6.9: Representation of Memory/Data nodes.

6.2.6 Communication

A substantial part of programming IoT is the communication of the device. This is why it is essential to provide some means of communication with other services - mainly by utilizing HTTP and MQTT protocol. This is being achieved with the use of the following nodes Logger, MQTT Server, MQTT Publish, MQTT Subscribe, HTTP Get, and HTTP Post. For detailed description refer to C.6.

Logger serves as a node, which sends feedback of values in the program to the frontend client console.

MQTT Server serves for configuration of MQTT server parameters. It is being used in connection with MQTT Publish and MQTT Subscribe nodes.

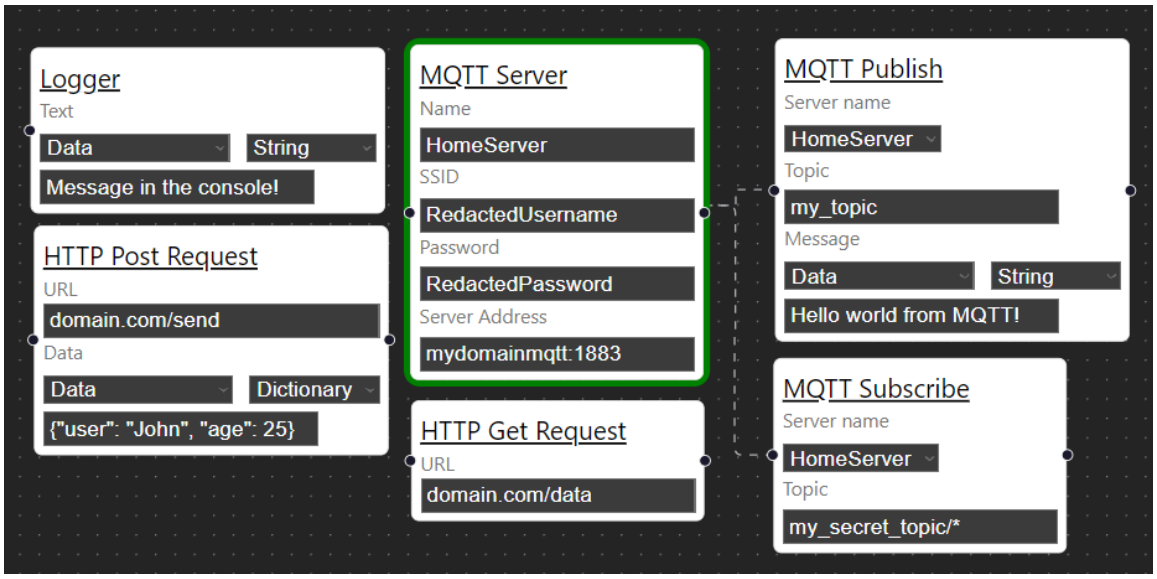
MQTT Publish publishes a message to a specified topic.

MQTT Subscribe subscribes to a specified topic and calls connected target nodes as a callback when a message is received.

HTTP Get Request sends an HTTP GET request to a specified endpoint and outputs the response.

HTTP Post Request sends an HTTP POST request to a specified endpoint and outputs a response (for example to check the status code).

The communication nodes with example configuration are depicted in Figure 6.10.



Logger
Text
Data | String
Message in the console!

HTTP Post Request
URL
domain.com/send
Data
Data | Dictionary
{"user": "John", "age": 25}

MQTT Server
Name
HomeServer
SSID
RedactedUsername
Password
RedactedPassword
Server Address
mydomainmqtt:1883

HTTP Get Request
URL
domain.com/data

MQTT Publish
Server name
HomeServer
Topic
my_topic
Message
Data | String
Hello world from MQTT!

MQTT Subscribe
Server name
HomeServer
Topic
my_secret_topic/*

Figure 6.10: Representation of Communication nodes.

6.2.7 Machine

For the sake of having control over individual components of the microcontroller it is essential to implement nodes to interact with GPIO pins, Analog-digital converter, SPI, I2C or Timer. For detailed description refer to C.7.

Pin represents a configuration of GPIO pin, with mode (Input, Output, Open-Drain) and specification of whether the pin has the pull resistor attached.

Set Pin sets the pin to High or Low.

Analog-digital converter converts the analog input to a digital value based on selected reading type and selected Pin ID.

Write to SPI configures the SPI bus to write and writes to the bus a value based on the configuration.

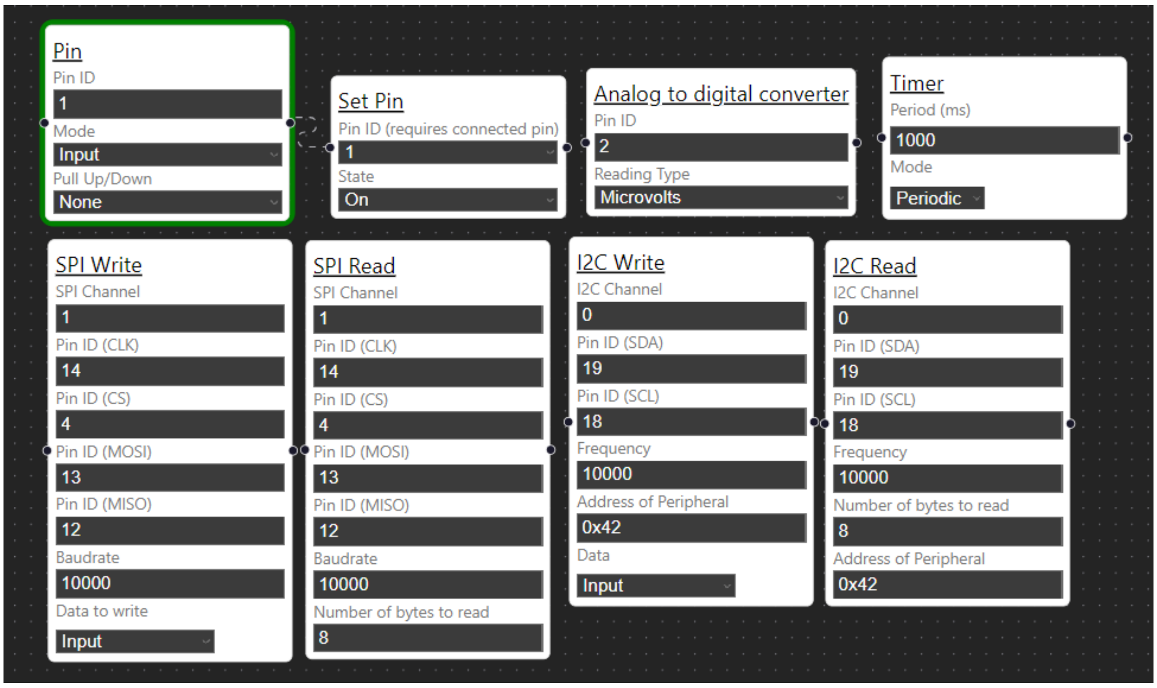
Read from SPI reads data from the SPI bus based on the configuration and outputs it.

Write to I2C writes data to I2C based on the configuration.

Read from I2C reads data from I2C based on configuration.

Timer configures a timer for either one-time or periodical execution after some period.

In figure 6.11 are shown all machine nodes with some example configuration of properties.


Pin
Pin ID: 1
Mode: Input
Pull Up/Down: None

Set Pin
Pin ID (requires connected pin): 1
State: On

Analog to digital converter
Pin ID: 2
Reading Type: Microvolts

Timer
Period (ms): 1000
Mode: Periodic

SPI Write
SPI Channel: 1
Pin ID (CLK): 14
Pin ID (CS): 4
Pin ID (MOSI): 13
Pin ID (MISO): 12
Baudrate: 10000
Data to write: Input

SPI Read
SPI Channel: 1
Pin ID (CLK): 14
Pin ID (CS): 4
Pin ID (MOSI): 13
Pin ID (MISO): 12
Baudrate: 10000
Number of bytes to read: 8

I2C Write
I2C Channel: 0
Pin ID (SDA): 19
Pin ID (SCL): 18
Frequency: 10000
Address of Peripheral: 0x42
Data: Input

I2C Read
I2C Channel: 0
Pin ID (SDA): 19
Pin ID (SCL): 18
Frequency: 10000
Number of bytes to read: 8
Address of Peripheral: 0x42

Figure 6.11: Representation of Machine nodes.

6.2.8 Peripherals

In general, it should not be complicated to interact with peripherals connected to IoT devices. Therefore the implementation supports several drivers for commonly used sensors and peripherals. These include DHT sensors (DHT11, DHT22), Led Matrix 8x8, LED Display, NeoPixel LED, Barometric (Environment) Sensors (BMP180, BMP280, BME280), Ultrasonic distance sensor. For detailed description refer to C.8. It is important to note that open-source third-party libraries were used to integrate these peripherals with the backend. The authors of these libraries are acknowledged in the respective library files.

DHT Sensors reads a temperature and humidity from a selected DHT sensor and outputs it in a dictionary.

LED Matrix displays text on LED matrix.

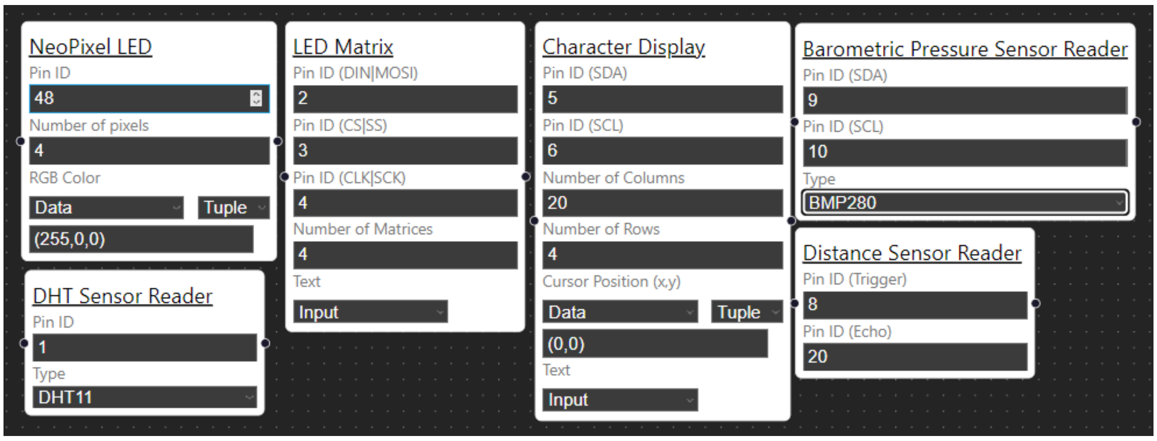
Character display displays text on the backlit character display.

NeoPixel LED lights up an LED with the specified color.

Barometric Sensors measures values (mixture of temperature, humidity, and pressure, depending on selected sensor) from a selected barometric sensor and outputs it in a dictionary.

Ultrasonic Distance Sensor reads a distance from the ultrasound distance sensor. Supported sensors are specified in C.8 and there is no need to select them as both of them work the same way.

Peripheral nodes with example values of properties are illustrated in Figure 6.12.


NeoPixel LED
Pin ID: 48
Number of pixels: 4
RGB Color: Data, Tuple
(255,0,0)

DHT Sensor Reader
Pin ID: 1
Type: DHT11

LED Matrix
Pin ID (DIN|MOSI): 2
Pin ID (CS|SS): 3
Pin ID (CLK|SCK): 4
Number of Matrices: 4
Text: Input

Character Display
Pin ID (SDA): 5
Pin ID (SCL): 6
Number of Columns: 20
Number of Rows: 4
Cursor Position (x,y): Data, Tuple
(0,0)
Text: Input

Barometric Pressure Sensor Reader
Pin ID (SDA): 9
Pin ID (SCL): 10
Type: BMP280

Distance Sensor Reader
Pin ID (Trigger): 8
Pin ID (Echo): 20

Figure 6.12: Representation of Periperals nodes.

6.2.9 Miscellaneous

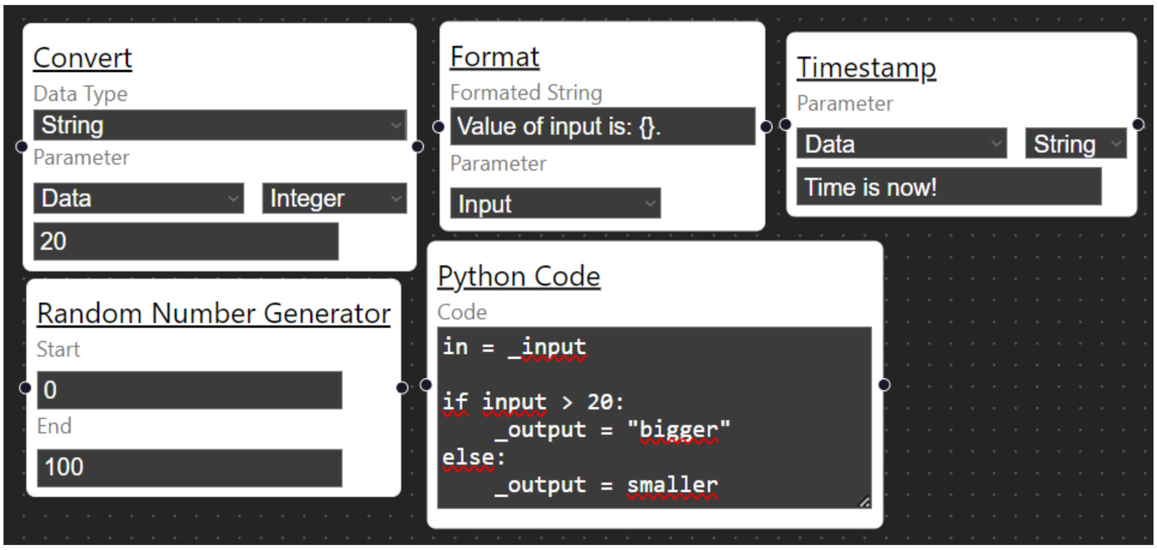
There are various essential tasks and utilities that do not fall into specific categories. This concerns nodes Format String, Convert, Random Number Generator, Python Code, Timestamp. For a detailed description refer to C.9.

Format String formats a value to a string.

Convert casts a parameter to the specified type.

Random Number Generator generates a random number in the specified interval.

Python Code executes a custom Python code written by the user. This node leverages the ability to execute code in a text format. All variable names defined prior to this node, by **Define Variable** (they need to be connected before this block) can be used in the code. Output can be specified by assigning value to **_output** variable in the code.



Convert
Data Type: String
Parameter: Data, Integer
20

Format
Formated String: Value of input is: {}.
Parameter: Input

Timestamp
Parameter: Data, String
Time is now!

Random Number Generator
Start: 0
End: 100

Python Code
Code:

```
in = _input

if input > 20:
    _output = "bigger"
else:
    _output = smaller
```

Figure 6.13: Representation of Miscellaneous nodes.

6.3 Principles Behind Micropython Backend

This section will discuss the setup of the ESP device, backend, communication, receiving, loading, and execution of the flow in more detail.

6.3.1 Setup

To run MicroPython on an ESP device, it needs to be flashed with MicroPython binary with the use of flashing software, for example **esptool**. To run the backend the device needs to contain provided *main.py* - containing configuration loading and HTTP Server setup, *models.py* - containing the code for the interpretation of the flow from JSON, *config.py* - containing configuration for Wi-Fi credentials and DEBUG flag and all the libraries required for sensors, communication, and utilities. These files can be transferred to the ESP device with the use of **Thonny IDE** (see 4.2.3) or **mpremote** (see 4.2.3).

6.3.2 Main

After the ESP device is set up the main program can be run by running the main.py file. To enable an ESP32 device to interact with a frontend application, it first needs to connect to Wi-Fi by loading credentials from the configuration file and calling a function that establishes an internet connection. After that, it creates a Microdot instance to run an HTTP server. This server can then upgrade HTTP connections to WebSockets upon request. In this state, the flow can be received from the frontend via WebSocket. When the device receives the request with the flow, it is saved to the ESP32's internal storage, and the program is restarted to load the new flow. In case the DEBUG flag is set to True, the program will wait, until the user clicks the connect button on frontend in the sidebar to add its connection to the list of WebSocket connections for feedback purposes.

6.3.3 Representing the flow

Two base classes are important for encapsulating the information about the nodes - The **BaseNode class** and **Output class** shown on Listings 6.1 and 6.2. The BaseNode class contains the *id* of the node, its *properties* as a Python dictionary, and *outputs* as a list of Output objects. Output object consists of the *id* of the edge, *type* of the edge, and the *object* which is the target of the connection.

```python
class BaseNode:
    def __init__(self, id, properties, outputs):
        self.id = id
        self.properties = properties
        self.outputs = outputs
```

Listing 6.1: BaseNode class.

```python
class Output:
    def __init__(self, id, type, obj):
        self.id = id
        self.type = type
        self.obj = obj
```

Listing 6.2: Output class.

The specific nodes are each represented by a unique class that will inherit from BaseNode and implement its own unique behavior. In the constructor of the class, the properties will be unpacked for more readable code. The execution logic is handled by the asynchronous **execute** function, which takes arguments containing the possible input, variables and code for the logic itself. It then creates asynchronous coroutines by calling instances of objects contained in the output property with the output value and passes variables in the process. For instance let's presume a node responsible for generating random numbers from interval $< start; end >$. Its class representation is shown at Listing 6.3.

```
class RandomNumberNode(Base):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.start = self.properties["start"]
        self.end = self.properties["end"]

    @deep_copy_params
    async def execute(self, *args, variables):
        random_number = randrange(self.start, self.end)
        for output in self.outputs:
            asyncio.create_task(output.obj.execute(random_number, variables=
                variables))
```
Listing 6.3: Random number generator node.

So in summary, the initialization of attributes is handled by the superclass, and the *execute* method will contain instructions for the execution of the logic of the node. After the completion of the internal logic, all nodes connected to the random number generator node have their own execute methods called with *asyncio.create_task()* effectively creating asynchronous coroutines, which are handled by Asyncio library and their results via event loop.

Loading of the flow has been intentionally skipped so far in order to clarify the functionality of the nodes itself. When loading the flow from the JSON, the class of each node is determined by using the type of the JSON node object as a key to the dictionary with class values. An example of such a dictionary can be seen on listing 6.4.

```
NODES = {
    'timer': _Timer,
    'logger': Logger,
    'random_number': RandomNumberBlock,
    'if': _If,
    ...
}
```
Listing 6.4: Example of dictionary classifying JSON object nodes to MicroPython classes.

Afterward, the object is initialized with the properties stored in the JSON object and added to the dictionary of the flow, where *node id* represents a key and the object itself represents a value while checking the *isStarter* property of each node and adding to the list of starting nodes. Subsequently, the Output instances are created by iterating over the input JSON flow and the outputs of each node, getting the objects by using the *target_id* as key in the dictionary of the flow and setting them as an argument for the *obj* property together with *id* and *type* of the edge. Finally, the last iteration over the list of starting nodes initiates the flow by creating coroutines with the objects *execute* method as the input. This starts the execution of the flow by creating chained coroutines, which are handled by asyncio library.

6.3.4 Variables

Variables are represented by a simple Python Dictionary. The key represents the name of the variable and the value is the actual value the variable contains. Upon start, the flow is initialized with an empty dictionary and this dictionary is being passed from node to node. Variables are added when a **Define Variable** is executed and modified when

Assign Variable, or some other nodes interacting with the variables are being executed. When they are passed as an argument, they are being deep-copied, so each node has its own snapshot of the variables (approach explained in 7.1.3).

6.3.5 Feedback messages

When a client on the frontend connects to an ESP device, the connection is added to a list of active connections. If the WebSocket connection becomes inactive, it is removed from this list, which helps in tracking currently active connections to the microcontroller. This active connection is then used to send feedback messages back to the frontend, and these messages eventually appear in the console component. The backend informs the user on the frontend if the flow has been received successfully and if the user has been added to the list of active connections. It is also important to have a feature which helps user to debug whether the flow is being executed as he expects. This functionality is a responsibility of the **Logger** node (see C.6).

6.3.6 Type mapping

In the case of handling data in the Selection Field which consists of type and value, it is important to define a mapping of types in order to cast the value correctly. This is being achieved by using the type defined on frontend as key, where the value will be the MicroPython function, which will be able to convert the value to the desired data type. For more complex data structures the **eval** function evaluates the expression to be for example a list, a dictionary, or a set. An example of such a dictionary can be seen on Listing 6.5.

```
type_mapping = {
    "String": str,
    "Integer": int,
    "Float": float,
    "Boolean: bool,
    "List": eval,
    "None": lambda x: None,
    ...
}
```

Listing 6.5: Example of dictionary responsible for type mapping.

6.3.7 Operations

Evaluation of mathematical and logical expressions is handled by defining a dictionary, where the key is an operation in string representation and the value is a lambda function, which takes two parameters and evaluates the result according to the provided operation. An example can be seen on Listing 6.6.

```
operations = {
    "+": lambda x, y: x + y,
    "-": lambda x, y: x - y,
    "*": lambda x, y: x * y,
    "in": lambda x, y: x in y,
    "not in": lambda x, y: x not in y, ...
}
```

Listing 6.6: Example of dictionary responsible operations.

6.3.8 Convert field

The Selection Field (see 6.1.5) offers a handful of possible values for a property, so it needs to be handled carefully. Each option in the selection field must be well-defined on the backend to reflect the choices provided on the frontend. This is the purpose of **convert_field** function. It takes a field that represents either „input“, dictionary {„variable“: „<name>“} or {„type“: „<type>“, „value“: „<value>“}, _input which is the input of the node and variables dictionary and returns the value of the property based on the field. The function can be seen on Listing 6.7.

```python
def convert_field(field, _input, variables):
    if isinstance(field, str):
        if _input and field == "input":
            return _input[0]
    elif "variable" in field:
        return variables[field["variable"]]
    elif "type" in field:
        return type_mapping.get(field["type"])(field["value"])
    raise UndefinedFieldError
```

Listing 6.7: Function for returning Selection Field value.

6.3.9 Asynchronous execution

It is important to mention that the node output can be connected to multiple inputs of other nodes. This results in asynchronous execution of both „sub flows“, which improves the overall performance on the backend, as the code is interpreted in MicroPython. This behavior is being achieved with the use of Asyncio library, which executes each node as a task. An example of flow utilizing asynchronous execution is shown in Figure 6.14

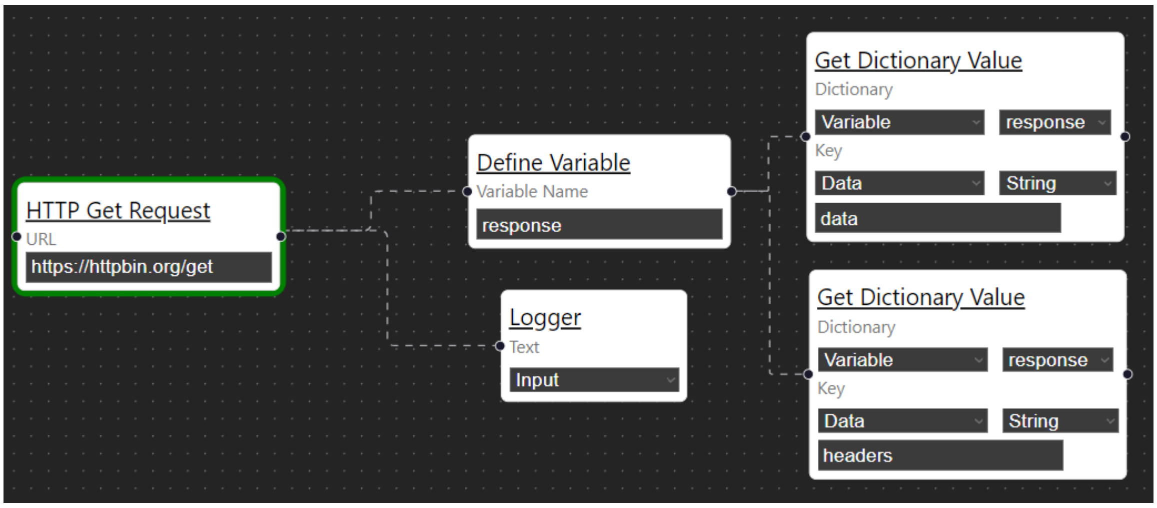


Figure 6.14: An example of asynchronous flow.

Chapter 7

Testing and Evaluation

This chapter will discuss how were the frontend visual toolkit and MicroPython backend tested, what approaches were taken in the development, what obstacles appeared and how were they solved.

7.1 Problems and Challenges

During the development, I encountered several challenges and problems that I needed to deal with. These obstacles ranged from technical difficulties and design flaws to unexpected bugs and errors that required thorough troubleshooting.

7.1.1 Synchronization of global state

The first challenge I faced was related to getting the most recent state of the application as React framework works in cumbersome ways. To explain the problem, I will have to tackle a little bit, of how React components work. Each component consists of a state and its HTML which is shown in the browser. When certain events happen by user interaction, the state changes and is re-rendered (e.g. changing the property of the node). However, this is being done asynchronously, so while the code changing the state has been executed and the UI has been re-rendered in the browser, the state variable may not contain the most recent information. This is why I had to use the global state management library Zustand as mentioned in 6.1.7. When the information of the state finishes updating I use **useEffect hook** to update the global state of the whole flow with the most recent data.

7.1.2 Saving and loading of the flow

It is vital to save all the information about the flow into the file, as it has to be reconstructed in the same way when loaded. Saving has been an easy part. The trouble I encountered was with the loading of the flow. When the nodes were being reconstructed from the JSON file, the data had not been passed to the properties, so the information had not reached the actual nodes. This has been solved by passing data as a prop, setting the properties state on initialization to the provided data, and setting the default value for each property of the node from the state. When testing, I found out about another problem. When a node is created, an ID counter for the node is incremented to uniquely identify each node and keep track of their number. This information is not contained in the ReactFlow component instance so it had to be added to the file explicitly. Previously, when the application was restarted and the flow was loaded, everything was working correctly, until the user tried to add a new node to the flow. When this happened, the node from the saved file with the same ID was overwritten and deleted.

7.1.3 Variables handling in backend

Initially, the variables were implemented as a global dictionary with the key being the variable name and the value being the variable value. This has shown to work in practice unless there was a **Timer** node involved with periodic execution. The main problem with this approach was, that the Timer node executes the rest of the flow periodically while there may be some coroutines finishing their execution, which concludes in a race condition and other nodes not obtaining the required value from the variable. The first attempt to solve this problem was to initialize each flow with an empty variables dictionary, which is being modified in **Define variable** and **Assign Variable** nodes (there are also other nodes, that modify variables). These variables are then passed down the chain of coroutines, essentially creating a situation, where each node has it's own snapshot of variables. This would essentially work, if the variables were not passed by reference. The last piece of the puzzle is to create a @*deep_copy_params* decorator which handles the deep copying of parameters and used it, to decorate *execute* function of each node. This has successfully achieved the desired result and the integrity of variables has been accomplished.

7.1.4 Memory management

As stated in 4.1.1 and 4.1.2 the ESP microcontrollers have a limited amount of memory. As it turns out, all the libraries required to support every node implemented on the frontend take up a lot of memory space when loaded. When I was testing on an ESP32 device, this problem escalated to such a point, that there was no memory left for Wi-Fi initialization and the device could not function as intended. One approach that worked really well was precompiling the libraries. MicroPython provides a tool called **mpy-cross** which is capable of converting regular Python module into a pre-compiled version. Further optimization can be achieved by freezing the pre-compiled bytecode into the firmware image as a part of the main firmware compilation process, meaning that the bytecode will be executed from ROM [8]. With models of ESP microcontrollers such as ESP32-C3 with external PSRAM there is no need to do such optimizations as 8 megabytes is a decent memory size for everything to run perfectly without any pre-compilation.

Another thing to keep in mind is that memory usage can become an issue during *runtime* in MicroPython. To prevent memory overflow, MicroPython provides a garbage collector library called **gc**. In my backend implementation, I am calling garbage collector periodically every 10 seconds, to ensure that objects no longer needed are removed from memory, freeing up space for new objects.

7.1.5 Execution speed

Although MicroPython combines the bytecode execution with the interpretation of the code, the fact that some code is being interpreted implies a slower execution speed compared to compiled languages. To improve the speed as much as possible, most of the code is being executed asynchronously with the help of Asyncio library. This means, that some tasks can be executed concurrently, resulting in faster performance.

7.2 Experiments and Testing

This section will describe what experiments were performed and what was their outcome. It further explains what methods and metrics were chosen and what was the performance of the toolkit.

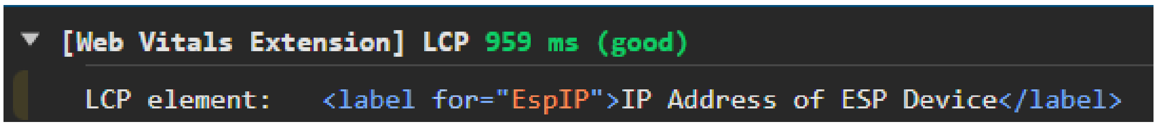
7.2.1 Visual programming toolkit

In order to properly test the visual programming toolkit, I decided to simulate the worst possible scenario. In the browser development tools, I chose the option of a slow mobile device, which should result in emulating a very slow CPU. The visual programming toolkit handled this really well and I have not experienced any performance issues regarding the functionality or user interface (even animations were smooth). Furthermore, I placed focus on the WebSocket communication with the ESP device. This turned out to be functioning the same as with no simulated restrictions to the performance of the browser. For further frontend testing, I decided to use the core web vitals metrics, which are the metrics defined by Google. Values for these metrics have been acquired by web vitals browser extension.

Largest Contentful Paint (LCP)

LCP metric defines the loading performance of the website. It tries to measure when a website is visible to the user and shows the largest content element visible within the user's viewport [6].

The largest contentful paint in the visual programming toolkit has proved to be on a very good level. As shown in Figure 7.1, the website loads within one second, which is considered a good result. Additionally, the element that takes the longest to load is a label, suggesting that the website is free of performance-impeding elements such as images, which typically require more time to display.



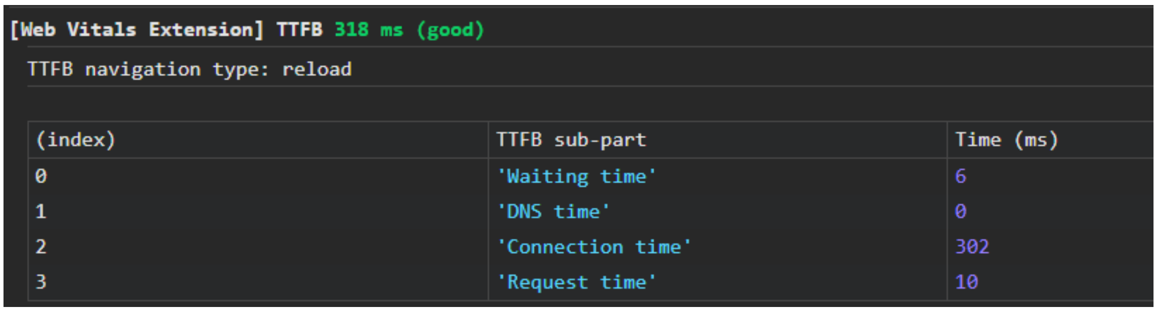
```
▼ [Web Vitals Extension] LCP 959 ms (good)
    LCP element:   <label for="EspIP">IP Address of ESP Device</label>
```

Figure 7.1: Depiction of LCP in visual programming toolkit.

Time to First Byte

Time to First Byte (TTFB) is the time that it takes for a user's browser to receive the first byte of page content [6]. It is a measurement used as an indication of the responsiveness of a webserver or other network resource.

The visual programming toolkit demonstrated effective performance in terms of time to first byte metric. As illustrated in figure 7.2, the most significant portion of time was spent establishing a connection to the website. Meanwhile, the durations for waiting time, DNS lookup, and request time were comparatively negligible.



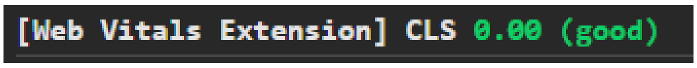
```
[Web Vitals Extension] TTFB 318 ms (good)
  TTFB navigation type: reload

  (index)                    TTFB sub-part              Time (ms)
  0                          'Waiting time'             6
  1                          'DNS time'                 0
  2                          'Connection time'          302
  3                          'Request time'             10
```

Figure 7.2: Depiction of TTFB in visual programming toolkit.

Cumulative Layout Shift

CLS measures the sum total of all individual layout shift scores for every unexpected layout shift that occurs during the entire lifespan of the page. A layout shift occurs any time a visible element changes its position from one frame to the next [6].

As shown in Figure 7.3, there is absolutely no cumulative layout shift present in the visual programming toolkit. This means that while the website is being loaded, no elements are being moved or shifted. Please note, that dynamic actions such as expanding of a drawer in the sidebar or moving the node on the canvas are not considered as the cumulative layout shift, as these are supposed to be functioning by changing the layout on user actions, not when the page is loading.



```
[Web Vitals Extension] CLS 0.00 (good)
```

Figure 7.3: Depiction of CLS in visual programming toolkit.

7.2.2 Experiments

This subsection will cover experiments. They will utilize the visual programming toolkit to create a representation deserializable and interpretable by MicroPython. The used models of microcontrollers will be ESP32, ESP32-S3 and ESP32-C3.

First experiment

In the first experiment, I chose to create a simple flow utilizing MQTT communication with the use of an LED Matrix display. The program sets up the MQTT server configuration to a HiveMQ testing MQTT broker and subscribes to a topic _test_subscribe. If string „HELL“ is received from the MQTT, it displays a string „Yeah“ on the LED Matrix, otherwise, the program writes „NAH“ to the console. The flow is illustrated in Figure 7.4 and sent messages via the MQTT Broker are depicted in Figure 7.7. As can bee seen from results of the flow depicted in Figures 7.5 and 7.6 the program has executed correctly. When the message received from the broker was anything other than the expected string, the „NAH“ message was logged to the frontend console and when the string was „HELL“, the Matrix displayed the string „Yeah“.

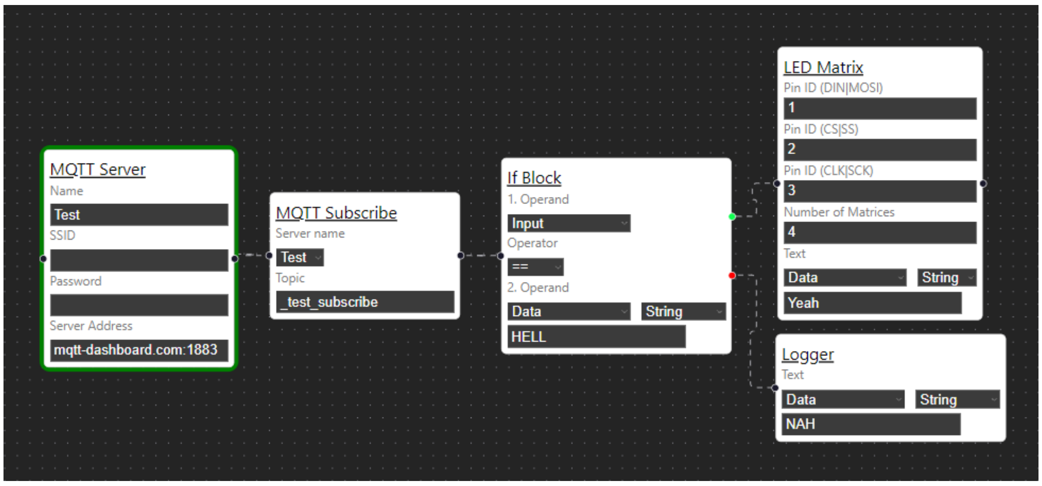
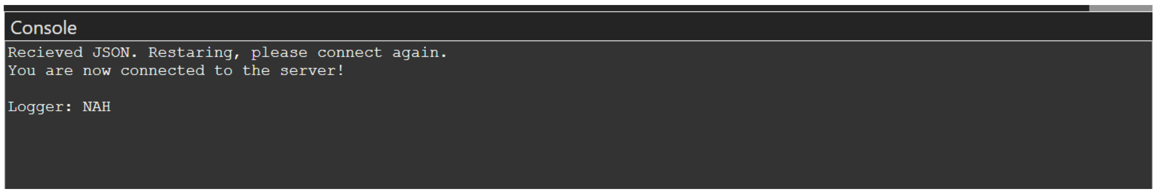


Figure 7.4: Depiction of the first experiment flow.



Console

```
Recieved JSON. Restaring, please connect again.
You are now connected to the server!

Logger: NAH
```

Figure 7.5: Depiction of the first experiment console feedback.

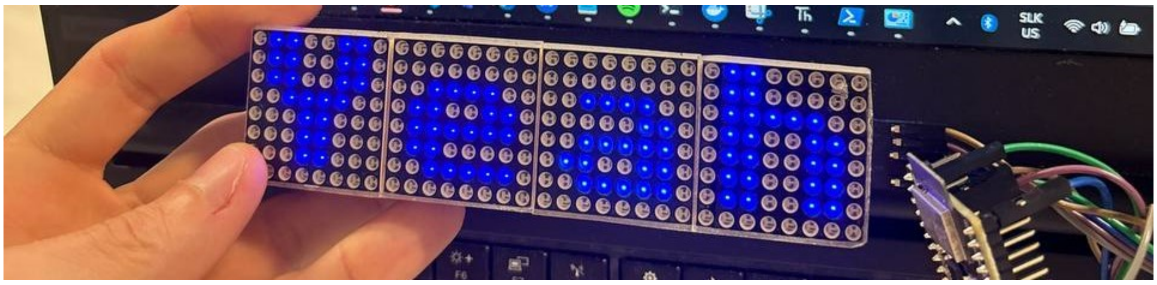
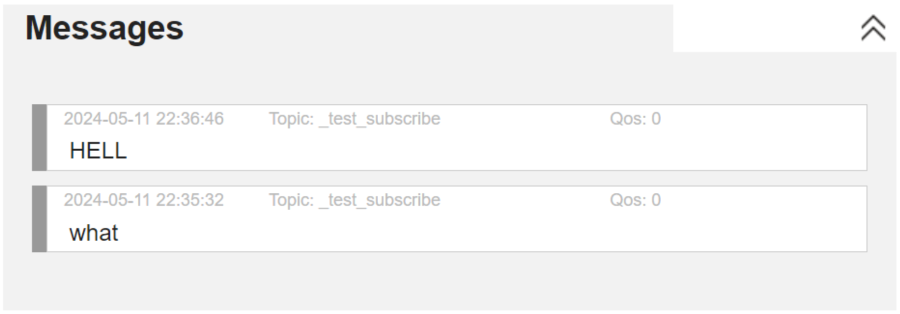


Figure 7.6: Illustration of the LED Matrix showing expected output.



Messages

2024-05-11 22:36:46 Topic: _test_subscribe Qos: 0
HELL

2024-05-11 22:35:32 Topic: _test_subscribe Qos: 0
what

Figure 7.7: Picture of messages published to the MQTT broker.

Second experiment

For a second experiment, I decided to choose a more complex example but stick to the „Pythonic“ program as I wanted to test the Python constructions. The aim was to find out, how the visual toolkit handles a considerable amount of different nodes. The goal of the program was to generate random numbers and append them to the list until the length of the list was larger than 20. I could have used a simple for loop for this, but I decided to test the Length node and use for loop later. After the list containing 21 numbers has been generated, the flow continues to sort the list and in one continuation it prints them to the console via the Logger node. In the other continuation, it iterates over the list and sends to the console component each number with the statement, whether it is odd or even. The flow is shown in Figure 7.8. As can be seen from the console output shown in Figure 7.9, the program has indeed generated 21 random numbers based on the condition that the length of the list must be greater than 20, appended them to a list, sorted the list, iterated over the list and displayed the sorted list in the console, while iterating over the list and assessing whether each number is even or odd.

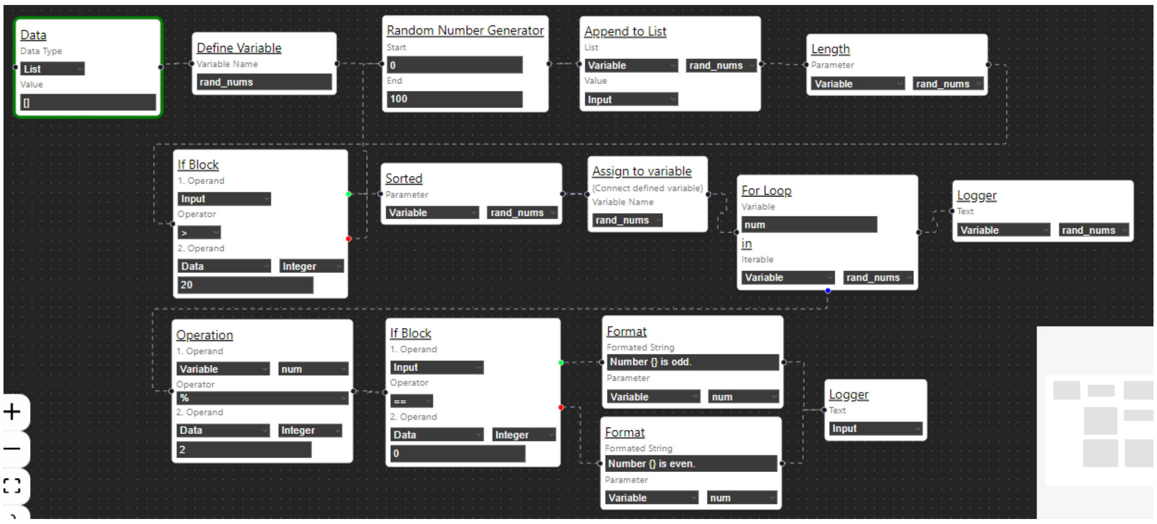
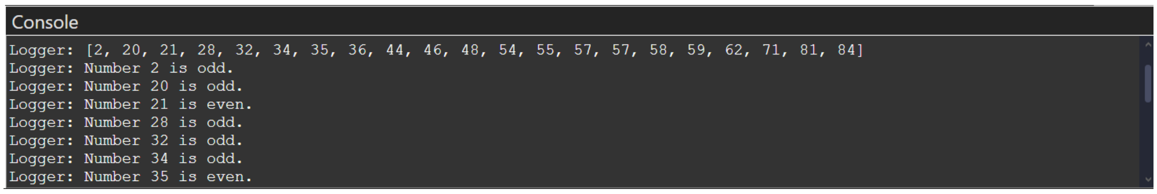


Figure 7.8: Flow program of the second experiment.


Console

```
Logger: [2, 20, 21, 28, 32, 34, 35, 36, 44, 46, 48, 54, 55, 57, 57, 58, 59, 62, 71, 81, 84]
Logger: Number 2 is odd.
Logger: Number 20 is odd.
Logger: Number 21 is even.
Logger: Number 28 is odd.
Logger: Number 32 is odd.
Logger: Number 34 is odd.
Logger: Number 35 is even.
```

Figure 7.9: Console output of the second experiment.

Third experiment

The third experiment tests work with collection operations, DHT sensor, and NeoPixel LED. The program periodically triggers DHT sensor to read temperature and humidity. Afterwards, the measured values are stored in a variable and the items of the dictionary are displayed in the frontend console. Additionally, if the temperature value is bigger than 20, the LED will light up with red color signaling that the temperature is warm. In case the measured temperature is below 20, the LED lights up blue signaling that the temperature is low. On the Figure 7.10 is shown the frontend representation of the program. For the purpose of this testing, the sensor has been measuring temperature in a room with standard temperature and later has been cooled down with ice cubes wrapped in a plastic bag. This can be seen in Figures 7.11 and 7.12. The results of the measurements in the console can be seen in Figure 7.13 for the temperature measurement above 20 degrees Celcius and in Figure 7.14 for measurement of temperature below 20 degrees Celcius.

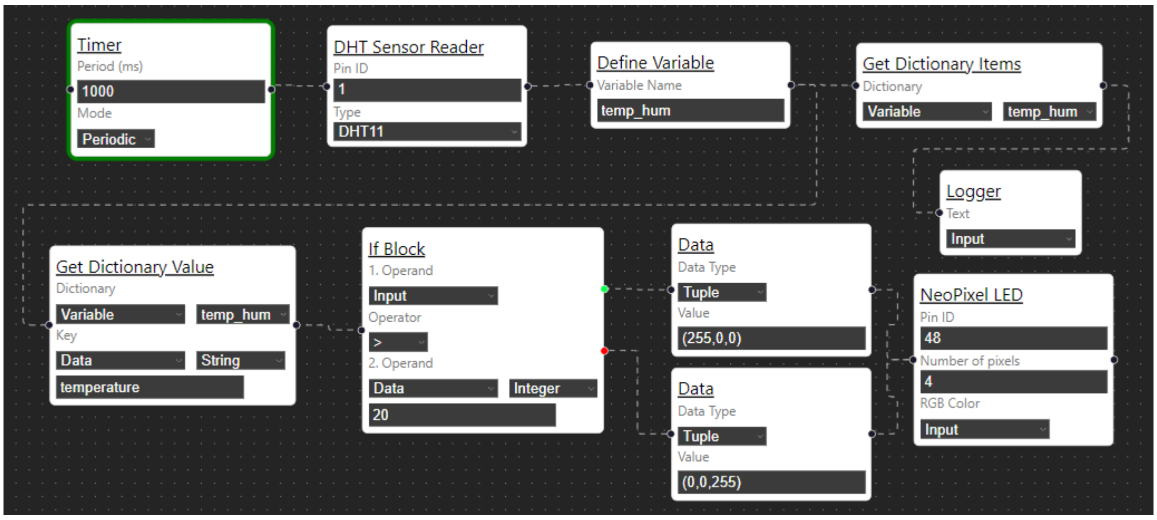


Figure 7.10: Flow program of the third experiment.

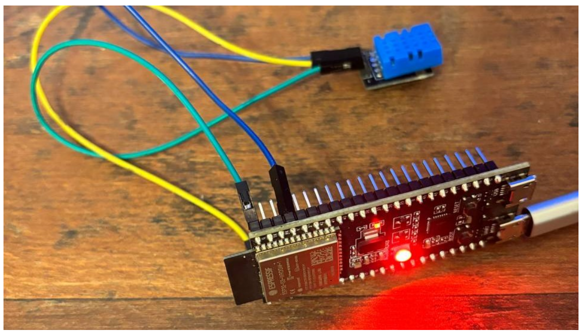


Figure 7.11: Image of an ESP device with the temperature above 20 degrees Celsius.

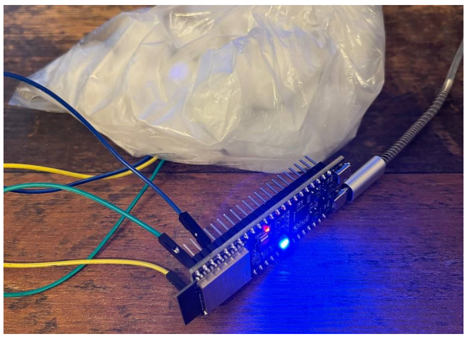
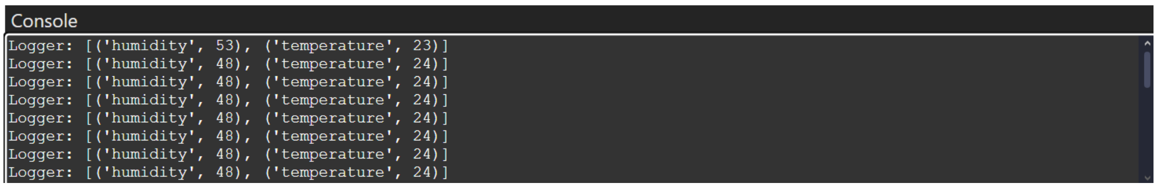


Figure 7.12: Image of an ESP device with the temperature below 20 degrees Celsius.



Console
```
Logger: [('humidity', 53), ('temperature', 23)]
Logger: [('humidity', 48), ('temperature', 24)]
Logger: [('humidity', 48), ('temperature', 24)]
Logger: [('humidity', 48), ('temperature', 24)]
Logger: [('humidity', 48), ('temperature', 24)]
Logger: [('humidity', 48), ('temperature', 24)]
Logger: [('humidity', 48), ('temperature', 24)]
Logger: [('humidity', 48), ('temperature', 24)]
```

Figure 7.13: Console displaying measurements when the temperature was above 20 degrees Celsius.

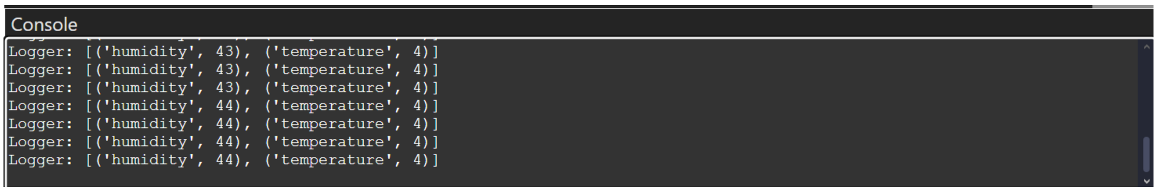
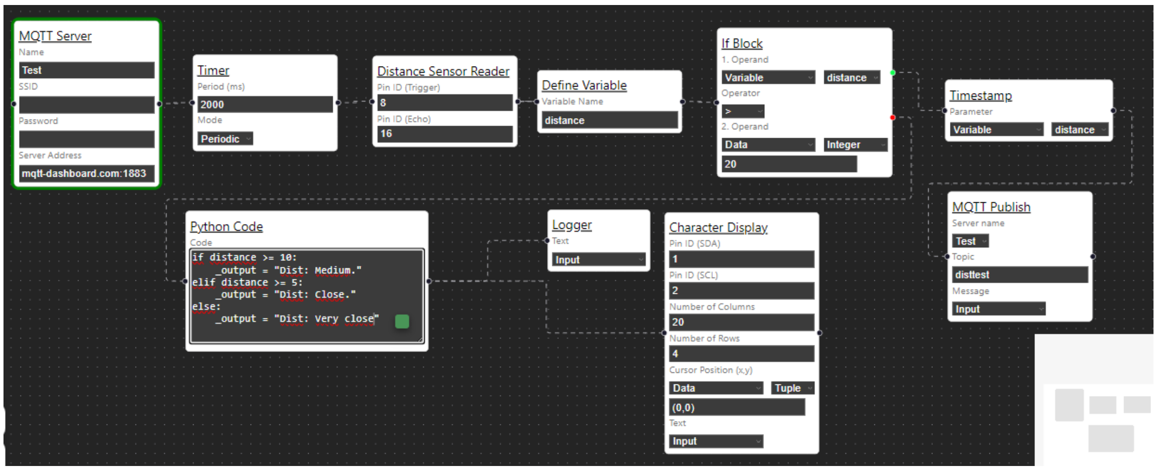

Console
```
Logger: [('humidity', 43), ('temperature', 4)]
Logger: [('humidity', 43), ('temperature', 4)]
Logger: [('humidity', 43), ('temperature', 4)]
Logger: [('humidity', 44), ('temperature', 4)]
Logger: [('humidity', 44), ('temperature', 4)]
Logger: [('humidity', 44), ('temperature', 4)]
Logger: [('humidity', 44), ('temperature', 4)]
```

Figure 7.14: Console displaying measurements when the temperature was below 20 degrees Celsius.

Fourth experiment

The final experiment was to create a program, that will use ultrasonic distance sensor. If the measured distance is greater than 20, it will put a timestamp on it and publish it via MQTT to topic disttest. In case the distance is lower than 20, it will utilize a custom Python node, which will specify in the **__output** variable, how close the object is to the sensor. The custom Python node contains Python's *elif* construction which is not created as a standalone node in the current implementation of the visual programming toolkit, so this is the convenient use case. Afterwards, the text with a specified distance is displayed on the backlit character display. The frontend program flow can be seen in Figure 7.15. The MQTT Output is illustrated in Figure 7.16. Output of the distance on the display has been converted to grayscale as the character display has not been very bright and it was hard to see the text from the photos. The outputs of the display can be seen in Figures 7.17, 7.18 and 7.19. The console output of the program is depicted in Figure 7.20.


MQTT Server
Name: Test
SSID:
Password:
Server Address: mqtt-dashboard.com:1883

Timer
Period (ms): 2000
Mode: Periodic

Distance Sensor Reader
Pin ID (Trigger): 8
Pin ID (Echo): 16

Define Variable
Variable Name: distance

If Block
1. Operand: Variable distance
Operator: >
2. Operand: Data Integer
20

Timestamp
Parameter: Variable distance

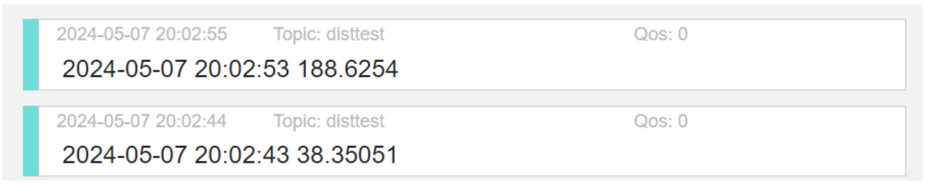
MQTT Publish
Server name: Test
Topic: disttest
Message: Input

Python Code
Code:
```
if distance >= 10:
    _output = "Dist: Medium."
elif distance >= 5:
    _output = "Dist: Close."
else:
    _output = "Dist: Very close"
```

Logger
Text: Input

Character Display
Pin ID (SDA): 1
Pin ID (SCL): 2
Number of Columns: 20
Number of Rows: 4
Cursor Position (x,y): Data Tuple
(0,0)
Text: Input

Figure 7.15: Illustration of fourth experiment program.


2024-05-07 20:02:55 Topic: disttest Qos: 0
2024-05-07 20:02:53 188.6254

2024-05-07 20:02:44 Topic: disttest Qos: 0
2024-05-07 20:02:43 38.35051

Figure 7.16: Delivered messages via MQTT Publish.

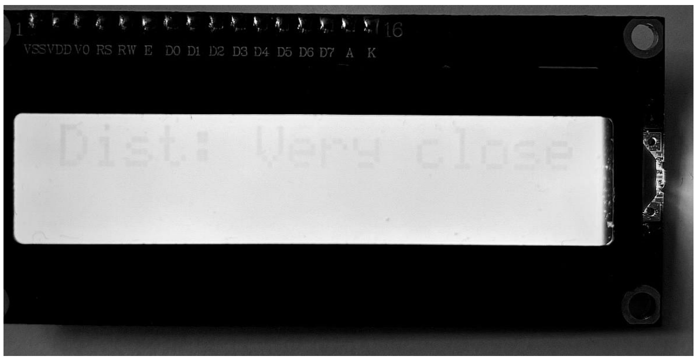


Figure 7.17: Very close distance shown on character display.

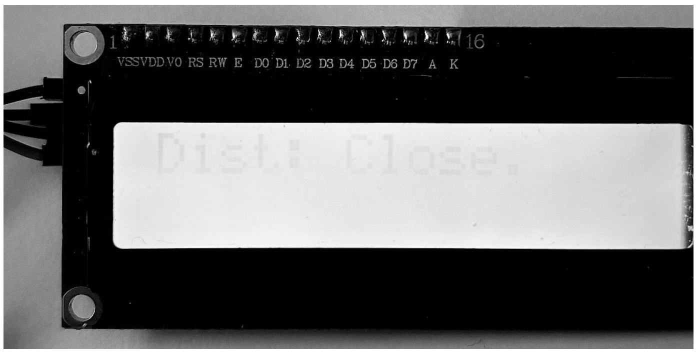


Figure 7.18: Close distance shown on character display.

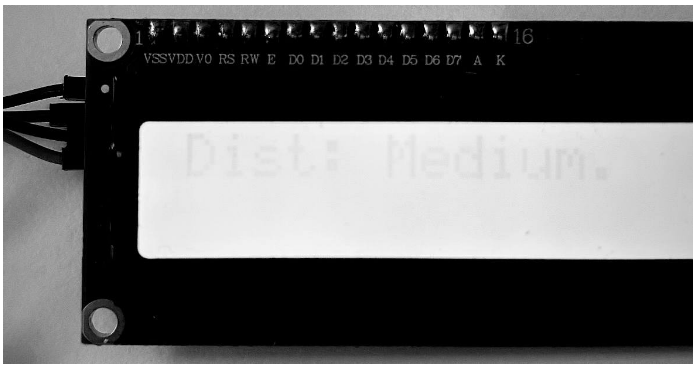
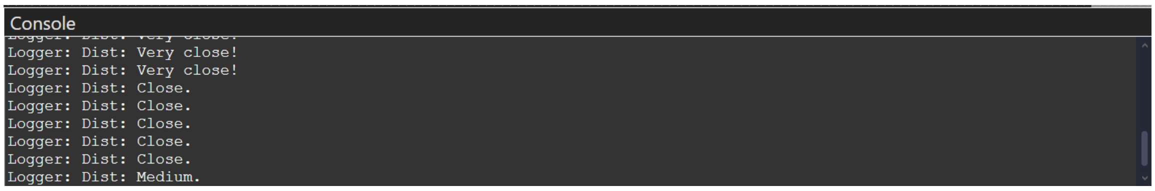


Figure 7.19: Medium distance shown on character display.



```
Logger: Dist: Very close!
Logger: Dist: Very close!
Logger: Dist: Close.
Logger: Dist: Close.
Logger: Dist: Close.
Logger: Dist: Close.
Logger: Dist: Close.
Logger: Dist: Medium.
```

Figure 7.20: Distance output in the frontend console.

7.2.3 Assessment

As shown by the results of experiments, the visual programming toolkit has been thoroughly tested. While using the toolkit, the browser has been set to emulate a slow CPU and weak internet connection. The metrics - Largest Contentful Paint, Time to First Byte and Cumulative Layout Shift - all had excellent results. The created programs tested both „Pythonic" programming, while placing focus on the IoT part of the testing as the code is being interpreted on ESP devices, which commonly have some peripherals attached. The flows were deserialized and interpreted on ESP32, ESP32-S3 and ESP32-C3. Each program performed very well on all three devices without any performance issues. All nodes in the visual programming toolkit were properly tested under various scenarios, the provided experiments only illustrate, what were the approaches and how approximately complex were the flows being tested. In these experiments I proved, that this toolkit works without a problem with the common MicroPython programming regarding work with common data structures, control flow, loops and variables. Additionally, communication protocols work as expected together with supported sensors and peripherals like DHT, Matrix Display, Character Display, NeoPixel LEDs, Ultrasonic Distance sensor and handful of more. The programming toolkit is quite robust as it does not allow for user to work with variables which are not connected to the flow and do not start with a starter node, making exceptions for dead code impossible.

Chapter 8

Conclusion

To summarize, the goal of this thesis is to implement a visual toolkit, which will be able to create programs interpretable by MicroPython for ESP32 platform. I studied theory and approaches of various visual programming languages, their cognitive effects in comparison to text-based programming language with the aim to create the best visual representation of a toolkit for programming ESP32 platform. Furthermore I gathered information about numerous techniques and technologies used to implement web-based applications and studied means of programming of ESP32 devices with focus placed specifically on MicroPython. I designed and implemented a visual toolkit capable of creating a flow-based programs, which run asynchronously on the backend. The nodes contained in the toolkit contain control structures, mathematical/logical expressions, sequence operations, collection operations, support of common data and variables, communication for debug logging, HTTP Requests and MQTT Subscribe/Publish, support of GPIO pins, ADC, I2C, SPI, Timer, a handful of sensors and miscellaneous nodes, such as timestamp, random number generator, or custom Python code. The toolkit is able to export the flow locally.

Above the requirements of the thesis, the visual toolkit is able to export the flow to the ESP device via WebSocket, it is able to save the client-side representation and load it from the file. I also implemented a console component which serves as a display element for feedback to the client in the browser about what is happening on the backend device. In addition, the current implementation contains drivers for a handful of common sensors and peripherals, which are not natively supported by MicroPython.

8.1 Possible Extensions

There are many ways the visual programming toolkit could be improved and extended. First of all the I see a room for UI improvement in terms of how the nodes are shown and possibly sidebar UI improvement, although I think the current implementation is user friendly and sufficient. The second thing that is a thing to consider is adding support for more sensors - mainly more models of sensors. Thirdly I think it should be possible to make use of more communication protocols like BLE and ESP-NOW. Lastly, the toolkit could have a functionality to encapsulate the part of already configured flow into a single node which could be reusable.

Bibliography

[1] Allington, M. *Introducing svelte 5 - what's new* [online]. Oct 2023 [cit. 2024-01-21]. Available at: https://marcallington.com/blog/introducing-svelte5-whats-new.

[2] Annamaa, A. Introducing Thonny, a Python IDE for learning programming. In: *Proceedings of the 15th Koli Calling Conference on Computing Education Research*. ACM, November 2015. Koli Calling '15. DOI: 10.1145/2828959.2828969. Available at: http://dx.doi.org/10.1145/2828959.2828969.

[3] Annamaa, A. *Learn to code with Thonny - A Python IDE for Beginners* [online]. Fedora magazine, 19. Feb 2018 [cit. 2024-01-25]. Available at: https://fedoramagazine.org/learn-code-thonny-python-ide-beginners/.

[4] Baer, E. *What react is and why it matters* [online]. O'Reilly Media, Inc. [cit. 2024-01-21]. Available at: https://www.oreilly.com/library/view/what-react-is/9781491996744/ch01.html.

[5] Bottoni, P., Costabile, M. F., Levialdi, S. and Mussio, P. Specification of Visual Languages as Means for Interaction. In: *Visual Language Theory*. Springer New York, 1998, p. 353–375. DOI: 10.1007/978-1-4612-1676-6_13. ISBN 9781461216766. Available at: http://dx.doi.org/10.1007/978-1-4612-1676-6_13.

[6] Chandak, V. *Web Vitals Metrics – What you should know* [online]. 4. jun 2020 [cit. 2024-04-29]. Available at: https://www.virendrachandak.com/techtalk/web-vitals-metrics/.

[7] Damien, G. P. and Sokolovsky, P. *1. getting started with MicroPython on the ESP32* [online]. George Robotics Limited, 2014. 2024-05-07 [cit. 2024-01-21]. Available at: https://docs.micropython.org/en/latest/esp32/tutorial/intro.html.

[8] Damien, G. P. and Sokolovsky, P. Optimizations. *MicroPython Internals* [online]. 2014. 2024-07-05 [cit. 2024-01-24]. Available at: https://docs.micropython.org/en/latest/develop/optimizations.html.

[9] Damien, G. P. and Sokolovsky, P. MicroPython remote control: mpremote. *MicroPython language and implementation* [online]. 2014. 2024-07-05 [cit. 2024-05-04]. Available at: https://docs.micropython.org/en/latest/reference/mpremote.html.

[10] Damien, G. P. and Sokolovsky, P. *MicroPython libraries* [online]. George Robotics Limited, 2014. 2024-05-07 [cit. 2024-01-21]. Available at: https://docs.micropython.org/en/latest/library/index.html.

[11] Damien, G. P. and Sokolovsky, P. asyncio-aynchronous I/O scheduler. *MicroPython libraries* [online]. 2014. 2024-07-05 [cit. 2024-01-24]. Available at: https://docs.micropython.org/en/latest/library/asyncio.html.

[12] Damien, G. P. and Sokolovsky, P. machine — functions related to the hardware. *MicroPython libraries* [online]. 2014. 2024-07-05 [cit. 2024-01-24]. Available at: https://docs.micropython.org/en/latest/library/machine.html.

[13] Damien, G. P. and Sokolovsky, P. class Pin – control I/O pins. *MicroPython libraries* [online]. 2014. 2024-07-05 [cit. 2024-01-24]. Available at: https://docs.micropython.org/en/latest/library/machine.Pin.html.

[14] Damien, G. P. and Sokolovsky, P. class Signal – control and sense external I/O devices. *MicroPython libraries* [online]. 2014. 2024-07-05 [cit. 2024-01-24]. Available at: https://docs.micropython.org/en/latest/library/machine.Signal.html.

[15] Damien, G. P. and Sokolovsky, P. class ADC – analog to digital conversion. *MicroPython libraries* [online]. 2014. 2024-07-05 [cit. 2024-01-24]. Available at: https://docs.micropython.org/en/latest/library/machine.ADC.html.

[16] Damien, G. P. and Sokolovsky, P. class PWM – pulse width modulation. *MicroPython libraries* [online]. 2014. 2024-07-05 [cit. 2024-01-24]. Available at: https://docs.micropython.org/en/latest/library/machine.PWM.html.

[17] Damien, G. P. and Sokolovsky, P. class UART – duplex serial communication bus. *MicroPython libraries* [online]. 2014. 2024-07-05 [cit. 2024-01-24]. Available at: https://docs.micropython.org/en/latest/library/machine.UART.html.

[18] Damien, G. P. and Sokolovsky, P. class SPI – a Serial Peripheral Interface bus protocol (controller side). *MicroPython libraries* [online]. 2014. 2024-07-05 [cit. 2024-01-24]. Available at: https://docs.micropython.org/en/latest/library/machine.SPI.html.

[19] Damien, G. P. and Sokolovsky, P. class I2C – a two-wire serial protocol. *MicroPython libraries* [online]. 2014. 2024-07-05 [cit. 2024-01-24]. Available at: https://docs.micropython.org/en/latest/library/machine.I2C.html.

[20] Damien, G. P. and Sokolovsky, P. class Timer – control hardware timers. *MicroPython libraries* [online]. 2014. 2024-07-05 [cit. 2024-01-24]. Available at: https://docs.micropython.org/en/latest/library/machine.Timer.html.

[21] Damien, G. P. and Sokolovsky, P. class SDCard – secure digital memory card. *MicroPython libraries* [online]. 2014. 2024-07-05 [cit. 2024-01-24]. Available at: https://docs.micropython.org/en/latest/library/machine.SDCard.html.

[22] Espressif, S. *Espressif SOC serial Bootloader Utility* [online]. 2016 [cit. 2024-01-16]. Available at: https://docs.espressif.com/projects/esptool/en/latest/esp32/.

[23] Ferreira, F., Borges, H. S. and Valente, M. T. On the (un-)adoption of JavaScript front-end frameworks. *Software: Practice and Experience*. Wiley. october 2021, vol. 52, no. 4, p. 947–966. DOI: 10.1002/spe.3044. ISSN 1097-024X. Available at: http://dx.doi.org/10.1002/spe.3044.

[24] George, D. *MicroPython* [online]. George Robotics Limited, 2014 [cit. 2024-01-18]. Available at: https://micropython.org/.

[25] Gevv. *Flowcode examples* [online]. 15. october 2016. 2019-08-04 [cit. 2024-01-13]. Available at: https://320volt.com/en/flowcode-examples/.

[26] Hope, C. *What is block-based programming?* [online]. 16. Nov 2019 [cit. 2024-01-10]. Available at: https://www.computerhope.com/jargon/b/block-based-programming.htm.

[27] ICDSoft. *One-page vs. Multi-page website. which one is better?* [online]. 28. Mar 2022 [cit. 2024-01-20]. Available at: https://www.icdsoft.com/blog/one-page-vs-multi-page-website-which-one-is-better/.

[28] Itemis. Model and simulate decision logic using state machines and flow charts. *Stateflow* [online]. [cit. 2024-01-13]. Available at: https://www.mathworks.com/products/stateflow.html.

[29] itemis. *What is a state machine?* [online]. [cit. 2024-01-13]. Available at: https://www.itemis.com/en/products/itemis-create/documentation/user-guide/overview_what_are_state_machines.

[30] Limited, M. M. *Hardware Supported by Flowcode* [online]. [cit. 2024-01-13]. Available at: https://www.flowcode.co.uk/hardware/.

[31] Lucidchart. What is a flowchart? *What is a Flowchart* [online]. [cit. 2024-01-13]. Available at: https://www.lucidchart.com/pages/what-is-a-flowchart-tutorial.

[32] Meta Platforms, I. JSX Definition. *JSX* [online]. 4. Aug 2022 [cit. 2024-01-21]. Available at: https://facebook.github.io/jsx/.

[33] Miguel, G. *Microdot* [online]. 2021 [cit. 2024-01-26]. Available at: https://microdot.readthedocs.io/en/latest/.

[34] MozDevNet. *Getting started with Svelte - learn web development: MDN* [online]. 2023-10-05 [cit. 2024-01-21]. Available at: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Svelte_getting_started.

[35] mpyblockly. What is MPY Blockly? *Mpyblockly* [online]. 2022 [cit. 2024-01-26]. Available at: https://mpyblockly.github.io/.

[36] NAVARRO PRIETO, R. and CAÑAS, J. J. Are visual programming languages better? The role of imagery in program comprehension. *International Journal of Human-Computer Studies* [online]. Elsevier BV. june 2001, vol. 54, no. 6, p. 799–829, [cit. 2024-01-11]. DOI: 10.1006/ijhc.2000.0465. ISSN 1071-5819. Available at: https://www.sciencedirect.com/science/article/pii/S1071581900904658.

[37] Poimandres. Introduction. *Zustand* [online]. [cit. 2024-01-26]. Available at: https://docs.pmnd.rs/zustand/getting-started/introduction.

[38] rs esp. Overview of Development Approaches. *The Rust on ESP Book* [online]. [cit. 2024-01-23]. Available at: https://esp-rs.github.io/book/overview/index.html.

[39] RS esp. Using the Standard Library (std). *The Rust on ESP Book* [online]. [cit. 2024-01-23]. Available at: https://esp-rs.github.io/book/overview/using-the-standard-library.html.

[40] RS esp. Using the Core Library (no_std). *The Rust on ESP Book* [online]. [cit. 2024-01-23]. Available at: https://esp-rs.github.io/book/overview/using-the-core-library.html.

[41] RS esp. *Rust on ESP Community* [online]. [cit. 2024-01-23]. Available at: https://github.com/esp-rs.

[42] RS esp. *ESP-rs/espflash: Serial flasher utility for ESPRESSIF socs and modules based on esptool.py* [online]. 1. sep 2020. 2024-01-09 [cit. 2024-01-23]. Available at: https://github.com/esp-rs/espflash.

[43] SOLIDOWORKS. *Flowcode* [online]. Jan 2024 [cit. 2024-01-13]. Available at: https://www.solidworks.com/partner-product/flowcode.

[44] SOLOMON, B. *Async IO in python: A complete walkthrough* [online]. Real Python, Jan 2019 [cit. 2024-01-24]. Available at: https://realpython.com/async-io-python/.

[45] SOURCE, M. O. *Built-in React Hooks* [online]. [cit. 2024-01-21]. Available at: https://react.dev/reference/react/hooks.

[46] S.R.O., E. Vývojová deska ESP32 2,4GHz Dual-Mode Wi-Fi + Bluetooth modul antény. *IOT Vývojové platformy* [online]. [cit. 2024-01-24]. Available at: https://dratek.cz/arduino/1581-esp-32s-esp32-esp8266-development-board-2.4ghz-dual-mode-wifi-bluetooth-antenna-module.html.

[47] SYSTEMS, E. ESP32-S3-DevKitC-1 v1.1. *ESP-IDF Programming Guide* [online]. [cit. 2024-01-24]. Available at: https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/hw-reference/esp32s3/user-guide-devkitc-1.html.

[48] SYSTEMS, E. *Welcome to Espressif IoT Development Framework!* [online]. [cit. 2024-01-24]. Available at: https://idf.espressif.com/.

[49] SYSTEMS, E. Get Started. *ESP-IDF Programming Guide* [online]. 2016 [cit. 2024-01-24]. Available at: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html.

[50] SYSTEMS, E. *ESP32-S3 series Datasheet* [online]. 1.8th ed. 2023 [cit. 2024-01-17]. Available at: https://www.espressif.com/sites/default/files/documentation/esp32-s3_datasheet_en.pdf.

[51] SYSTEMS, E. *ESP32 series Datasheet* [online]. 4.5th ed. 2024, 2024 [cit. 2024-01-17]. Available at: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[52] SYSTEMS, E. *A feature-rich MCU with integrated Wi-Fi and Bluetooth connectivity for a wide-range of applications* [online]. 2024 [cit. 2024-01-15]. Available at: https://www.espressif.com/en/products/socs/esp32.

[53] Systems, E. *Your Complete AIoT Solution Provider* [online]. 2024 [cit. 2024-01-15]. Available at: https://www.espressif.com/.

[54] Systems, S. Construct and Run a Stateflow Chart. *Stateflow charts* [online]. [cit. 2024-01-13]. Available at: https://www.mathworks.com/help/stateflow/gs/stateflow-charts.html.

[55] Systems, S. *Stateflow Integration* [online]. [cit. 2024-01-13]. Available at: https://sparxsystems.com/enterprise_architect_user_guide/16.0/model_simulation/stateflow_integration.html.

[56] Team, Q. E. *Innovating with Ease: How Visual Programming Transforms Software Development?* [online]. Oct 2023 [cit. 2024-01-10]. Available at: https://quixy.com/blog/visual-programming-in-software-development/.

[57] W3Schools. *React Components* [online]. [cit. 2024-01-21]. Available at: https://www.w3schools.com/react/react_components.asp.

[58] W3Schools. *React State* [online]. [cit. 2024-01-21]. Available at: https://www.w3schools.com/react/react_state.asp.

[59] xyflow. Wire Your Ideas with React Flow. *React Flow* [online]. [cit. 2024-01-26]. Available at: https://reactflow.dev/.

[60] xyflow. Quickstart. *Svelte Flow* [online]. 2023 [cit. 2024-01-26]. Available at: https://svelteflow.dev/learn.

Appendix A

Contents of the included storage media

```
VUT_FIT_Diploma_Code
└── frontend
    ├── index.html
    ├── .eslintrc.cjs
    ├── package.json
    ├── vite.config.js
    ├── yarn.lock
    ├── public
    │   └── vite.svg
    └── src
        ├── App.jsx
        ├── App.css
        ├── main.jsx
        ├── store.js
        ├── index.css
        ├── assets
        └── components
            ├── static
            │   └── common.css
            ├── shared
            │   └── SelectField.jsx
            ├── common
            │   ├── Console
            │   │   ├── Console.jsx
            │   │   └── Console.css
            │   ├── SidebarComponent
            │   │   ├── Sidebar.jsx
            │   │   └── Sidebar.css
            │   └── ContextMenu
            │       ├── ContextMenu.jsx
            │       └── ContextMenu.css
            └── blocks (Custom Nodes)
```

```
│
├── AssignVariable
│   └── AssignVariable.jsx
├── SPI
│   ├── SPIWrite.jsx
│   └── SPIRead.jsx
├── ForLoop
│   └── ForLoop.jsx
├── LedMatrix
│   └── LedMatrix.jsx
├── BuiltIn
│   ├── Append
│   │   └── Append.jsx
│   ├── GetDictKeys
│   │   └── GetDictKeys.jsx
│   ├── Format
│   │   └── Format.jsx
│   ├── Sum
│   │   └── Sum.jsx
│   ├── Sorted
│   │   └── Sorted.jsx
│   ├── GetDictItems
│   │   └── GetDictItems.jsx
│   ├── SetInDict
│   │   └── SetInDict.jsx
│   ├── Abs
│   │   └── Abs.jsx
│   ├── GetIndex
│   │   └── GetIndex.jsx
│   ├── SetIndex
│   │   └── SetIndex.jsx
│   ├── DeleteFromDict
│   │   └── DeleteFromDict.jsx
│   ├── Convert
│   │   └── Convert.jsx
│   ├── GetDictValues
│   │   └── GetDictValues.jsx
│   ├── GetFromDict
│   │   └── GetFromDict.jsx
│   ├── Len
│   │   └── Len.jsx
│   ├── RemoveIndexValue
│   │   └── RemoveIndexValue.jsx
│   └── Count
│       └── Count.jsx
├── Timestamp
│   └── Timestamp.jsx
├── Timer
│   └── Timer.jsx
│   │
│   │   │
│   │   ├── LiquidLed
│   │   │   └── LiquidLed.jsx
│   │   ├── Data
│   │   │   └── Data.jsx
│   │   ├── Operation
│   │   │   └── Operation.jsx
│   │   ├── DefineVariable
│   │   │   └── DefineVariable.jsx
│   │   ├── HYSRF05
│   │   │   └── HYSRF05.jsx
│   │   ├── Python
│   │   │   └── Python.jsx
│   │   ├── DHT
│   │   │   └── DHT.jsx
│   │   ├── ADC
│   │   │   └── ADC.jsx
│   │   ├── I2C
│   │   │   ├── I2CWrite.jsx
│   │   │   └── I2CRead.jsx
│   │   ├── Pin
│   │   │   ├── Pin.jsx
│   │   │   └── SetPin.jsx
│   │   ├── If
│   │   │   └── If.jsx
│   │   ├── RandomNumber
│   │   │   └── RandomNumber.jsx
│   │   ├── NeoPixel
│   │   │   └── NeoPixel.jsx
│   │   ├── BME
│   │   │   └── BME.jsx
│   │   ├── MQTT
│   │   │   ├── MQTTSubscribe.jsx
│   │   │   ├── MQTTServer.jsx
│   │   │   └── MQTTPublish.jsx
│   │   ├── Logger
│   │   │   └── Logger.jsx
│   │   └── HTTP
│   │       ├── HTTPGet.jsx
│   │       └── HTTPPost.jsx
│   └── hooks
│       └── useComponent.jsx
└── mpy_libs
    ├── hysrf05.mpy
    ├── models.mpy
    ├── umqttsimple.mpy
    ├── BME280.mpy
    ├── protocol.mpy
    └── websocket.mpy
```

```
│   │   │
│   │   ├── client.mpy
│   │   ├── i2c_api.mpy
│   │   ├── i2c_lcd.mpy
│   │   ├── max7219.mpy
│   │   ├── microdot_new.mpy
│   │   ├── bmp280.mpy
│   │   ├── bmp180.mpy
│   │   ├── boot.mpy
│   │   └── utils.mpy
│   ├── src
│   │   ├── bmp280.py
│   │   ├── umqttsimple.py
│   │   ├── main.py
│   │   ├── hysrf05.py
│   │   ├── max7219.py
│   │   ├── websocket.py
│   │   ├── protocol.py
│   │   ├── models.py
│   │   ├── microdot_new.py
│   │   ├── BME280.py
│   │   ├── bmp180.py
│   │   ├── boot.py
│   │   ├── config.py
│   │   ├── i2c_api.py
│   │   ├── i2c_lcd.py
│   │   ├── client.py
│   │   └── utils.py
│   └── setup_microcontroller.sh
└── VUT_FIT_Diploma_Text
    ├── Diploma_Thesis.pdf
    └── Diploma_Thesis_Text_Source.zip
```

Appendix B

Manual

B.1 Prerequisities

The following software versions are required:

- Python $\geq$ 3.8
- Node $\geq$ v18.10.0
- Yarn $\geq$ 1.22.19

B.2 Installation

B.2.1 Frontend

The first step is to navigate to the frontend directory and install the dependencies via

```
yarn install
```

After the dependencies are installed the frontend can be run via

```
yarn dev
```

This will run the local server on localhost:5173.

B.2.2 Backend

The first step is to flash MicroPython on the ESP device. For this purpose **esptool.py** can be used. It can be installed via `pip install esptool`. The next step is to download the MicroPython binary for the ESP device. The binaries are available at https://micropython.org/download/. Download the latest version for the ESP device. First, use the esptool.py to erase flash on the ESP device via

```
esptool.py --chip <chip> --port /dev/ttyUSB0 erase_flash
```

and then flash the downloaded binary onto the ESP device via

```
esptool.py --chip <chip> --port /dev/ttyUSB0 write_flash -z <starting address>
<binary name>,
```

where the chip is the ESP chip being used, the starting address is either 0x1000 for ESP32 or 0x0000 for other ESP chips and the binary name is the name of the MicroPython binary file.

Now that the ESP device has been flashed with MicroPython, the next step is to put the MicroPython files on the ESP file system. For this, either Thonny IDE or mpremote CLI tool can be used (the latter is recommended).

To use **mpremote**, first install it via *pip install –user mpremote.*

Subsequently, it is required to set the config in src/config.py. The SSID and password of the Wi-Fi connection are required. Additionally, the DEBUG flag can be set to False, but this is not recommended, as it will turn off the ability to display feedback in the frontend console, as the ESP device won't wait for the connection and start executing the flow right away. The Wi-Fi should be running at 2.4 GHz. When the configuration is set up, the script *setup_microcontroller.sh* can be executed in the root directory. This script will copy MicroPython libraries compiled to the bytecode required to run the backend together with the config to the ESP device file system. Finally the program can be run via `python -m mpremote run ./src/main.py`.

The user will be greeted with the „Connecting...“ string and when the ESP device connects to the Wi-Fi, the IP Address of the ESP device will be displayed. This address is necessary to be used in order to connect to the ESP device from the frontend.

B.3 Usage

In order to use the visual programming toolkit, the user needs to open it in the browser. The next step is to copy the IP Address of the ESP device printed when running the main.py into the frontend field found on the sidebar. Afterward, the user can create a flow by clicking on the drawers to open them and drag-and-dropping the nodes onto the canvas. At the start of making a flow, a user should always pick a starting node by right-clicking the node on the canvas and clicking „Toggle starter“. When creating the flow, connect the nodes before filling out their properties, so the information from the previous nodes is transferred (in the case of defined variables for example). If the nodes are connected by mistake, the connection can be removed by hovering over either the start of the connection and dragging it on the empty canvas. When a user finishes a flow, he can click the menu and export it on the ESP device via WebSocket. Confirmation about receiving the JSON will be displayed in the console and the ESP device will be restarted. Afterward, the user can close the menu and connect to the ESP device, as in DEBUG mode, it will wait for a WebSocket connection before trying to execute the flow on the frontend.

Appendix C

Detailed Description of Nodes

In Appendix A I will describe nodes in more detail - their description, properties and outputs. I will often refer to the value of some parameter being Selection Field, please see section 6.1.5.

C.1 Control Structures

Name: If Statement
Description: Directs the flow of the program based on a condition.
Properties:

1. **Operand** - Selection Field with Constant types: String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, *None*

   Operation - set of operations (<, >, <=, >=, ==, !=, in, not in)

2. **Operand** - Selection Field with Constant types: String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, *None*

Outputs:

Data: Nothing
Control: True (marked green), False (marked red)

Name: For Loop
Description: Iterates over provided iterable and executes body with values of that iterable being available.
Properties:

Variable - Variable name representing individual values of iterable
Iterable - Selection Field with Constant types: String, List, Dictionary, Set, Tuple, Range, Enumerate

Outputs:

Data: Nothing
Control: Body (marked blue), Continue

C.2 Mathematics

Name: Absolute Value
Description: Calculates absolute value.
Properties:

Parameter - Selection Field with Constant types: Integer, Float, Complex

Outputs:

Data: Integer, Float, Complex (Returns magnitude of the complex number)
Control: Next node

Name: Operation
Description: Evaluates mathematical or logical expression based on operation.
Properties:

1. Operand - Selection Field with Constant types: String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, $None$
Operation - Set of operations $(+, -, *, /, //, \%, **, \&, |, \hat{}, <<, >>, ==, ! =, >, <, >=, <=$, in, not in, is, is not)
2. Operand - Selection Field with Constant types: String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, $None$

Outputs:

Data: String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, $None$
Control: Next node

C.3 Sequence Operations

Name: Length
Description: Evaluate the length of the provided parameter.
Properties:

Parameter - Selection Field with Constant types: String, List, Dictionary, Set, Tuple

Outputs:

Data: Integer
Control: Next node

Name: Sorted
Description: Sort the provided parameter into the list.
Properties:

- **Parameter** - Selection Field with Constant types: String, List, Dictionary, Set, Tuple

Outputs:

- **Data:** List
- **Control:** Next node

Note: If the parameter is dictionary, the sorted keys will be returned.

Name: Sum
Description: Calculate the sum of the provided input.
Properties:

- **Parameter** - Selection Field with Constant types: List, Dictionary, Set, Tuple

Outputs:

- **Data:** Integer, Float
- **Control:** Next node

Note: If the parameter is a dictionary, the sum of keys will be returned.

Name: Count
Description: Counts occurrences of a value in a data structure.
Properties:

- **Data Structure** - Selection Field with Constant types: String, List, Dictionary, Set, Tuple
- **Value** - Selection Field with Constant types: String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, *None*

Outputs:

- **Data:** Integer
- **Control:** Next node

C.4 Collection Operations

Name: Set Dictionary Value
Description: Sets the value of a key in a dictionary.
Properties:

- **Dictionary** - Selection Field with Constant type Dictionary
- **Key** - Selection Field with Constant types String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, *None*
- **Value** - Selection Field with Constant types String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, *None*

Outputs:

- **Data:** Provided Dictionary
- **Control:** Next node

Note: If the key does not exist, it will be created

Name: Get Dictionary Value
Description: Gets the value of a key in a dictionary.
Properties:

- **Dictionary** - Selection Field with Constant type Dictionary
- **Key** - Selection Field with Constant types String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, *None*

Outputs:

- **Data:** Value of a dictionary - String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, *None*
- **Control:** Next node

Note: If the key does not exist, the output will be *None*

Name: Remove from Dictionary
Description: Removes the value of a key in a dictionary.
Properties:

- **Dictionary** - Selection Field with Constant type Dictionary
- **Key** - Selection Field with Constant types String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, *None*

Outputs:

- **Data:** Provided Dictionary with the removed value of provided key
- **Control:** Next node

Note: If the key does not exist, nothing will be deleted.

Name: Get Dictionary Values
Description: Gets values of a dictionary.
Properties:

> **Dictionary** - Selection Field with Constant type Dictionary

Outputs:

> **Data:** Dictionary Values in a List
> **Control:** Next node

Name: Get Dictionary Keys
Description: Gets keys of a dictionary.
Properties:

> **Dictionary** - Selection Field with Constant type Dictionary

Outputs:

> **Data:** Dictionary Keys in a List
> **Control:** Next node

Name: Get Dictionary Items
Description: Gets items of a dictionary.
Properties:

> **Dictionary** - Selection Field with Constant type Dictionary

Outputs:

> **Data:** Dictionary items in a List of tuples, where each tuple is key:value pair
> **Control:** Next node

Name: Get Index Value
Description: Gets value on the index of a data structure.
Properties:

> **Data Structure** - Selection Field with Constant types String, List, and Tuple
> **Index** - Selection Field with Constant type Integer representing the index of a data structure

Outputs:

> **Data:** Value on the index - String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, $None$
> **Control:** Next node

Name: Set Index Value
Description: Sets value on the index of a data structure.
Properties:

- **Data Structure** - Selection Field with Constant types String, List and Tuple
- **Index** - Selection Field with Constant type Integer representing index of a data structure
- **Value** - Selection Field with Constant types String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, *None*

Outputs:

- **Data:** Provided Data Structure - String, List, and Tuple
- **Control:** Next node

Name: Remove Index Value
Description: Removes value on the index of a data structure.
Properties:

- **Data Structure** - Selection Field with Constant types String, List and Tuple
- **Index** - Selection Field with Constant type Integer representing index of a data structure

Outputs:

- **Data:** Provided Data Structure - String, List, or Tuple with removed value on the provided index
- **Control:** Next node

Name: Append to list
Description: Appends a value to a list.
Properties:

- **List** - Selection Field with Constant type List, List for value to be appended to.
- **Value** - Selection Field with Constant types String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, *None*

Outputs:

- **Data:** Provided List with appended value
- **Control:** Next node

C.5 Memory/Data

Name: Define Variable
Description: Defines a variable and sets it to an input value.
Properties:

> **Name** - Text name for a variable

Outputs:

> **Data:** Nothing
> **Control:** Next node

Name: Assign Variable
Description: Selects a variable and sets it to an input value.
Properties:

> **Variable name** - Select for variable name (Requires a connected Define Variable node)

Outputs:

> **Data:** Nothing
> **Control:** Next node

Name: Data
Description: Creates a value of selected data type.
Properties:

> **Data Type** - Select for data type - String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, *None*
> **Value** - Input for value

Outputs:

> **Data:** Nothing
> **Control:** Next node

C.6 Communication

Name: Logger
Description: Sends feedback messages to the frontend console.
Properties:

> **Text** - Selection Field with Constant types String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, *None*

Outputs:

> **Data:** Nothing
> **Control:** Nothing - Logger's use is only to send messages to frontend.

Name: MQTT Server
Description: Sets up MQTT Server for Publishing/Subscribing to a topic.
Properties:

- **Name** - Text input, unique name to identify the server in the flow.
- **SSID** - Text input, username credential for the MQTT server
- **Password** - Text input, password credential for the MQTT server
- **Server Address** - Text input, Address of the MQTT server in format [IP|DomainName]:[Port] (e.g. 192.168.1.1:1883)

Outputs:

- **Data:** Nothing
- **Control:** Next node

Name: MQTT Publish
Description: Publishes message to a topic.
Properties:

- **Server name** - Selection input, MQTT server (Requires connected MQTT Server Node)
- **Topic** - Text input, topic to publish to
- **Message** - Selection Field with Constant types String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, $None$

Outputs:

- **Data:** Nothing
- **Control:** Next node

Name: MQTT Subscribe
Description: Subscribes to a topic.
Properties:

- **Server name** - Selection input, MQTT server (Requires connected MQTT Server Node)
- **Topic** - Text input, topic to publish to
- **Message** - Selection Field with Constant types String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, $None$

Outputs:

- **Data:** Received message
- **Control:** Next node

Name: HTTP Get
Description: Sends a GET request to the HTTP endpoint.
Properties:

> **URL** - Text input, URL of a website with endpoint

Outputs:

> **Data:** Received response in a Dictionary
> **Control:** Next node

Name: HTTP Post
Description: Sends a POST request to the HTTP endpoint.
Properties:

> **URL** - Text input, URL of a website with endpoint
> **Data** - Selection Field with Constant types Dictionary and None, Data to be sent with the request

Outputs:

> **Data:** Received response in a Dictionary
> **Control:** Next node

C.7 Machine

Name: Pin
Description: Configures a I/O pin (GPIO).
Properties:

> **Pin ID** - Number field, identification number of GPIO pin
> **Mode** - Select with options of setting the pin as an output, input, or open-drain
> **Pull Up/Down** - Select specifying whether the pin has a pull resistor attached

Outputs:

> **Data:** Nothing
> **Control:** Next node

Name: Set Pin
Description: Sets pin to High/Low
Properties:

> **Pin ID** - Select (Requires connected Pin node), Select a Pin to set a value on
> **State** - Select with options of setting pin on or off

Outputs:

> **Data:** Nothing
> **Control:** Next node

Name: Analog-digital converter (ADC)
Description: Converts analog input to a digital value
Properties:

Pin ID - Number field, identification number of GPIO pin
Reading Type - Select with values Raw (u16) and Microvolts, represents a type of reading from ADC

Outputs:

Data: Measured value - Integer
Control: Next node

Name: Write to SPI
Description: Writes data to SPI bus
Properties:

SPI Channel - Number field, specifies an SPI channel
Pin ID (CLK) - Number field, identification number of GPIO connected to Clock
Pin ID (CS) - Number field, identification number of GPIO connected to Chip Select
Pin ID (MOSI) - Number field, identification number of GPIO connected to Master Out Slave In
Pin ID (MISO) - Number field, identification number of GPIO connected to Master In Slave Out
Baudrate - Number field, represents the rate at which information is transferred
Data to write - Selection Field with Constant types String, Integer, Float

Outputs:

Data: Nothing
Control: Next node

Name: Read from SPI
Description: Reads data from SPI bus
Properties:

- **SPI Channel** - Number field, specifies an SPI channel
- **Pin ID (CLK)** - Number field, identification number of GPIO connected to Clock
- **Pin ID (CS)** - Number field, identification number of GPIO connected to Chip Select
- **Pin ID (MOSI)** - Number field, identification number of GPIO connected to Master Out Slave In
- **Pin ID (MISO)** - Number field, identification number of GPIO connected to Master In Slave Out
- **Baudrate** - Number field, represents rate at which information is transferred
- **Number of bytes to read** - Number field, represents the number of bytes to read from a bus

Outputs:

- **Data:** Bytes, data read from SPI bus
- **Control:** Next node

Name: Read from I2C
Description: Reads data from I2C bus
Properties:

- **I2C Channel** - Number field, specifies an I2C channel
- **Pin ID (SDA)** - Number field, identification number of GPIO connected to Synchronous data line
- **Pin ID (SCL)** - Number field, identification number of GPIO connected to Serial clock line
- **Frequency** - Number field, represents maximum frequency for SCL
- **Number of bytes to read** - Number field, specifies the number of bytes to read from the bus
- **Address of Peripheral** - Text field, specifies the address of the peripheral

Outputs:

- **Data:** Bytes, data read from I2C bus
- **Control:** Next node

Name: Write to I2C
Description: Writes data from I2C bus
Properties:

- **I2C Channel** - Number field, specifies an I2C channel
- **Pin ID (SDA)** - Number field, identification number of GPIO connected to Synchronous data line
- **Pin ID (SCL)** - Number field, identification number of GPIO connected to Serial clock line
- **Frequency** - Number field, represents maximum frequency for SCL
- **Address of Peripheral** - Text field, specifies the address of the peripheral
- **Data** - Selection Field with Constant type String, specifies data being written to I2C

Outputs:

- **Data:** Nothing
- **Control:** Next node

Name: Timer
Description: Sets a timer to execute a flow
Properties:

- **Period (ms)** - Number field, represents the timer period in milliseconds
- **Mode** - Select with options Periodic and Once, specifies whether the timer should trigger periodically or only one

Outputs:

- **Data:** Nothing
- **Control:** Next node

C.8 Peripherals

Name: DHT Sensors
Description: Reads from DHT Sensor
Properties:

- **Pin ID** - Number field, identification number of GPIO connected to the data bus
- **Type** - Select with options DHT11 and DHT22, specifies sensor type

Outputs:

- **Data:** Dictionary in a format {„temperature“: <value>, „humidity“: <value>}, where <value> is integer value measured by the sensor
- **Control:** Next node

Name: LED Matrix
Description: Displays text on LED matrix
Properties:

- **Pin ID (DIN|MOSI)** - Number field, identification number of GPIO connected to the data bus (MOSI)
- **Pin ID (CS|SS)** - Number field, identification number of GPIO connected to the card select (SS)
- **Pin ID (CLK|SCK)** - Number field, identification number of GPIO connected to the clock (SCK)
- **Number of Matrices** - Number field, represents the number of matrices connected in a row
- **Text** - Selection Field with Constant type String, represents text to be displayed

Outputs:

- **Data:** Nothing
- **Control:** Next node

Name: Character Display
Description: Displays text on the backlit character display
Properties:

- **Pin ID (SDA)** - Number field, identification number of GPIO connected to the Synchronous Data pin (SDA)
- **Pin ID (SCL)** - Number field, identification number of GPIO connected to the Synchronous Clock pin (SCL)
- **Number of Columns** - Number field, represents the number of columns on the display
- **Number of Rows** - Number field, represents the number of rows on the display
- **Cursor Position** - Selection Field with Constant type Tuple, represents x and y coordinates to display the text from
- **Text** - Selection Field with Constant type String, represents text to be displayed

Outputs:

- **Data:** Nothing
- **Control:** Next node

Name: NeoPixel LED
Description: Lights up an LED with specified color
Properties:

Pin ID - Number field, identification number of GPIO connected to the LED
Number of Pixel - Number field, represents the number of pixels that the LED contains
RGB Color - Selection Field with Constant type Tuple, represents RGB color to be shown in format (R, G, B)

Outputs:

Data: Nothing
Control: Next node

Name: Barometric Sensors
Description: Reads from Barometric (environment) Sensor
Properties:

Pin ID (SDA) - Number field, identification number of GPIO connected to the Synchronous Data pin (SDA)
Pin ID (SCL) - Number field, identification number of GPIO connected to the Synchronous Clock pin (SCL)
Type - Select with options BMP180, BMP280 and BME280, representing the type of the sensor

Outputs:

Data BMP180, BMP280: Dictionary in a format {„temperature“: <value>, „pressure“: <value>}, where <value> is integer value measured by the sensor
Data BME280: Dictionary in a format {„temperature“: <value>, „pressure“: <value>, „humidity“: <value>}, where <value> is integer value measured by the sensor
Control: Next node

Name: Ultrasonic Distance Sensor
Description: Reads from Ultrasound Distance Sensor (HY-SRF05/HC-SR04)
Properties:

Pin ID (Trigger) - Number field, identification number of GPIO connected to the Trigger
Pin ID (Echo) - Number field, identification number of GPIO connected to the Echo

Outputs:

Data: Distance in centimeters - Float
Control: Next node

C.9 Miscellaneous

Name: Format String
Description: Formats a value into a string
Properties:

Formatted String - Text field, specifying the formatted string. This field has to contain {} which represents the place where the value is going to be formatted.
Parameter - Selection Field with Constant types String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, *None*

Outputs:

Data: Formatted string - String
Control: Next node

Name: Convert
Description: Casts a parameter to the specified type.
Properties:

Data type - Select with options String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, specifies the casting data type
Parameter - Selection Field with Constant types String, Integer, Float, Boolean, List, Dictionary, Set, Tuple, *None*

Outputs:

Data: Parameter converted to a specified type
Control: Next node

Name: Random Number Generator
Description: Generates a random number from the specified interval.
Properties:

Start - Number input, specifies the lower bound of the interval
End - Number input, specifies the higher bound of the interval

Outputs:

Data: Random number - Integer
Control: Next node

Name: Python Code
Description: Executes a custom Python code
Properties:

Code - Text input, specifies the Python code to be executed

Outputs:

Data: Specified by user in _output variable or *None*
Control: Next node

Name: Timestamp
Description: Adds a timestamp before a parameter
Properties:

- **Parameter** - Selection Field with Constant type String, specifies the parameter before which the timestamp will be added

Outputs:

- **Data:** Parameter with prefixed timestamp - String
- **Control:** Next node