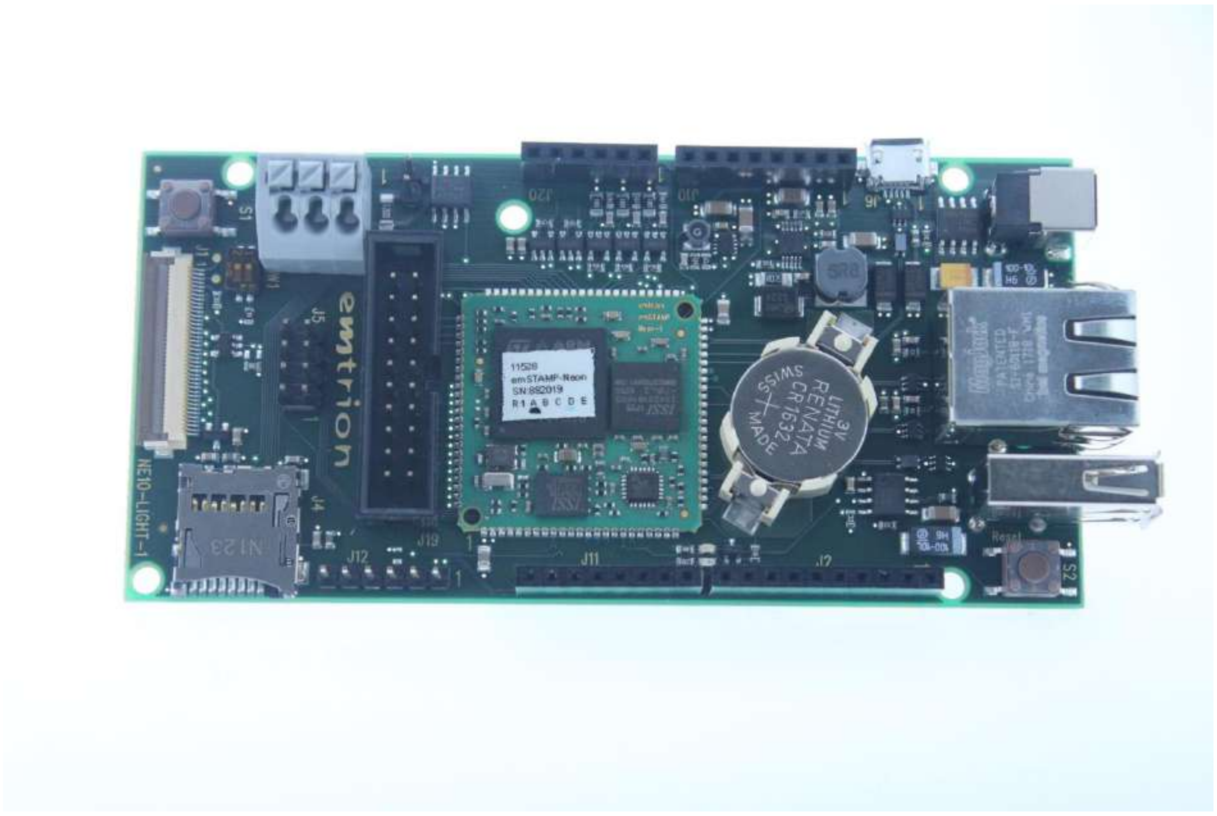
emSTAMP Neon Zephyr Developer Kit

Software Manual

Rev01 / 26.10.2023



emtrion GmbH

Revision: **01/ 26.10.2023**

Rev	Date/Signature	Changes
1	26.10.2023/Pa	Initial release

Table of contents

1 Introduction

Welcome to emSTAMP-Neon developer kit documentation. This manual will give you a startup software guideline for our developer kit. It will describe how to use the different free software to program your developer kit.

It is assumed that users of emtrion developer kits are already familiar with software development. Programming knowledge is out of the scope of this document. emtrion will gladly assist you in acquiring this knowledge. If you are interested in training courses or getting support, please contact the emtrion sales department.

The examples in this manual are demonstrated on specific hardware but if not mentioned otherwise they all work on all supported emtrion devices.

Please refer to the “Hardware Description” of emSTAMP-Neon available on the emtrion support website (http://support.emtrion.de) for more detailed information on the capability of the product.

1.1 Zephyr Project

Zephyr is a real-time operating system (RTOS) designed specifically for resource-constrained embedded systems and IoT devices. It focuses on providing a lightweight and efficient kernel for small-scale, low-power devices.

Zephyr is designed to be extremely lightweight and can run on devices with minimal memory and storage requirements. It is well-suited for resource-constrained embedded systems.

Zephyr uses a microkernel architecture, which means only essential services are placed in kernel space and functionality is placed in user space. Whereas Linux uses a monolithic kernel architecture where most of OS functionality is placed in kernel space. This way zephyr improves stability and security.

Zephyr is a real-time operating system, which is specifically designed for real-time tasks, whereas Linux is not a real-time OS.

1.2 Features

1) **Extensive suite of Kernel services**

   Zephyr offers familiar services for development:

   - Multithreading services
   - Interrupt services
   - Memory allocation services
   - Inter-thread synchronization services
   - Inter-thread data passing
   - Power management services

2) **Multiple scheduling algorithms**
   Zephyr provides a set of thread scheduling choices: for example, cooperative and primitive scheduling, Earliest deadline first, Time slicing, Multiple queueing strategy etc.
3) **Highly configurable / modular for flexibility**
4) **Cross Architecture**
5) **Memory Protection**
6) **Compile time resource definition**
7) **Optimized Device Driver Model**
8) **Device tree support**
9) **Bluetooth low energy 5.0 support**
10) **User friendly**
11) **Linux, Mac and Windows development**
12) **Native POSIX support**

2 Hardware requirement

2.1 JTAG debugger/programmer

In this manual, the in-circuit debugger/programmer used is the ST-LINK/V2 (https://www.st.com/en/development-tools/st-link-v2.html).

3 Workstation Software installation

Before starting you need to prepare your workstation. To build an image that runs on the target, you need to install the following set of free software available online on the Zephyr Project official site (https:/zephyrproject.org/).

3.1 Zephyr Project Development Environment

Zephyr Project is a small real-time operating system (RTOS) for connected, resource-constrained and embedded devices supporting multiple architectures and was released under the Apache License 2.0. Zephyr includes a kernel, and all components and libraries, device drivers, protocol stacks, file systems, and firmware updates, needed to develop full application software. To set up command line Zephyr development environment on Ubuntu.

Step 1: Select and update the OS

```
sudo apt update

sudo apt upgrade
```

Step 2: Install dependencies

Minimum required version for dependencies

Tool	Min. Version
CMake	3.20.5
Python	3.8
Device tree compiler	1.4.6

1. If Ubuntu version is older than 22.04, then this step is important to add an extra repository

```
wget https://apt.kitware.com/kitware-archive.sh

sudo bash kitware-archive.sh
```

2. Use apt to install dependencies

```
sudo apt install
```

3. Verify the versions of the main dependencies installed in the system

```
cmake --version

python3 --version

dtc --version
```

Step 3: Get zephyr and install python dependencies

1. Use apt to install Python venv package

```
sudo apt install python3-venv
```

2. Create new virtual environment:

```
python3 -m venv ~/zephyrproject/.venv
```

3. Activate the virtual environment:

```
source ~/zephyrproject/.venv/bin/activate
```

4. Install west:

```
pip install west
```

5. Get the zephyr source code:

```
west init ~/zephyrproject

cd ~/zephyrproject

west update
```

6. Export a Zephyr CMake package:

```
west zephyr-export
```

7. Install zephyr's requirements.txt file with pip:

```
pip install -r ~/zephyrproject/zephyr/scripts/requirements.txt
```

Step 4: Install Zephyr SDK

1. Download and verify the zephyr SDK bundle

```
cd ~

wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.1/zephyr-sdk-
0.16.1_linux-x86_64.tar.xz

wget -O - https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.1/sha256.sum
| shasum --check --ignore-missing
```

2. Extract the zephyr SDK bundle

```
tar xvf zephyr-sdk-0.16.1_linux-x86_64.tar.xz
```

3. Run the Zephyr SDK bundle setup script

```
cd zephyr-sdk-0.16.1

./setup.sh
```

4. Install udev rules

```
sudo cp ~/zephyr-sdk-0.16.1/sysroots/x86_64-pokysdk-linux/usr/share/openocd/contrib/60-
openocd.rules /etc/udev/rules.d

sudo udevadm control --reload
```

To set up Zephyr development environment for Windows or Mac please follow the below link (https://docs.zephyrproject.org/latest/develop/getting_started/index.html).

→This way zephyr development environment is set up.

3.2 Zephyr Project for Development

There are some key aspects of using zephyr for development. It provides kernel and RTOS features (threads, timers, etc.). It supports a wide range of architectures and boards. It allows you to develop applications using C or C++ programming languages. It places a strong emphasis on security and safety.

It integrates well with popular IDEs such as VS code, Eclipse, and others.

3.3 STM32 ST-LINK utility

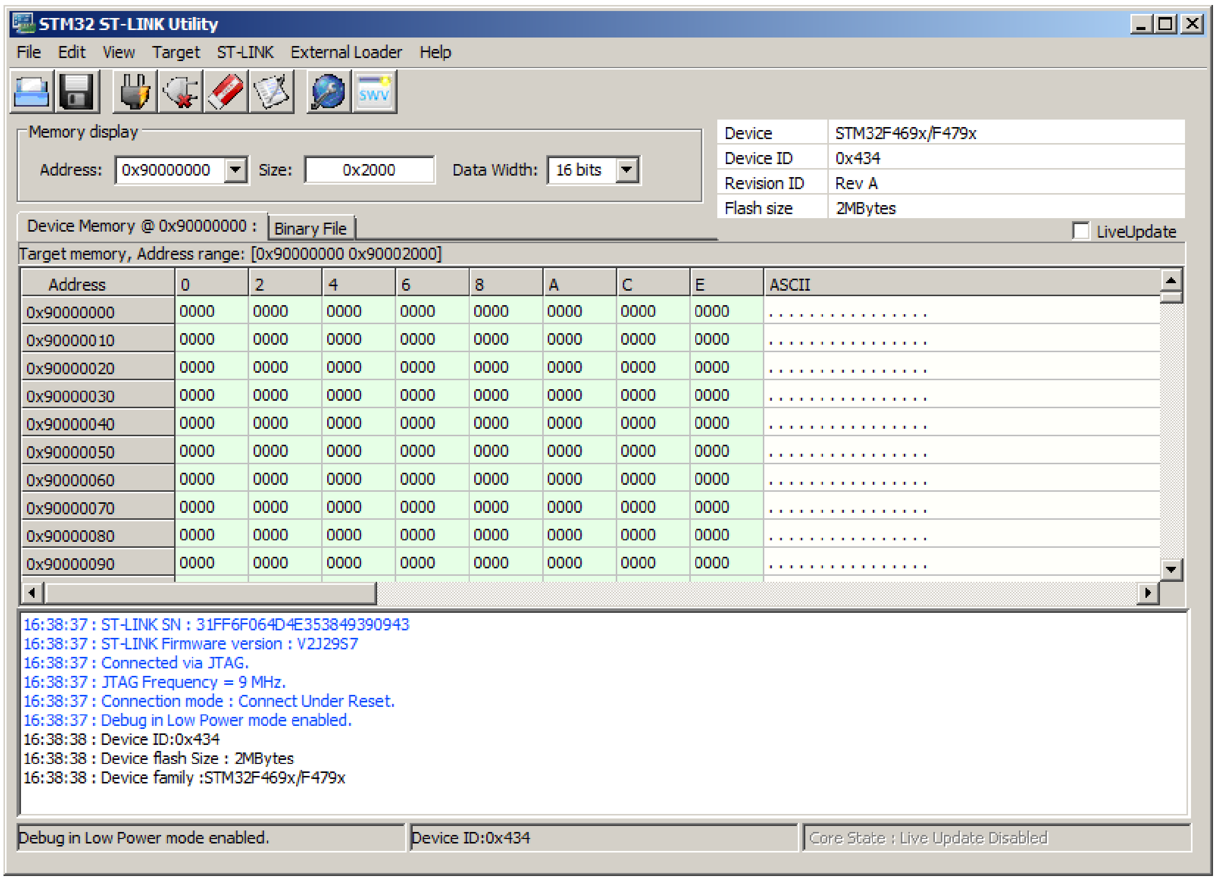
STM32 ST-LINK Utility is a full-featured software interface for programming STM32 microcontrollers (https://www.st.com/en/development-tools/stsw-link004.html)

The tool offers a wide range of features to program STM32 internal memories (Flash, RAM, OTP and others), and external memories to verify the programming content (checksum, verify during and after programming, compare with file) and to automate STM32 programming.STM32 ST-LINK Utility is delivered as a graphical user interface (GUI) with a command line interface (CLI).

4 Testing your in-circuit Debugger/programmer on your emStamp-Neon

It is strongly advised to test the communication between your workstation and your target using the ST-LINK utility tool before starting any debugging and/or programming with SW4STM32.

The output of a good communication should look like this:



STM32 ST-LINK Utility

File Edit View Target ST-LINK External Loader Help

Memory display

Address: 0x90000000 Size: 0x2000 Data Width: 16 bits

Device	STM32F469x/F479x
Device ID	0x434
Revision ID	Rev A
Flash size	2MBytes

Device Memory @ 0x90000000 : Binary File

LiveUpdate

Target memory, Address range: [0x90000000 0x90002000]

Address	0	2	4	6	8	A	C	E	ASCII
0x90000000	0000	0000	0000	0000	0000	0000	0000	0000	....
0x90000010	0000	0000	0000	0000	0000	0000	0000	0000	....
0x90000020	0000	0000	0000	0000	0000	0000	0000	0000	....
0x90000030	0000	0000	0000	0000	0000	0000	0000	0000	....
0x90000040	0000	0000	0000	0000	0000	0000	0000	0000	....
0x90000050	0000	0000	0000	0000	0000	0000	0000	0000	....
0x90000060	0000	0000	0000	0000	0000	0000	0000	0000	....
0x90000070	0000	0000	0000	0000	0000	0000	0000	0000	....
0x90000080	0000	0000	0000	0000	0000	0000	0000	0000	....
0x90000090	0000	0000	0000	0000	0000	0000	0000	0000	....

```
16:38:37 : ST-LINK SN : 31FF6F064D4E353849390943
16:38:37 : ST-LINK Firmware version : V2J29S7
16:38:37 : Connected via JTAG.
16:38:37 : JTAG Frequency = 9 MHz.
16:38:37 : Connection mode : Connect Under Reset.
16:38:37 : Debug in Low Power mode enabled.
16:38:38 : Device ID:0x434
16:38:38 : Device flash Size : 2MBytes
16:38:38 : Device family :STM32F469x/F479x
```

Debug in Low Power mode enabled. | Device ID:0x434 | Core State : Live Update Disabled

5 Information on the downloaded Zephyr repository folder

- ➔ Once your Zephyr environment is set up, now you are all set for building and flashing the project but before that firstly, we need information about the downloaded Zephyr repository folder what does it contain and what happens if we make changes.
- ➔ Zephyr project repository folder contains 4 folders bootloader, modules, tools, and zephyr.

mcuboot: It contains MCUboot Project, which is an open-source bootloader for controllers. It provides a secure and flexible way to update the firmware on microcontrollers.

zephyr: It is the heart of the Zephyr project. It contains RTOA source code, configurations and associated files. In this folder, Zephyr RTOS itself is maintained and developed. It includes kernel, device drivers, libraries and components that make up Zephyr RTOS. Developers working on the Zephyr project interact with this folder to configure and build their applications.

tools: This folder contains various tools and scripts that are useful for working with the Zephyr Project. It houses a collection of scripts and utilities that help developers with tasks like building Zephyr applications, managing dependencies, and performing various development tasks. For example, it includes the Zephyr Software Development Kit (SDK) and tools for flashing firmware onto target devices.

modules: It contains additional libraries and modules that are not part of the core Zephyr RTOS code base. These additional modules can be used to extend the functionality of your Zephyr-based application. They include components like networking stacks, Bluetooth profiles, and various middleware libraries. Developers can choose to include these modules in their projects as needed.

6 Important steps before starting development with Neon board and Zephyr Project

Once the Zephyr environment is set up, now Zephyr repository folder contains only files that come with the Zephyr Project. It does not contain supportive or working files for the neon board.

- ➔ To work with the Neon board, we have to add necessary device tree files of the neon board in the Zephyr repository.
- ➔ You can download the necessary device tree files of the Neon board from the emtrion site.
- ➔ After downloading that folder, you can copy the whole folder named "**stm32f769i_neon**" and paste it at the given path.
  Path for pasting the folder: **zephyrproject/zephyr/boards/arm/**

- ➔ After pasting it at the given path location now we are all set to do development with Neon board.
- ➔ To check that it's working or not you can try out the blinky project example on the Neon board.

  Note: If you are trying out applications that gives serial output do not forget to connect the UART connector with Neon board.

7 Working with Zephyr Project to build and flash your application

Emtrion is providing the configuration file that gives you the possibility to load the entire pin muxing, clock configuration and middleware of the MCU STM32F479NIHx used in the emStamp-Neon. Once you have done everything according to the getting started guide you are all set to develop and load your application on your neon board.

Step 1: To start working open the terminal and check if the virtual environment is activated or not. if not then activate it. Command to activate the virtual environment:

```
source ~/zephyrproject/.venv/bin/activate
```

Step 2: Go to the Zephyr directory:

```
cd ~/zephyrproject/zephyr
```

Step 3: You can build any application of your choice but here we are taking the example of a blinky application. command to build an application:

```
west build -p always -b stm32f769i_neon samples/basic/blinky
```

→Once you build the project, you will see one folder created with "**build**" name in the zephyr folder of the zephyr project repository. The contents of the build folder may vary depending on specific project configuration, target hardware, and build options. Build folder contains folders like app, CMakeFiles, Kconfig, modules, etc., and files like CMakeCache.txt, build.ninja ...etc.

→If you want to work with another controller then **"stm32f769i_neon"** must be replaced with the other controller's device tree file name. If you want to work with other applications than blinky application then the path will be changed.

Step 4: Once it's built successfully then flash it to board. command to flash application:

```
west flash
```

Now your blinky application is directly flashed to the neon board and you can see that red LED on the neon board started blinking.

8 Working with Zephyr Project to develop your application

There are three types of Zephyr applications based on their location.

1. **Repository application**: located in Zephyr source code repository
2. **Workspace application:** located in the workspace outside the Zephyr repository
3. **Freestanding application:** located outside of workspace

There are many sample applications provided by the zephyr.

Here is the path for sample applications: ~/zephyrproject/zephyr/samples/

Creating new Application:

Step 1: Create a basic directory

```
mkdir app
```

Step 2: Create your source code files.

```
cd app

mkdir src
```

Step 3: Place your application source code in a subdirectory named src.

Step 4: Create a file named CMakeLists.txt in the app directory with the following contents.

```
cmake_minimum_required(VERSION 3.20.0)

find_package(Zephyr)

project(my_zephyr_app)

target_sources(app PRIVATE src/main.c))
```

Note:

- The cmake_minimum_required() call is required by CMake. CMake will error out if its version is older than either the version in your CMakeLists.txt or the version number in the Zephyr package.
- find_package(Zephyr) pulls in the Zephyr build system, which creates a CMake target named app.
- project(my_zephyr_app) defines your application's CMake project.
- target_sources(app PRIVATE src/main.c) is to add your source file to the app target.

Step 5: Create at least one Kconfig fragment for your application (usually named prj.conf) and set Kconfig option values needed by your application there. See Kconfig Configuration. If no Kconfig options need to be set, create an empty file.

Step 6: Configure any device tree overlays needed by your application, usually in a file named app.overlay.

Step 7: Set up any other files you may need and then your new application is created.

Here are the files given in a simple Zephyr application structure:

```
<app>
├── CMakeLists.txt
├── app.overlay
├── prj.conf
├── VERSION
└── src
    └── main.c
```

- **CMakeLists.txt:** This file tells the build system where to find the other application files, and links the application directory with Zephyr's CMake build system. This link provides features supported by Zephyr's build system, such as board-specific configuration files, the ability to run and debug compiled binaries on real or emulated hardware, and more.
- **app.overlay:** This is a device tree overlay file that specifies application-specific changes that should be applied to the base device tree for any board you build for. The purpose of device tree overlays is usually to configure something about the hardware used by the application.
- **prj.conf:** This is a Kconfig fragment that specifies application-specific values for one or more Kconfig options. These application settings are merged with other settings to produce the final configuration. The purpose of Kconfig fragments is usually to configure the software features used by the application.
- **VERSION:** A text file that contains several version information fields. These fields let you manage the lifecycle of the application and automate providing the application version when signing application images. (Not compulsory to have)
- **main.c:** A source code file. Applications typically contain source files written in C, C++, or assembly language. The Zephyr convention is to place them in a subdirectory of <app> named src.

These files are necessary for creating and modifying an application.

To create a new application or to modify it you can follow the instructions given on the site(https://docs.zephyrproject.org/latest/develop/application/index.html)

9 Working with Zephyr Project to develop a device tree

For adding extra features to the controller board as well as to make changes inside the device tree of the board.

here is path for device tree of neon board:
~/zephyrproject/zephyr/boards/arm/stm32f769i_neon/stm32f769i_neon.dts

Steps to develop the device tree:

1. **Locate the Device Tree Files:** typically located in 'zephyr/dts/' directory.
2. **Choose the appropriate Device Tree File:** According to hardware you're using.
3. **Edit the Device Tree File:** Open .dts or .dtsi file and make your desired changes to it.
4. **Device Tree Syntax:** Familiarize yourself with device tree syntax and conventions in the device tree. Otherwise, it may lead to errors.
5. **Use Device Tree Overlays(optional)**
6. **Rebuild the Zephyr Application**

7. **Flash and Test:** Flash your updated firmware and test your changes.
8. **Document your changes:** It's necessary to document the device tree modifications to ensure that others can understand your changes and facilitate troubleshooting.

Note: Please don't make modification directly inside the device tree. It's safe to create separate overlay files and make changes.

Here to make changes in the device tree you must need inside knowledge of the neon board. That's why it's recommended to use the emSBC-Neon hardware manual(https://www.emtrion.de/de/products/emsbc-neonm7-mit-st-stm32f769ni.html).

Set Devicetree Overlays

The CMake variable **DTC_OVERLAY_FILE** contains a space- or semicolon-separated list of overlay files to use. If **DTC_OVERLAY_FILE** specifies multiple files, they are included in that order by the C preprocessor. A file in a Zephyr module can be referred to by escaping the Zephyr module dir variable like \${ZEPHYR_<module>_MODULE_DIR}/<path-to>/dts.overlay when setting the **DTC_OVERLAY_FILE** variable.

You can set **DTC_OVERLAY_FILE** to contain exactly the files you want to use.

If you don't set the **DTC_OVERLAY_FILE** then the build system will follow these steps, looking for device tree overlay files:

1. If the file `boards/<BOARD>.overlay` exists, it will be used.
2. If the current board has multiple revisions and `boards/<BOARD>_<revision>.overlay` exists, it will be used. This file is used in addition to the `boards/<BOARD>.overlay` if both exist.
3. If one or more file exists, then build systems stop looking and uses those files.
4. Otherwise, if `<BOARD>.overlay` exists, then it will be used and the build system will stop looking for more files.
5. Otherwise if `app.overlay` exists, it will be used.

Overlays can override the node properties in multiple ways. If your board contains this node:

```
/ {
        soc {
                serial0: serial@40002000 {
                        status = "okay";
                        current-speed = <115200>;
                        /* ... */
                };
        };
};
```

➔ These are the ways to override the `current-speed` value in an overlay:

```
/* Option 1 */
&serial0 {
    current-speed = <9600>;
};
/* Option 2 */
&{/soc/serial@40002000} {
    current-speed = <9600>;
};
```

➔ This way, an overlay has been created for making changes inside the device tree.