Mendel University in Brno

Faculty of Business and Economy

MicroPython Utilizing Zephyr Port and NXP FRDM-MCXN947

Bachelor thesis

Supervisor:
Ing. Jan Kolomazník, Ph.D.

Dmitrii Titarenko

Brno 2025

Declaration

I hereby declare that this thesis entitled **MicroPython Utilizing Zephyr Port and NXP FRDM-MCXN947** was written and completed by me. I also declare that all the sources and information used to complete the thesis are included in the list of references. I agree that the thesis could be made public in accordance with Article 47b of Act No. 111/1998 Coll., Higher Education Institutions and on Amendments and Supplements to Some Other Acts (the Higher Education Act), and in accordance with the current Directive on publishing of the final thesis.

I am aware that my thesis is written in accordance to Act No. 121/2000 Coll., on Copyright and therefore Mendel University in Brno has the right to conclude licence agreements on the utilization of the thesis as a school work in accordance with Article 60(1) of the Copyright Act.

Before concluding a licence agreement on utilization of the work by another person, I will request a written statement from the university that the licence agreement is not in contradiction to legitimate interests of the university, and I will also pay a prospective fee to cover the cost incurred in creating the work to the full amount of such costs.

Brno 03.08.2025

....

signature

Abstrakt

Titarenko, D. MicroPython Utilizing Zephyr Port and NXP FRDM-MCXN947. Bachelor thesis. Brno, 2025.

Práce zkoumá podporu MicroPython na Zephyr RTOS s využitím vývojové desky FRDM-MCXN947 od NXP. Samo o sobe Zephyr RTOS poskytuje širokou podporu a snadno použitelné API pro mnoho embedded zařízení a jejích periferii, ale podpora MicroPython je stále limitovaná a nekonzistentní s MicroPython porty vyvíjený pro jiná zařízení. Práce analyzuje součastné limity MicroPython na Zephyr RTOS a vybírá funkčnost pro implementaci. Implementace zahrnuje inicializaci vývojovýho prostředí pro Zephyr a MicroPython, které bude upravené pro FRDM-MCXN947, provádění funkčního testování periferii vývojové desky v vývojovém prostředí pro Zephyr a MicroPython a porovnáním kompatibility nativného poru a Zephyr portu MicroPython.

Klíčová slova: MicroPython, závěrečná práce, Zephyr RTOS, FRDM-MCXN947

Abstract

Titarenko, D. MicroPython Utilizing Zephyr Port and NXP FRDM-MCXN947. Bachelor thesis. Brno, 2025.

This thesis explores MicroPython support on Zephyr RTOS using the NXP FRDM-MCXN947 development board. Zephyr RTOS itself provides extended support and easy-to-use APIs to many embedded devices and their peripherals, but the support of MicroPython remains limited and not consistent with MicroPython ports developed for other devices. This work analyzes the current limitations of MicroPython on Zephyr RTOS and selects specific functionality for implementation. The implementation involves initializing Zephyr and MicroPython development environments adjusted to FRDM-MCXN947, conducting functional testing of the development board peripherals with Zephyr and MicroPython environment, and comparing the compatibility of native and Zephyr ports of MicroPython.

Key words: MicroPython, thesis, Zephyr RTOS, FRDM-MCXN947

Contents

List of Figures

Listings

1 Introduction

The world of microcontrollers and embedded devices continues to grow, and such devices become more common with every day. Even though they are seldom notice-able we more often might find ourselves surrounded by them. From home appliances, to cars, to factory machines to city-wide networks embedded devices reach wide and deep in our lives.

But with the growing count of embedded devices grows complexity of functions they implement. Hence arises a need for Operating Systems(OSs) to manage sets of complex programs and provide a layer of abstraction to ease development in such constrain but demanding environments.

To cover demand of an Operating System in embedded devices the Zephyr Real Time Operating System (RTOS) was created. Zephyr RTOS is an open-source op-erating system with build-in security and optimization for resource limited devices. Zephyr kernel supports ARM, Intel x86, ARC, RISC-V, Nios II, Tensilica Xtensa and large number of development boards, among others NXPs FRDM-MCXN947. ALso Zephyr has rich API that allows developers to write high-level code for em-bedded devices.

Writing software for embedded devices is still a complex task regardless of what underlying technologies are used. Writing it in a language such as C adding an additional complexity due to need of managing program memory allocation and deallocation by hand. Leaving unhandled memory sector could lead to memory leaks or worse opens an opportunity for an attacker to execute malicious code on an embedded device. The consequences of such problems become much grater when occurring in the embedded world. MicroPython is an optimized subset of Python 3 programming language for embedded devices. It aims to ease writing software by managing memory using its garbage collector system, using easy to read Python-like syntax and providing various modules to enable work with different peripherals. Additionally, MicroPython allows for code portability, meaning that code written for FRDM-MCXN947 could be ported and ran on ESP32 with minimal updates to code.

But MicroPython does not support every single device straightaway – ported versions of MicroPython are submitted to the MicroPython repository, by a man-ufacturer or an enthusiasts. Later submitted port will be reviewed and tested by MicroPython maintainers, which is a lengthy process, for example MicroPython port for Zephyr was under review for 2 years.

By combining Zephyr RTOS and MicroPython in one technological stack the best of both technologies could be utilized. Potential exists for developers to write highly readable, easy-to-understand and efficient code with MicroPython that make use of various hardware support introduced by Zephyr RTOS. Yet the state of this development environment is not yet firm and have plenty of rough edges and unap-parent problems that could arise during the process of software development.

The aim of this thesis is to construct a method for configuring the development

environment for MicroPython, Zephyr RTOS and FRDM-MCXN947 development board, provide insight of compatibility challenges of both platforms and propose potential solutions for extending MicroPython port to Zephyr RTOS. The finding of this thesis will contribute to understanding of the state of both platforms and their integration, informing future developers and increasing usability of MicroPython and Zephyr.

1.1   Goal of this thesis

Goal of the thesis is to establish development environment and workflow for developing applications for embedded devices with the MicroPython Zephyr port. This work could be used in future to ease start of application development and as a reference.

2 Background

This chapter introduces the reader to an information about Zephyr RTOS, MicroPython and FRDM- MCXN947 board that is needed for understanding this thesis.

2.1 Zephyr RTOS

Zephyr is an Operation System designed for resource-constrained and embedded system from simple sensors to smart industrial embedded solutions with emphasis on safety. It supports a broad list of embedded devices, development boards and peripherals. Zephyr offers extensive number of features and services including multi-threading, inter-thread data passing, inter-thread synchronization, dynamic memory allocation, interrupt service, power management, networking, file system. Zephyr project is open-source, distributed under Apache 2.0 license and was created under Linux Foundation organization.(Zephyr Project, Introduction, 2024)
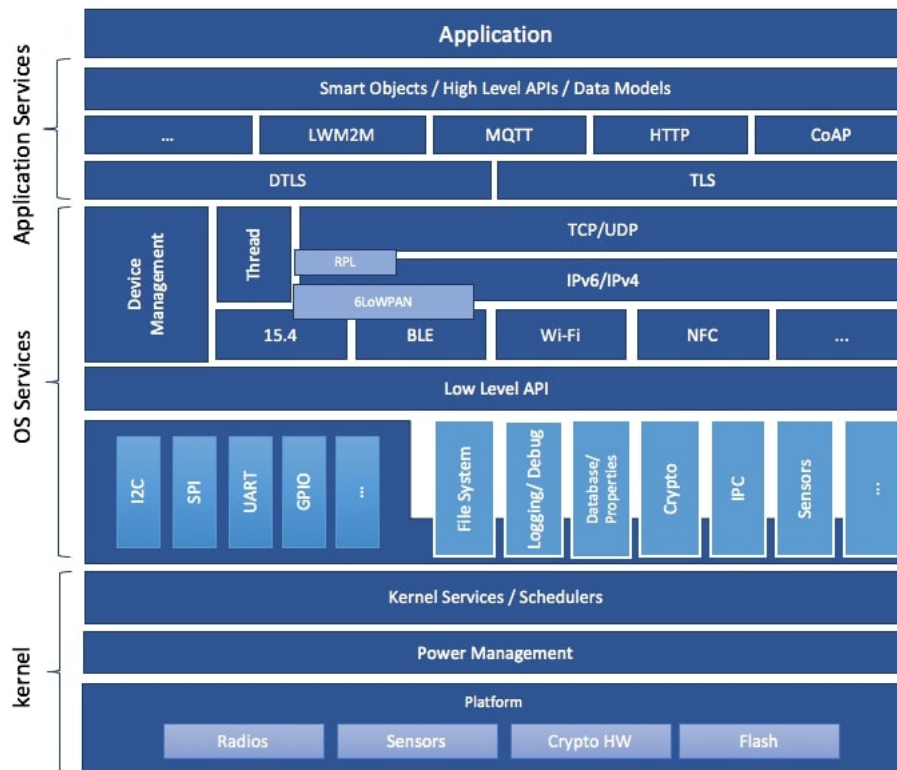
Figure 1: Zephyr System Architecture
Source: (Zephyr Project, Zephyr Security Overview, 2024)

2.1.1 West

West is a part of Zephyr's tool-chain used for building and configuring. West can initiate Zephyr workspace from official upstream repository, update or change version of a local Zephyr workspace to any version in official repository, build Zephyr application from source, flash built application to a board.(Zephyr Project, West (Zephyr's meta-tool), 2024)

2.1.2 Kconfig

Kconfig is Zephyr's kernel, peripheral drivers and subsystems configuration system that allow to configure Zephyr at a build time. Kconfig goal is to enable configuration without introducing changes to the source code.

The initial board configuration can be found in **<board>_defconfig** files. For example configuration file for FRDM-MCXN947 is located at **boards/nxp/frdm_mcxn947/ frdm_mcxn947_mcxn947_cpu0_defconfig**. The board configuration for NXPs' FRDM-MCXN947 is as follows:

Listing 1: FRDM-MCXN947 Kconfig configuration

```
CONFIG_CONSOLE=y
CONFIG_UART_CONSOLE=y
CONFIG_SERIAL=y
CONFIG_UART_INTERRUPT_DRIVEN=y
CONFIG_GPIO=y
CONFIG_PINCTRL=y
CONFIG_ARM_MPU=y
CONFIG_HW_STACK_PROTECTION=y
CONFIG_TRUSTED_EXECUTION_SECURE=y
```

Kconfig values can be set to a **<board>_defconfig** files, temporarily with terminal graphical interfaces or with a **prj.conf** file at application level which overrides the initial configuration during application build.(Zephyr Project, Configuration System (Kconfig), 2022)

2.1.3 Devicetree

Devicetree is a data structure to describe hardware. It is a community driven standard that is heavily used in Zephyr project. In Zephyr devicetrees are usually build inherently meaning that for example FRDM-MCXN947 has a devicetree configuration **board/nxp/frdm_mcxn947/frdm_mcxn947_mcxn947_cpu0.dts** which mainly enables peripheral devices, but includes FRDM-MCXN947 specific configuration from **frdm_mcxn947.dtsi** (include file), which in turn includes **frdm_mcxn947-pinctrl.dtsi** file that mostly defines pinmux groups. Additionally the **frdm_mcxn947_mcxn947_cpu0.dts** includes **nxp_mcxn94x.dtsi** file that defines memory ranges for SRAM, FLEXSPI and peripherals and includes

nxp_mcxn94x_common.dtsi include file where most of devices including CPU, GPIO, CTIMER and others are defined and assigned memory ranges.(devicetree.org, Devicetree Specification Release v0.4, 2023)

Same as Kconfig Devicetrees can be overwritten or have some specific devices configured differently with *overlay* files, which as well needs to be placed in build directory, from there *west* tool will use it to edit the Devicetree configuration.

2.2   MicroPython

MicroPython is an open-source project founded by Damien George. MicroPython is an implementation of the Python programming language that is optimized to be run on embedded and resource constraint devices. It implements the entire Python 3.4 syntax with some selected features from the later versions such as *async/await* from Python 3.5 , additionally on par with Python it uses garbage collection system for memory management. MicroPython final build include a compiler that compiles MicroPython code to bytecode and an runtime interpreter of the compiled byte-code. Programs could be written directly to the MicroPython REPL(Read–eval–print loop) or be loaded onto MicroPython host device with use of serial connection and utility programs like *ampy.*

MicroPython's core development is focused on implementing and maintaining core features of the MicroPython like Python language features, libraries, memory management and MicroPython interpreter. The responsibility for adapting and porting MicroPython to different platforms lies on the community around it. Every MicroPython port introduces required adaptations and addresses hardware features and limitations of its platform. Consequently, MicroPython support is not linear on all platforms, because some might lack the necessary configuration for enabling a part of functionality or even lack reimplementation of a number of core libraries. Additionally, the slow pace of adding to source code features for various platforms created by the community means that even fully functional and tested ports or features might wait for months before being reviewed. Despite all of this, there are already many supported devices and architectures that MicroPython can run on. MicroPyton has additional support to be run on operating system Zephyr RTOS and on OSes from UNIX family, as well as experimental Windows port.

MicroPython remains in beta-stage, hence it is a subject to possible API and code-base changes in the future.(Nicholas H. Tollervey, 2017)

2.3   FRDM-MCXN947

The FRDM-MCXN947 is a low-cost development board designed by NXP semi-conductors. FRDM-MCXN947 integrates Dual Arm Cortex-M33 microcontroller, a neural processing unit, P3T1755DP I3C temperature sensor, TJA1057GTK/3Z CAN PHY, Ethernet PHY, SDHC circuit, RGB LED, touch pad, high-speed USB, MCU-Link debuger, push buttons and has an option to be extended with external

devices. (NXP semiconductors, UM12018 FRDM-MCXN947 Board User Manual, 2024)
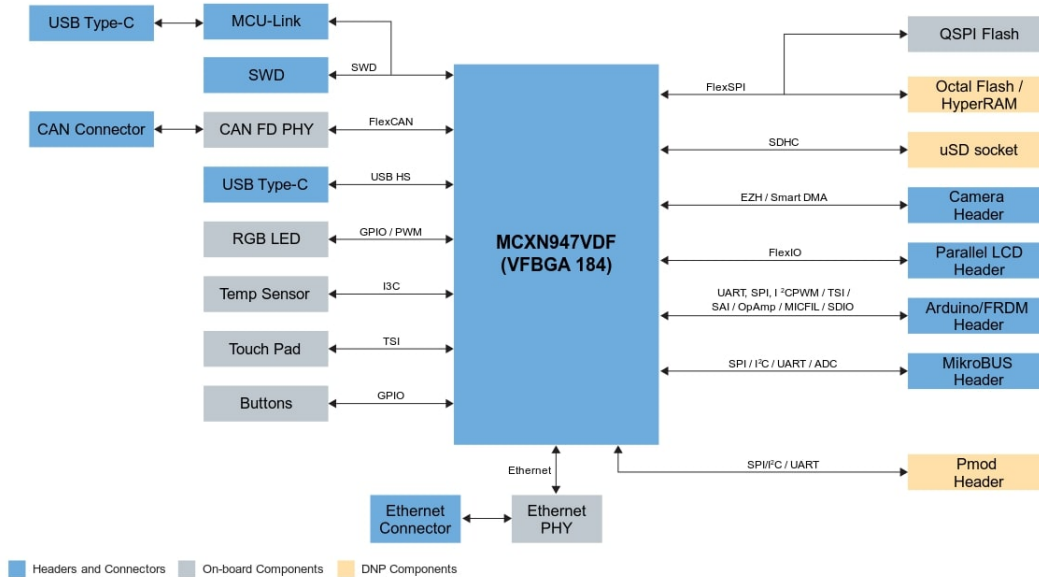
Figure 2: FRDM-MCXN947 Block diagram
Source: (NXP semiconductors, FRDM Development Board for MCX N94/N54 MCUs , 2025)

2.3.1 Signal Multiplexing

FRDM-MCXN947 enables use of several functions for different pins by utilizing Signal Multiplexing. For example pin **P0__10** which is an red RGB pin can use **GPIO** functionality directly, **FLEXCOMM** by utilizing **FC0__P6** FLEXCOMM device, **CTIMER** by utilizing **CT0__MAT0** CTIMER device, and **FLEXIO** functionality by utilizing **FLEXIO0__D2** device.

Only one function can be used at a time on a pin and only one pin can be assigned to a peripheral device. (NXP semiconductors, MCX Nx4x Reference Manual, 2025)

2.3.2 LinkServer

LinkServer is an NXP command-line utility that provides target flashing capabilities and firmware updates for FRDM-MCXN947. This is typically used as a backend for flashing FRDM-MCXN947 in *west* utility. (NXP semiconductors – LinkServer for Microcontrollers, 2025)

2.4   MicroPython port to Zephyr

While Zephyr RTOS provides a feature-packed and expandable development and system to be used in embedded world standard development in C can be time consuming. MicroPython's Zephyr port brings advantages of both MicroPython and Zephyr to single environment, allowing to write hight-level Python like code and rapidly prototype and debug, while also leveraging hardware agnostic Zephyr APIs and wide support of different embedded devices and their peripherals.

But despite growth in support between Zephyr and MicroPython there are still features that lack in the port and issues with coupling of both technologies. For a long time the MicroPythons' Zephyr port have been using an older versions of Zephyr and MicroPython itself. It used MicroPython 1.19.1 and Zephyr 3.1.0 versions which both came out in period between May and June 2022 until September 2024. In September 2022 began work by Maureen Helm to introduce a CI pipeline into MicroPython repository to ease porting MicroPython to latest Zephyr release. From this work emerged last MicroPython Zephyr port version based on MicroPython 1.24.0 and Zephyr 3.7.0.

And though MicroPython Zephyr port already supports the MicroPython modules like socket, time, math, machine and other are implemented and usable in the final MicroPython build they may lack support of some sub-modules like machine's PWM sub-module or functionality of modules and sub-modules like not yet implemented features of machine's I2C sub-module that do not have ability to set clock and data lines.

A MicroPython's Zephyr port is built in a same way any Zephyr application is built. A *west* utility is used and MicroPython port to Zephyr source code as a build target. The final build could be configured and some features or peripherals could be activated or deactivated with *Kconfig* and a final devicetree be overwritten with devicetree *overlays*. Then the result binary file is fleshed to a target board with *west* utility.

2.5   Summary

For utilizing MciroPython Zephyr port on FRDM-MCXN947 board it is needed to build MicroPython port as a Zephyr application, and then flash this application onto the development board, both operations are made with use of Zephyr's *west* utility. Program flashing is made using *LinkServer*.

Pre-build configuration is possible using *Kconfig*, for setting what peripherals and sub-system are to be enabled or disable, and *Devicetree*, for creating or updating a structured description of the underling hardware.

FRDM-MCXN947 is a programmable and extendable development board with many devices, peripherals and systems available.

3   Methodology

3.1   Introduction

This chapter introduces the reader to the set of tools used in this thesis and outlines a methodological approach. First Zephyr and MicroPython development environments will be setup and tested, following with their integration for building MicroPython's Zephyr port. Next the MicroPython's Zephyr port will be extended with use of configuration and additional code to bring it closer to the level of native port support. Finally, the thesis will conclude by building a web server that will run on FRDM-MCXN947 utilizing MicroPython's Zephyr port.

3.2   Zephyr development environment setup

This section will detail the process of setting up a development environment for working with Zephyr RTOS. The setup will be broken into several steps to provide clear and easy-to-follow instructions to the reader.

Firstly, the prerequisites and dependencies would be introduced along with the installation process. The reader will be briefed on purpose of the key dependencies. After the necessary preparation the Zephyr Software Development Kit (SDK) and source code will be obtained. For the Zephyr SDK two methods for obtaining it will be presented and discussed. Then the *west* tool will be described in detail and it's abilities for managing the project will be discussed. Next, the installation and use of the *LinkServer* utility and its alternatives will be discussed. Finally, the development setup will be verified with a functional test on a FRDM-MCXN947 board and test prerequisites discussed.

3.3   Use of MicroPython on FRDM-MCXN947

This section will describe how to use MicroPython on FRDM-MCXN947 board.The *LinkServer* utility will be used and aspects of it will be discussed.

The section will begin with an explanation on what are the possibilities to obtain MicroPython binary. Particularly, the current process of that takes place to obtain MicroPython binary working on FRDM-MCXN947 will be shown. After obtaining the binary it will be shown how to load it to the board. The section will conclude with a functional test of MicroPython possibilities.

3.4   MicroPython port to Zephyr RTOS

The section will present an approach on how to setup and start using MicroPython's Zephyr port.

It will start with setting up a development environment for MicroPython. After this process of building a MicroPython Zephyr port will be shown with ways of how to influence the final build. The section will conclude with an illustrative program.

3.5   Comparing native and Zephyr ports

This section will compare two MicroPython builds the NXP native MicroPython implementation and Zephyr port focusing on availability and functionality of the built-in modules. The goal of this section is to identify what modules are supported by each implementation and to highlight the missing features. In process how modules are getting added to the MicroPython's Zephyr port will be explored, which will allow to extend modules support in feature section.

3.6   Extending Zephyr Port functionality

This section will try to extend the MicroPython's Zephyr port with modules.

First, list of modules available to use in MicroPython's Zephyr port will be extended by methods explored in previous section. Next, a community extension will be added to bring a PWM support and the board build will be configured with *Kconfig* for PWM to work. Finally, how a new module could be created, and how to add this module to the MicroPython build.

3.7   Creating an HTTP server of FRDM-MCXN947 with MicroPython's Zephyr port

In this section the process and steps to create a web server for controlling the FRDM-MCXN947 board will be discussed. Zephyr's strong networking stack will be utilized for local network IP configuration as well as data transition and reception. The configuration process using *Kconfig* and *Devicetree* will be needed to enable certain parts of the Zephyr core libraries or to configure the hardware interface. The necessary configuration will be explained along with the code for the server.

4   Implementation

4.1   Introduction

This chapter describes configuration and development of the MicoPython's Zephyr port itself as of the web server built utilizing this port. The main outcome of this thesis is a MicroPython-based web server running on a FRDM-MCXN947 board that host Zephyr RTOS.

The first part of this thesis is focusing on describing the setup and a necessary configuration of the development environment for building and running MicroPython on Zephyr RTOS. Next the MicroPython port for Zephyr is compared against a MicroPython port for FRDM- MCXN947 developed by NXP. The MicroPython's Zephyr port then is synchronized with the port by NXP in terms of capabilities and modules supported.

This chapter concludes with implementation of a web server that demonstraites the capabilities of the extended MicroPython's Zephyr port.

4.2   Zephyr development environment setup

4.2.1   Installing Zephyr and it's dependencies

Before starting to setting up the development environment for Zephyr the needed dependencies need to be obtained. Those include :

- *git* – Zephyr's *west* tool uses *git* as a backend

- *cmake* – Zephyr use *cmake* as it's build system for configuration and build files generation.

- *ninja* – is a performance based build system that is used by CMake for building a project.

- *make* – some tools expecting or fall back to Make.

- *gperf* – GNU utility for fast hash generation used in Zephyr for lookup.

- *ccache* – an utility that speeds up compilation time by caching previous compilation.

- *dtc* – compiles *Devicetree* files.

- *xz* – is a set of data compression utilities used for decompressing Zephyr SDK.

- *wget* – used to download Zephyr SDK.

- *python-pip* – used to install Zephyr's *west* tool.

- *python-setuptools* – Python build helper package.

- *python-wheel* – Python build helper package.

- *dfu-util* – implementation of Direct Firmware Update and is used in Zephyr to flash firmware and programs to a board.

The next step after obtaining all the dependencies is to install the Zephyr SDK. There are two options of installing Zephyr SDK: using the *west* tool (which is shown at page 24) or performing a manual installation. To install the SDK in desired installation directory execute:

Listing 2: Installing Zephyr SDK: obtaining SDK

```
wget  https://github.com/zephyrproject−rtos/sdk−ng/releases/
    download/v0.17.0/zephyr−sdk−0.17.0_linux−x86_64.tar.xz
wget −O − https://github.com/zephyrproject−rtos/sdk−ng/
    releases/download/v0.17.0/sha256.sum | shasum −−check −−
    ignore−missing
```

These commands download the archived Zephyr SDK and verify it's integrity using SHA-256 checksum. The download *URL* could be altered to download the SDK for other operating systems like *Windows, Mac Os* by changing the system name to *windows*(windows SDK version uses 7z archive format and does not have ARM variant) or *macos* and for ARM processors by changing architecture name to *aarch64*.

The content of the downloaded archive is the extracted file and the *setup* script is run. Additionally for Linux systems it is recommend to install *udev* rules, which will allow to flash boards with Zephyr as a regular user:

Listing 3: Installing udev rules

```
sudo cp ~/zephyr−sdk−0.17.0/sysroots/x86_64−pokysdk−linux/
    usr/share/openocd/contrib/60−openocd.rules /etc/udev/
    rules.d
sudo udevadm control −−reload
```

For building and working with Zephyr applications it is required to obtain Zephyr source code and configure a Python virtual environment. First, Python virtual environment *venv* needs to be installed. In the preferred working directory then the new *venv* environment is initialized and activated:

Listing 4: Creating and activating Python venv environment

```
python3 −m venv ~/zephyrproject/.venv
source ~/zephyrproject/.venv/bin/activate
```

Zephyr uses it's tool *west* for to manage it's source code, dependencies and to build and flash applications. *West* could be installed to *venv* environment using:

Listing 5: Installing Zephyr's West tool

```
pip install west
```

With *west* installed it could be used to obtain Zephyr's source code:

Listing 6: Obtaining Zephyr's source code

```
west init
west update
```

This command registers the current Zephyr installation as a CMake config package in the CMake user package registry.(Zephyr Project, Additional Zephyr extension commands, 2025)

```
west zephyr−export
```

At this point Zephyr's SDK could be installed if it was not already using *west*:

Listing 7: Installing Zephyr SDK using West

```
west sdk install
```

4.2.2   Installing LinkServer

To flash application to FRDM-MCXN947 board Zephyr uses the NXP's *LinkServer* utility. The installation files for different Operating systems are available on the web page of the *LinkServer* utility.(NXP semiconductors – LinkServer for Microcontrollers, 2025)

Also J-Link could be used as an alternative though it needs additional hardware peripherals. (Zephyr Project, FRDM-MCXN947, 2025)

4.2.3   Configuring serial port

This step is not required for the development and correct work with Zephyr, but it's useful to have a capability to see board logs and to interact with the board through a command line interface from the development environment. For Linux and Mac OS *minicom*(Jayantilal, S. H., Interfacing of AT command based HC-05 serial Bluetooth module with minicom in Linux, 2014) and *screen*(Free Software Foundation, GNU Screen, 2016) open-source utilities could be used, for Windows *PuTTY* serial port capabilities could be utilized.

For establishing serial communication with a FRDM-MCXN947 board the following configuration parameters should be used(NXP semiconductors, Getting Started with FRDM-MCXN947,2024):

Listing 8: Serial port configuration

```
Baud rate: 115200
Data size: 8
Parity: None
Stop bit: 1
```

4.2.4   Building and flashing a program to FRDM-MCXN947

In this subsection to carry out testing of the development environment and demonstrate the flashing and building process of Zephyr applications, a Zephyr-provided sample program will be built.

To build the sample application *west* utility is used:

Listing 9: Building the Blinky Program

```
west build −p always −b frdm_mcxn947/mcxn947/cpu0 samples/
    basic/blinky
```

where arguments *-p always* makes a pristine build (Zephyr Project, Building, Flashing and Debugging, 2024), *-b frdm_mcxn947/mcxn947/cpu0* specifies the target(*cpu0* is specified because only it can be targeted standalone, second core is enabled after configuring (Zephyr Project, FRDM-MCXN947, 2025)) and *samples/basic/blinky* specifies program to build.

The built binary then can be flashed to the board using *west* and *LinkServer*. For this, board need to be connected with USB-C via *J-17* port. Flashing is run with:

Listing 10: Flashin a program

```
west flash
```

After flashing, the red led will start to blink with a one-second period.

With the successful build and flashing of the program to the FRDM-MCXN947 board, it is verified that the Zephyr development environment functions correctly and now prepared for further development.

4.3   Use of MicroPython on FRDM-MCXN947

4.3.1   Installing MicroPython on FRDM-MCXN947

In this section the process of installing MicroPython interpreter on FRDM-MCXN947 board is described.

There are many boards that have direct MicroPython ports or support through their architecture(MicroPython, MicroPython downloads, 2025), but FRDM-MCXN947 is not among them due to the port made by NXP developers not yet being reviewed and merged into the main MicroPython repository. Instead the MicroPyhton binary must be obtained through NXP's article(NXP semiconductors, Quick start guide for MicroPython on FRDM-MCXN947 board , 2024) or compiled from source code of the development branch from NXP with port to FRDM-MCXN947. For sake of time saving the MicroPython binary version used in this thesis was downloaded from the said NXP article.

Before proceeding with flashing the binary to the target board the *LinkServer* utility must be installed on the host computer and the board itself is connected to the host computer with USB-C via *J-17* port.

To ensure the board has no prior binaries install the following command is used:

Listing 11: LinkServer erase
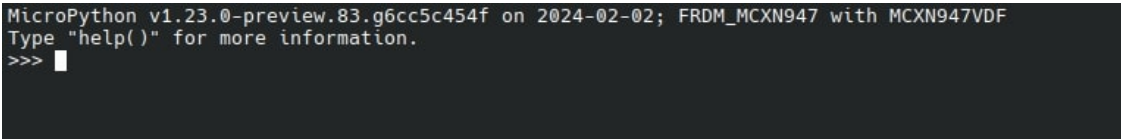
```
LinkServer flash MCXN947:FRDM–MCXN947 erase
```

To flash the binary to the FRDM-MCXN947 this command is used:

Listing 12: LinkServer flash

```
LinkServer flash MCXN947:FRDM–MCXN947 load firmware.bin —
    addr 0
```

After the flash process has ended, serial port connection could be made to the board, which will reveal REPL environment for code execution.


```
MicroPython v1.23.0-preview.83.g6cc5c454f on 2024-02-02; FRDM_MCXN947 with MCXN947VDF
Type "help()" for more information.
>>> 
```

Figure 3: MicroPython REPL environment on the FRDM-MCXN947 board, access via the *minicom* utility

4.3.2   Writing and flashing MicroPython programs

There are two common approaches to writing programs for MicroPython: either by entering the code directly to REPL, which allows fast prototyping and modules or periphery testing, or to load the python code onto the board using utilities like *rshell* or *ampy*.

Both *ampy* and *rshell* utilities are installed via Python's *pip* package manager and to the Python virtual environment.

To test MicroPython capability and program loading to the board with MicroPython the following code for reading temperature data from the *P3T1755* temperature sensor on the FRDM-MCXN947 board:

Listing 13: Read tempreture data from P3T1755 sensor using MicroPython

```
import machine, time

P3T_ADDR = 72

i2c = machine.I2C(5, scl=machine.Pin('P1_17'), sda=machine.
    Pin('P1_16'))

class P3T1755:
    def __init__(self, i2c, addr):
        self.addr = addr
```

```
        self.i2c = i2c

        self.i2c.writeto(self.addr, b'\x00')

    def read(self):
        temp_raw = i2c.readfrom(self.addr, 2)
        temp_converted = ((temp_raw[0] << 4) | (temp_raw[1]
            >> 4)) * 0.0625
        return temp_converted

ts = P3T1755(i2c, P3T_ADDR)

while(True):
    print("Temperature: {:.2f} Deg.C".format(ts.read()))
    time.sleep_ms(100)
```
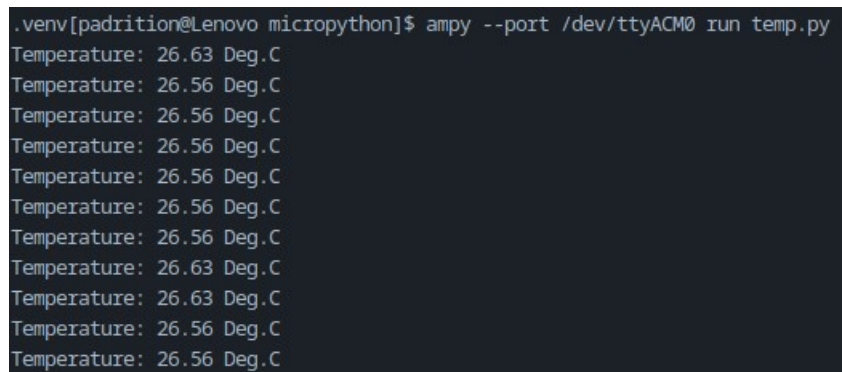
To load and run this code on the board with MicroPython the code must be saved to a file and loaded with:

Listing 14: Loading a program to FRDM-MCXN947 board with MicroPython using ampy utility

```
ampy ——port /dev/ttyACM0 run you_micropython_code.py
```

The output from the board will be shown in the console the program was loaded from, to only load the program the *–no-output* parameter could be passed to the *ampy* utility.

Figure 4: Output of the FRDM-MCXN947 board after loading temperature reading program

It is now verified that the MicroPython port by NXP works correctly on the FRDM-MCXN947 board.

4.4   MicroPython port to Zephyr RTOS

4.4.1   Setup of the MicroPython Zephyr port

This section details the process of building and running MicroPython's Zephyr port on FRDM- MCXN947 development board. To begin firs the MicroPython source code should be obtained, either from *git* repository or as a archived file from the project's web.

Before proceeding with the build creating a Python virtual environment in the Zephyr installation directory if not done yet.

The build of the MicroPython's Zephyr port is done from Zephyr development environment using *west* utility:

Listing 15: Building MicroPython's Zephyr port

```
west build −p always −b frdm_mcxn947/mcxn947/cpu0 ~/
    micropython/ports/zephyr
```

Where the target program is a path to */ports/zephyr* directory in MicroPython's source code.

During the initial build the following error may occure:

Listing 16: Error building MicroPython's Zephyr port

```
/micropython/ports/zephyr/modzephyr.c:52:5: error: too few
    arguments to function 'thread_analyzer_print'
    52 |     thread_analyzer_print();
       |     ^~~~~~~~~~~~~~~~~~~~~
```

The issue is due to an API mismatch introduced in newer versions of Zephyr. To solve this, the Zephyr version should be lowered to 3.7.0. This could be done using *git* and it's *checkout* command to switch to *v3.7-branch* branch.

After switching to an older branch the building process finishes without further errors and the resulting binary could be flashed to the FRDM-MCXN947 board using *west* utility.

After flash with the help of serial port program a similar to NXP's MicroPython port REPL environment is seen with additional information on hosting OS.

Figure 5: MicroPython's Zpehyr port REPL environment on the FRDM-MCXN947 board, access via the *minicom* utility

The same applies to program loading as with NXP's MicroPython port. Programs can be either written directly into REPL environment or uploaded using tools such as *ampy*.

To verify the functionality of the built MicroPython's Zephyr port and compare it to NXP's port a simple program was developed, which utilizes *GPIO* pins of green and red on board LEDs to switch between low and high voltage.

Listing 17: Led program for MicroPython's Zephyr port

```python
import time
from machine import Pin

red = Pin(("gpio0", 10), Pin.OUT)
red.value(1)
green = Pin(("gpio0", 27), Pin.OUT)
green.value(1)

while True:
    green.value(0)
    time.sleep(1.5)
    red.value(0)
    time.sleep(1.5)
    green.value(1)
    time.sleep(1.5)
    red.value(1)
```

After loading the program to the FRDM-MCXN947 board the red and green LEDs will be periodically set to high and low voltage.

Note how the program needs to refer to a pin in the MicroPython's Zephyr port as a tuple while in NXP's port it uses a string name of the pin. MicroPython's Zephyr port use definition of the board and periphery from *Devicetree* files where *gpio0* is a peripheral which exposes various pins, such as pins 10 and 27, each configured for specific functions.

4.5   Comparing native and Zephyr ports

4.5.1   Modules available on NXP's port

This section compares module support of NXP's and Zephyr MicroPython ports. The analysis includes a list of modules available for both ports. The analysis was made by importing modules into the REPL environment and passing them to the *help()* function.

NXP port provides a variety of modules specific for the hardware of the FRDM-MCXN947 board and common MicroPython modules. To get a list of supported modules on NXP port *help('modules')* command could be used in the REPL environment which returns the following list:

Listing 18: NXP's port modules

__main__	asyncio/stream
_asyncio	binascii
_boot	builtins
_onewire	cmath
array	collections
asyncio/__init__	deflate
asyncio/core	dht
asyncio/event	ds18x20
asyncio/funcs	errno
asyncio/lock	framebuf
gc	os
hashlib	platform
heapq	random
io	re
json	select
machine	struct
math	sys
mcx	time
micropython	uasyncio
onewire	uctypes

Modules could also be passed as parameters to the *help()* function to reveal what functions does the module have and what constants it defines.

4.5.2   Modules available on Zephyr port

In contrast to the NXP's port, running *help('modules')* does not yield the supported modules. Instead to get the list of supported modules it is needed to either try to import modules and pass them to the *help()* function or to read the source-code of the MicroPython's Zephyr port, specifically the *CMakeLists.txt CMake* file where some of the imported modules are listed :

Listing 19: Zephyr port CMake list

```
set(MICROPY_SOURCE_PORT
    main.c
    help.c
    machine_i2c.c
    machine_spi.c
    machine_pin.c
    modbluetooth_zephyr.c
    modsocket.c
    modzephyr.c
    modzsensor.c
```

```
    mphalport.c
    uart_core.c
    zephyr_device.c
    zephyr_storage.c
    mpthreadport.c
)
list(TRANSFORM MICROPY_SOURCE_PORT PREPEND ${
    MICROPY_PORT_DIR}/)
```

Those modules are direct ports of the equivalent Python or MicroPython modules with use of Zephyr APIs.

Additionally, MicroPython's Zephyr port contains *prj.conf* file that is passed to Zephyr when compiling the port and it hav *CONFIG_MICROPY_CONFIGFILE* parameter that specifies a modules configuration file *mpconfigport.h* :

Listing 20: Part of MicroPython's Zephyr port mpconfigport.h module conifguration file

```
...
#define  MICROPY_ENABLE_SOURCE_LINE    (1)
#define  MICROPY_STACK_CHECK           (1)
#define  MICROPY_ENABLE_GC             (1)
#define  MICROPY_ENABLE_FINALISER      (MICROPY_VFS)
#define  MICROPY_HELPER_REPL           (1)
#define  MICROPY_REPL_AUTO_INDENT      (1)
#define  MICROPY_KBD_EXCEPTION         (1)
#define  MICROPY_PY_ASYNC_AWAIT        (0)
#define  MICROPY_PY_BUILTINS_BYTES_HEX (1)
#define  MICROPY_PY_BUILTINS_FILTER    (0)
#define  MICROPY_PY_BUILTINS_MIN_MAX   (0)
#define  MICROPY_PY_BUILTINS_PROPERTY  (0)
...
```

The *mpconfigport.h* file contains a list of C macros(Kernighan, B. W., Ritchie, D. M., The C programming language, 2nd ed., 1988) that act like a switch to enable and disable said modules.

An analysis yielded the following list of the supported modules in MicroPython's Zephyr port:

Listing 21: Zephyr port modules

```
binascii      hashlib
builins       machine
errno         math
gc            micropython
os            usys
```

```
socket        utime
sys           vfs
time          zephyr
zsensor
```

4.5.3 Comparing modules availability in two ports

The table below summarizes the difference in module support between NXP's and Zephyr MicroPython ports:

Table 1: Difference in module support of NXP's and Zephyr MicroPython ports

MicroPython Port to Zephyr	NXP's MicroPython implementation
	array
	asyncio
binascii	binascii
builtins	builtins
	cmath
	collections
	deflate
	dht
	ds18x20
errno	errno
	framebuf
gc	gc
hashlib	hashlib
	heapq
	io
	json
machine	machine
math	math
	mcx
micropython	micropython
	onewire
os	os
	platform
	random

(continued)

MicroPython Port to Zephyr	NXP's MicroPython implementation
	re
	select
	struct
socket	
sys	sys
time	time
	uasyncio
	uctypes
usys	usys
utime	utime
vfs	
zephyr	
zsensor	

While NXP's MicroPyton port offers more extensive set of supported modules, the MicroPython's Zephyr port offers core modules enough for many embedded applications.

4.6   Extending Zephyr Port functionality

4.6.1   Enabling and Disabling Modules

This subsection aims to bring two MicroPython ports closer together in modules support by adding modules to the Zephyr port. This will be achieved mainly through use *mpconfigport.h* file shown in the previous section.

As discussed in subsection 4.5.2 modules are added to the Zephyr port of MicroPython either via *mpconfigport.h* or *CMakeList.txt* files, which enable customization of the final build.

Thus, by switching some macro values in mpconfigport.h :

Listing 22: Extending Zephyr port modules support, Part 1

```
#define  MICROPY_PY_ARRAY            (1)
#define  MICROPY_PY_COLLECTIONS      (1)
#define  MICROPY_PY_IO               (1)
#define  MICROPY_PY_STRUCT           (1)
```

and rebuilding the port, the *array, collections, io* and *struct* modules become usable in the MicroPython's Zephyr port.

Some modules, namely *deflate, framebuf, heapq, json, platform, random, re, select, uctypes, asyncio*, are not available to configuration neither in *mpconfigport.h* nor *CMakeList.txt* files. Implementations of these modules are found in */extmod* directory of MicroPython source code, which is meant to host non-core modules implemented in C. In those implementations it is possible to find macros that enable them. By adding those macros to the *mpconfigport.h* file of Zephyr port:

Listing 23: Extending Zephyr port modules support, Part 2

```
#define  MICROPY_PY_DEFLATE       (1)
#define  MICROPY_PY_FRAMEBUF      (1)
#define  MICROPY_PY_HEAPQ         (1)
#define  MICROPY_PY_JSON          (1)
#define  MICROPY_PY_PLATFORM      (1)
#define  MICROPY_PY_RANDOM        (1)
#define  MICROPY_PY_RE            (1)
#define  MICROPY_PY_SELECT        (1)
#define  MICROPY_PY_UCTYPES       (1)
#define  MICROPY_PY_ASYNCIO       (1)
```

and rebuilding the port all the modules expect for *asyncio* become available to use in the Zephyr port of MicroPython. The issue with *asyncio* module lies in how it is registered in the build:

Listing 24: Asyncio module in MicroPython's Zephyr port

```
MP_REGISTER_MODULE(MP_QSTR___asyncio, mp_module_asyncio);
```

it's name thus is *_asyncio* and importing it by this name too makes it available in MicroPython's Zephyr port .

Finally, to add *cmath* support a board needs to have a floating point numbers capabilities (MicroPython, cmath – mathematical functions for complex numbers, 2025) which FRDM-MCXN947 has. The *cmath* modules is implemented in */py* MicroPython directory and is enabled with combination of the macros *MICROPY_PY_BUILTINS_FLOAT*, *MICROPY_PY_CMATH* and *MICROPY_PY_BUILTINS_COMPLEX*. The last macro is already defined in the *mpconfigport.h* file and only needs to be enabled, the first two need to be added and enabled:

Listing 25: Extending Zephyr port modules support, Part 3

```
#define  MICROPY_PY_BUILTINS_FLOAT    (1)
#define  MICROPY_PY_BUILTINS_COMPLEX  (1)
#define  MICROPY_PY_CMATH             (1)
```

and after rebuilding the Zephyr port those modules are too available in the port.

After the process of adding modules the difference in supported modules between two ports have narrowed and the Table 1 can be updated now to show off the progress:

Table 2: Difference in module support of NXP's and Zephyr MicroPython ports after adding modules to Zephyr port

MicroPython Port to Zephyr	NXP's MicroPython implementation
array	array
_asyncio	asyncio
binascii	binascii
builtins	builtins
cmath	cmath
collections	collections
deflate	deflate
	dht
	ds18x20
errno	errno
framebuf	framebuf
gc	gc
hashlib	hashlib
heapq	heapq
io	io
json	json
machine	machine
math	math
	mcx
micropython	micropython
	onewire
os	os
platform	platform
random	random
re	re
select	select
struct	struct

(continued)

MicroPython Port to Zephyr	NXP's MicroPython implementation
socket	
sys	sys
time	time
	uasyncio
uctypes	uctypes
usys	usys
utime	utime
vfs	
zephyr	
zsensor	

4.6.2 Community contributed features

In this subsection the *PWM* support will be added to the Zephyr port. Although the *PWM* is not officially supported in the upstream repository by the Zephyr MicroPython port, at the time of writing, it is available as a pull request that has not yet been merged by the maintainer. First, the branch with the *PWM* support will be downloaded and configured in a similar way to the previous subsection to bring support of the other modules. Next, an additional configuration will be made via *Kconfig* to make the *PWM* support available after building the MicroPython's Zephyr port.

First, the code with *PWM* support needs to be cloned from the contributor:

Listing 26: Obtaining MicroPython's Zephyr port version with PWM support

```
git  clone  https://github.com/Ayush1325/micropython.git
```

In the cloned project *Kconfig* file *frdm_mcxn947_mcxn947_cpu0.conf* for FRDM-MCXN947 configuration needs to be created in the *ports/zephyr/boards* directory containing :

Listing 27: Kconfig configuration for PWM support on FRDM-MCXN947 board

```
CONFIG_PWM=y
CONFIG_PWM_MCUX=y
```

Both *CONFIG_PWM*(Zephyr Project, CONFIG_PWM, 2023) and *CONFIG_PWM_MCUX*(Zephyr Project, CONFIG_PWM_MCUX, 2023) are Zephyr build flags to enable *PWM* and *mcux PWM driver* respectively.

After configuration, the port with *PWM* support can be build by passing to Zephyr's *west* the path to *ports/zephyr/* directory of newly configured MicroPython

version. Following the build and flash processes the *PWM* functionality becomes
enabled and can be tested via REPL environment on the *pwm0* device, defined in
FRDM-MCXN947 *Devicetree* configuration in the Zephyr source code:

Listing 28: Testing PWM in REPL environment of the Zephyr MicroPython port

```
from machine import PWM
pwm = PWM(("pwm0", 0), freq=500, duty_ns=200)
print(pwm)
print(pwm.duty_ns())
```

The program outputs the defined *PWM* object and the duty cycle, indicating that
the community feature was successfully added.

4.6.3   Implementing new modules

This subsection will touch on the process of implementing completely new modules
for MicroPython's Zephyr port by creating a simple demonstrative module.

First, the macro that will enable the new module after building and flashing
the MicroPython to FRDM-MCXN947 board is defined in the *mpconfigport.h* file:

Listing 29: Implementing a new module for the Zephyr port, Part 1

```
#define MICROPY_NEW_MODULE          (1)
```

Next, the module file is created and added to a list of included files from the
Zephyr port directory in the *CMakeList.txt* file:

Listing 30: Implementing a new module for the Zephyr port, Part 2

```
set(MICROPY_SOURCE_PORT
    ...
    mod_new_mod.c
    ...
)
list(TRANSFORM MICROPY_SOURCE_PORT PREPEND ${
   MICROPY_PORT_DIR}/)
```

This will add the *mod_new_mod.c* file to a list of sources under the variable *MI-
CROPY_SOURCE_PORT* and then prepend those sources with a path to the
Zephyr port directory from the *MICROPY_PORT_DIR* variable. Doing so will
ensure that the compiler can later find *mod_new_mod.c* module file and include it
in the final build.(Swidzinski, R., Modern CMake for C++, 2022)

The module implementation is as follows:

Listing 31: Implementing a new module for the Zephyr port, Part 3

```
#include "py/runtime.h"
```

```
#if MICROPY_NEW_MODULE

static mp_obj_t newmod_info(void) {
    mp_print_str(MP_PYTHON_PRINTER, "function that prints
        out information about the new module\n");
    return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_0(newmod_info_obj,
    newmod_info);

static const mp_rom_map_elem_t newmod_module_globals_table
    [] = {
    { MP_OBJ_NEW_QSTR(MP_QSTR___name__), MP_OBJ_NEW_QSTR(
        MP_QSTR_newmod) },
    { MP_OBJ_NEW_QSTR(MP_QSTR_newstr), MP_OBJ_NEW_QSTR(
        MP_QSTR_newstringvalue) },
    { MP_ROM_QSTR(MP_QSTR_info), MP_ROM_PTR(&
        newmod_info_obj) },
};
static MP_DEFINE_CONST_DICT(newmod_module_globals,
    newmod_module_globals_table);

const mp_obj_module_t newmod_module = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t *)&newmod_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_newmod, newmod_module);

#endif
```

#if MICROPY_NEW_MODULE ensures the module is active only when enabled in the *mpconfigport.h* file. A function *newmod_info* is defined that prints out text to MicroPython's standard output. All module objects are listed in *newmod_module_globals_table*, where a string constant *newstr* is also defined. The table is converted into a dictionary using *MP_DEFINE_CONST_DICT*, which is then added to the *newmod_module* object representing the new module. The module object is then registered as a module under the name *newmod*.(MicroPython, Implementing a Module, 2025)

After building and flashing the Zephyr port with the new module added, it becomes fully usable after importing.

4.7   Creating an HTTP server of FRDM-MCXN947 with MicroPython's Zephyr port

4.7.1   Introduction

One of the objectives of the thesis was to build a lightweight implementation of HTTP server that can run on the FRMD-MCXN947 development board that will utilize capabilities of MicorPython's Zephyr port. The server provides simple web interface that could be used in browser and provides board interaction, specifically interaction with *CTIMER* functionality of LED pins of the FRDM-MCXN947 board.

The implementation of the HTTP server is built on top of a custom MicroPython's Zephyr port. The build also includes additional hardware configuration with *Devicetree* and Zephyr's network capabilities configuration with *Kconfig*. Both configurations are defined in port directory and are compiled as *overlay* configurations.

4.7.2   Devicetree configuration

This subsection describes the necessary *Devicetree* configuration to enable *PWM* output through *CTIMER* function available to the pins *P0_10, P0_27, P1_2* via *pin multiplexing* technology(NXP semiconductors, MCX Nx4x Reference Manual, 2025), which is available on the FRDM-MCXN947 board. The *Devicetree* configuration is as follows:

Listing 32: Configuring PWM support through CTIMER functionality on FRDM-MCXN947 pins P0_10, P0_27 and P1_2 with Devicetree

```
#include <zephyr/dt-bindings/pwm/pwm.h>
#include <nxp/mcx/MCXN947VDF-pinctrl.h>
/ {
    aliases {
        red-pwm-led = &red_pwm_led;
        green-pwm-led = &gree_pwm_led;
        blue-pwm-led = &blue_pwm_led;
    };

    pwmleds {
        compatible = "pwm-leds";
        red_pwm_led: pwm_led_0 {
            pwms = <&ctimer0 0 255 PWM_POLARITY_NORMAL>;
            label = "red led pwm";
        };

        gree_pwm_led: pwm_led_1 {
            pwms = <&ctimer0 3 255 PWM_POLARITY_NORMAL>;
            label = "green led pwm";
```

```
        };

        blue_pwm_led: pwm_led_3 {
            pwms = <&ctimer1 0 255 PWM_POLARITY_NORMAL>;
            label = "green led pwm";
        };
    };
};

&ctimer0 {
    compatible = "nxp,ctimer-pwm";
    #pwm-cells = <3>;
    status = "okay";
    pinctrl-0 = <&pinmux_ctimer0_pwm_red_green>;
  pinctrl-names = "default";
};

&ctimer1 {
    compatible = "nxp,ctimer-pwm";
    #pwm-cells = <3>;
    status = "okay";
    pinctrl-0 = <&pinmux_ctimer0_pwm_blue>;
  pinctrl-names = "default";
};

&pinctrl {
    pinmux_ctimer0_pwm_red_green:
        pinmux_ctimer0_pwm_red_green {
          group0 {
              pinmux = <CT0_MAT0_PIO0_10>,
                  <CT0_MAT3_PIO0_27>;
              slew-rate = "fast";
              drive-strength = "low";
          };
    };
    pinmux_ctimer0_pwm_blue: pinmux_ctimer0_pwm_blue {
          group0 {
              pinmux = <CT1_MAT0_PIO1_2>;
              slew-rate = "fast";
              drive-strength = "low";
          };
    };
};
```

In *&pinctrl* node, which controls *pin multiplexing* and configuration parameters(Ramírez-Sánchez, Earl O., A Practical Start with Zephyr RTOS, 2024), the two labels *pinmux_ctimer0_pwm_blue* and *pinmux_ctimer0_pwm_red_green* are used to configure *CT0_MAT0_PIO0_10, CT0_MAT3_PIO0_27* and *CT1_MAT0_PIO1_2 pin multiplexing functions.* Those functions are mapped to the hardware addresses in *MCXN947VDF-pinctrl.h* file that is included at the top of the configuration.

The labels are then passed to the lists of pin configuration nodes *pinctrl-0* in their corresponding peripheral nodes *&ctimer0* and *&ctimer1.* The peripherals *ctimer0* and *ctimer1* are configured to use *nxp,ctimer- pwm* driver for *PWM* support. The peripherals are then enabled by being set to status *okay.*

In the *pwmleds* node the *PWM* channels are set to the desired frequency and polarity. In *aliases* node aliases are created for the pin functions.

After this configuration the MicroPython's Zephyr port built is ready to use *CTIMER* function in *PWM* mode for FRDM-MCXN947 pins *P0_10, P0_27, P1_2.*

4.7.3   Kconfig configuration

This subsection describes the neccessary *Kconfig* configuration to enable *PWM* and networking use and their various components. The *Kconfig* configuration is as follows:

Listing 33: Configuring PWM and networking support in MicroPython's Zephyr port with Kconfig

```
CONFIG_NETWORKING=y
CONFIG_NET_SOCKETS=y
CONFIG_NET_SOCKETS_POLL_MAX=4

CONFIG_NET_IPV4=y

CONFIG_NET_TCP=y

CONFIG_NET_L2_ETHERNET=y

CONFIG_PWM=y
CONFIG_PWM_MCUX=y

CONFIG_NET_BUF_FIXED_DATA_SIZE=y
CONFIG_NET_BUF_DATA_SIZE=2048
```

The flags *CONFIG_NETWORKING, CONFIG_NET_SOCKETS, CONFIG_NET_SOCKETS_POLL_MAX* are Zephyr flags used to enable network and allow basic socket configuration.   The flags *CONFIG_NET_IPV4* and *CONFIG_NET_TCP* are enabling *IPv4* and *TCP* support respectively.   The

CONFIG_NET_L2_ETHERNET enables *Ethernet* support, it is needed because the board is connected to the local network via Ethernet cable. *CONFIG_PWM* and *CONFIG_PWM_MCUX* are for *PWM* support and to enable the right driver. The *CONFIG_NET_BUF_FIXED_DATA_SIZE* and *CONFIG_NET_BUF_DATA_SIZE* flags are used for increasing default data buffer size for data transmission of the network.

After this configuration the After this configuration the MicroPython's Zephyr port built is ready to utilize Zephyr's networking layer and have *PWM* support enabled.

4.7.4 HTTP server setup

After building and flashing new MicroPython's Zephyr port build with configuration from the previous subsections the HTTP server can be setup on the FRDM-MCXN947 board. The prerequisite to running the server is to have the board connected to the local network via *Ethernet* cable. Zephyr handles *DHCP* communication with the router to obtain an IP address, when *CONFIG_NET_DHCPV4* flag is enabled(*CONFIG_NET_DHCPV4* is enabled in default MicroPython's Zephyr port *Kconfig*). The server code is as follows:

Listing 34: HTTP server implementation with MicroPython for MicroPython's Zehpyr port runing on FRDM-MCXN947 board

```
import socket
import errno
from machine import PWM

red_pwm = PWM(("ctimer0", 0), freq=1000, duty_ns=0)
green_pwm = PWM(("ctimer0", 3), freq=1000, duty_ns=0)
blue_pwm = PWM(("ctimer1", 0), freq=1000, duty_ns=0)

CONTENT = b"""\
HTTP/1.0 200 OK
Connection: close
Content-Type: text/html

<html>
<head>
    <title>FRDM-MCXN947</title>
    <script>
        function setColor(event) {
            let color = event.target.value;
            fetch('/set_color?hex=${encodeURIComponent(
                color)}')
```

```
                    . then ( response => console . log ( 'Color set to
                        ${ color } ') )
                    . catch ( error => console . error ( "Error
                        setting color :", error ) ) ;
        }

        document . addEventListener ( "DOMContentLoaded " ,
            function ( ) {
                document . getElementById ( "colorPicker ") .
                    addEventListener ( "input ", setColor ) ;
        }) ;
    </script >
</head>
<body style ="background−color : #333; color : #ffffff ;">
    <h1>FRDM–MCXN947</h1>

    <input type ="color " id ="colorPicker ">
</body>
</html>

"""
def set_color ( hex_color ) :
    r , g , b = int ( hex_color [0:2] , 16) , int ( hex_color [2:4] ,
        16) , int ( hex_color [4:6] , 16)
    max_duty_cycle = 500_000
    r = int (( r / 255) ∗ max_duty_cycle )
    g = int (( g / 255) ∗ max_duty_cycle )
    b = int (( b / 255) ∗ max_duty_cycle )
    red_pwm . duty_ns ( r )
    green_pwm . duty_ns ( g )
    blue_pwm . duty_ns ( b )

def main ( micropython_optimize=False ) :
    print ( " starting ")
    s = socket . socket ( socket . AF_INET , socket .SOCK_STREAM)

    addr = ( "0.0.0.0" , 8080)

    s . setsockopt ( socket .SOL_SOCKET, socket .SO_REUSEADDR, 1)
    try :
        s . bind ( addr )
    except OSError as e :
```

```
        print ("Error :", e. errno , errno . errorcode . get (e .
            errno , "Unknown Error"))
        return
s. listen (5)
print ("Listening , connect your browser to http://<
    this_host >:8080/")

while True:
    res = s.accept ()
    client_sock = res [0]
    client_addr = res [1]
    print ("Client address :", client_addr )
    print ("Client socket :", client_sock )

    if not micropython_optimize :
        client_stream = client_sock . makefile ("rwb")
    else :
        client_stream = client_sock

    print ("Request :")
    req = client_stream . readline ()
    print (req)

    request_line = req . decode (). split ()
    if len ( request_line ) > 1:
        path = request_line [1]
        print ("request line :", request_line )
    else :
        path = "/"

    while True:
        h = client_stream . readline ()
        if h == b"" or h == b"\r\n":
            break

    if path . startswith ("/ set_color "):
        hex_value = path . split ("? hex=%23")[1]
        set_color ( hex_value )

        client_stream . write (CONTENT)
    else :
        client_stream . write (CONTENT)
```

```
        client_stream.close()
        if not micropython_optimize:
            client_sock.close()
        print()

main()
```

First, three *PWM* objects are created, each corresponding to one of the three *CTIMER* channels for controlling red,green and blue LED lights. Each *PWM* object is configured with a 1 kHz frequency and a duty cycle of 0 nanoseconds, which initially turns off the LED lights. The variable *CONTENT* contains the *HTML* code that is served to the client on connection. The *HTML* has simple *JavaScript* for sending a request back to the server. The *set_color* function parses *HEX* color string and updates each *PWM* light with a corresponding configuration to give an appearance of the right color. The *main* function establishes *TCP* socket on port 8080 and listens for an incoming connection. When the client client establishes connection, the server reads the request. If the path begins with */set_color*, the server reads the hexadecimal color of the query string and passes it to the *set_color* function to set the color of LED lights. After reconfiguring *PWM* objects or when the client request does not have */set_color* string in the path, the server serves the *HTML* content.

5 References

Zephyr Project, Introduction [online], 2024 Available from:
    https://docs.zephyrproject.org/latest/introduction/index.html.

Zephyr Project, West (Zephyr's meta-tool) [online], 2024 Available from:
    https://docs.zephyrproject.org/latest/develop/west/index.html.

Zephyr Project, Configuration System (Kconfig) [online], 2022 Available from:
    https://docs.zephyrproject.org/latest/build/kconfig/index.html.

Zephyr Project, FRDM-MCXN947 [online], 2025
    https://docs.zephyrproject.org/latest/boards/nxp/frdm_mcxn947/doc/index.html.

Zephyr Project, Zephyr Security Overview [online], 2024 Available from:
    https://docs.zephyrproject.org/latest/security/security-overview.html.

Zephyr Project, Additional Zephyr extension commands [online], 2025 Available
    from: https://docs.zephyrproject.org/latest/develop/west/zephyr-cmds.html.

Zephyr Project, Building, Flashing and Debugging, [online], 2024 Available from:
    https://docs.zephyrproject.org/latest/develop/west/build-flash-debug.html.

Zephyr Project, CONFIG_PWM, [online], 2023 Available from:
    https://docs.zephyrproject.org/2.7.5/reference/kconfig/CONFIG_PWM.html.

Zephyr Project, CONFIG_PWM_MCUX, [online], 2023 Available from:
    https://docs.zephyrproject.org/2.7.5/reference/kconfig/CONFIG_PWM_MCUX.html.

devicetree.org, Devicetree Specification Release v0.4, 2023.

Ramírez-Sánchez, Earl O., A Practical Start with Zephyr RTOS, 2024.

NXP semiconductors, MCX Nx4x Reference Manual, 2025 MCXNX4XRM.

NXP semiconductors, UM12018 FRDM-MCXN947 Board User Manual, 2024.

NXP semiconductors, Getting Started with FRDM-MCXN947 [online] ,2024
    Available from: https://www.nxp.com/document/guide/getting-started-with-
    frdm-mcxn947:GS-FRDM-MCXNXX?section=build-and-run.

NXP semiconductors, FRDM Development Board for MCX N94/N54 MCUs
    [online], 2025 Available from: https://www.nxp.com/design/design-
    center/development-boards-and-designs/FRDM-MCXN947.

NXP semiconductors, Quick start guide for MicroPython on FRDM-MCXN947
    board [online], 2024 Available from:
    https://community.nxp.com/t5/NXP-Tech-Blog/Quick-start-guide-for-

MicroPython-on-FRDM-MCXN947-board/ba-p/1799949.

NXP semiconductors, LinkServer for Microcontrollers [online], 2025
Available from: https://www.nxp.com/design/design-
center/software/development-software/mcuxpresso-software-and-tools-
/linkserver-for-microcontrollers:LINKERSERVER.

Nicholas H. Tollervey *Programming with MicroPython: embedded
programming with microcontrollers and Python.* O'Reilly Media, Inc., 2017.
ISBN 978-1-491-97273-1.

MicroPython, MicroPython downloads [online], 2025 Available from:
https://micropython.org/download/.

MicroPython, cmath – mathematical functions for complex numbers [online], 2025
Available from: https://docs.micropython.org/en/latest/library/cmath.html.

MicroPython, Implementing a Module [online], 2025 Available from:
https://docs.micropython.org/en/latest/develop/library.html.

Jayantilal, S. H., Interfacing of AT command based HC-05 serial Bluetooth module
with minicom in Linux, International Journal for Scientific Research and
Development, 2014; 2(3): 329–332.

Kernighan, B. W., Ritchie, D. M., The C programming language, 2nd edition,
Englewood Cliffs: Prentice Hall, 1988. ISBN 0-13-110362-8.

Swidzinski, R., Modern CMake for C++: Discover a better approach to building,
testing, and packaging your software, Birmingham: Packt Publishing Ltd,
2022. ISBN 9781803239729.

Free Software Foundation, GNU Screen [online], 2016 Available from:
https://www.gnu.org/software/screen/.