DWM3001CDK Developer Manual

Qorvo

Release QM33SDK-1.0.2

Contents

1 Revision History

Version	Date	Comment
DW3_QM33_SDK_1.0.0	2024-11-06	• Restructured documentation into individual document for each supported development kit. • Updated *Board Connections* chapter with information on communication interfaces. • Updated building and flashing description: – Merged *Flashing SDK Firmware* and *Building SDK Firmware* chapters into one - *Building and flashing*, – Added list of required tools, – Added VS Code integration description and standalone CMake build description, – Removed Segger Embedded Studio build description. • Updated CLI chapter: – Updated overall description, – Removed CLI commands: DECA$, UWBCFG, VERSION, TXPOWER, ANTENNA, ANTTXA, ANTRXA, XTALTRIM, PDOAOFF, TCFM, TCWM, LISTENER2, STSKEYIV, – Added CLI commands: LISTENER, LCFG, SETAPP, GETOTP, CALKEY, LISTCAL, – Updated CLI commands: SAVE, RESTORE, DIAG, SAVE, UART, INITF, RESPF, HELP. • Updated tasks description in *SDK Runtime* chapter. • Updated *OTP Memory Map* chapter: – Content moved out of *Customizing the firmware* to a separate chapter, – Added customer registers description. • Updated *Licenses in this SDK* chapter. • Updated *UCI - UWB Command Interface* chapter: – Chapter content removed, – Added references to external documents. • Completed *SDK SW Architecture* chapter. • Completed *Customizing the firmware* chapter. • Added *Calibration and Configuration* chapter. • Added *Qorvo One TWR GUI* chapter. • Removed *List of Tables* and *List of Figures*.
DW3_QM33_SDK_0.1.0	2022-09-27	• Initial version

2 Introduction

2.1 Overview

The SDK package is designed to support developers with comprehensive resources, including a Developer Manual and a Quick Start Guide, to help working with the QM33 SDK. Below is a structured breakdown of the package contents and associated documentation:

Quick Start Guide

The Quick Start Guide includes essential guidelines for getting started with the SDK package.

Developer Manual

The Developer Manual contains comprehensive information about the SDK package.

Tools

- The UWB Qorvo Tools are provided as a set of Python scripts to demonstrate key features of the UCI.
- The Qorvo One GUI is provided as a visualization application for ranging capabilities of the device.

Binary and Source Code

The package includes a precompiled binary for the target and source code, enabling developers to start developing their solutions immediately.

Additional Documentation

The SDK package also includes extensive documentation for various components and APIs, as listed below:

- **DW3xxx/QM33xxx Device Driver Application Programming Interface (API) Guide** - refer to `Documentation/QM33XXX_DW3XXX_Software_API_Guide.pdf`
- **UWB FiRa Protocol Documentation** - refer to `Documentation/uwb-fira-protocol.pdf`
- **UWB L1 API Documentation** - refer to `Documentation/uwb-l1-api.pdf`
- **UWB-Stack L1 Configuration Documentation** - refer to `Documentation/uwb-l1-configuration.pdf`
- **UWB QHAL API Documentation** - refer to `Documentation/uwb-qhal-api.pdf`
- **UWB QOSAL API Documentation** - refer to `Documentation/uwb-qosal-api.pdf`
- **UWB Qplatform API Documentation** - refer to `Documentation/uwb-qplatform-api.pdf`
- **UWB UCI Message API Documentation** - refer to `Documentation/uwb-uci-messages-api.pdf`
- **UWB UWBMAC API Documentation** - refer to `Documentation/uwb-uwbmac-api.pdf`

Note: For anything related to hardware design and custom board creation using Qorvo UWB transceivers, it is recommended to read the documentation provided in APH301 Hardware Design Guide for DW3000 and QM33100 Series ICs[1]. This document will provide you with the necessary guidelines and instructions for designing and manufacturing your own custom hardware.

[1] https://www.qorvo.com/products/p/QM33120W#documents

For better understanding the Angle of Arrival (AoA) feature, it is advisable to refer to the documentation in APH511 UWB AoA Antenna Fundamentals[2]. These resources will provide you with the necessary answers to commonly asked questions on antennas to implement or use the AoA functionality.

By following the guidelines and recommendations provided in these documents, you will be able to get the best performance from your own hardware.

2.2 Package Content

The following table shows SDK package content once unzipped.

Table 2.1: SDK package content

Folder name	Folder description
Binaries	Executable files to flash the board (UCI and CLI).
Documentation	Documentation of libraries, SDK User Manual, SDK Quick Start Guide, release notes.
Firmware	SDK source code.
Tools	Qorvo One GUI application and UWB Qorvo Tools.

Warning: *On Windows*, when extracting the zipped package inside the `Firmware` directory, it is recommended to enable Long Path support. Otherwise, the following error may occur: "Error 0x80010135: Path too long".

To resolve this issue, you can either:

- Run following PowerShell command from a terminal window with elevated privileges:

```
New-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Control\FileSystem" -Name
↪"LongPathsEnabled" -Value 1 -PropertyType DWORD -Force
```

- Change the following registry key: `Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem\LongPathsEnabled` in the Registry Editor opened with elevated privileges.

Alternatively, you can extract the zipped directory at the root directory instead of inside the `Firmware` directory.

For more information, refer to the following sections:

- *Building SDK Firmware* - Instructions on building the firmware.
- *SDK SW Architecture* - Details of the source code.
- *Qorvo One TWR GUI* - Information on the Qorvo One GUI.
- *UWB Qorvo Tools* - Information on Ultra-Wideband Qorvo Tools.

[2] https://www.qorvo.com/products/p/QM33120W#documents

3 Board Connections

3.1 DWM3001CDK

Connect the two micro-USB connectors available on the board as in *DWM3001CDK connections*

- J9 (interface MCU): used for flashing/debugging via J-Link OB and UART communication with MCU through a virtual COM port.
- J20 (nRF USB): used for UART/USB communication with MCU.

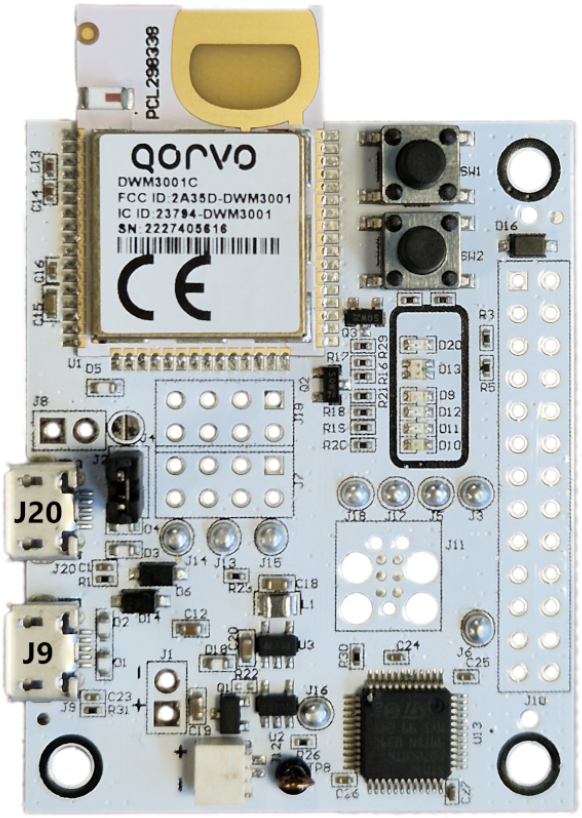

Fig. 3.1: DWM3001CDK connections

3.1.1 Communication Interfaces on DWM3001CDK

In SDK for DWM3001CDK, two different communication interfaces can be utilized for the communication:

- UART over USB (USB CDC ACM) allows for the communication with the MCU over USB connection, enabled by default.
- UART allows for the communication with the MCU through the UART pins, disabled by default.

4 Building and flashing

4.1 Overview

This chapter offers detailed instructions on building firmware and flashing development kit boards.

For development, we recommend utilizing the integrated building and flashing functionalities provided by *Visual Studio Code* environment. Nevertheless, the SDK is built upon CMake and Make, ensuring compatibility with various IDEs. If you prefer to use an alternate development environment, please follow *Standalone building and flashing* for guidance on executing a standalone build.

Warning: Please be aware that the *Setup* subchapter is essential and must be completed prior to proceeding with either the integrated or standalone build approach.

4.2 Setup

4.2.1 Required tools

Before you can build and run the examples, you need to have the following tools installed on your system. Please ensure you install each one before proceeding.

Note: VS Code tool is not mandatory. However, please keep in mind that provided automated tasks for building, flashing and debugging are only supported on VS Code.

4.2.1.1 Windows

- **ARM Toolchain**:
  - Download **arm-none-eabi-gcc-10.3-2021.10-win32** from the ARM download page[3].
  - Install it inside: **C:\GnuToolsArmForEmbedded\gcc-arm-none-eabi-10.3-2021.10\bin**.
  - Add `C:\\GnuToolsArmForEmbedded\\gcc-arm-none-eabi-10.3-2021.10\\bin` to your system/user environment variables.
- **CMake**:
  - Download CMake from the CMake download page[4]. The required version is higher than 3.23.
  - Install CMake to a directory of your choice, referred to as `<cmake_path>` in these instructions.
  - Add `<cmake_path>/bin` to your system/user environment variables.

[3] https://developer.arm.com/downloads/-/gnu-rm/10-3-2021-10
[4] https://cmake.org/download/

- **Python**:
  - Download Python from the Python download page[5]. Recommended version is 3.10.
  - Install Python and make sure to select the option “Add Python to PATH” during the installation.
- **MinGW and make**:
  - MinGW can be downloaded from the MinGW download page[6].
  - Install MinGW in the directory of your choice.
  - Using the MinGW Installation Manager, install `mingw32-base` to a directory of your choice, referred to as `<mingw_path>` in these instructions.
  - Create a copy of `<mingw_path>/bin/mingw32-make.exe` and name it `<mingw_path>/bin/make.exe`.
  - Add `<mingw_path>/bin` to your system/user environment variables.
- **SEGGER J-Link**:
  - Install J-Link Software and Documentation Pack from the Segger downloads page[7]. Recommended version is higher than V7.96j.
- **VS Code**:
  - Install VS Code from VS code download page[8].

4.2.1.2 Linux

- **ARM Toolchain**:
  - Create an install directory:

```
mkdir /opt/gcc
```

  - Go to the install directory:

```
cd /opt/gcc
```

  - Download **gcc-arm-none-eabi-10.3-2021.10-x86_64-linux** from the ARM page[9].
  - Extract the tarball:

```
tar -xvf gcc-arm-none-eabi-10.3-2021.10-x86_64-linux.tar.bz2
```

  - Remove the tarball:

```
rm gcc-arm-none-eabi-10.3-2021.10-x86_64-linux.tar.bz2
```

  - Open **~/.bashrc** and add this line `export PATH="/opt/gcc/gcc-arm-none-eabi-10.3-2021.10/bin:${PATH}"`.
- **Make**:
  - Install Make:

```
sudo apt-get install build-essential
```

- **CMake**:

[5] https://www.python.org/downloads/
[6] https://sourceforge.net/projects/mingw/
[7] https://www.segger.com/downloads/jlink/
[8] https://code.visualstudio.com/Download
[9] https://developer.arm.com/-/media/Files/downloads/gnu-rm/10.3-2021.10/gcc-arm-none-eabi-10.3-2021.10-x86_64-linux.tar.bz2

- Download CMake from the CMake github[10]. The required version is higher than 3.23.
- Create a cmake directory in usr/bin:

```
sudo mkdir /usr/bin/cmake
```

- Execute command:

```
sudo cmake-<cmake_version>-Linux-x86_64.sh --skip-license --prefix=/usr/bin/cmake
```

- Open **~/.bashrc** and add this line `export PATH="/usr/bin/cmake/bin:${PATH}"`.

- **Python**:
  - Install Python. Recommended version is 3.10:

```
sudo apt-get install -y python3 python3-pip
```

- **SEGGER J-Link**:
  - Install J-Link Software and Documentation Pack from the Segger downloads page[11]. Recommended version is higher than V7.96j.
- **VS Code**:
  - Install VS Code from the VS code download page[12].

4.2.2 Setup environment

This step is applicable to both Windows and Linux operating systems. Please execute the following commands in your terminal (on Windows please use PowerShell), within the root directory of the project.

1. Create a virtual environment:

```
python -m venv .venv
```

2. Activate the virtual environment:
   - On Linux

```
source .venv/bin/activate
```

   - On Windows

```
.\.venv\Scripts\Activate.ps1
```

3. Install all requirements:

```
pip install -r requirements.txt
```

[10] https://github.com/Kitware/CMake/releases/
[11] https://www.segger.com/downloads/jlink/
[12] https://code.visualstudio.com/Download

4.2.3 Hardware tools

Development kit board is equipped with Segger J-Link OB (on-board programmer), no additional hardware tool is needed to flash the device.

4.3 Visual Studio Code

SDK provides support to build, flash and debug firmware directly from VS Code. Below you can find brief description of required setup and possible functionalities.

If you wish to use different IDE/code editor, please refer to *Standalone building and flashing*.

4.3.1 Setup

4.3.1.1 Open workspace

To make use of all VS Code features please remember to open project as a workspace:

1. Pick *File* → *Open Workspace from File...* from the menu on top.

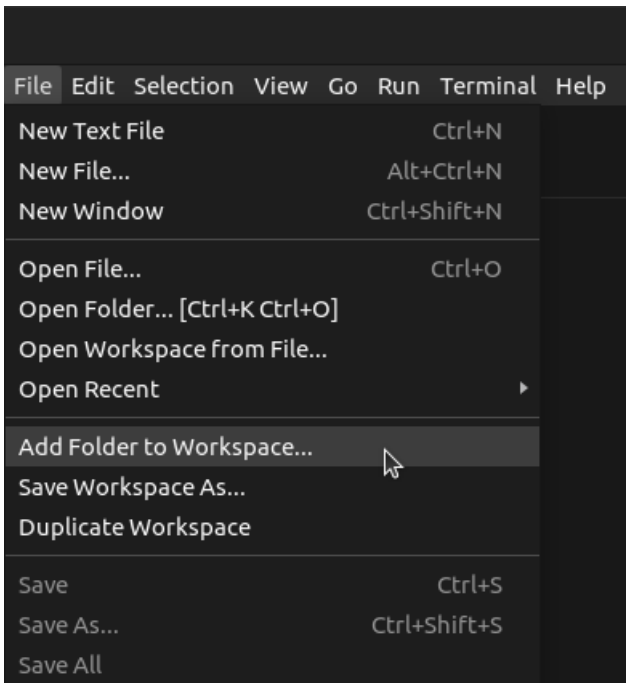

Fig. 4.1: VS Code: Open workspace from file.

2. Find and open *DW3_QM33_SDK.code-workspace*.

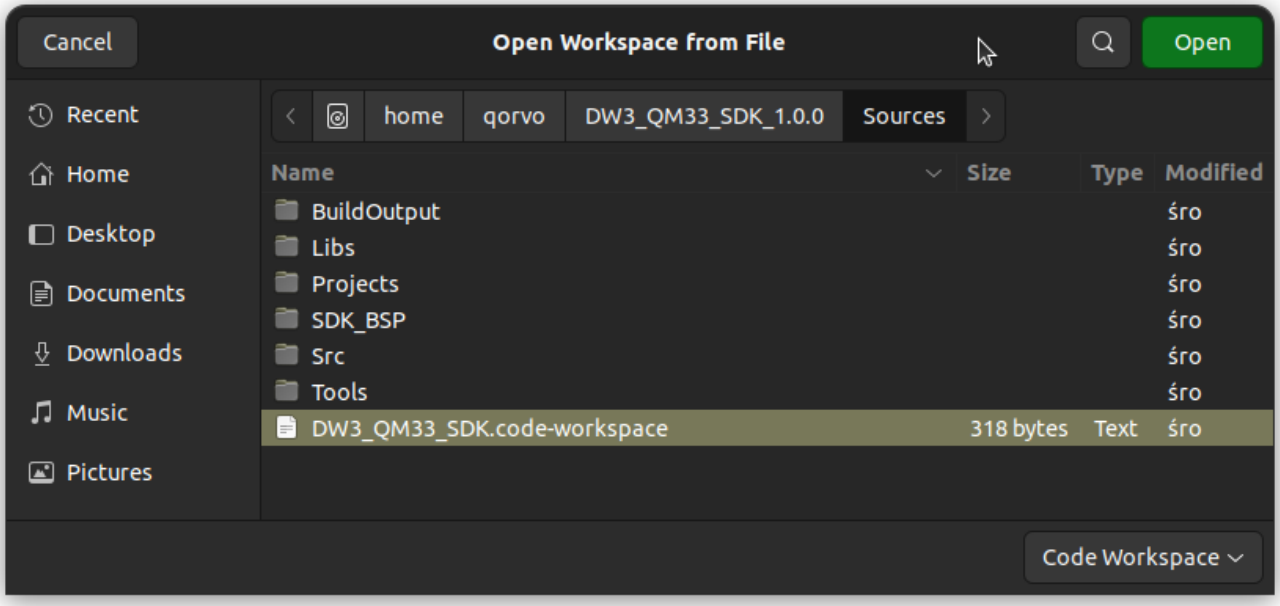

Fig. 4.2: VS Code: Find workspace file.

4.3.1.2 Install extensions

To build and run the firmware directly from VS Code, please install all the recommended extensions.

1. Navigate to the Extensions tab located on the left side of the window, or use the keyboard shortcut `Ctrl+Shift+X`.

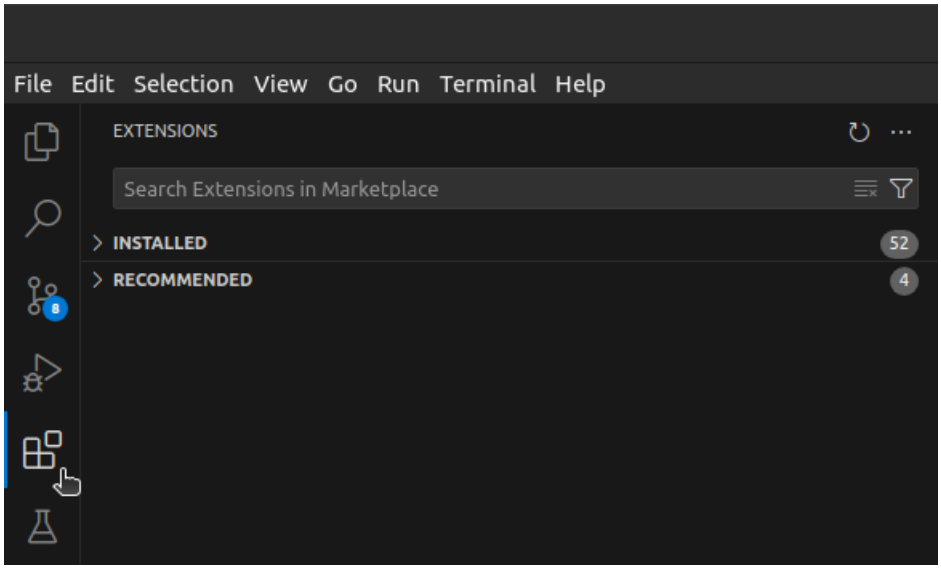

Fig. 4.3: VS Code: Extensions tab.

2. Click on the **Filter extensions** button (next to search field).

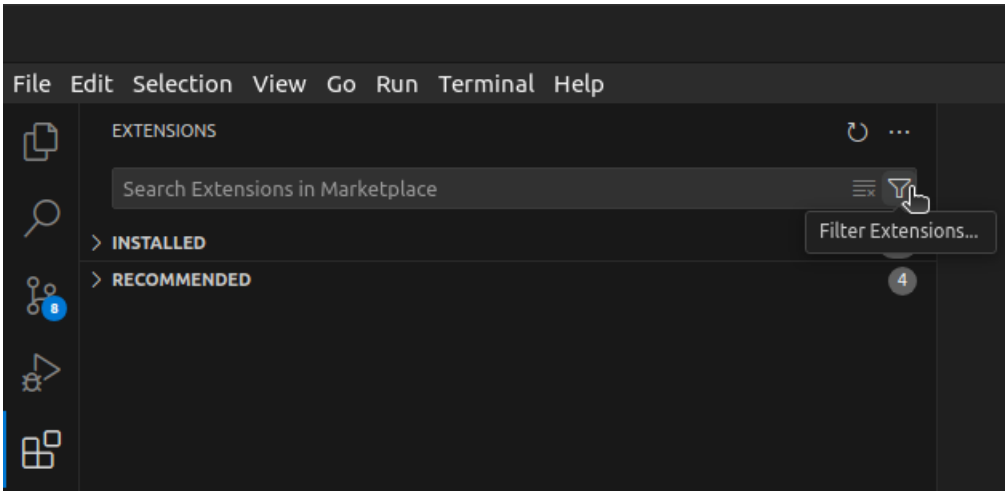

Fig. 4.4: VS Code: Filter extensions.

3. Pick the **Recommended** category.

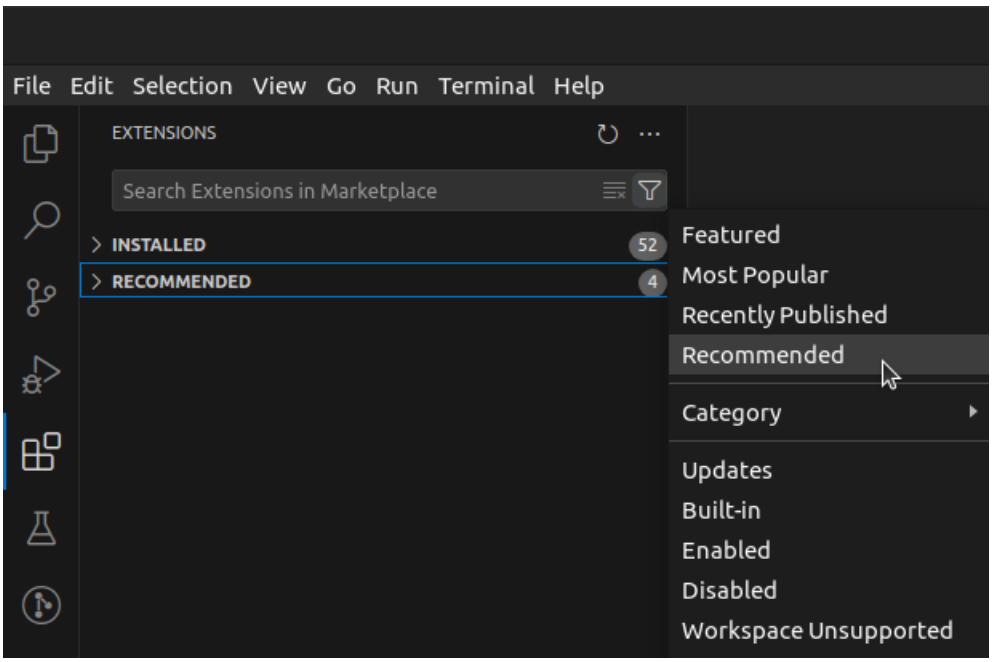

Fig. 4.5: VS Code: Filter recommended extensions.

4. Click on the cloud icon located just below to install all recommended extensions.

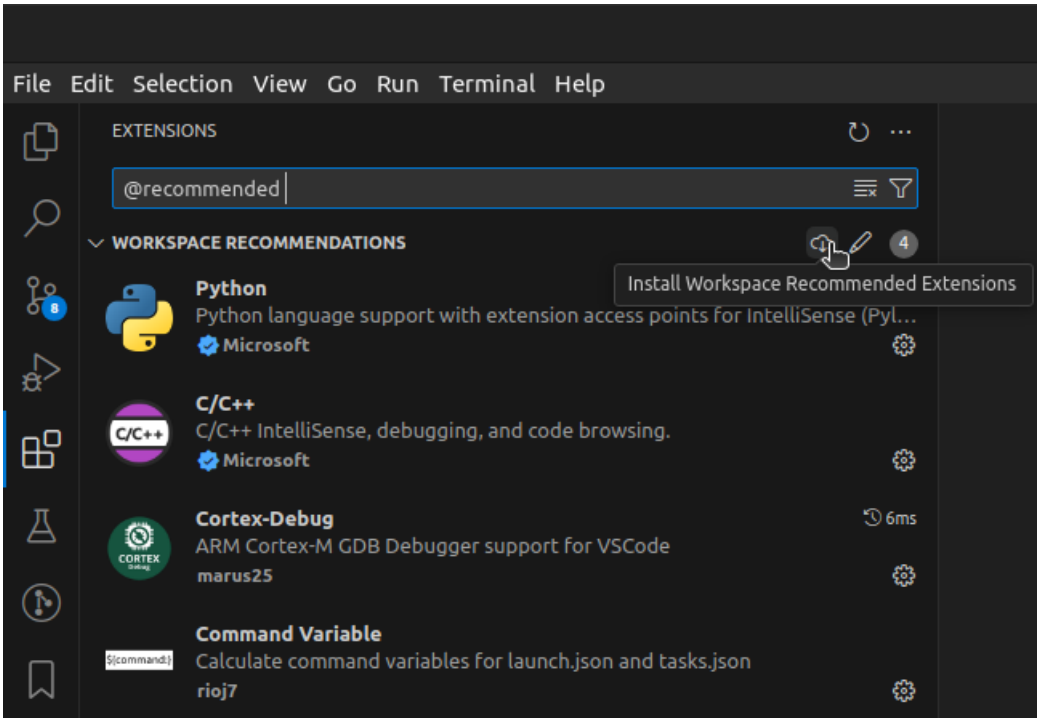

Fig. 4.6: VS Code: Install all extensions.

If you wish to install extensions manually, here you can find the list of recommended extensions:

- **ms-vscode.cpptools** (C language support)
- **marus25.cortex-debug** (debugging firmware on target)
- **rioj7.command-variable** (automated tasks for building and debugging)
- **ms-python.python** (Python language support)

Note: We encourage you to use the latest version of VS Code and its extensions. However, if you encounter issues, please revert to the specific extension versions:

- ms-vscode.cpptools **v1.20.5**
- marus25.cortex-debug **v1.12.1**
- rioj7.command-variable **v1.63.0**
- ms-python.python **v2024.6.0**

For compatibility with these extensions, use Visual Studio Code version **1.89.1**.

4.3.1.3 Other

VS Code automatically finds and picks Python interpreter from virtual environment. However, if your global settings were changed for other projects, please make sure to pick the proper interpreter.

1. Open **Command Palette** (*View* → *Command Palette...*) or use the keyboard shortcut `Ctrl+Shift+P`.

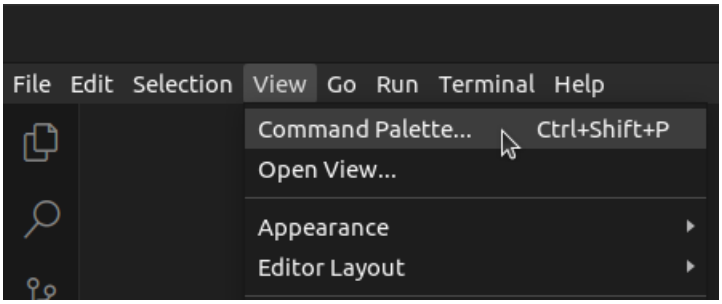

Fig. 4.7: VS Code: Open command palette.

2. Write `Python: Select interpreter`.

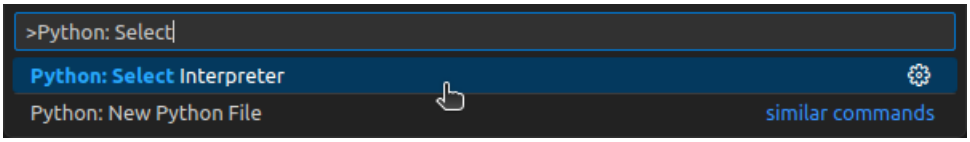

Fig. 4.8: VS Code: Choose Python interpreter.

3. Choose path to your local env, e.g. *./.venv/bin/python*.

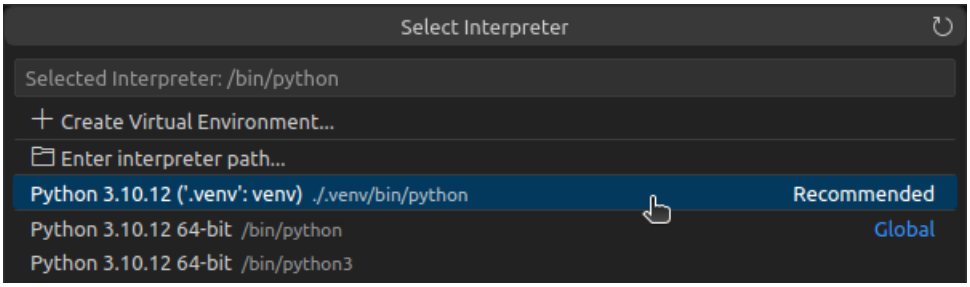

Fig. 4.9: VS Code: Pick venv interpreter.

Warning: On Windows, the recommended and working Terminal Default Profile is PowerShell. Git Bash and other may not be working properly because of differences in path resolution.

1. Open **Command Palette** (*View* → *Command Palette...*) or use the keyboard shortcut `Ctrl+Shift+P`
2. Write `Terminal: Select Default Profile` and choose **PowerShell**.
3. Choose **PowerShell** from the list.

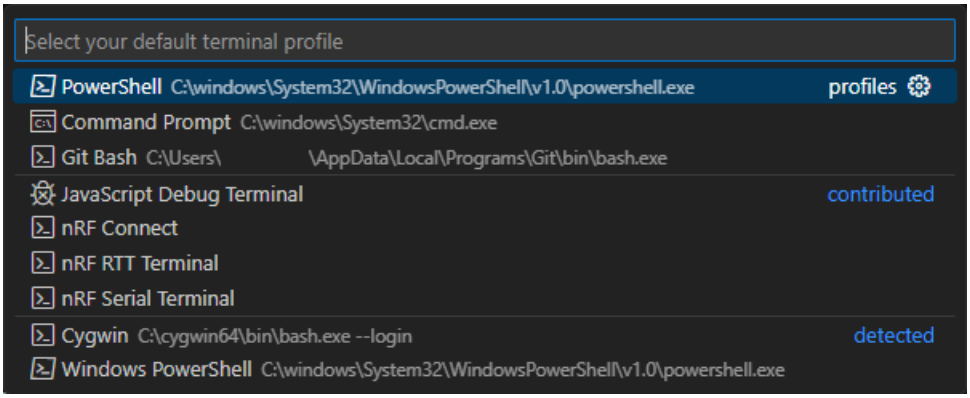

Fig. 4.10: VS Code: Pick PowerShell on Windows.

4.3.2 Tasks

VS Code built-in tasks and launch configuration can be utilized to build, flash and debug the firmware. You can find short description of available tasks below:

Build:

- **Build firmware** - build the firmware (compile only changed files).
- **Build clean firmware** - rebuild the firmware (delete output files and compile everything again).
- **Flash target** - flash target with recently built firmware and reset the target.
- **Build & flash target** - Build firmware + Flash target.

Launch:

- **Debug firmware** - runs GDB sessions with target.
- **Build & debug firmware** - builds firmware if needed and runs GDB sessions with target.

To provide a short list of tasks, we utilize **Command Variable** extension. This tool enables the passing of arguments between tasks and retains those arguments even after restarting VS Code.

4.3.2.1 Choose configuration

You can choose persistent build configuration which will be saved and used for all other tasks.

1. Open the **Run Build Task** dialog (*Terminal* → *Run Build Task...*) or use the keyboard shortcut `Ctrl+Shift+B`.

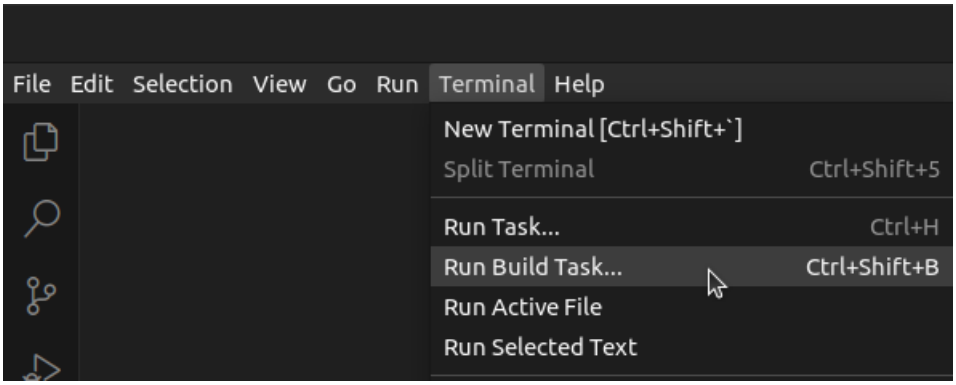

Fig. 4.11: VS Code: Open build tasks.

2. Pick **Choose configuration** and VS Code will prompt about possible configurations.

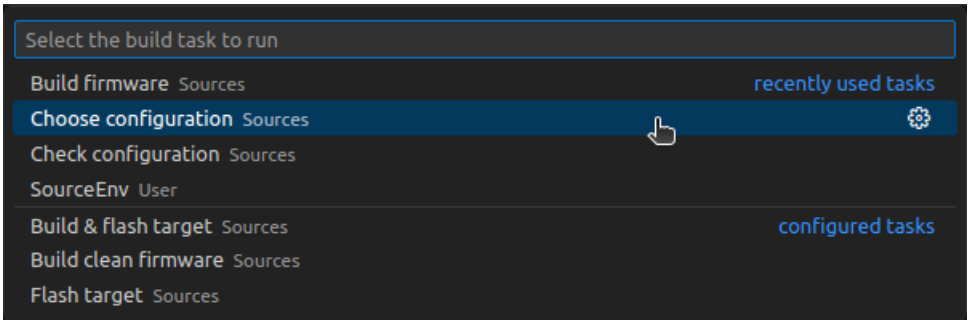

Fig. 4.12: VS Code: Choose configuration.

Fig. 4.13: VS Code: Choose build configuration.

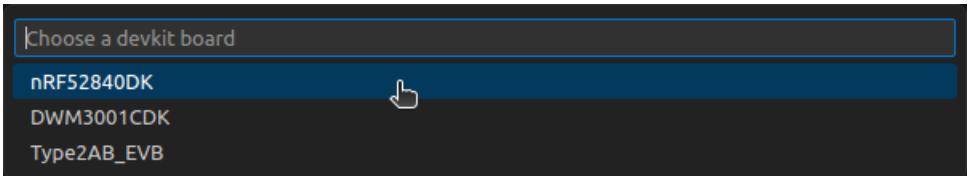

Fig. 4.14: VS Code: Choose board configuration.

4.3.2.2 Build firmware

1. Open the Run Build Task dialog (*Terminal* → *Run Build Task...*) or use the keyboard shortcut `Ctrl+Shift+B`.
2. To build a target you can use **Build firmware**.

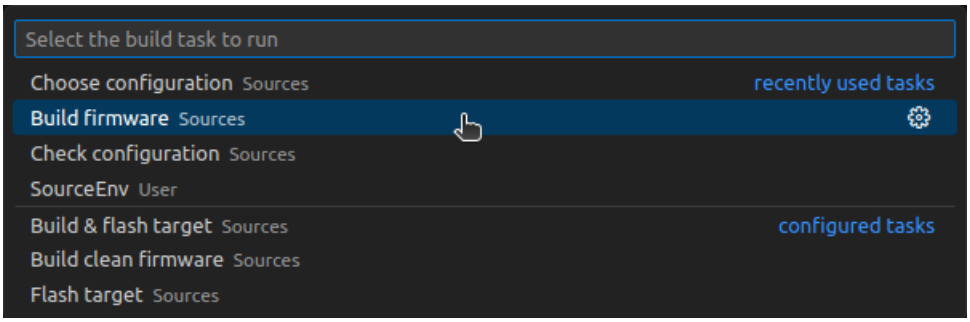

Fig. 4.15: VS Code: Pick build firmware.

3. New terminal tab will appear, after compilation completes you can see following message.

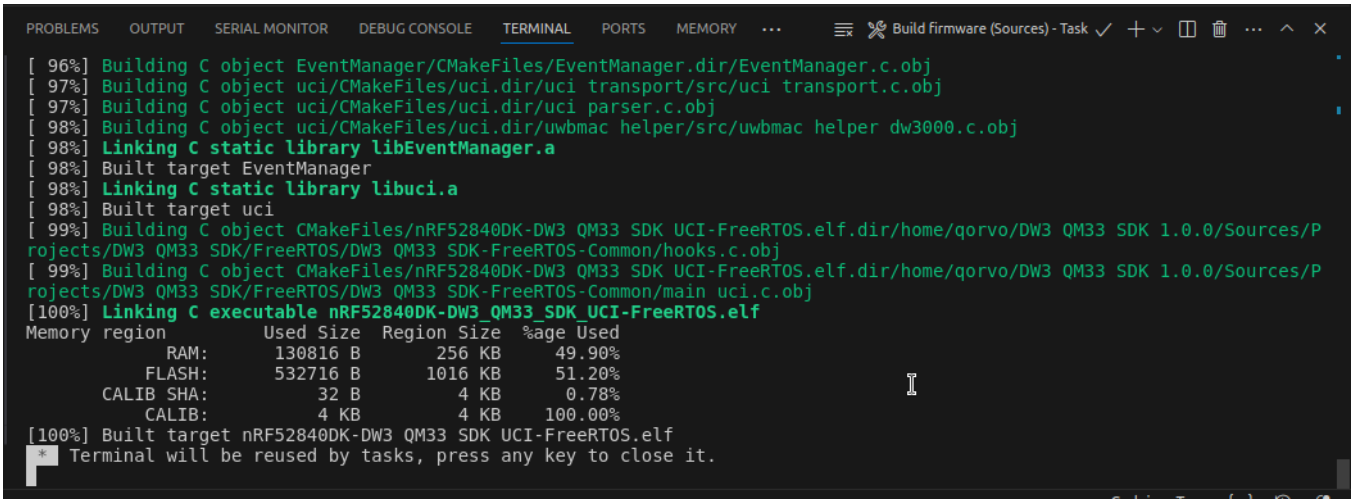

Fig. 4.16: VS Code: Firmware build successfully.

4. Alternatively, you can pick **Build clean firmware** to perform build from scratch.

4.3.2.3 Flash firmware

1. Make sure the firmware has been already built.
2. Open the Run Build Task dialog (*Terminal* → *Run Build Task...*) or use the keyboard shortcut `Ctrl+Shift+B`.
3. To flash the target, pick **Flash target**.

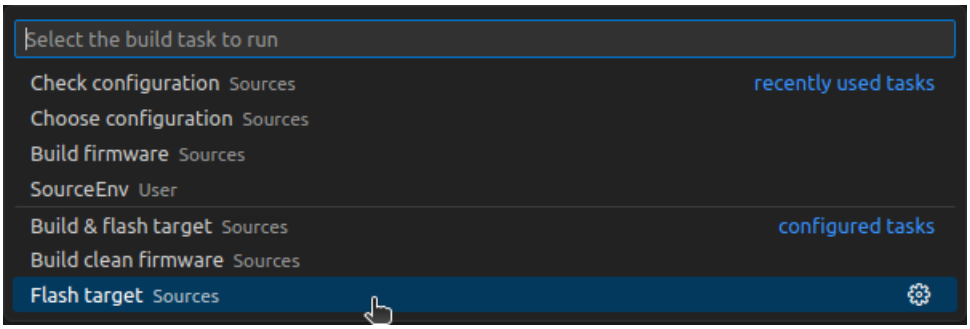

Fig. 4.17: VS Code: Pick Flash target.

4. If you have more than one development kit connected to your PC, you will be prompted to choose which one should be flashed.

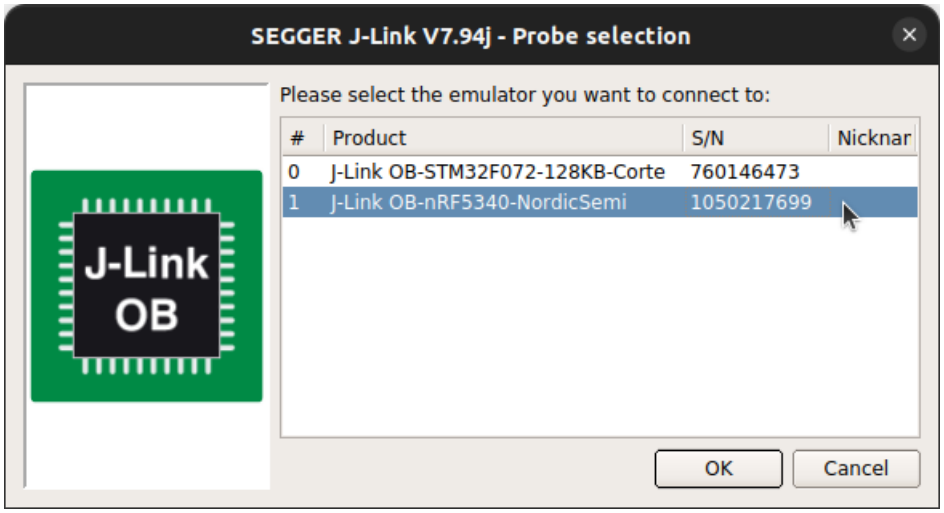

Fig. 4.18: VS Code: Pick proper J-Link probe.

5. After this, the flashing progress bar will show up.

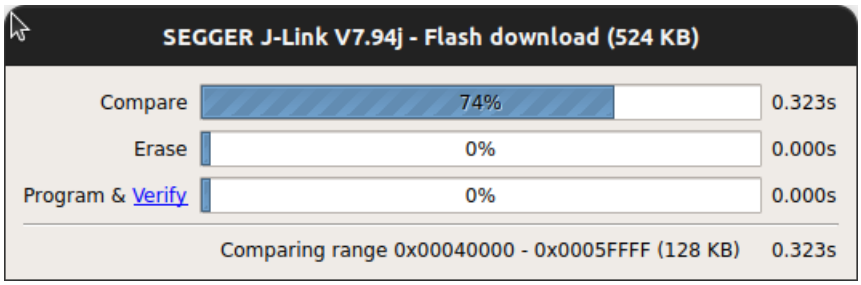

Fig. 4.19: VS Code: Flashing progress window.

6. When flashing completed successfully, you will see following message in a dedicated terminal tab. Firmware will now start execution on the device.

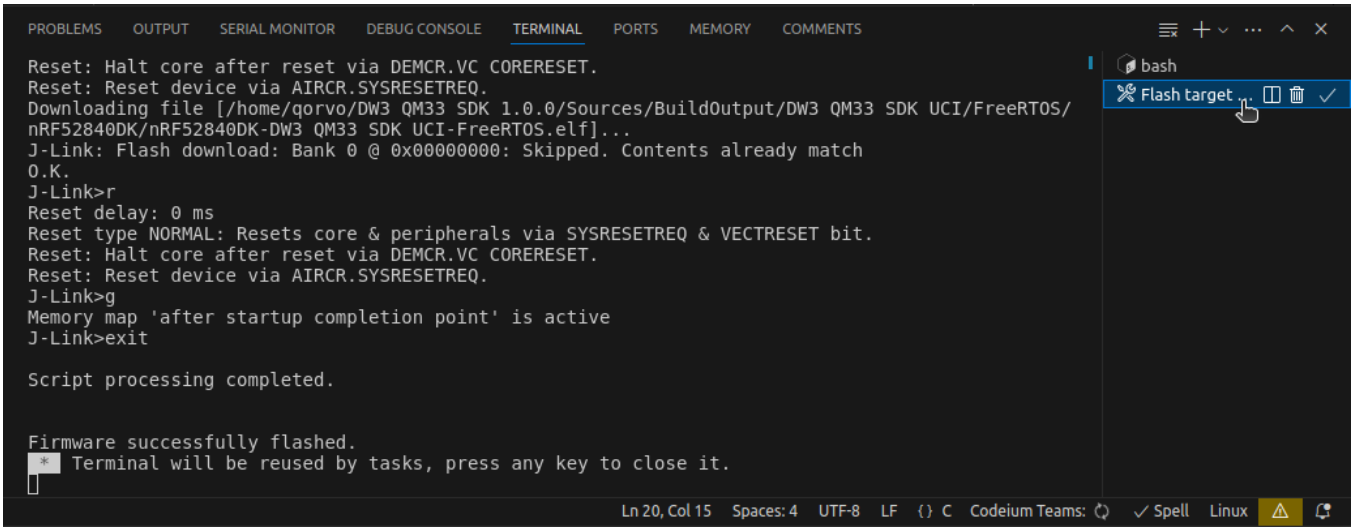

Fig. 4.20: VS Code: Flushing succeeded.

7. Alternatively, you can pick **Build & flash target** to build firmware before flashing.

4.3.2.4 Debug firmware

1. To debug firmware, open **Run and Debug** tab on the left panel, or use the keyboard shortcut `Ctrl+Shift+D`

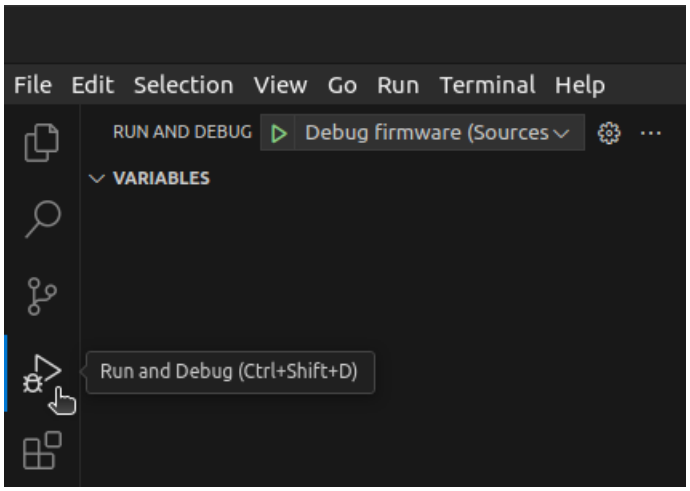

Fig. 4.21: VS Code: Open Run and Debug tab.

2. Choose build configuration (If you pick **Debug firmware**, make sure the firmware has been already built).

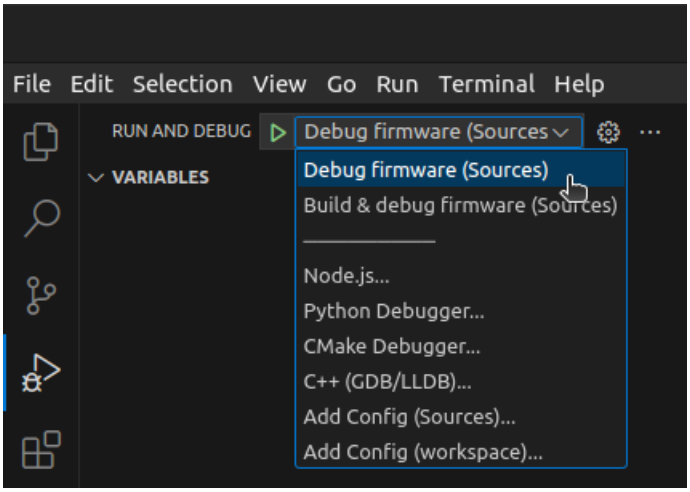

Fig. 4.22: VS Code: Pick debugging profile.

3. Start debug session by clicking green *Play* button or use the keyboard shortcut `F5`

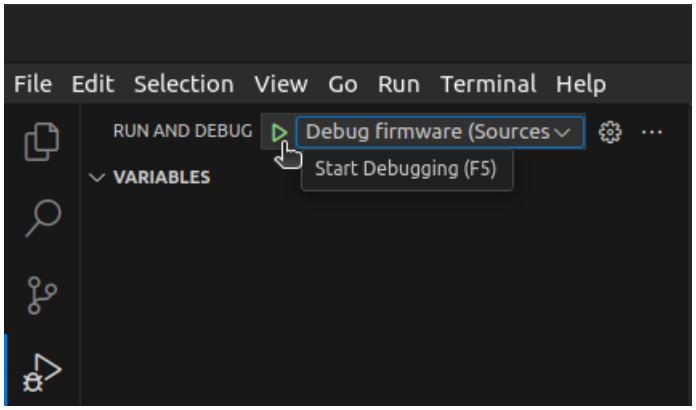

Fig. 4.23: VS Code: Start debug session.

4. Debug session will start. Now you can add breakpoints, variables to watch and execute code step-by-step.

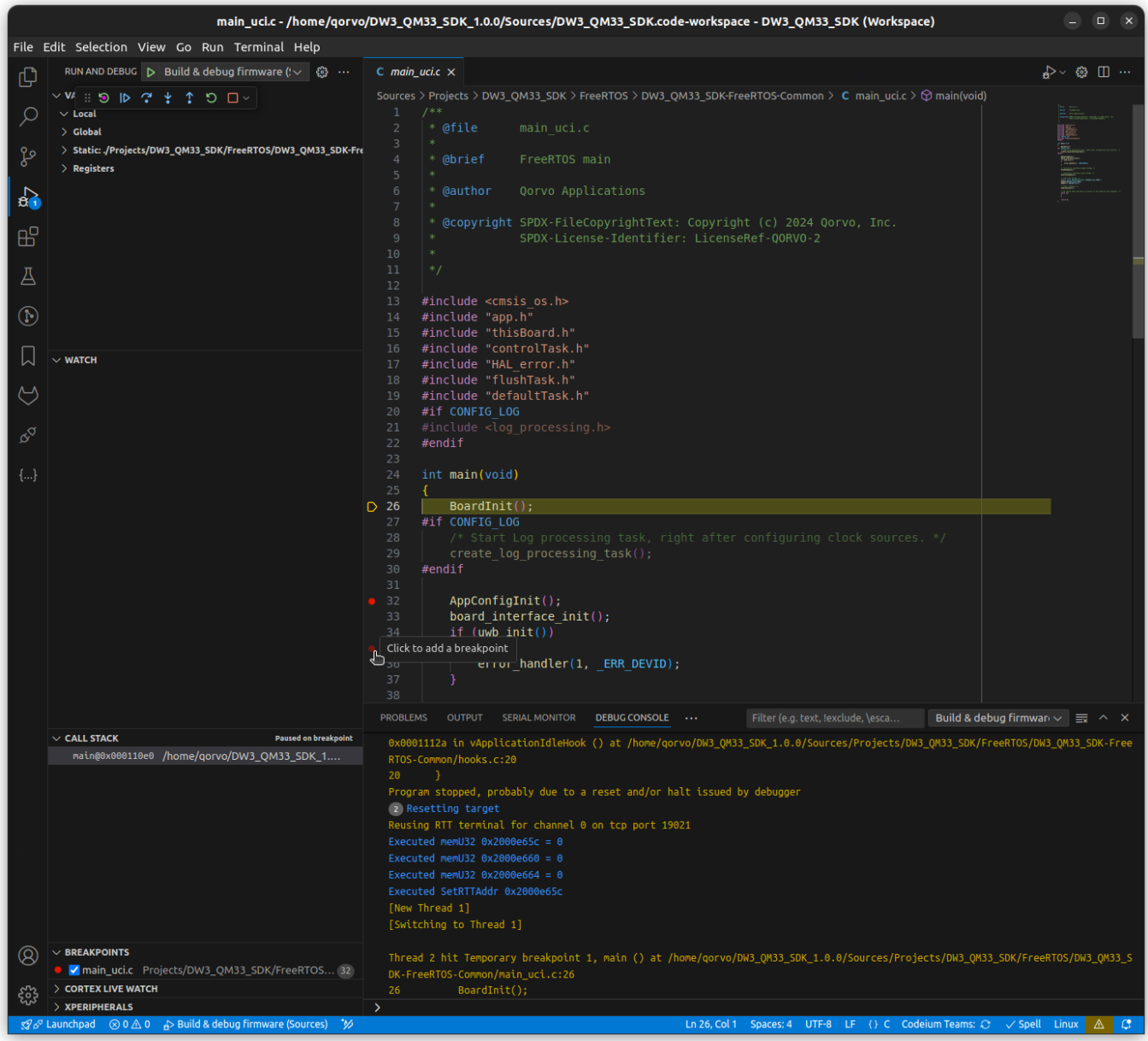

Fig. 4.24: VS Code: Window with active debug session.

5. In Terminal tab you can also find RTT console, where you can access logs from the device.

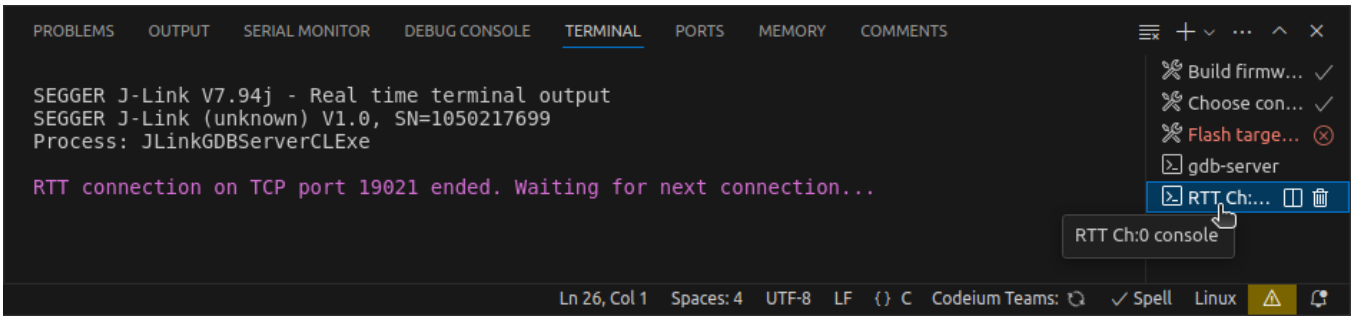

Fig. 4.25: VS Code: RTT console.

4.3.2.5 Predefined tasks

If you find the generic tasks not handy or have issues with **Command Variable** extension, you can choose predefined tasks. They have the same end-functionality, but you have to choose every time a proper task for your devkit board/build type from long list of possible combinations.

1. To use predefined tasks, please replace *.vscode/tasks.json* and *.vscode/launch.json* with files from directory *.vscode/legacy*.

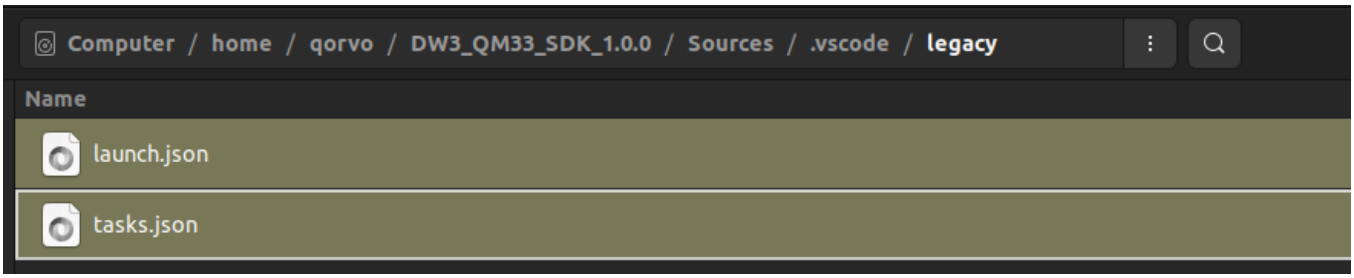

Fig. 4.26: VS Code: Files for predefined configurations.

2. When you open the build dialog you will see a list of all build and flash configurations.

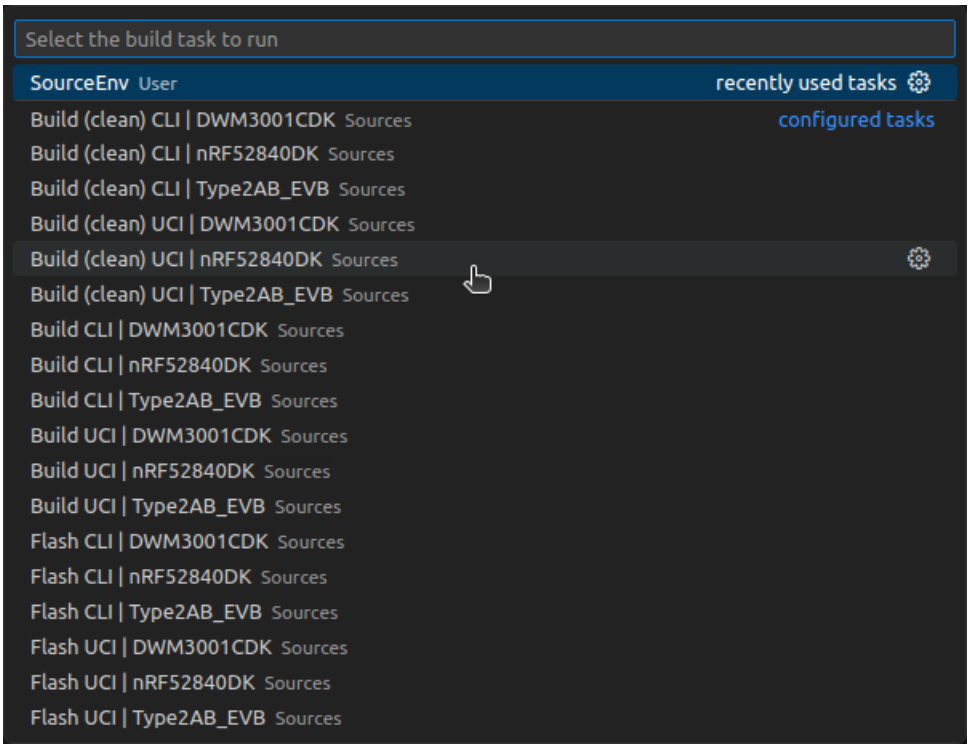

Fig. 4.27: VS Code: Predefined build and flash configurations.

3. When you open the Build and Debug tab, you will see a list of all debug configurations.

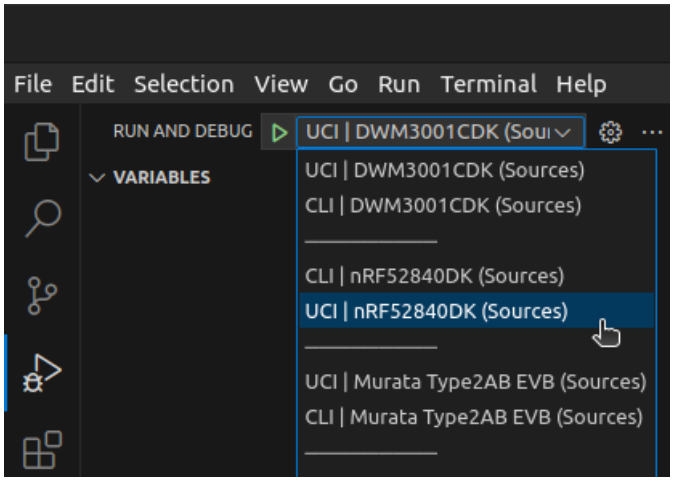

Fig. 4.28: VS Code: Predefined launch configurations.

Note: If you work only with one type of build and board, feel free to comment or remove tasks (from files *tasks.json* / *launch.json*) you are not going to use, list displayed in VS Code will be shorter and easier to use.

4.4 Standalone building and flashing

4.4.1 Building SDK Firmware

4.4.1.1 Create a Target

`CreateTarget.py` automates the process of generating a target for a project using common settings. It calls CMake with the appropriate options to configure the build environment and generate the target.

Warning: Make sure your Python virtual environment is activated before running this script.

4.4.1.1.1 Usage

Navigate to the project folder `<project_dir>/Projects/DW3_QM33_SDK/FreeRTOS/<board_name>` and execute:

- On Linux:

```
./CreateTarget.py [build_type] [-no-force]
```

- On Windows:

```
python ./CreateTarget.py [build_type] [-no-force]
```

4.4.1.1.2 Options

- **`build_type`: Required positional argument with choices `[cli, uci]`:**
  - `cli`: builds the CLI only,
  - `uci`: builds the UCI only.
- `-no-force`: Optional flag to prevent the removal of the old build folder.

4.4.1.1.3 Example

Navigate to the project folder `<project_dir>/Projects/DW3_QM33_SDK/FreeRTOS/<board_name>` and execute:

- On Linux:

```
./CreateTarget.py uci
```

- On Windows:

```
python ./CreateTarget.py uci
```

4.4.1.1.4 Help output

```
./CreateTarget.py --help
usage: CreateTarget.py [-h] [-no-force] {cli,uci}

Build specified target

positional arguments:
{cli,uci}   Build type

options:
-h, --help  show this help message and exit
-no-force   Do not remove old build folder
```

4.4.1.2 Build the Firmware

This step compiles the firmware, producing output files in three formats: `.hex`, `.bin`, and `.elf`.

4.4.1.2.1 Usage

Navigate to the project folder `<project_dir>/BuildOutput/<build_type>/FreeRTOS/<board_name>` and execute:

```
make -j
```

4.4.1.2.2 Examples

- compile CLI build for QM33120WDK1:

```
cd BuildOutput/DW3_QM33_SDK_CLI/FreeRTOS/nRF52840DK
make -j
```

- compile UCI build for DWM30001CDK:

```
cd BuildOutput/DW3_QM33_SDK_CLI/FreeRTOS/DWM3001CDK
make -j
```

Note: To speed up the building process, we use the `-j` flag so all CPU cores will be utilized to build the firmware. You can specify to use e.g. only 4 cores with `make -j4`.

4.4.2 Flashing the development kit

There is a wide variety of tools which can be used to flash the target. Below is an example demonstrating how to use the JFLashLite tool.

Warning: DWM3001CDK boards are not shipped preprogrammed. UCI version needs to be flashed to continue.

1. To flash the board, connect a micro-USB cable to the board (J9 USB connector shown in the figure dwm3001cdk-board.)
2. Locate the JFLashLite tool in your installation directory (e.g. `C:\Program_Files\SEGGER\FlashLite.exe` in Windows or `/usr/bin/JFlashLite` in Ubuntu) and launch the application.

   Note: When you connect the development kit for the first time, you may be prompted to update firmware of the on-board programmer. Please approve the update by clicking **OK** and wait for the update to complete.

3. Upon starting JFlashLite, the Device and Interface selection dialog will appear. Select **NRF52833_xxAA** device by clicking on "**...**" button. On the right side of dialog window, select **SWD** Interface with clock speed of **4000 kHz**. Click **OK** to proceed.

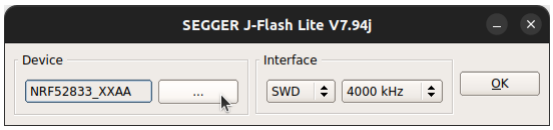

Fig. 4.29: Select CPU settings.

4. Next, select the firmware file you wish to flash onto the board. Click the "**...**" button to browse for the firmware (e.g. `DW3_QM33_SDK/Binaries/DWM3001CDK-DW3_QM33_SDK_UCI-FreeRTOS.hex`). After selecting the file, click **Program Device** to start the flashing process.

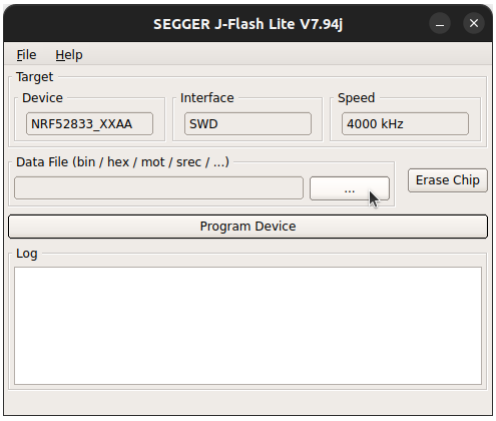

Fig. 4.30: Select .hex file to flash.

5. Once the progress bar is completed, the device has been successfully flashed. Perform a power cycle by disconnecting and reconnecting the power supply to reset the board. Your device should be now ready to use.

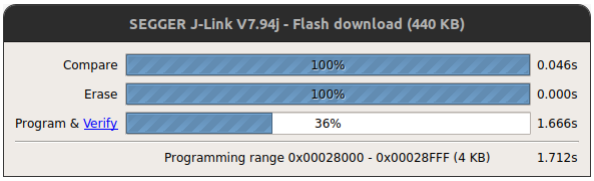

Fig. 4.31: Programming progress window.

5 CLI - Command Line Interface

5.1 Overview

CLI stands for Command Line Interface.

CLI build provides an easy way to send and receive commands to test the different features offered by the software.

This section describes how to use the different CLI commands.

5.2 Setup

Note: See for details *Board Connections*

Open a Serial Terminal (example using Tera Term Open Source Project[13]) and set “Serial Port” as follow:

- Baud rate: 115200
- Number of data bits: 8
- Parity: None
- Number of stop bits: 1
- Flow control: None

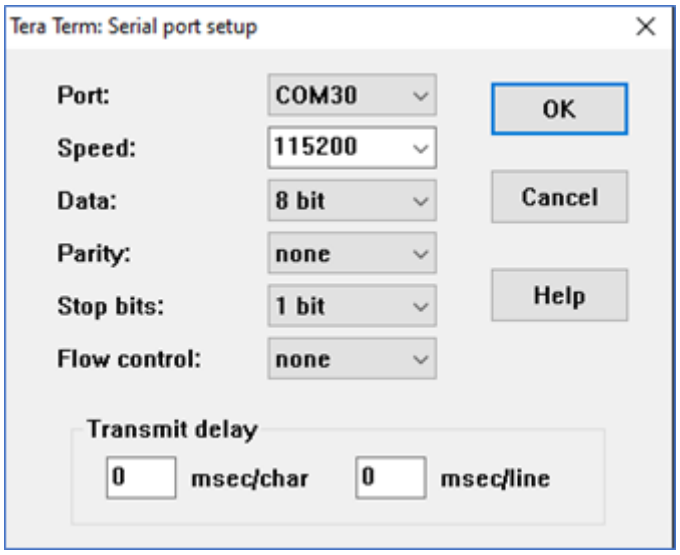

Fig. 5.1: Com serial port settings

[13] https://teratermproject.github.io/index-en.html

To use the numpad operation keys in Tera Term, go to Setup -> Load Key Map... and select `FUNCTION.CNF`.

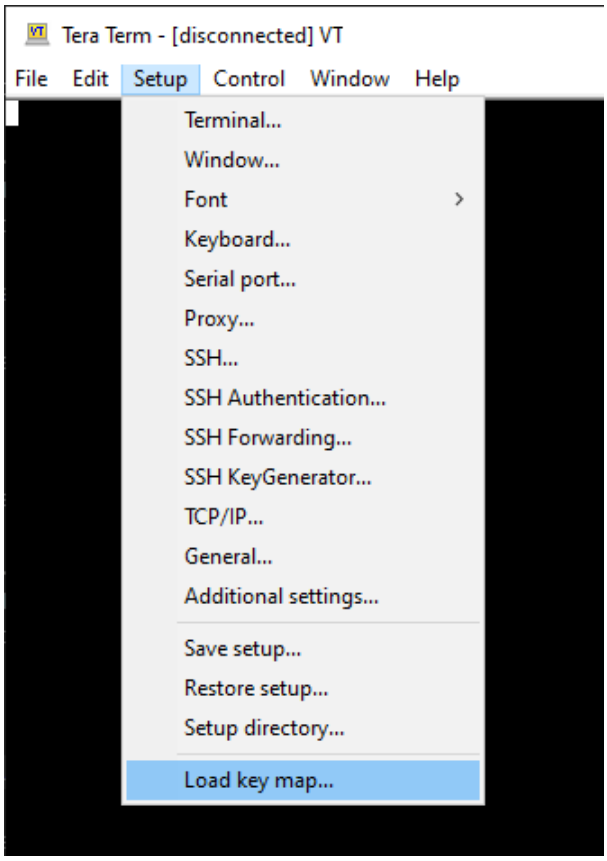

Fig. 5.2: Load Key Map option

Warning: Before starting FiRa applications, please check *Calibration important notes*.

5.3 Anytime Commands

5.3.1 Overview

Commands listed in table below can be executed at any time.

Table 5.1: Anytime Commands

Command	Description
HELP	Outputs a list of all known commands available in the current mode of operation or shows the help of the command <CMD>. Equivalent shortcut is "?".
STAT	Reports the status. Gives a dump of software version info, configuration values and the current operation mode.
STOP	Stops running any of top-level applications and places the device in the NONE mode, where only core tasks are running.
THREAD	Reports information about running threads and memory consumption.

Application or setting commands will return "ok" if the command was properly executed.

5.3.2 HELP

This command displays the help information.

Command:

```
HELP
```

or

```
?
```

Response:

```
nRF52840DK - DW3_QM33_SDK - FreeRTOS

---       Anytime commands       ---
HELP      ?         STOP      THREAD
STAT

---    Application selection     ---
LISTENER  RESPF     INITF

---      IDLE time commands      ---
UART      CALKEY    LISTCAL

---       Service commands       ---
RESTORE   DIAG      LCFG      DECAID
SAVE      SETAPP    GETOTP

---       LISTENER Options       ---
LSTAT
```

Note: This command may also take any command as a parameter. It allows you to display its description. Examples will follow.

Command:

```
HELP INITF
```

Response:

```
INITF:
INITF [Option1] [Option2] ...
Options: (only default values are shown, check the SDK Manual to know more about the available␣
↪configurations)
  -CHAN=9       --> Channel number
  -PRFSET=BPRF4 --> PRF set
  -PCODE=10     --> Preamble code index
  -SLOT=2400    --> Slot duration [RSTU]
  -BLOCK=200    --> Block duration [ms]
  -ROUND=25     --> Round duration [slots]
  -RRU=DSTWR    --> Ranging round usage
  -ID=42        --> Session ID
  -VUPPER=01:02:03:04:05:06:07:08   --> vUpper64
  -MULTI        --> Activate one-to-many mode
```

(continues on next page)

(continued from previous page)

```
  -HOP          --> Activate round hopping
  -ADDR=0       --> Device own address (Initiator address)
  -PADDR=1      --> 1st Responder address
    or to set multiple responders:
  -PADDR=[1,2,.,.,n]  --> to set n Responder addresses (for one-to-many)
```

Command:

```
HELP RESPF
```

Response:

```
RESPF:
RESPF [Option1] [Option2] ...
Options: (only default values are shown, check the SDK Manual to know more about the available␣
↪configurations)
  -CHAN=9       --> Channel number
  -PRFSET=BPRF4 --> PRF set
  -PCODE=10     --> Preamble code index
  -SLOT=2400    --> Slot duration [RSTU]
  -BLOCK=200    --> Block duration [ms]
  -ROUND=25     --> Round duration [slots]
  -RRU=DSTWR    --> Ranging round usage
  -ID=42        --> Session ID
  -VUPPER=01:02:03:04:05:06:07:08   --> vUpper64
  -MULTI        --> Activate one-to-many mode
  -HOP          --> Activate round hopping
  -ADDR=1       --> Device own address (Responder address)
  -PADDR=0      --> Peer address (Initiator address)
```

5.3.3 STAT

This command reports the status. It gives a dump of software version information, supported applications and the current operation mode.

Command:

```
STAT
```

Response:

```
MODE: NONE
JS0108{"Info":{
"Device":"nRF52840DK - DW3_QM33_SDK - FreeRTOS",
"Current App":"NONE",
"Version":"1.0.0",
"Build":"May 30 2024 08:38:01",
"Apps":["LISTENER","RESPF","INITF"],
"Driver":"DW3XXX Device Driver Version 08.00.26",
"UWB stack":"R12.7.0-227-gf8c9f0f27"}}
```

Command:

```
THREAD
```

Response:

```
THREAD NAME             Stack usage
Control                 960/2048
IDLE                    164/512
Flush                   176/512
Logger                  100/768
Default                 172/4304
Tmr Svc                 156/512
Total HEAP              51200
Current HEAP used       8544
Max HEAP used           8544
```

5.3.4 STOP

It stops running any of top-level applications and places the device in the NONE mode, where only core tasks are running.

Command:

```
STOP
```

5.3.5 THREAD

This command reports information about running threads and their stack usage.

For each Thread, there is a ratio displayed in the Stack usage column.

- The 1st value is the maximum stack depth reached during runtime.
- The 2nd value is the maximum stack size allocated to the thread.

5.4 Service Commands

Service commands can only be executed in the NONE mode when no applications are running.

Table 5.2: Service Commands

Command	Description
RESTORE	Restores the default configuration of CLI applications and L1 calibration keys.
LCFG	Sets/Gets a list of configuration parameters and values for LISTENER application.
DIAG	Enables diagnostic mode.
DECAID	Reports information about the chip.
SAVE	Saves default running application and all the configuration parameters to NVM.
GETOTP	Reads the content of OTP memory.

5.4.1 RESTORE

This command restores the default configuration of CLI applications and L1 calibration keys.

The restored parameters are automatically saved and written to NVM.

Command:

```
RESTORE
```

> **Warning:** Please see *Calibration important notes*.

5.4.2 LCFG

> **Note:** This command only concerns LISTENER application

Sets/Gets a list of configuration parameters and values for LISTENER application.

To see LISTENER configuration parameters, use “LCFG”.

Command:

```
LCFG
```

Response:

```
JS00C1{"LCFG PARAM":{
"CHAN":9,
"PAC":8,
"PCODE":10,
"SFDTYPE":3,
"DRATE":6810,
"PHRMODE":0,
"PHRRATE":0,
"STSMODE":0,
"STSLEN":64,
"PDOAMODE":1,
"XTALTRIM":46}}
```

To set LISTENER configuration parameters, use “LCFG <List of parameters>” with parameters listed in Table *Parameters list and default values*..

Command:

```
LCFG -PCODE=9
```

Response:

```
JS00C2{"LCFG PARAM":{
"CHAN":9,
"PAC":8,
"PCODE":9,
"SFDTYPE":3,
"DRATE":6810,
"PHRMODE":0,
```

(continues on next page)

(continued from previous page)

```
"PHRRATE":0,
"STSMODE":0,
"STSLEN":64,
"PDOAMODE":1,
"XTALTRIM":46}}
```

Table 5.3: Parameters list and default values

Order	Parameter	Default value	Description
1	CHAN	9	Channel selection [5, 9]
2	PAC	8	Preamble Acquisition Chunk size [4 or 8 when RX Preamble Length is <= 128, 16 for 256, 32 when higher]
3	PCODE	10	Preamble Code [9, 10, 11, 12]
4	SFDTYPE	3	Start Frame Delimiter [0 = 4A 8-bit, 1 = DW 8-bit, 2 = DW 16-bit, 3 = 4z 8-bit]
5	DRATE	6810	PSDU Data Rate in kbps [850, 6810]
6	PHRMODE	0	PHR Mode [0 - Standard, 1 - DW Extended]
7	PHRRATE	0	PHR Data Rate [0 - Standard, 1 - PHR at data rate]
8	STSMODE	0	Scrambled Time Sequence Mode [0 - off, 1 - STS mode 1, 2 - STS mode 2, 3 - STS with no data, 8 - STS mode SDC]
9	STSLEN	64	STS Length (power of 2) [32, . ., 2048]
10	PDOAMODE	1	PDoA Mode [0 - off, 1 - PDOA mode 1, 3 - PDOA mode 3]
11	XTALTRIM	46	Crystal oscillator trimming value [0, . ., 63]

5.4.3 DIAG

Reports diagnostics status or changes its value.

DIAG provides a way to display additional information about the performance while ranging. By default, it is set to 0.

To see diagnostics status, use “DIAG”.

Command:

```
DIAG
```

Response:

```
DIAG: 0
```

To set diagnostics status, use “DIAG <status>”,

- where <status> value:
  - 0: disables diagnostics,
  - 1: enables diagnostics.

Command:

```
DIAG 1
```

With DIAG set to 1, it allows to display RSSI for the whole ranging round and many more information about every frame sent/received.

RSSI calculations make use of IPATOV, STS1, STS2, and CIR (Channel Impulse Response) parameters. In particular, three points of the first path amplitude magnitude, the number of symbols accumulated, and the CIR power value. Details on RSSI can be found in the DW3000 User Manual (Diagnostic APIs sections).

Output when diagnostics enabled:

```
SESSION_INFO_NTF: {session_handle=1, sequence_number=263, block_index=263, n_measurements=1
[mac_address=0x0001, status="SUCCESS", distance[cm]=224, rmt_az=4.78, RSSI[dBm]=-73.5]}
RANGE_DIAGNOSTICS_NTF: {n_reports=6
[msg_id=CONTROL, action=TX, antenna_set=0, frame_status={SUCCESS: 1, WIFI_COEX: 0, GRANT_
↪DURATION_EXCEEDED: 0}, cfo_present=0, nb_aoa=0];
[msg_id=RANGING_INITIATION, action=TX, antenna_set=0, frame_status={SUCCESS: 1, WIFI_COEX: 0,␣
↪GRANT_DURATION_EXCEEDED: 0}, cfo_present=0, nb_aoa=0];
[msg_id=RANGING_RESPONSE, action=RX, antenna_set=0, frame_status={SUCCESS: 1, WIFI_COEX: 0,␣
↪GRANT_DURATION_EXCEEDED: 0}, cfo_present=1, cfo_ppm=0.48, nb_aoa=0];
[msg_id=RANGING_FINAL, action=TX, antenna_set=0, frame_status={SUCCESS: 1, WIFI_COEX: 0, GRANT_
↪DURATION_EXCEEDED: 0}, cfo_present=0, nb_aoa=0];
[msg_id=MEASUREMENT_REPORT, action=TX, antenna_set=0, frame_status={SUCCESS: 1, WIFI_COEX: 0,␣
↪GRANT_DURATION_EXCEEDED: 0}, cfo_present=0, nb_aoa=0];
[msg_id=RESULT_REPORT, action=RX, antenna_set=0, frame_status={SUCCESS: 1, WIFI_COEX: 0, GRANT_
↪DURATION_EXCEEDED: 0}, cfo_present=1, cfo_ppm=0.36, nb_aoa=0]}
```

5.4.4 DECAID

This command reports information about the chip.

Note: Qorvo SoC ID is defined as a concatenation of Qorvo Lot ID and Qorvo Part ID (in that order).

Command:

```
DECAID
```

Response:

```
Qorvo Device ID = 0xdeca0304
Qorvo Lot ID = 0x0000503639463438
Qorvo Part ID = 0x8124d5b7
Qorvo SoC ID = 00005036394634388124d5b7
```

5.4.5 SAVE

This command saves configuration parameters of CLI applications to NVM (Non-Volatile Memory). The saved parameters will stay after reboot or power cycle therefore making this ideal for battery powered configuration options.

Note: The SAVE command cannot be used during a ranging session.

Note: The default FiRa parameters could be changed manually in the common_fira.c or driver_app_config.c, if desired.

Command:

```
SAVE
```

This command can be used to run FiRa applications (INITF/RESPF) without providing parameters again.

Commands:

```
INITF -BLOCK=400
STOP
SAVE

INITF
<runs with BLOCK=400>
```

5.4.6 SETAPP

This command allows configuration of the default application that starts automatically after reboot or power cycle.

Note: Use SAVE command after SETAPP to store default application in NVM.

To check the default application, use “SETAPP”.

Command:

```
SETAPP
```

Response:

```
SETAPP: NONE
```

To set the default application, use “SETAPP <app>”,

- where <app> value:
  - LISTENER: UWB Sniffer,
  - INITF: Initiator in a FiRa two-way ranging session,
  - RESPF: Responder in a FiRa two-way ranging session,
  - NONE: No default application (IDLE state as default).

Command:

```
SETAPP RESPF
```

Note: The default parameters are set to a well-defined STS key/data pair and static mode.

5.4.7 GETOTP

Reads the content of OTP memory.

Command:

```
GETOTP
```

Response:

```
OTP CONTENT: {
"0x000":"0x00000000",
"0x001":"0x00000000",
"0x002":"0x00000000",
"0x003":"0x00000000",
"0x004":"0x00000001",
"0x005":"0x06867777",
"0x006":"0x8124d5b7",
"0x007":"0x00000000",
"0x008":"0x01758d40",
"0x009":"0x0000008d",
...
"0x07d":"0x00000000",
"0x07e":"0x00000000",
"0x07f":"0x00000000",
}
```

5.5 IDLE time Commands

5.5.1 Overview

IDLE time commands listed in table below can be executed only when the mode is NONE, corresponding to IDLE state.

Table 5.4: IDLE time Commands

Command	Description
UART	Enables selected UART.
CALKEY	Sets or gets a calibration key.
LISTCAL	Lists all available calibration keys and their respective values.

5.5.2 UART

Note: The “UART” command is not available when the compile flag “USB_ENABLE” was removed from the project.

Reports UART communication status or changes its value.

To see UART communication status, use “UART”.

Command:

```
UART
```

Response:

```
UART: 0
```

To set UART communication status, use “UART <status>”,

- where <status> value:
  - 0: disables communication using UART,
  - 1: enables communication using UART.

Command:

```
UART 1
```

5.5.3 CALKEY

Sets or gets a calibration key.

To get a value of a calibration key, use “CALKEY <KEY>”:

Command:

```
CALKEY xtal_trim
```

Response:

```
xtal_trim: 0x32 (len: 1)
```

See *Calibration and Configuration* chapter for more details on calibration possibilities.

To set a value of a calibration key, use “CALKEY <KEY> <VALUE>”:

Command:

```
CALKEY xtal_trim 1
```

Response:

```
xtal_trim: 0x1 (len: 1)
```

5.5.4 LISTCAL

Lists all available calibration keys and their respective values.

Command:

```
LISTCAL
```

Response:

```
restricted_channels: 0x0000 (len: 2)
wifi_coex_mode: 0x00 (len: 1)
wifi_coex_time_gap: 0x00 (len: 1)
ch5.wifi_coex_enabled: 0x01 (len: 1)
ch5.pll_locking_code: 0x00 (len: 1)
ch9.wifi_coex_enabled: 0x01 (len: 1)
...
ant_set0.rx_ants: 0x000001 (len: 3)
```

(continues on next page)

(continued from previous page)

```
ant_set0.tx_ant_path: 0x00 (len: 1)
ant_set0.nb_rx_ants: 0x01 (len: 1)
ant_set0.rx_ants_are_pairs: 0x00 (len: 1)
ant_set0.tx_power_control: 0x01 (len: 1)
experimental.mac.session_scheduler.id: 0x00 (len: 1)
```

5.6 Application Commands

5.6.1 SDK Applications

Top-level applications cannot run concurrently since they use the same resources.

Table 5.5: List of applications

Command	Description
INITF	Sets the board as INITIATOR in a FiRa two-way ranging session.
RESPF	Sets the board as RESPONDER in a FiRa two-way ranging session.
LISTENER	Puts the board into receive mode and reports any received packets.

5.6.2 FiRa applications: INITF/RESPF

5.6.2.1 INITF/RESPF command

Warning: If you are not using a QM33120WDK1, DWM3001CDK or Type2AB EVB (e.g. DWM3000EVB), you must flash the calibration before starting a ranging session.

Warning: Using the default calibration leads to decreased ranging accuracy (see section *Calibration important notes* for more details).

INITF sets the board as INITIATOR in a FiRa two-way ranging session. See table below for configuration parameters. Usage:

```
INITF [Option1] [Option2] ...
Options: (only default values are shown, check the SDK Manual to know more about the available␣
↪configurations)
  -CHAN=9       --> Channel number
  -PRFSET=BPRF4 --> PRF set
  -PCODE=10     --> Preamble code index
  -SLOT=2400    --> Slot duration [RSTU]
  -BLOCK=200    --> Block duration [ms]
  -ROUND=25     --> Round duration [slots]
  -RRU=DSTWR    --> Ranging round usage
  -ID=42        --> Session ID
  -VUPPER=01:02:03:04:05:06:07:08   --> vUpper64
  -MULTI        --> Activate one-to-many mode
  -HOP          --> Activate round hopping
  -ADDR=0       --> Device own address (Initiator address)
```

(continues on next page)

(continued from previous page)

```
  -PADDR=1      --> 1st Responder address
    or to set multiple responders:
  -PADDR=[1,2,.,.,n]  --> to set n Responder addresses (for one-to-many)
```

RESPF sets the board as RESPONDER in a FiRa two-way ranging session. See table below for configuration parameters. Usage:

```
RESPF [Option1] [Option2] ...
Options: (only default values are shown, check the SDK Manual to know more about the available␣
↪configurations)
  -CHAN=9       --> Channel number
  -PRFSET=BPRF4 --> PRF set
  -PCODE=10     --> Preamble code index
  -SLOT=2400    --> Slot duration [RSTU]
  -BLOCK=200    --> Block duration [ms]
  -ROUND=25     --> Round duration [slots]
  -RRU=DSTWR    --> Ranging round usage
  -ID=42        --> Session ID
  -VUPPER=01:02:03:04:05:06:07:08   --> vUpper64
  -MULTI        --> Activate one-to-many mode
  -HOP          --> Activate round hopping
  -ADDR=1       --> Device own address (Responder address)
  -PADDR=0      --> Peer address (Initiator address)
```

Table 5.6: INITF/RESPF parameters

Option	Default Value	Range	Description
CHAN	9	5 or 9	Channel number
PRFSET	BPRF4	BPRF3 BPRF4 BPRF5 BPRF6	Pulse Repetition Frequency parameter sets See *BPRF mode operating parameter sets*
PCODE	10	9 to 12	Preamble code index
SLOT	2400	1200 to 12000	Duration of the slot in RSTU. 1 ms = 1200 RSTU
BLOCK	200	100 to 500	Duration of the ranging block in ms
ROUND	25	2 to 255	Number of slots inside a ranging round
RRU	DSTWR	SSTWR DSTWR SSTWRNDEF DSTWRNDEF	Ranging round usage SSTWR: Single-Sided Two-Way Ranging with Deferred mode DSTWR: Double-Sided Two-Way Ranging with Deferred mode SSTWRNDEF: Single-Sided Two-Way Ranging with Non-Deferred mode DSTWRNDEF: Double-Sided Two-Way Ranging with Non-Deferred mode
ID	42	1 to 65535	Session ID
VUPPER	01:02:03:04:05:06:07:08	00:00:00:00:00:00:00:00 to FF:FF:FF:FF:FF:FF:FF:FF	vUpper64 Eight hexadecimal numbers, representing static part of the STS in FiRa standard
MULTI	Not activated		Activate one-to-many mode (instead of unicast mode)
HOP	Not activated		Activate round hopping
ADDR	Initiator: 0, Responder: 1	0 to 65535	Device own address
PADDR	Initiator: 1, Responder: 0	0 to 65535	Address(es) of peer devices For Responder, it is the Initiator address (only one) For Initiator, it is the Responder address or the list of multiple Responder addresses

Note: The options are all optional and can be set in any order and capital letters are not mandatory.

Table 5.7: BPRF mode operating parameter sets

BPRF Set#	SYNC PSR	SFD# (802.15.4 2020)	SFD Length	STS nr of Segments	STS Segment Length	PHR + data	Data Rate (Mbps)	Description
BPRF3	64	2	8	1	64	Yes	6.8	SP1 (Mandatory)
BPRF4	64	2	8	1	64	No	n/a	SP3 (Mandatory)
BPRF5	64	0	8	1	64	Yes	6.8	SP1 (Optional)
BPRF6	64	0	8	1	64	No	n/a	SP1 (Optional)

Listing 5.1: Example of default INITF command

```
INITF
```

Listing 5.2: Equivalent of default INITF command

```
INITF -CHAN=9 -PRFSET=BPRF4 -SLOT=2400 -BLOCK=200 -ROUND=25 -RRU=DSTWR -ID=42 -
↪VUPPER=01:02:03:04:05:06:07:08 -ADDR=0 -PADDR=1
```

Listing 5.3: Example of INITF command with multiple Responders

```
INITF -MULTI -ADDR=0 -PADDR=[1, 2, 3]
```

Listing 5.4: Example of default RESPF command

```
RESPF
```

Listing 5.5: Equivalent of default RESPF command

```
RESPF -CHAN=9 -PRFSET=BPRF4 -SLOT=2400 -BLOCK=200 -ROUND=25 -RRU=DSTWR -ID=42 -
↪VUPPER=01:02:03:04:05:06:07:08 -ADDR=1 -PADDR=0
```

Note: SAVE can be used to run the next INITF/RESPF (without specifying parameters) with previously saved parameters.

Listing 5.6: Run application with saved parameters

```
RESPF -CHAN=5 -BLOCK=400
STOP
SAVE
RESPF
<application running with CHAN=5 and BLOCK=400>
```

Next application command with provided any parameter will reset parameters and apply only provided one.

Listing 5.7: Change one of the parameters

```
RESPF -CHAN=5 -BLOCK=400
STOP
SAVE
RESPF -PRFSET=BPRF3
<application running with PRFSET=BPRF3 but CHAN=9 (default), BLOCK=200 (default)>
```

To restore all default parameters, run INITF/RESPF with at least one given default parameter or use RESTORE command.

Listing 5.8: Reset parameters to default

```
RESPF -CHAN=5 -BLOCK=400
STOP
SAVE
RESPF -CHAN=9
<application running with all default parameters>
STOP
SAVE
RESPF
<application running with all default parameters>
```

Listing 5.9: Reset parameters to default with RESTORE command

```
RESPF -CHAN=5 -BLOCK=400
STOP
SAVE
RESTORE
RESPF
<application running with all default parameters>
```

5.6.2.2 INITF/RESPF messages

INITF and RESPF commands display notifications during the ranging session.

Listing 5.10: Example log from INITF execution

```
SESSION_INFO_NTF: {session_handle=1, sequence_number=0, block_index=0, n_measurements=1
  [mac_address=0x0001, status="SUCCESS", distance[cm]=91, loc_az_pdoa=65.35, loc_az=24.90, loc_
↪el_pdoa=32.12, loc_el=12.01, rmt_az=22.84, rmt_el=13.59, RSSI[dBm]=-66.5]}
SESSION_INFO_NTF: {session_handle=1, sequence_number=1, block_index=1, n_measurements=1
  [mac_address=0x0001, status="SUCCESS", distance[cm]=101, loc_az_pdoa=57.79, loc_az=21.87, loc_
↪el_pdoa=35.54, loc_el=14.62, rmt_az=20.92, rmt_el=15.72, RSSI[dBm]=-66.5]}
SESSION_INFO_NTF: {session_handle=1, sequence_number=2, block_index=2, n_measurements=1
  [mac_address=0x0001, status="SUCCESS", distance[cm]=105, loc_az_pdoa=57.71, loc_az=21.85, loc_
↪el_pdoa=33.34, loc_el=12.89, rmt_az=21.37, rmt_el=14.02, RSSI[dBm]=-67.5]}
SESSION_INFO_NTF: {session_handle=1, sequence_number=3, block_index=3, n_measurements=1
  [mac_address=0x0001, status="SUCCESS", distance[cm]=110, loc_az_pdoa=56.48, loc_az=21.37, loc_
↪el_pdoa=34.47, loc_el=13.16, rmt_az=21.12, rmt_el=15.47, RSSI[dBm]=-65.5]}
```

Table 5.8: INITF/RESPF log legend

session_handle	Unique number generated by the device used to identify the UWB session.
sequence_number	Counter which starts from 0 and is incremented after every notification sent.
block_index	Current block index.
n_measurements	Number of measurements in this notification (one measurement = one pair of initiator/responder devices).
mac_address	MAC address of the peer device (the one that the current device is ranging with).
status	Status of the current ranging round.
distance[cm]	Distance between devices in centimeters.
loc_az_pdoa	Local raw PDoA measurements for Azimuth in deg [-180..+180].
loc_az	Local AoA Azimuth in deg [-90..+90].
loc_el_pdoa	Local raw PDoA measurements for Elevation in deg [-180..+180].
loc_el	Local AoA Elevation in deg [-90..+90].
rmt_az	Remote AoA Azimuth in deg [-90..+90].
rmt_el	Remote AoA Elevation in deg [-90..+90].
RSSI	RSSI (displayed only when DIAG is enabled, see *DIAG* command).

Set AoA/PDoA Enhancement Parameters

Some parameters can be used to improve AoA/PDoA performance. These settings need to be set on the device where the local AoA will be collected. This is where the AoA antenna is used and generally (but not always) it is the initiator.

- The calibration parameters can be used to enhance accuracy (see *Calibration and Configuration* chapter for more details).

5.6.3 UWB sniffer: LISTENER

PHY-level LISTENER application. This command will start a sniffer application based on the **LCFG** configuration.

Usage:

```
LISTENER:
Listen for the UWB packets using the UWB configuration and displays.
1/ Maximum 127 bytes of payload.
2/ "TS4ns": Timestamp of each valid frame reception.
3/ "O": Crystal frequency offset in ppm units.
4/ RSL, FSL: RX Level and First Path Power level (dBm).
```

LSTAT command displays the statistics inside the LISTENER application. It needs to be called while LISTENER application is running and it shows the following information:

Table 5.9: LSTAT details

Info	Description
CRCG	12-bit number of good CRC received frame events.
CRCB	12-bit number of bad CRC (CRC error) received frame events.
ARFE	8-bit number of address filter error events.
PHE	12-bit number of received header error events.
RSL	12-bit number of received frame sync loss event events.
SFDTO	12-bit number of SFD timeout events.
PTO	12-bit number of Preamble timeout events.
FTO	8-bit number of RX frame wait timeout events.
SFDD	Counts the number of SFD detections (RXFR shall be set also).

Example:

Responder side: start LISTENER with default LCFG settings

Command:

```
LISTENER
```

Response:

```
Found AOA chip. PDoA is available.
Listener Top Application: Started
```

Initiator side: start INITF with default settings

Command:

```
INITF
```

Response:

```
FiRa Session Parameters: {
SESSION_ID: 42,
CHANNEL_NUMBER: 9,
DEVICE_ROLE: INITIATOR,
RANGING_ROUND_USAGE: DS_TWR_DEFERRED,
SLOT_DURATION [rstu]: 2400,
RANGING_DURATION [ms]: 200,
SLOTS_PER_RR: 25,
MULTI_NODE_MODE: UNICAST,
```

(continues on next page)

(continued from previous page)

```
HOPPING_MODE: Disabled,
RFRAME_CONFIG: SP3,
SFD_ID: 2,
PREAMBLE_CODE_INDEX: 10,
STATIC_STS_IV: "01:02:03:04:05:06",
VENDOR_ID: "07:08",
DEVICE_MAC_ADDRESS: 0x0000,
DST_MAC_ADDRESS[0]: 0x0001
}
```

Listener logs

Type “LISTENER” to dump all the data captured. This cannot run faster than every 1ms. Please note, if the buffer is full, some data packets may be skipped. Type “STOP” to halt.

LISTENER application output of SP0 packets (STSMODE parameter set to 0):

Command:

```
LISTENER
```

Response:

```
Found AOA chip. PDoA is available.
Listener Top Application: Started
JS010C{"LSTN":[49,2B,01,00,26,13,00,FF,18,5A,08,08,08,08,08,08,08,08,2A,00,00,00,C1,4A,4C,7A,00,
↪3F,89,CF,EF,B6,B6,9A,3D,7E,A9,AE,54,CE,6E,7F,F9,A1,73,E4,A3,BF,2B,4B,CD,C0,75,60,7B,38,15,71,
↪1D,CD,88,07,F7,F5,C9,25,0A],"TS4ns":"0xD8B30876","O":-11,"rsl":-42.00,"fsl":-67.41}
JS010C{"LSTN":[49,2B,01,00,26,13,00,FF,18,5A,08,08,08,08,08,08,08,08,2A,00,00,00,C1,4A,4C,7A,00,
↪3F,89,CF,EF,B6,B6,9A,3D,7E,A9,AE,54,CE,6E,7F,F9,A1,73,E4,A3,BF,2B,4B,CD,C0,75,60,7B,38,15,71,
↪1D,CD,88,07,F7,F5,C9,25,0A],"TS4ns":"0xD8B30876","O":-11,"rsl":-42.00,"fsl":-67.41}
JS010B{"LSTN":[49,2B,01,00,26,13,00,FF,18,5A,08,08,08,08,08,08,08,08,2A,00,00,00,25,4B,4C,7A,00,
↪3F,89,CF,EF,B6,B6,9A,3D,7E,A9,AE,54,CE,6E,7F,F9,A1,73,E4,A3,BF,2B,4B,CD,C0,75,60,7B,38,15,B3,
↪0D,ED,AE,A4,4B,6D,AD,5A,62],"TS4ns":"0xDBACC224","O":-5,"rsl":-42.09,"fsl":-67.52}
JS010B{"LSTN":[49,2B,01,00,26,13,00,FF,18,5A,08,08,08,08,08,08,08,08,2A,00,00,00,25,4B,4C,7A,00,
↪3F,89,CF,EF,B6,B6,9A,3D,7E,A9,AE,54,CE,6E,7F,F9,A1,73,E4,A3,BF,2B,4B,CD,C0,75,60,7B,38,15,B3,
↪0D,ED,AE,A4,4B,6D,AD,5A,62],"TS4ns":"0xDBACC224","O":-5,"rsl":-42.09,"fsl":-67.52}
JS010B{"LSTN":[49,2B,01,00,26,13,00,FF,18,5A,08,08,08,08,08,08,08,08,2A,00,00,00,89,4B,4C,7A,00,
↪3F,89,CF,EF,B6,B6,9A,3D,7E,A9,AE,54,CE,6E,7F,F9,A1,73,E4,A3,BF,2B,4B,CD,C0,75,60,7B,38,15,2A,
↪52,7B,23,96,F6,7A,1D,83,E1],"TS4ns":"0xDEA67AEA","O":-1,"rsl":-47.79,"fsl":-73.37}
JS010B{"LSTN":[49,2B,01,00,26,13,00,FF,18,5A,08,08,08,08,08,08,08,08,2A,00,00,00,89,4B,4C,7A,00,
↪3F,89,CF,EF,B6,B6,9A,3D,7E,A9,AE,54,CE,6E,7F,F9,A1,73,E4,A3,BF,2B,4B,CD,C0,75,60,7B,38,15,2A,
↪52,7B,23,96,F6,7A,1D,83,E1],"TS4ns":"0xDEA67AEA","O":-1,"rsl":-41.79,"fsl":-67.37}
```

Command:

```
STOP
```

6 Qorvo One TWR GUI

6.1 Overview

The Qorvo One TWR application provides a graphical user interface to demonstrate features of the QM SDK like Two-Way Ranging, configuration and calibration.

6.2 User Manual

6.2.1 Installation

The Qorvo One TWR GUI supports desktop platforms running Windows, Linux or MacOS.

6.2.1.1 Windows

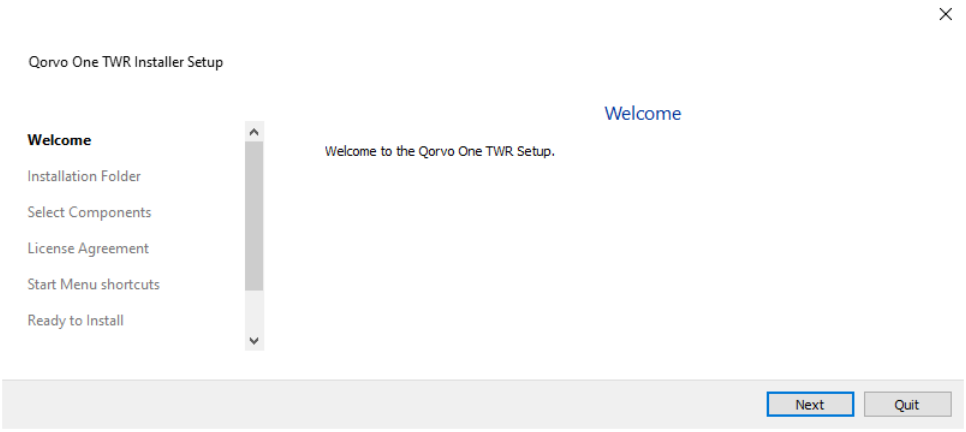

Fig. 6.1: Qorvo One TWR application installer.

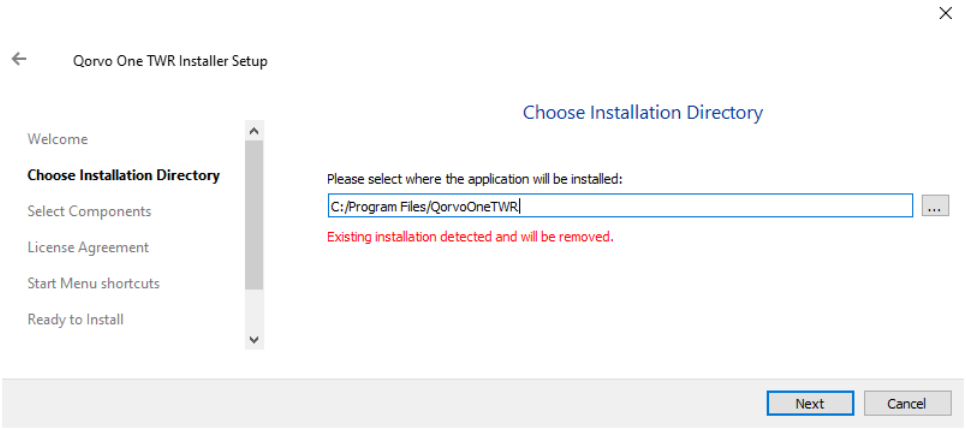

Fig. 6.2: Removing existing installation.

Fig. 6.3: License agreement.

The application can be installed on Windows using **QorvoOneTwr-2.1.0-setup.exe** installer. Successful installation requires the license agreement to be accepted. And the previous installation is uninstalled.

The application can be started by clicking on the shortcut automatically created by the installer.

6.2.1.2 Linux

Make Qorvo One TWR installer script executable and run it from terminal. The installer will extract the application into the same folder by default.

```
$ ./QorvoOneTWR-2.1.0-x86_64-install.sh
```

The Qorvo One TWR application is packaged for the Linux distribution as AppImage[14]. To start the application simply run the installed AppImage.

Warning:

- If you got the error:

[14] https://appimage.org/

```
dlopen():  error loading libfuse.so.2
```

Install libfuse2 using the following commands:

```
sudo apt update
sudo apt install libfuse2
```

- On Ubuntu 20.04, you may also need to install qt5-default:

```
sudo apt update
sudo apt install qt5-default
```

- On Ubuntu 22.04 & Ubuntu 24.04, you may also need to install qtbase5-dev:

```
sudo apt update
sudo apt install qtbase5-dev
```

6.2.1.3 Mac OS

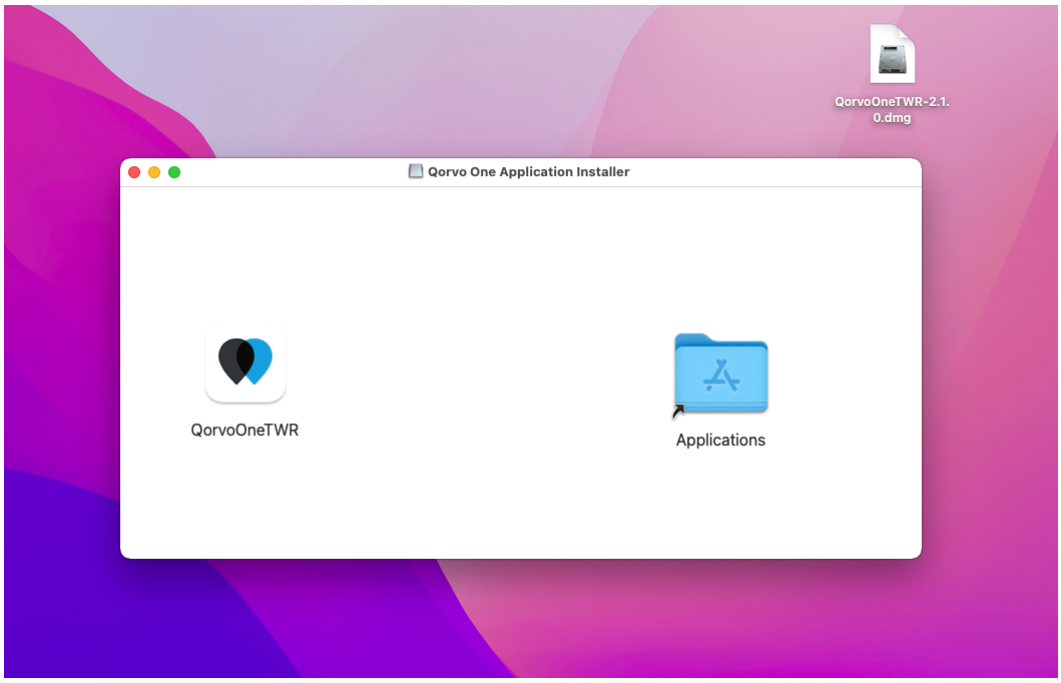

Fig. 6.4: Installing into the Applications folder.

To install the **QorvoOneTwr-2.1.0.dmg**:

1. Double-click the DMG file.
2. Drag the application from the DMG window into the Applications directory to install (you may need the administrator password).
3. Eject the DMG installer.
4. Optionally delete the DMG from the Downloads directory.

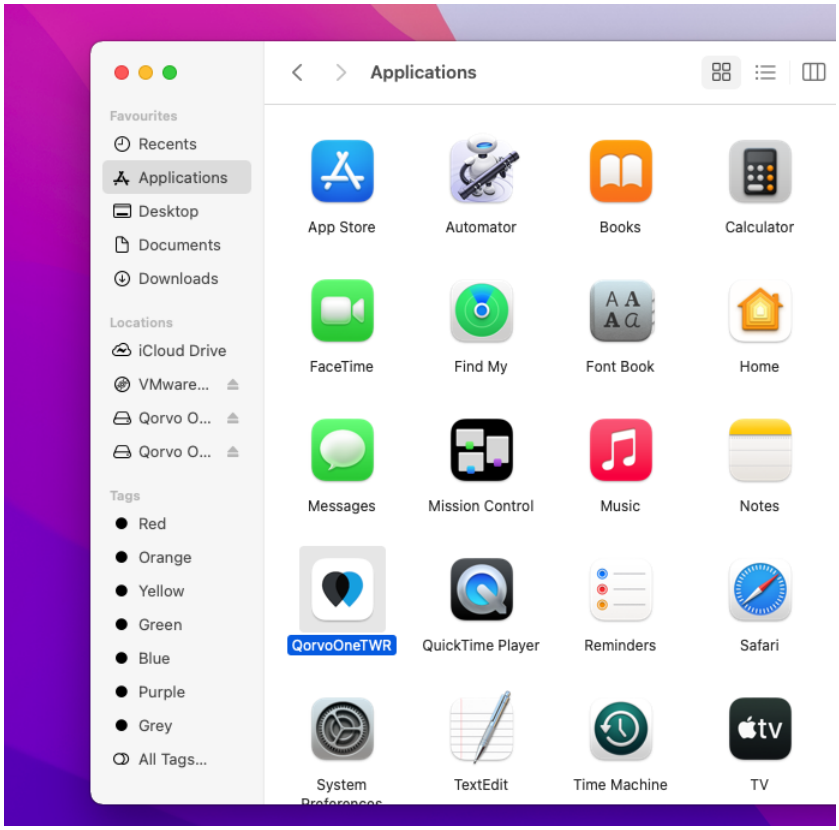

Fig. 6.5: Installed Qorvo One TWR application.

6.2.2 Welcome screen

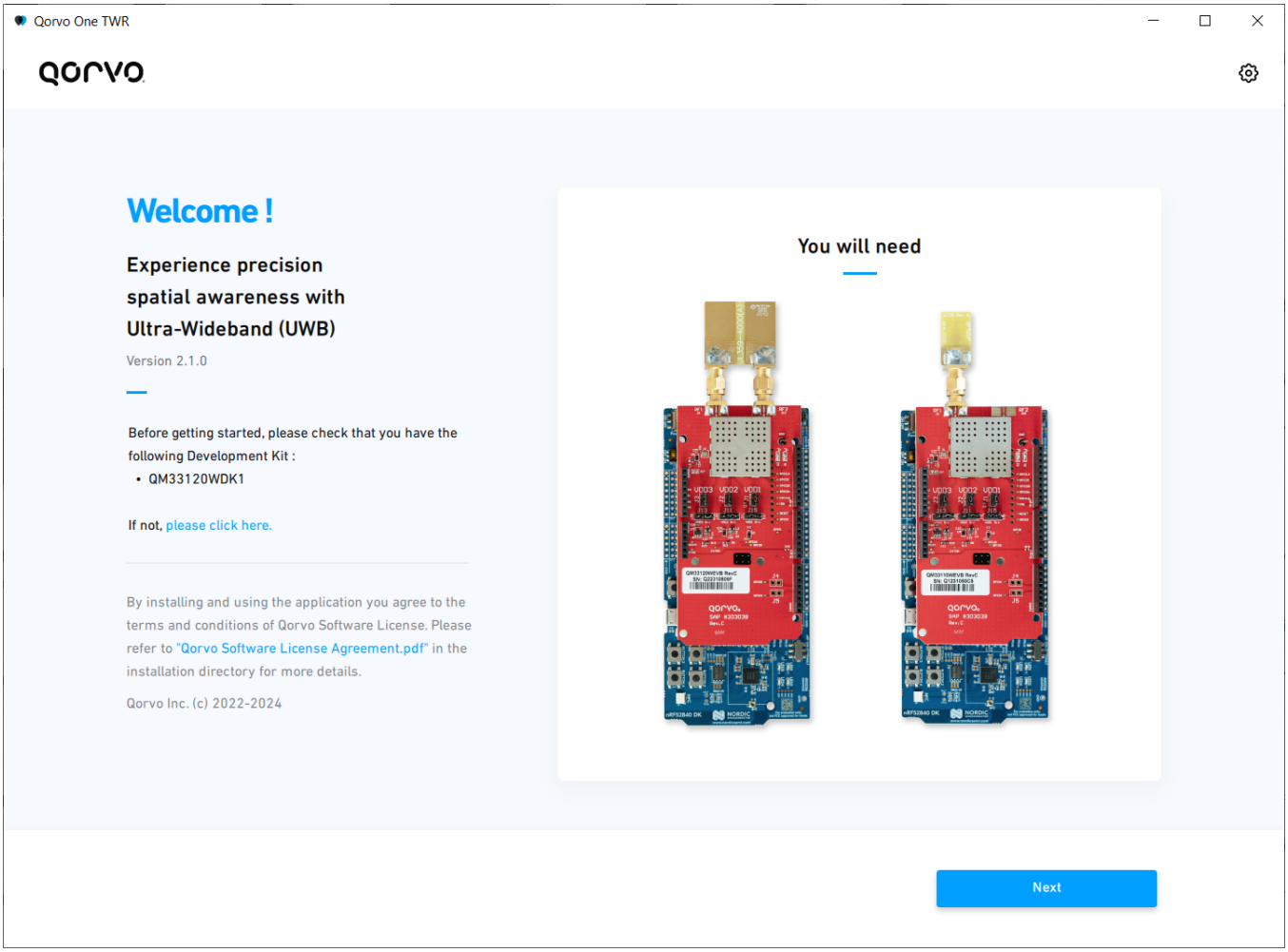

Fig. 6.6: Welcome screen.

The initial screen provides information about required tools and the actual version of the application. The license agreement can be viewed by clicking on the link.

6.2.3 Setup

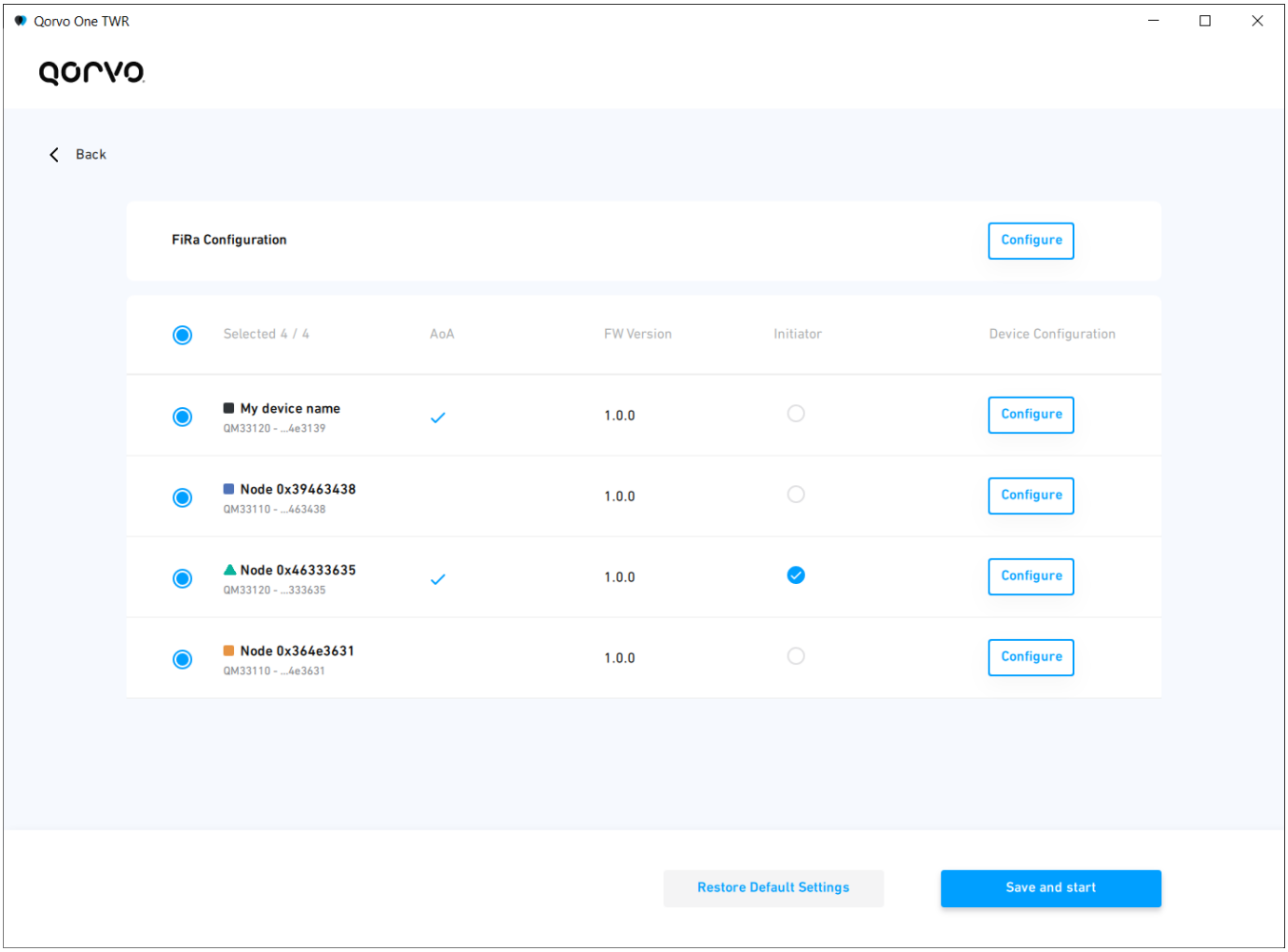

Fig. 6.7: Setup screen with the list of detected devices.

When the setup screen is displayed the application continuously scans all serial ports for new devices. Detected devices appear in the device list on the screen. Device that is disconnected from the host is removed from the list. Device can be deselected which will exclude it from the TWR session.

Double clicking the device name enables editing of the name. Default name is “Node” + unique ID as hexadecimal number. The part ID read from the device is used as the unique ID. If no part ID is provided, the application uses the serial number of the serial port or a hash of the port name respectively as the part ID.

Below the device name is the chip name and the unique ID as hexadecimal number where the chip ID takes the upper 4 bytes and the unique ID takes lower 4 bytes.

There can be at most one initiator (the controller) and rest of the devices are configured automatically as responders (the controlees).

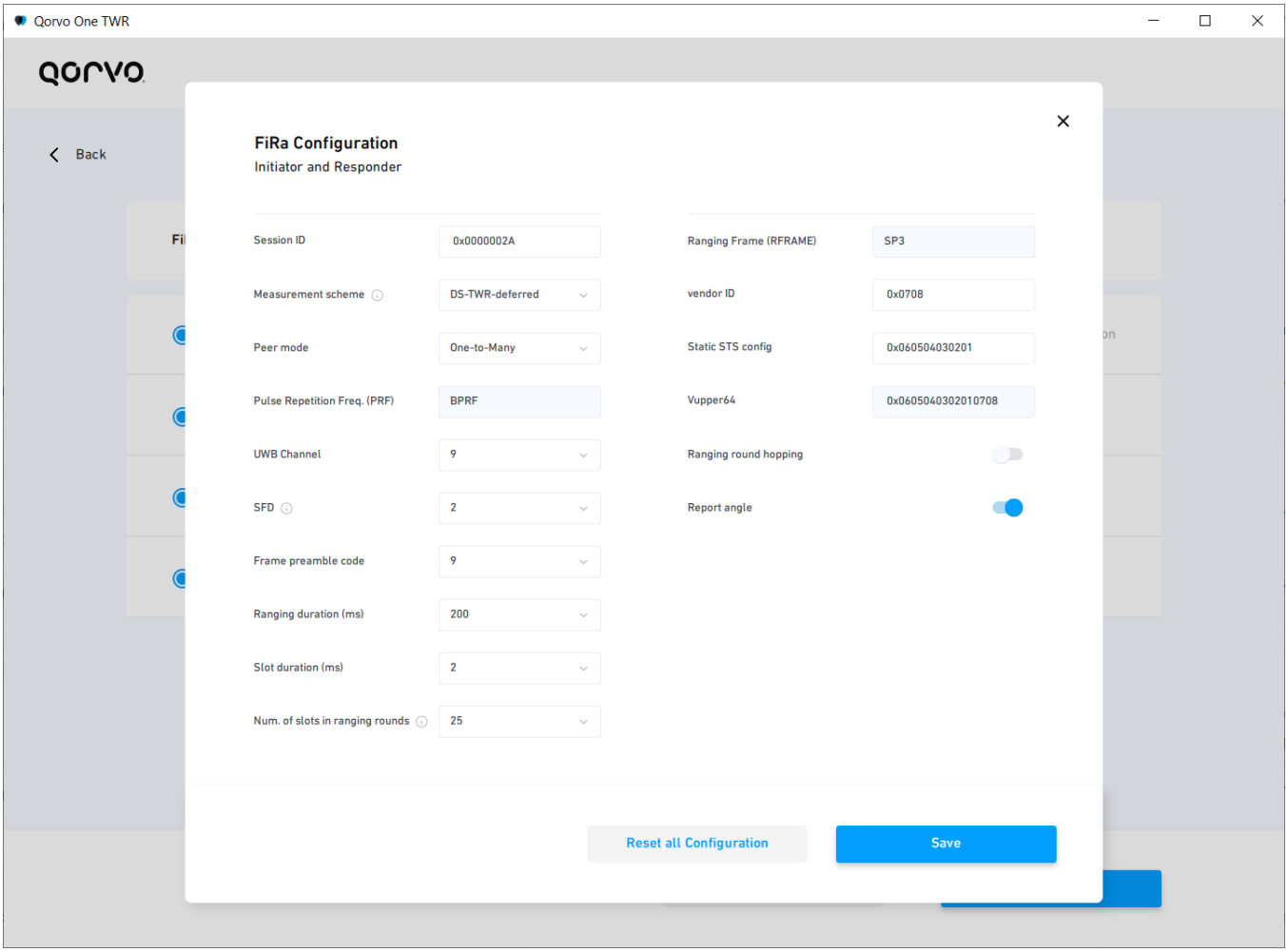

Fig. 6.8: FiRa configuration parameters.

The FiRa Configuration popup provides parameters defined in the FiRa[15] specification. The popup can be activated by the **Configure** button in the FiRa Configuration tab and the parameters are used for all the devices in the TWR session.

[15] https://www.firaconsortium.org/

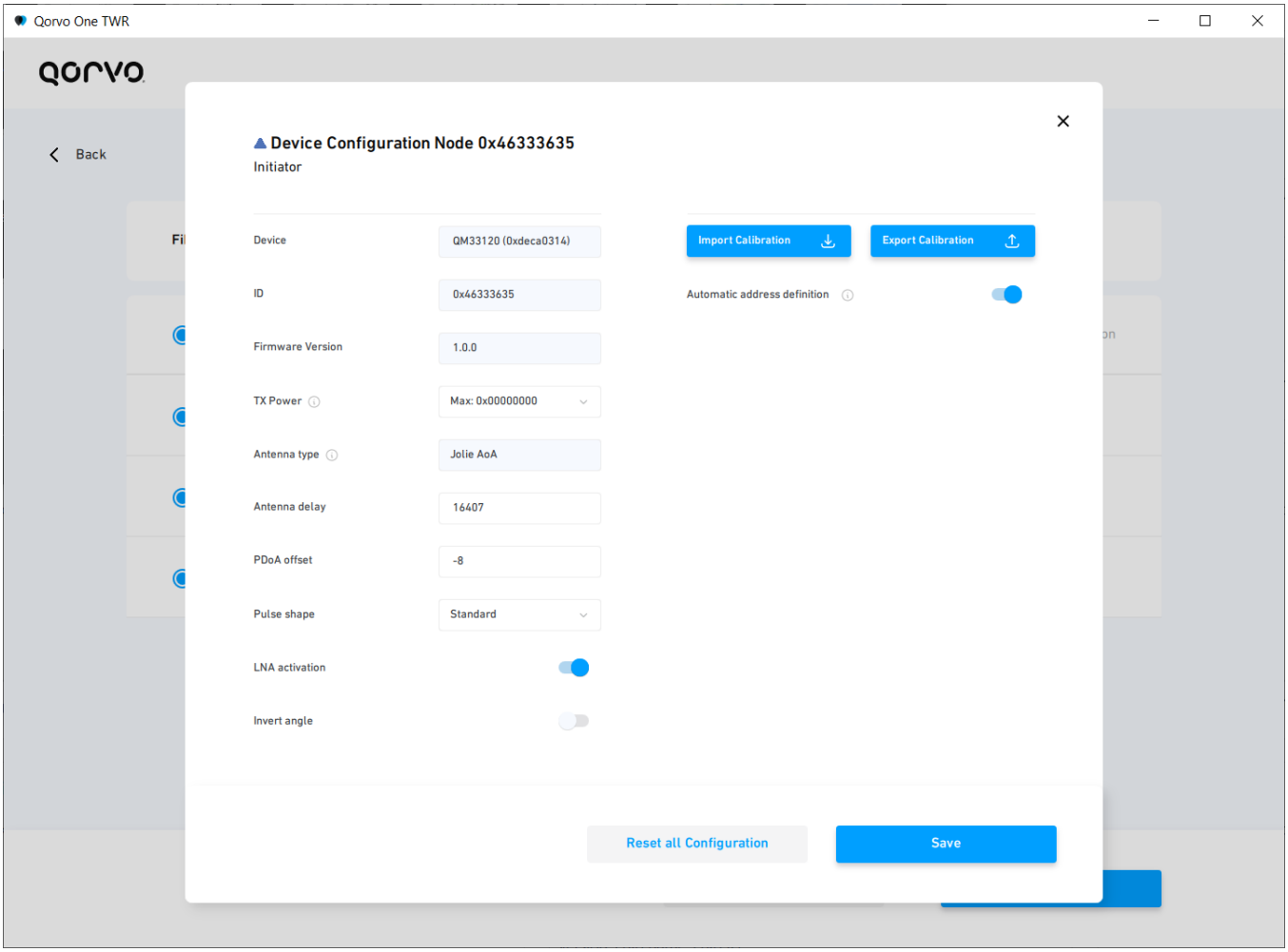

Fig. 6.9: Device-specific configuration.

Device-specific parameters can be configured in the **Device Configuration** popup. The popup is activated by the **Configure** button on the device tab. Some of the configuration widgets might not be visible for the device that does not support all of the API.

The content of the **Device** text box is the chip/device name + (chip/device ID as hexadecimal number).

The content of the **ID** text box represents the unique ID of the device.

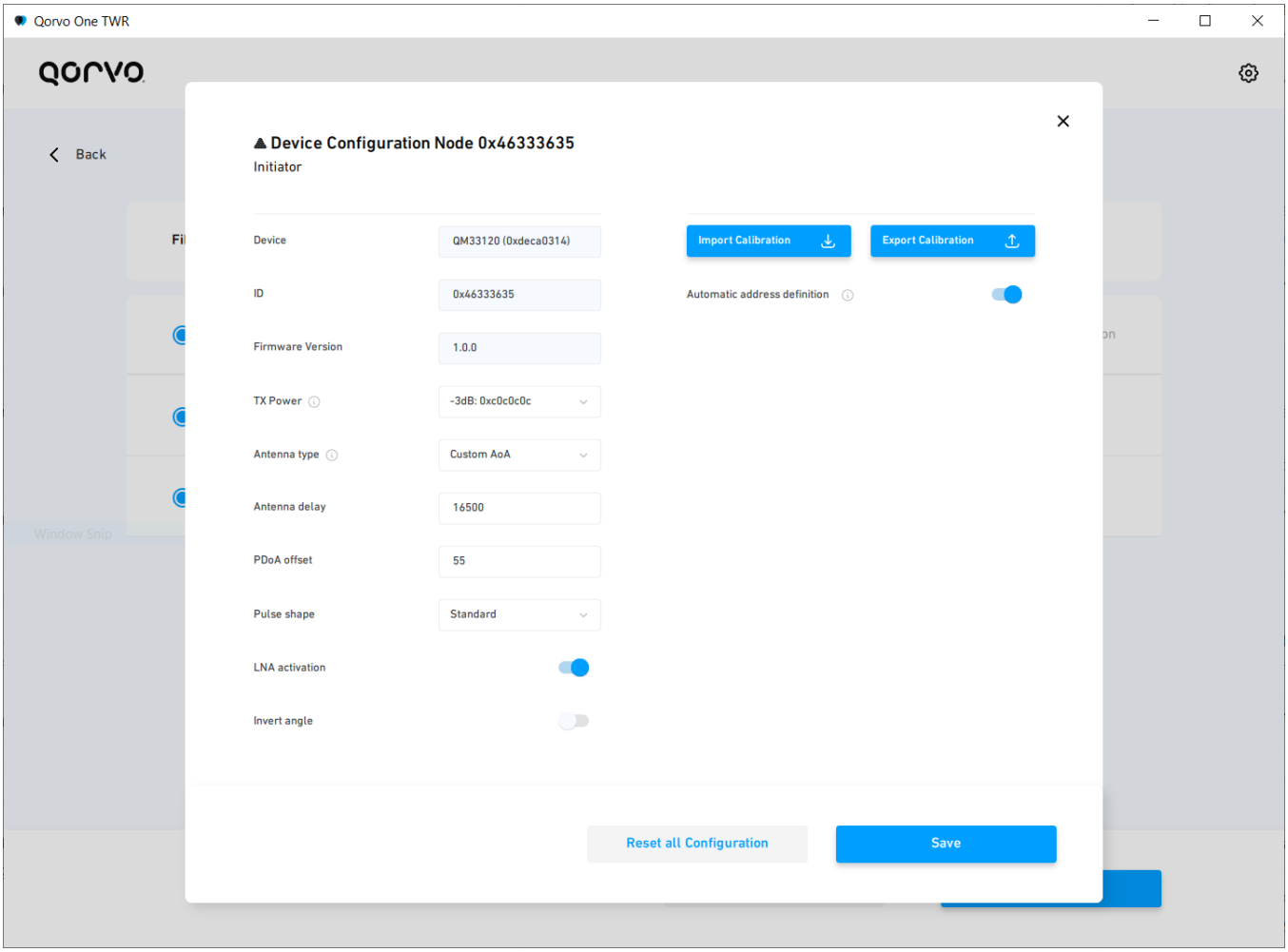

Fig. 6.10: Custom antenna configuration.

Antenna type can be **Custom** or **Jolie** for each of the UWB channels and depends on the actual PDoA look-up table (LUT) of the channel. The **Custom** represents configuration of the antenna imported by the user which LUT is different from the default **Jolie** antenna. Antenna delay and PDoA offset are stored along with the LUT to allow switching between the **Custom** and **Jolie** antenna configuration.

Suffix **AoA** or **non AoA** is added based on the AoA capability. Device not capable of the AoA has no LUT neither PDoA configuration options in the device configuration page.

Inverting the angle of the initiator will cause showing opposite angles of all the responders in the *Real-Time Location* screen since the AoA is measured on the initiator side. Inverting the angle of the responder has effect only in case when there is no local initiator and the AoA is read from the responders instead.

> **Warning:** Using the default calibration leads to decreased ranging accuracy (see section *Calibration important notes* for more details).

The calibration parameters can be exported or imported in the form of a JSON file. The imported file must respect calibration schema of the device to be successfully saved otherwise the import fails and user is notified by the state of the **Save** button. The button indicates busy state while the parameters are being saved into the device via the serial interface.

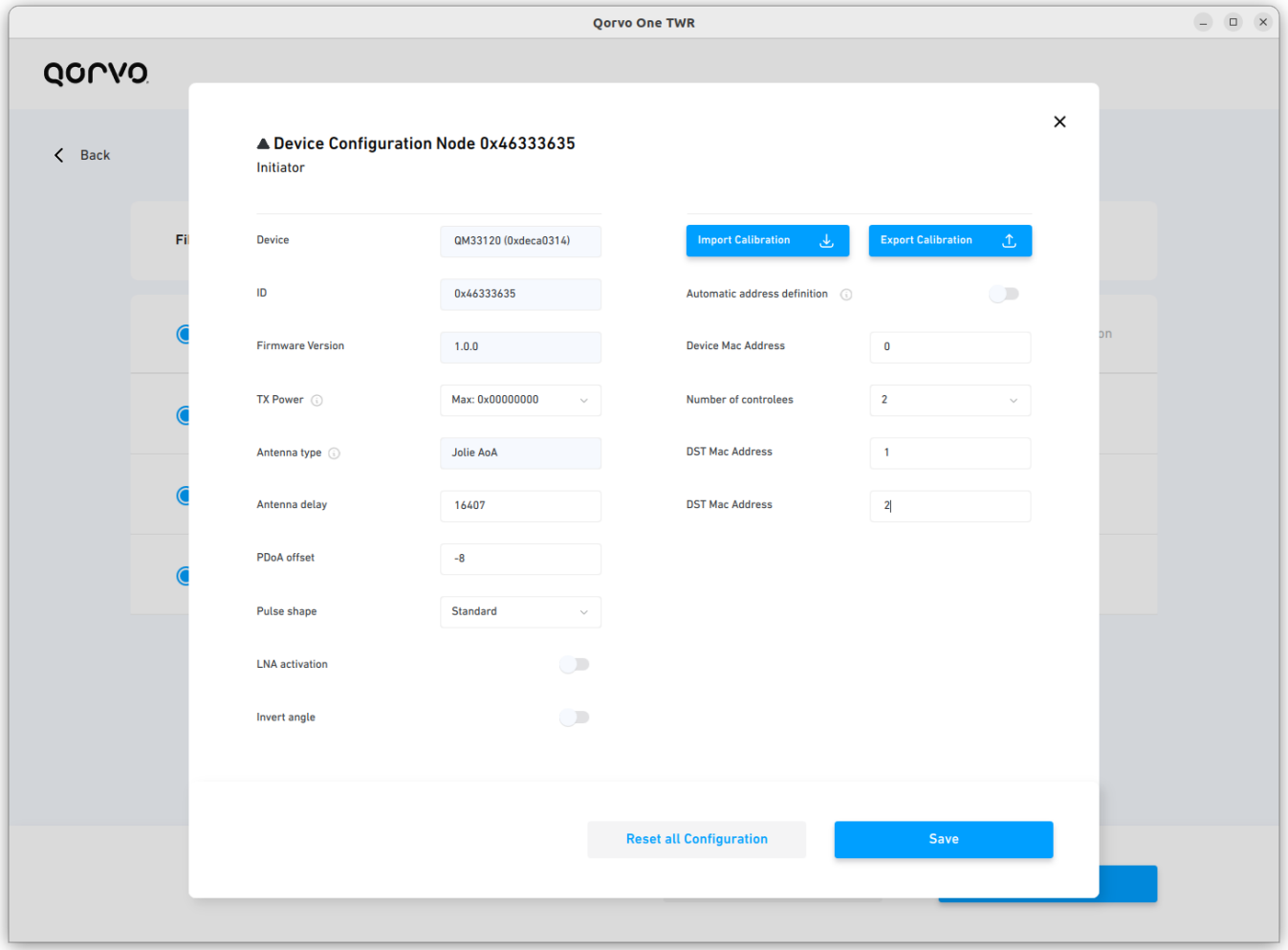

Fig. 6.11: Manual configuration of the MAC addresses.

The MAC address and the destination MAC address list as defined in the FiRa specification can be configured manually which is necessary when ranging with the devices that are controlled by the different host or application.

The configuration is applied by clicking the **Save** button or it can be reset to default by clicking the **Reset all Configuration**. Resetting all the configuration on the setup screen will reset the FiRa configuration, the configuration of all the devices, the floor plan, the grid and the geofencing configuration. The FiRa configuration can be reset separately on the *FiRa popup*. The configuration can be reset per device using *Device Configuration popup*.

The TWR session starts by clicking the **Save and Start** button.

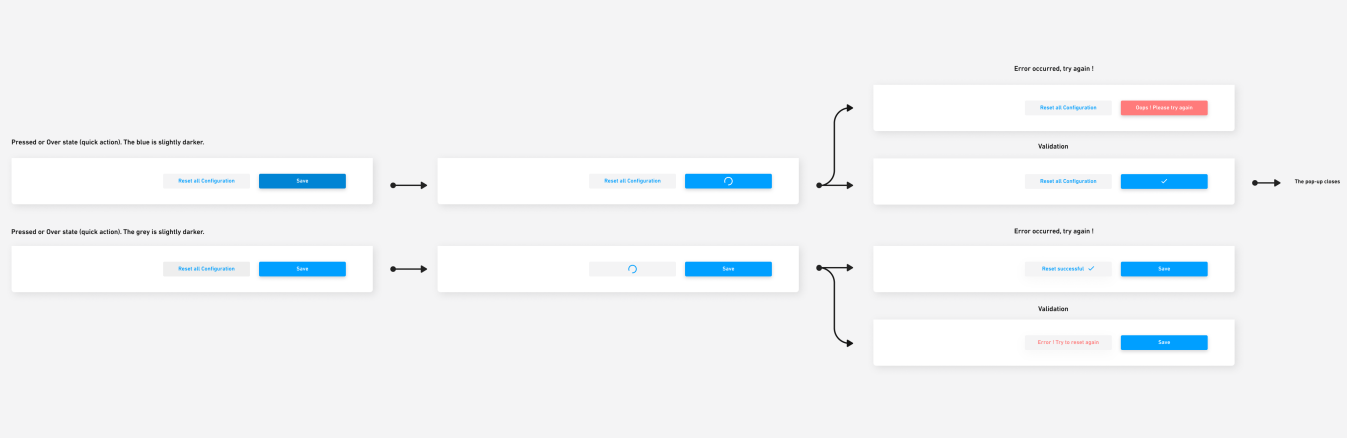

Fig. 6.12: Possible states of the buttons.

6.2.4 Real-Time Location

The screen demonstrates the TWR between the devices where the initiator is at the top of the screen at fixed position and the position of the responders is being updated. The card with information details can be enabled or disabled by clicking on the responder. Clicking on the card can bring it forward. If the initiator is one of the local devices the position of the responders is shown as measured by the initiator.

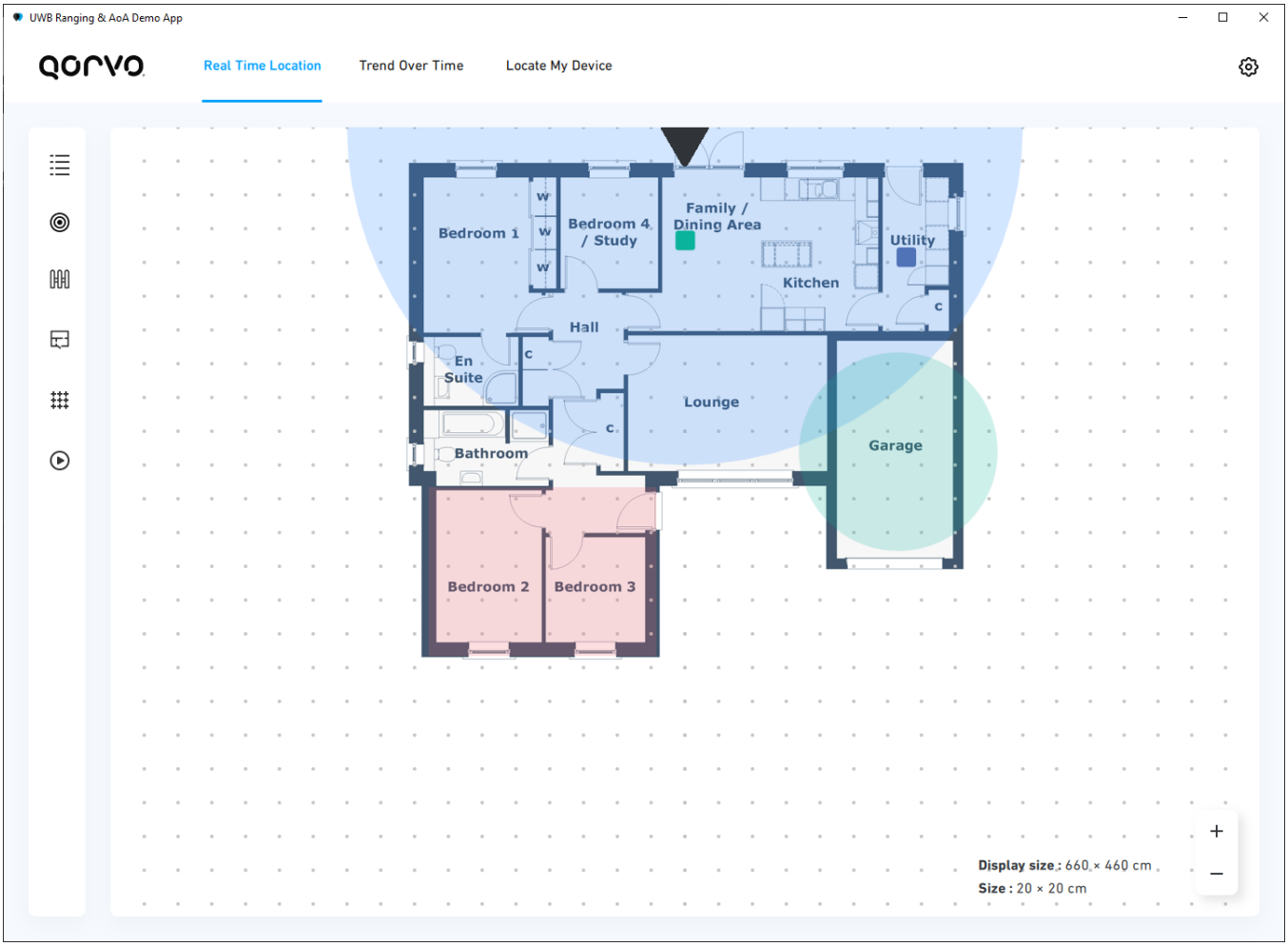

Fig. 6.13: TWR between 1 initiator and 2 responders.

If none of the local devices is the initiator, the position to the unknown initiator is the distance and angle measured by each of the responders. The unknown initiator is shown as grey polygon at fixed position at the top of the screen.

The left bar allows to configure the *Grid*, *Floor Plan* and *Geofencing* zones shown on the screen. TWR can be stopped by clicking the icon on the right side of the top bar.

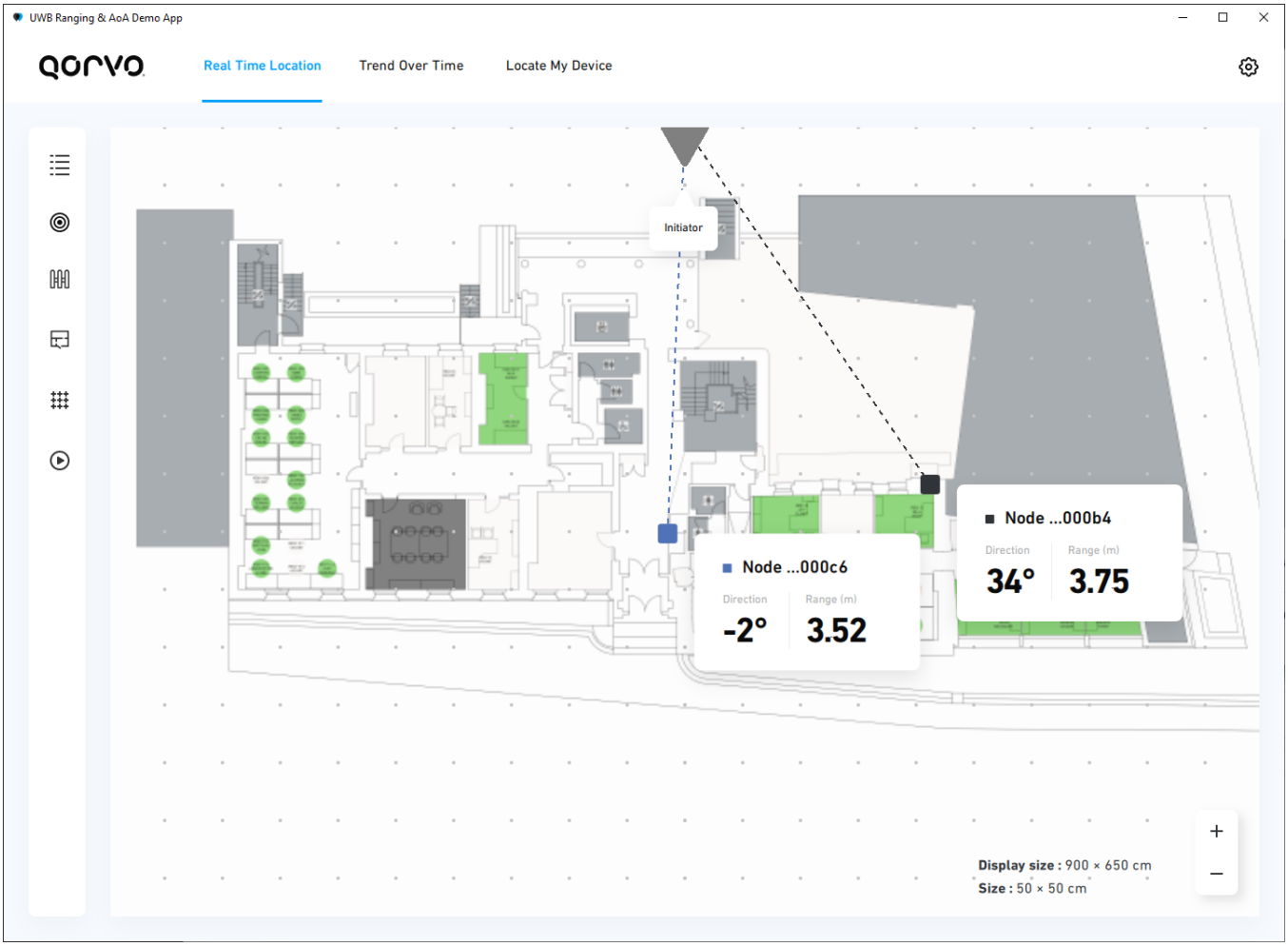

Fig. 6.14: Ranging with the initiator that is on different host.

6.2.5 Trend Over Time

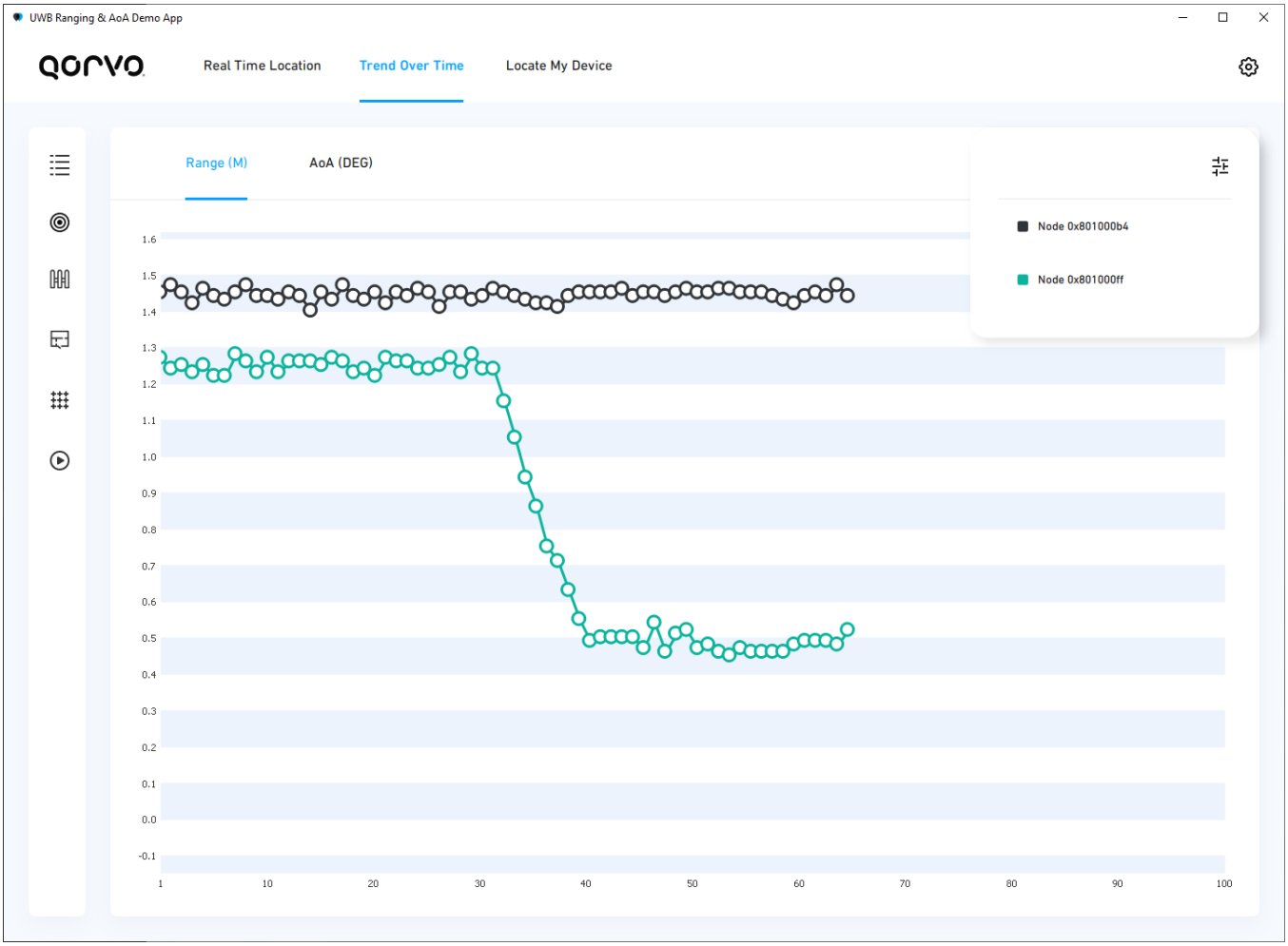

Fig. 6.15: Distance samples of the responders.

The graph shows samples of either distance (Range) in meters or angle (AoA) in degrees for each responder. The responder data can be hidden on the graph by clicking on the card in the top right corner.

6.2.6 Locate My Device

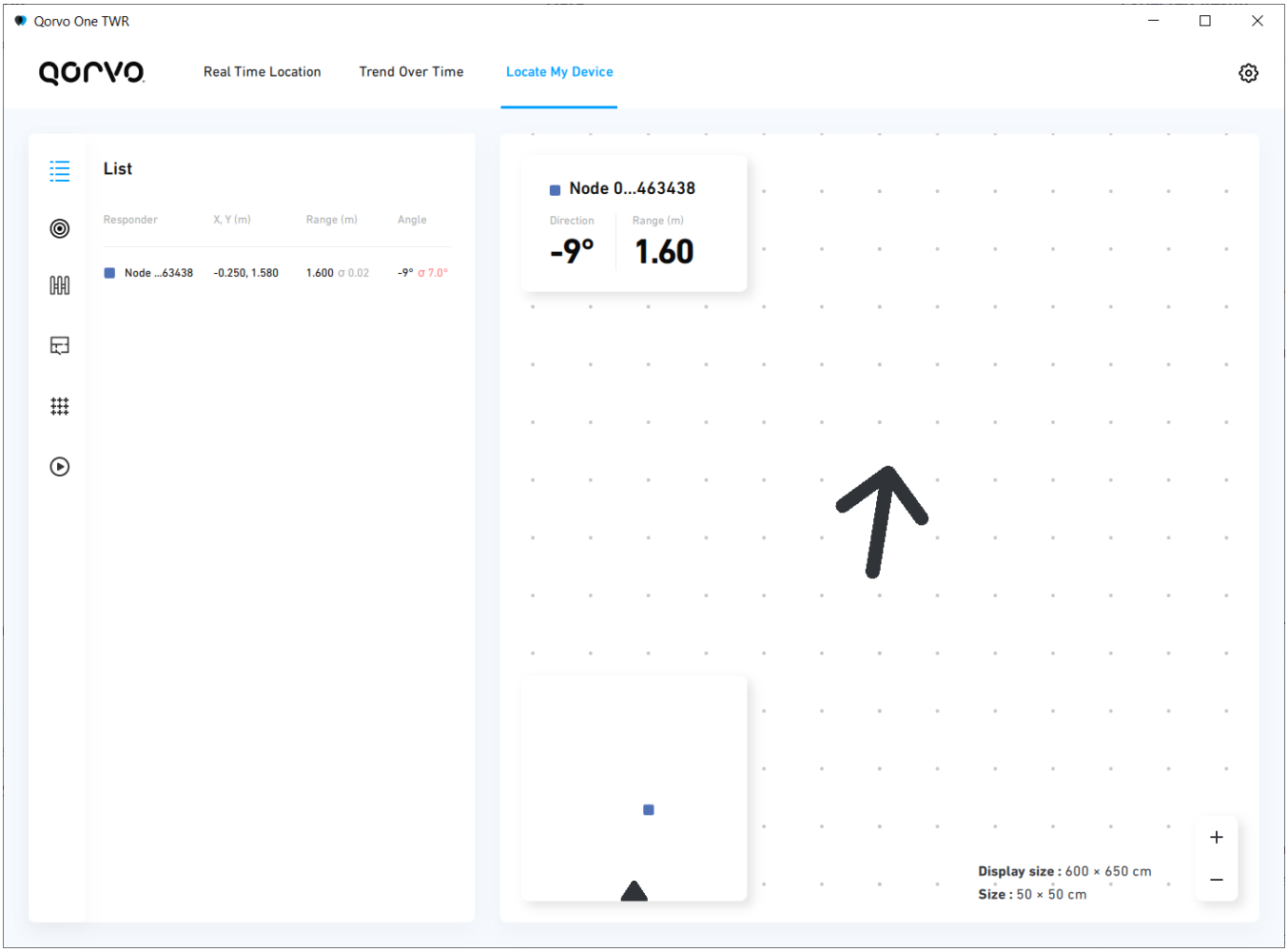

Fig. 6.16: The “Locate My Device” screen.

The angle of the arrow in the middle of the screen points in the direction of responder that can be selected by the combo box in the top left corner. The minimap in the bottom left corner shows position of all the responders.

6.2.7 Device List

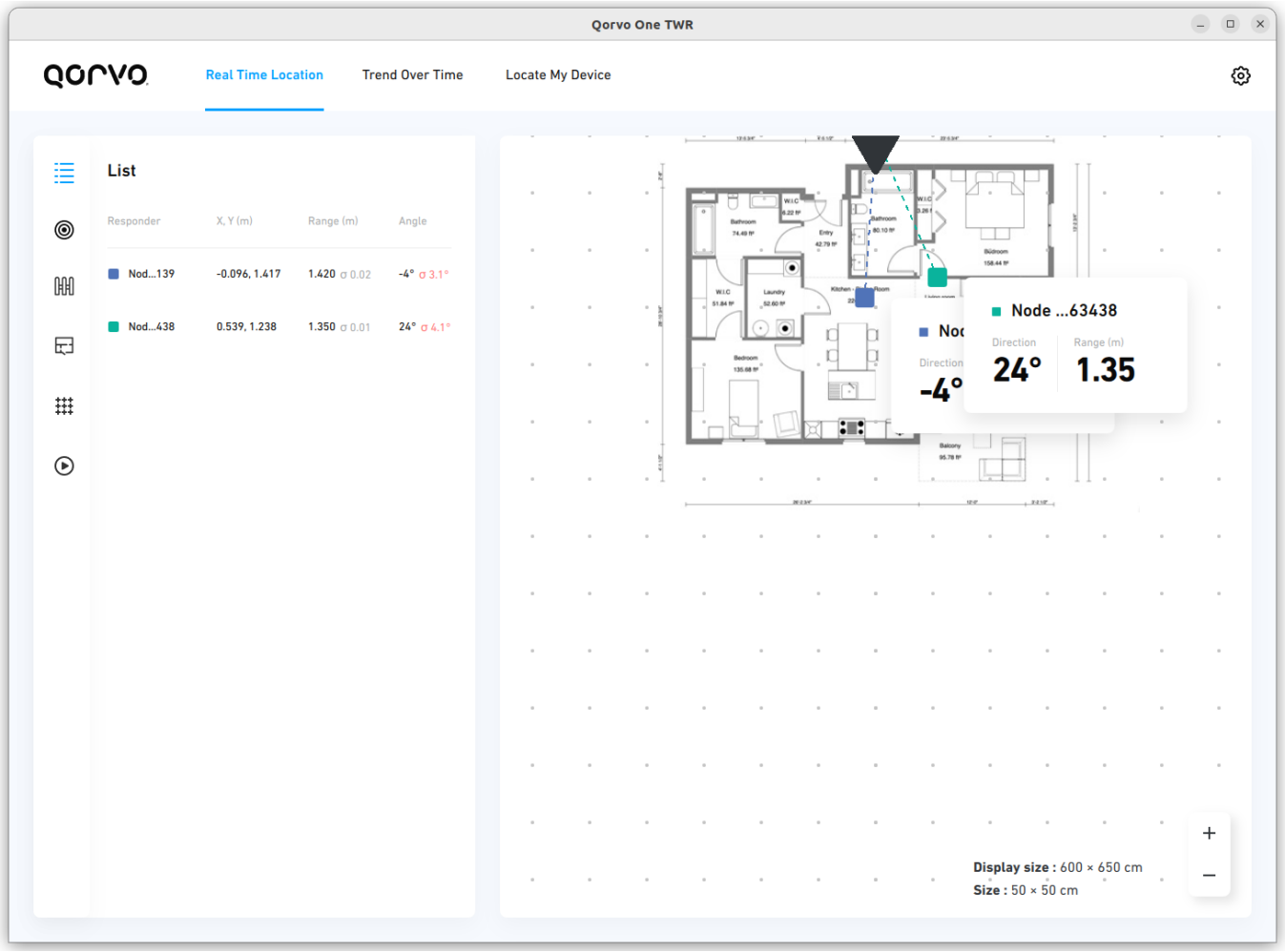

Fig. 6.17: List of the responders.

The list of the responders can be displayed by activating the **List** card on the left bar. The x, y coordinates and the polar coordinates are shown for each responder as well the standard deviation of the distance and angle. Responder can be identified by the color and by the name which can be edited by double clicking.

The standard deviation σ changes color to red when the corresponding value exceeds limit. The limit for the standard deviation for the distance is 0.03 meters and 3 degrees for the angle.

6.2.8 Calibration

Devices can be calibrated using the **Calibration** card on the left bar. The calibrated device has to be placed at 2 meters distance from the device that will be used as the peer (the calibrator) during the procedure. The responder always uses the initiator as the calibrator.

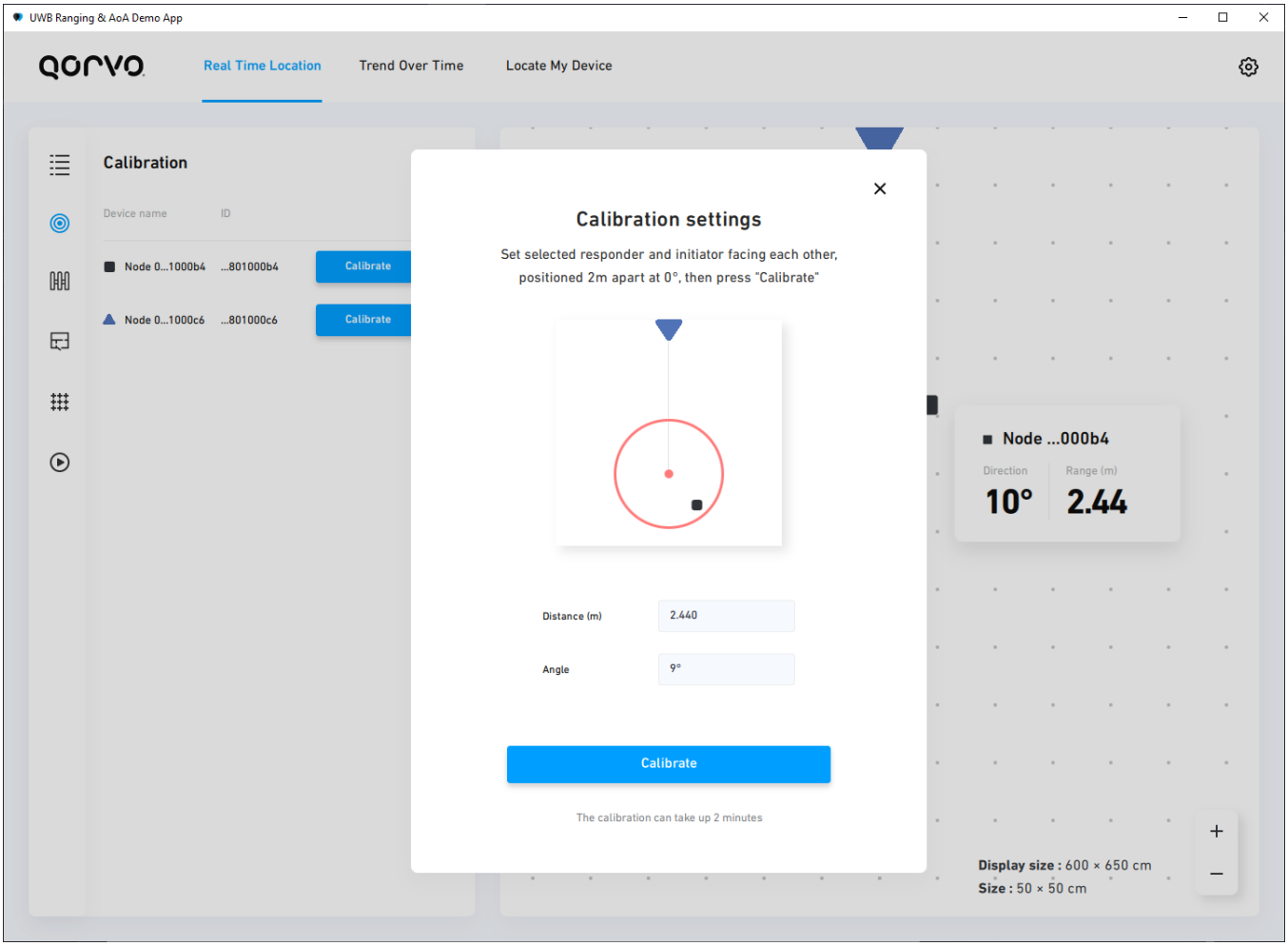

Fig. 6.18: Calibration popup.

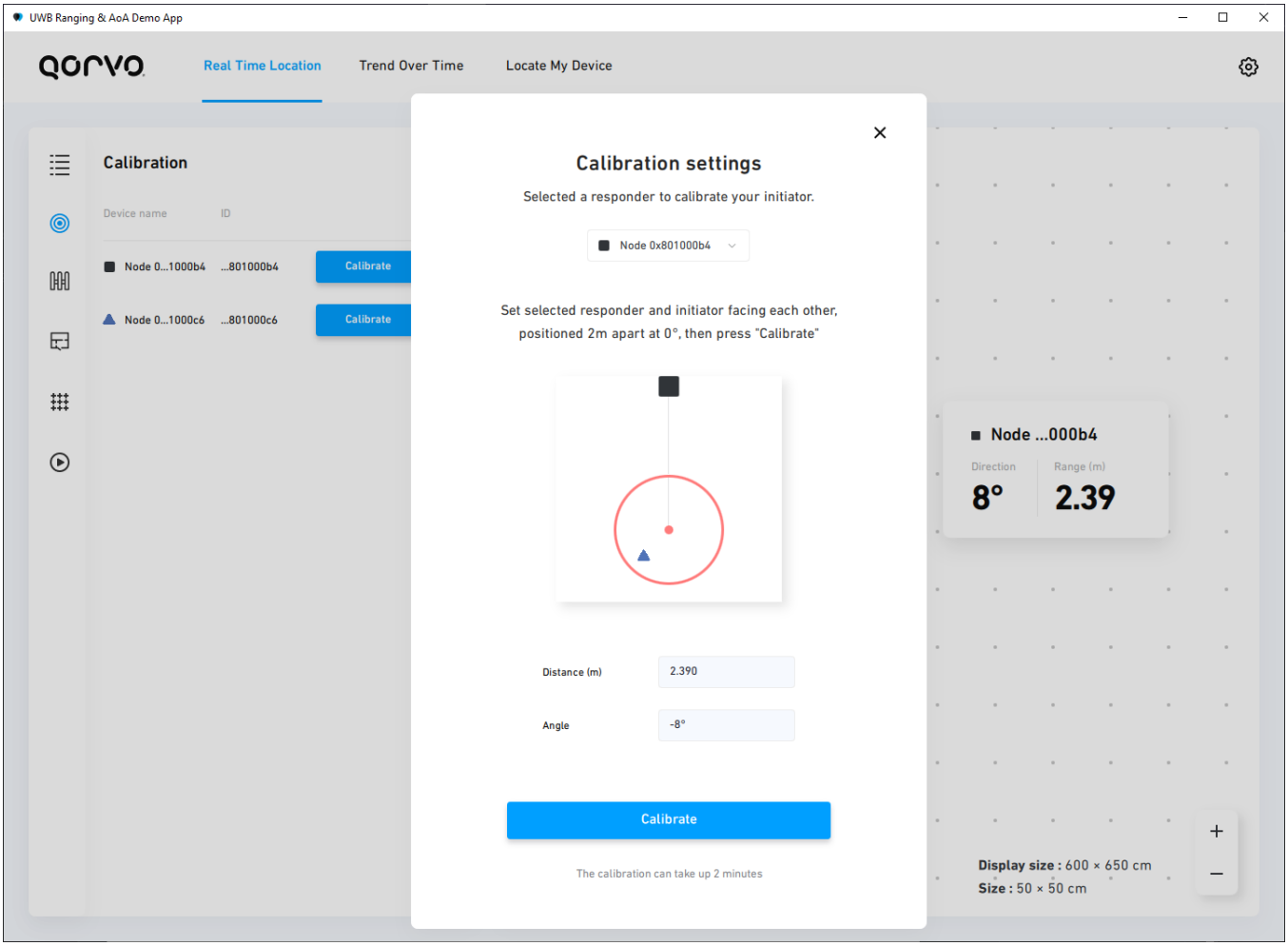

Fig. 6.19: Calibration of the initiator.

Initiator can choose the calibrator from one of the responders that are connected locally to the application. A device that is connected to the distant host cannot be used for calibration.

The time the calibration procedure can take depends on the ranging duration of the *FiRa configuration*.

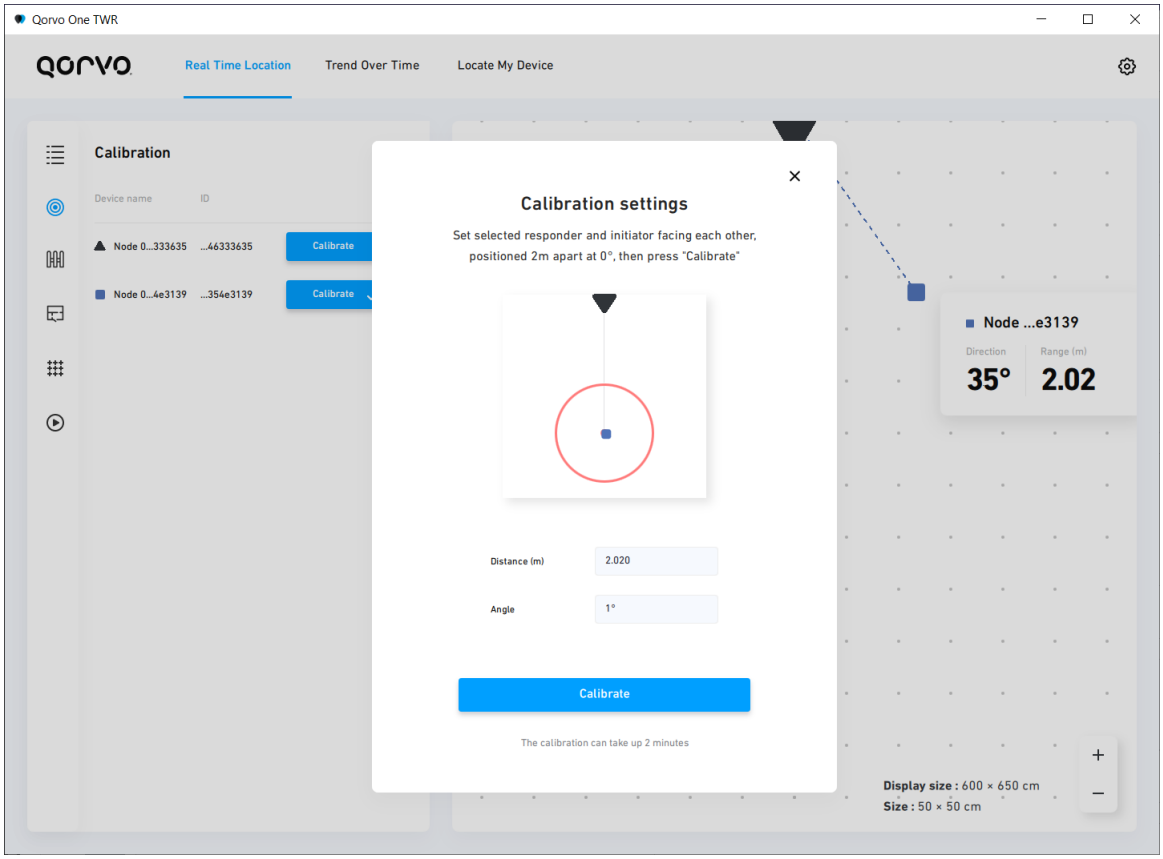

Fig. 6.20: Successful calibration.

The state of the calibration is displayed by the button. Closing the popup will stop the ongoing calibration. The calibration can fail either due to timeout or due to communication error with the devices.

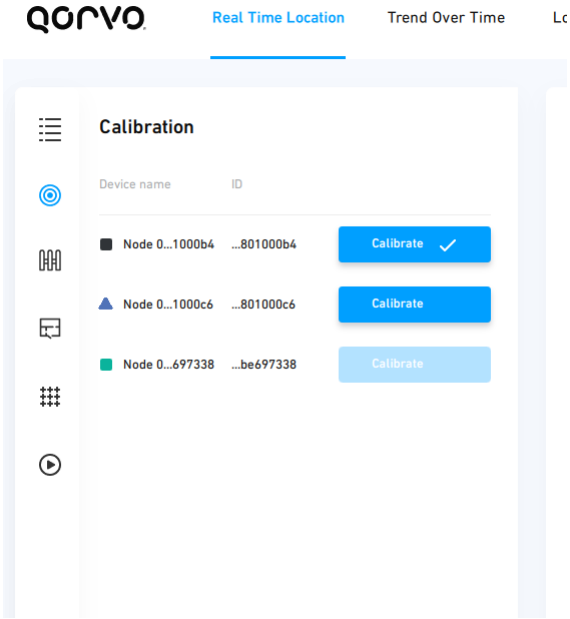

Fig. 6.21: List of devices for the calibration.

Device that has been already calibrated is marked by the check icon in the calibration button. Device that does not support calibration have the button disabled.

6.2.9 Geofencing

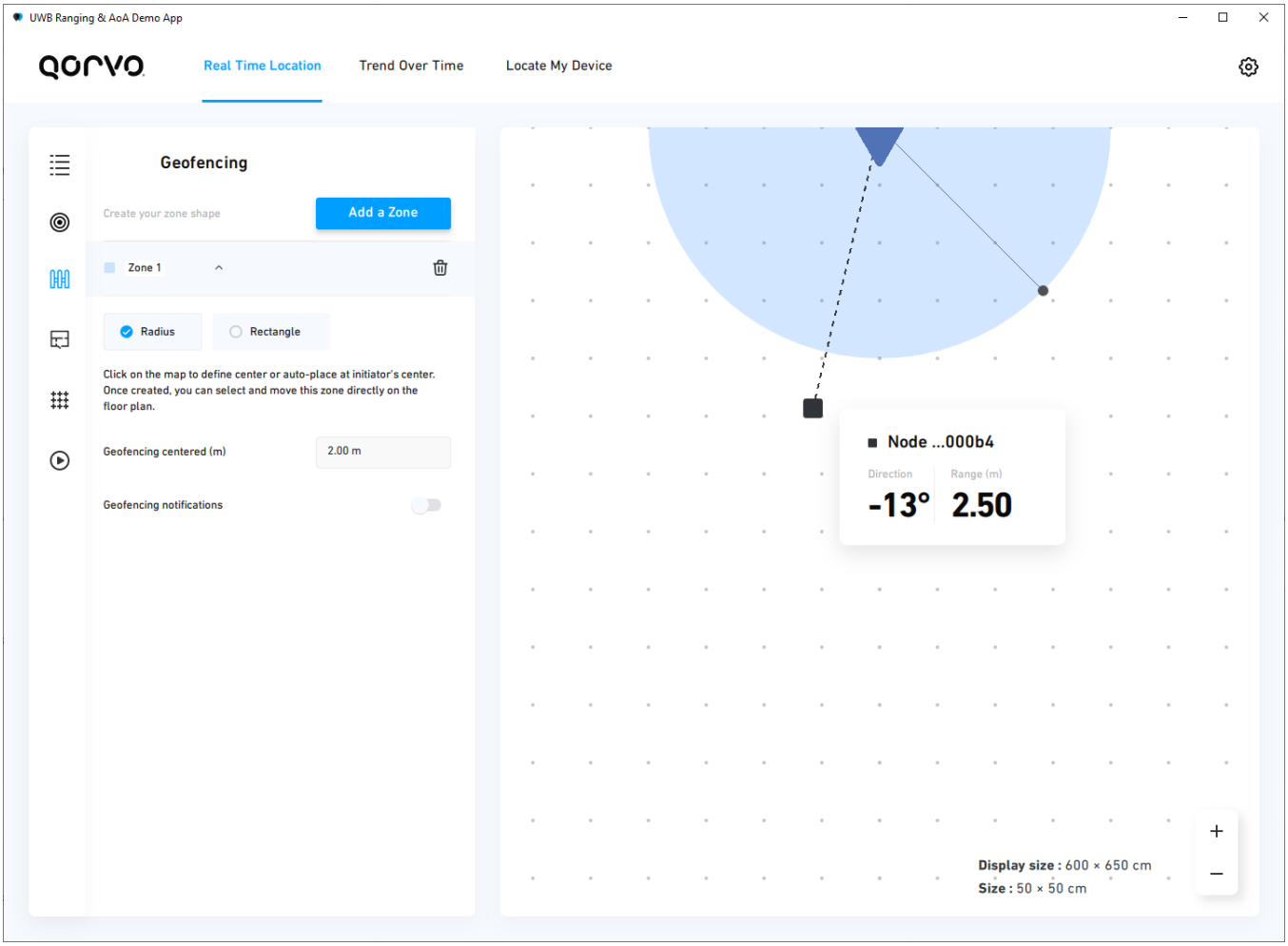

Fig. 6.22: Radius zone.

Up to 5 geofencing zones can be added to the *Real-Time Location* screen. To add or to remove the zone, click the **Geofencing** icon on the left bar. The radius zone is created either by clicking twice on the screen to place the center and the radius or by writing the radius value in meters into the dialog in which case the center of the zone is placed automatically at the initiator.

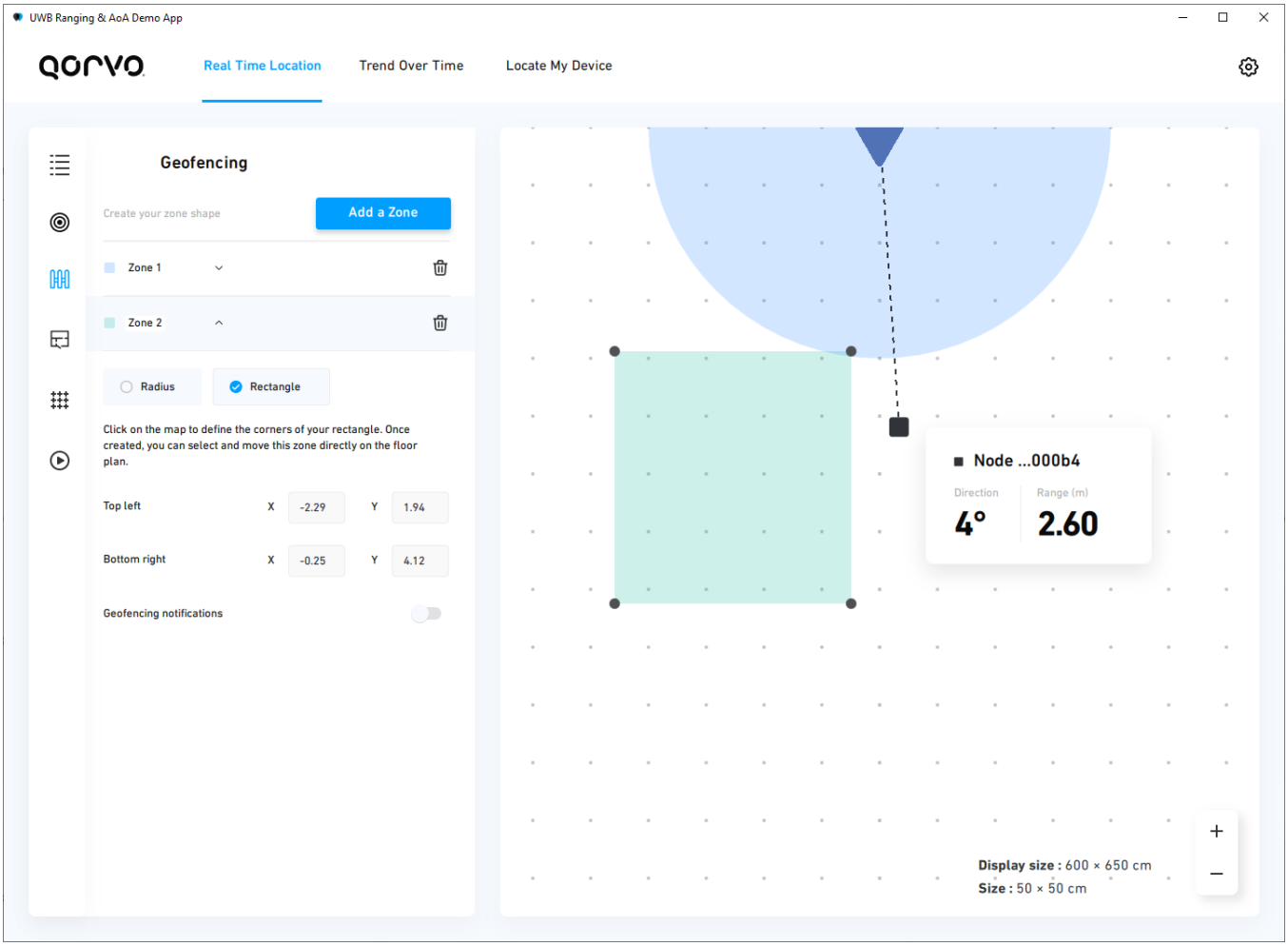

Fig. 6.23: Rectangular zone.

Rectangular zone can be created by clicking on the screen to define top left and bottom right corners or by specifying the coordinates in the text edit. Each zone can be resized, moved or configured when activated by clicking on the icon next to the zone name. The name of the zone can be edited by double clicking on the name.

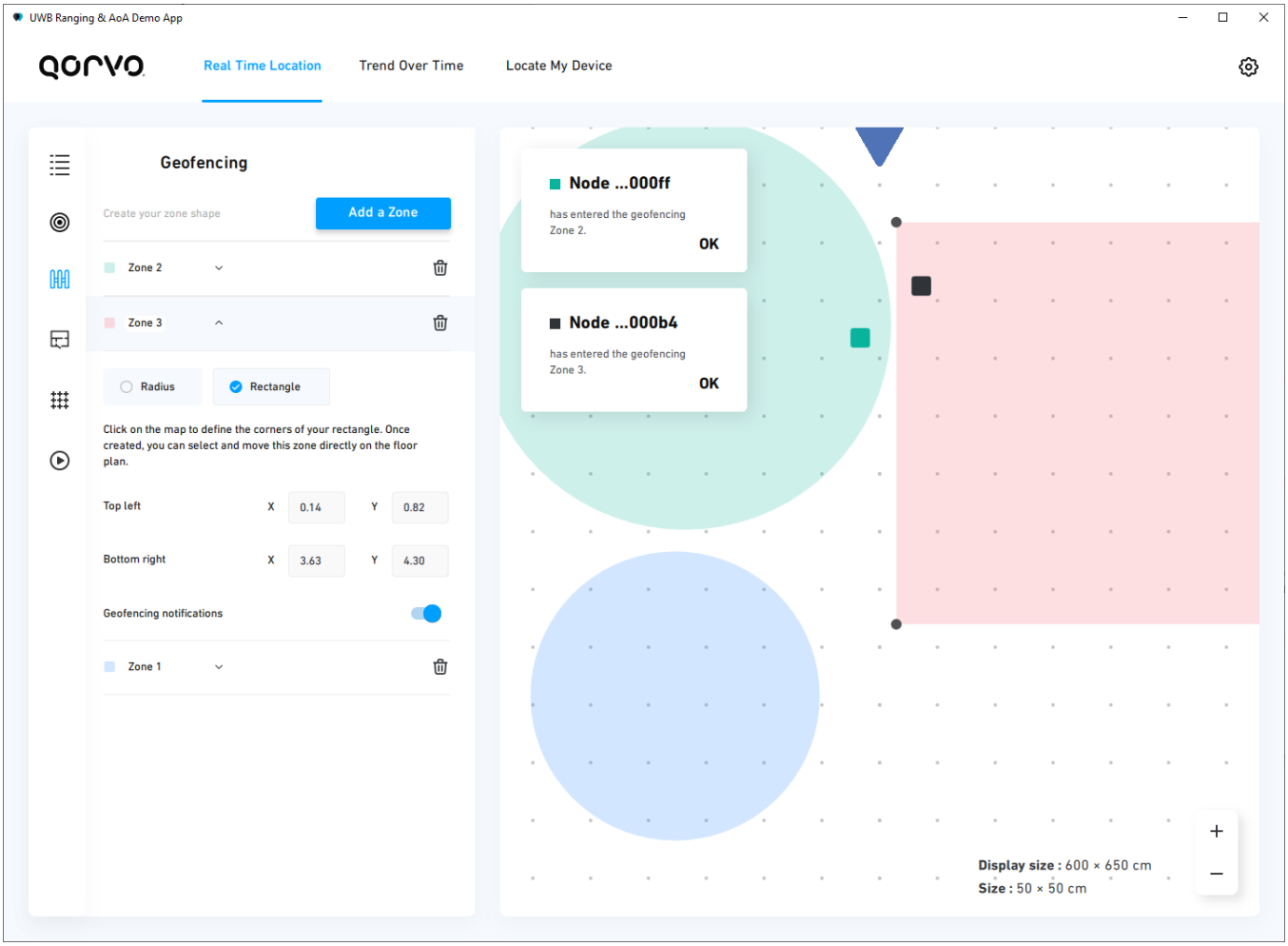

Fig. 6.24: Geofencing notifications.

The notifications about the responders entering the zone can be enabled by clicking the button in the zone card. The notification disappears automatically after 3 seconds.

6.2.10 Floor Plan

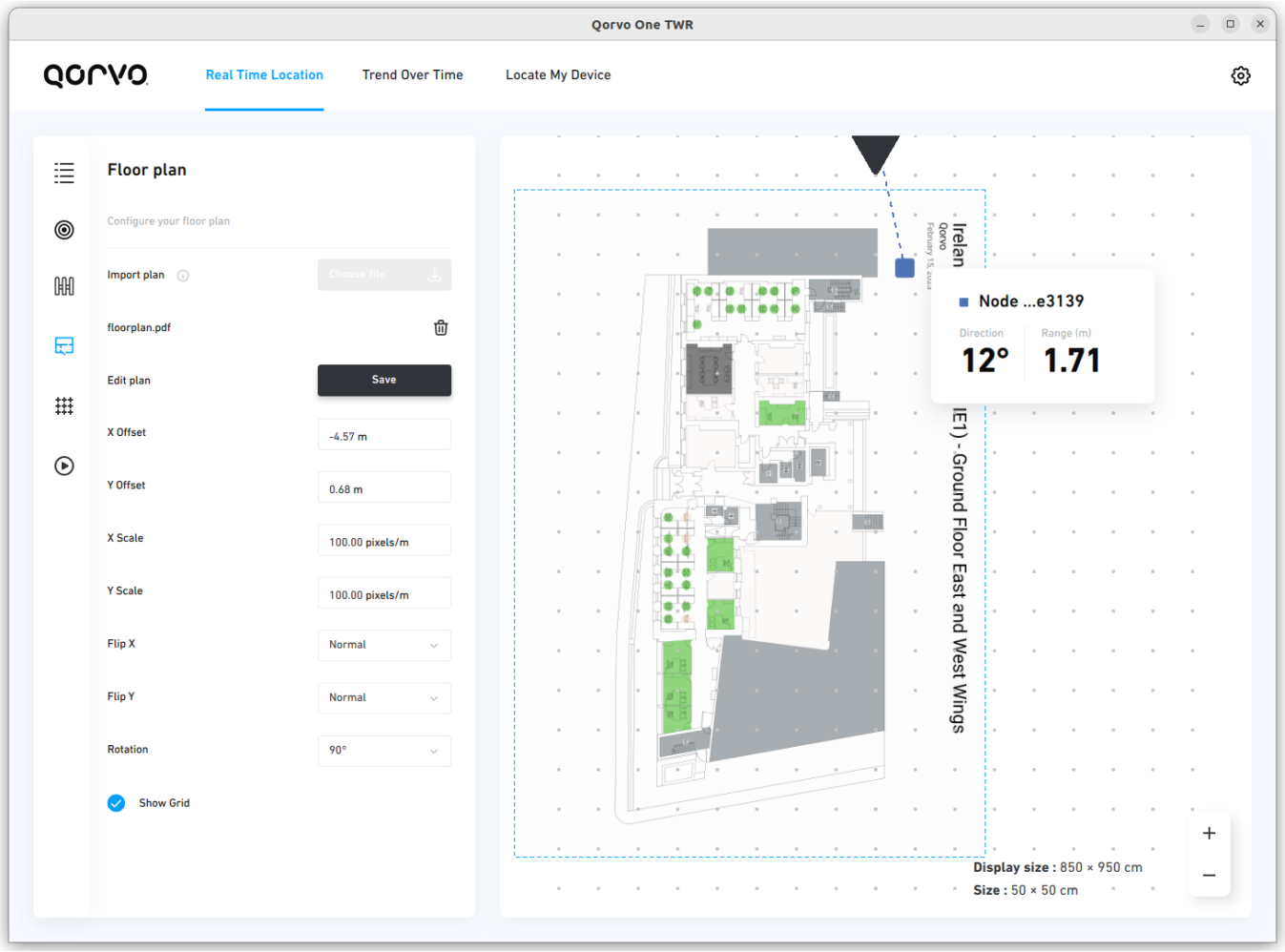

Fig. 6.25: Floor plan configuration.

The application supports following file types to be imported as floor plan: png, jpeg, jfif, jpg, jpe, tiff, tif, pdf, gif, bmp. The maximum file size is limited to 50 MB. The plan can be resized, flipped and rotated. The imported image is displayed on the *Real-Time Location* screen. The plan can be deleted by clicking on the trash icon.

6.2.11 Grid

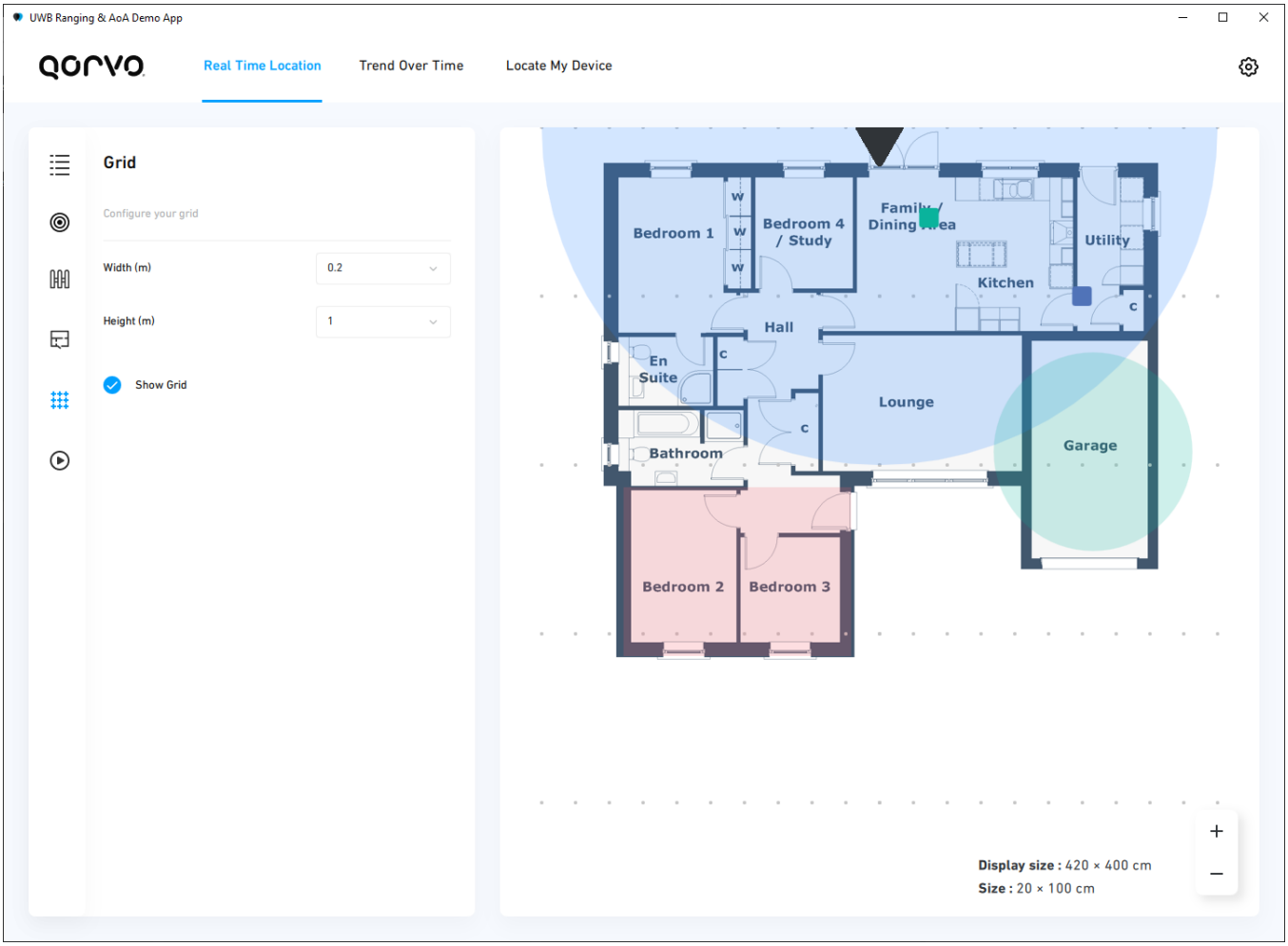

Fig. 6.26: The grid configuration.

The **Grid** card on the left bar allows the grid density and visibility to be configured. The grid is displayed on the *Real-Time Location* screen and on the *Locate My Device* screen. Total size of the grid and the grid resolution is shown in the bottom right corner.

The screen can be zoomed in and out either by clicking on the +/- button in the bottom right corner or using the mouse wheel together with the `Ctrl` key.

6.2.12 Logging

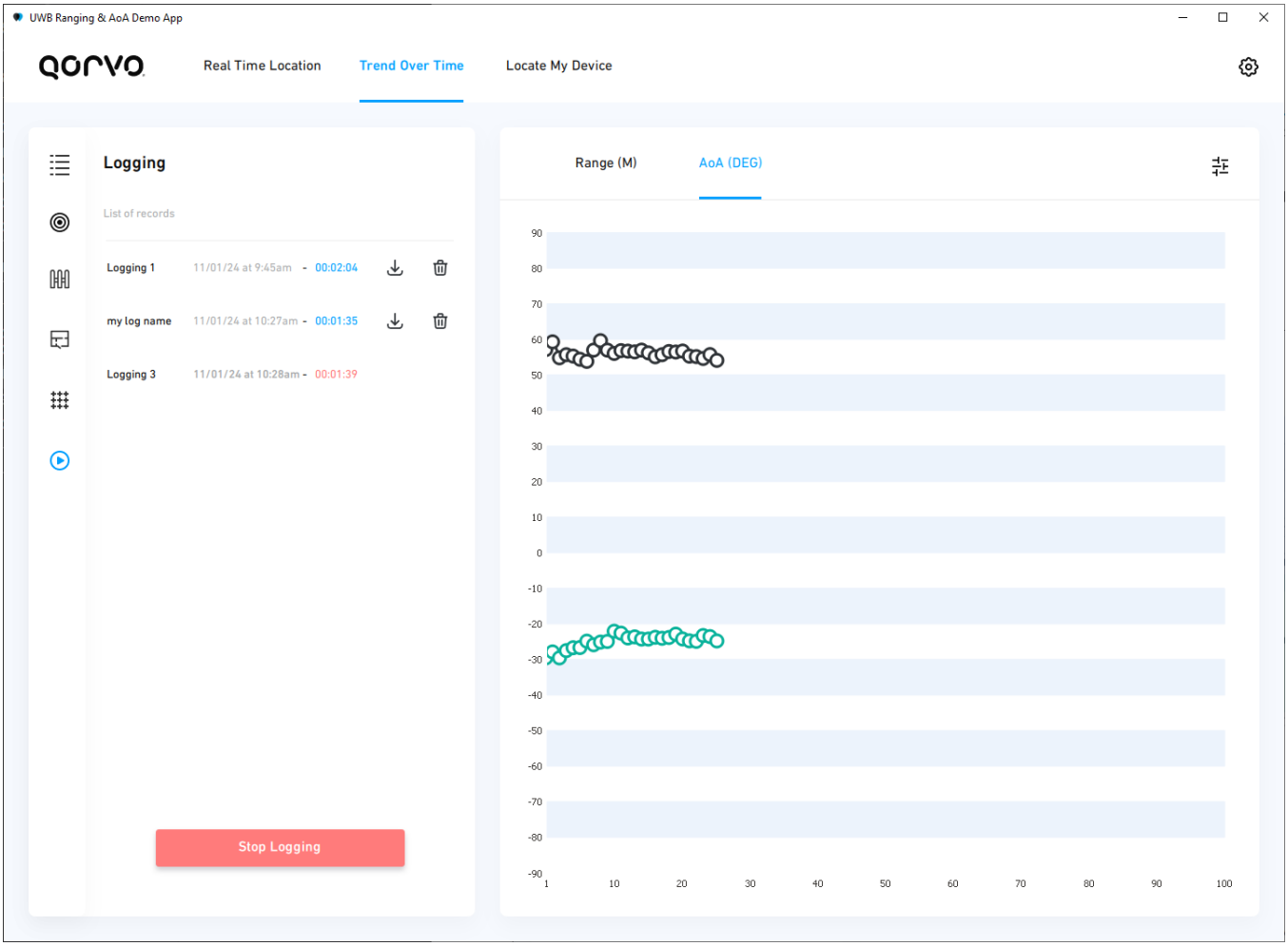

Fig. 6.27: List of logging records.

Logging is started automatically by the application when opened. New logging record can be appended to the list by clicking on the **Start Logging** button. The logging record that is not currently active can be deleted or exported by clicking on the icons in the tab.

The logging is exported as two files. One file contains the UCI communication frames. The second file contains the main logging messages. Name of the logging record can be edited by double clicking on the name.

7 UWB Qorvo Tools

7.1 Overview

The `UWB Qorvo Tools` gathers various tools provided by Qorvo for operating Ultra-Wideband (UWB) devices through the UCI (UWB Communication Interface). These tools are located in the `Tools/uwb-qorvo-tools` directory.

Warning: The `UWB Qorvo Tools` are delivered by Qorvo as UCI examples on an as-is basis and are not optimized nor meant to be used as production software.

Note: A separate documentation is provided specifically for the `UWB Qorvo Tools`. For information on how to use the `UWB Qorvo Tools`, please refer to this documentation located in `Tools/uwb-qorvo-tools`.

Warning: If you are not using a QM33120WDK1, DWM3001CDK or Type2AB EVB (e.g. DWM3000EVB), you must flash the calibration before starting a ranging session.

Warning: Using the default calibration leads to decreased ranging accuracy (see section *Calibration important notes* for more details).

8 UCI - UWB Command Interface

8.1 Overview

In UCI build, UCI will be the only available application and active by default.

For details about UCI protocol, please refer to:

- UCI[16] generic specifications can be found in FiRa[17] website (membership is needed to download the documents),
- `uwb-uci-messages-api` document.

[16] https://groups.firaconsortium.org/wg/members/document/folder/96
[17] https://www.firaconsortium.org/

9 SDK Runtime

9.1 Always running Threads

Once RTOS kernel has been started, the following threads are always running:

- Default Task
- Control Task
- Flush Task
- Logger Task
- Tmr Svc Task
- Idle Task

9.1.1 Default Task

The Default Task is responsible for starting one of the top-level applications. It receives events from the Control task to switch to a particular mode of operation and starts corresponding top-level application.

The Default task waits for a global event from EventManager, with a MESSAGE_QUEUE_TIMEOUT_MS (200ms) timeout.

- If an event is received within the timeout, Default task stops all running top-level tasks and it starts the requested top-level application.
- If no event is received within the timeout, Default task executes the USB_VBUS driver, which checks whether the VBUS pin of the MCU is attached to a power source or not.

This task is executed endlessly.

9.1.2 Control Task

The Control Task is responsible for reception and execution of commands from external IO interfaces USB and/or UART.

On reception of a valid command by the Control Task, the command is processed, and the corresponding reply is sent for output by triggering Flush task.

9.1.3 Flush Task

The Flush Task is responsible for transmitting any output data to the external I/O interfaces USB and/or UART. Any functions (including ISR functions) or tasks can request to send data to the non-blocking output (USB and/or UART).

When **port_tx_msg** function is called, the data is copied to an intermediate Report Buffer, which is a statically allocated circular buffer. The size of Report buffer is sufficient that any task/function can send a chunk of data for background output without delaying its throughput, even during an ISR.

The Flush task is then emptying the Report buffer onto the USB and/or the UART.

9.1.4 Logger Task

The Logger Task is a low-priority task which periodically calls *NRF_LOG_PROCESS()* function. It is mainly responsible for processing and transmitting log entries from a queue of deferred logs and flushing processed logs into configured backends (RTT by default). The task is running when *CONFIG_LOG* and *NRF_LOG_ENABLED* are defined.

9.1.5 Tmr Svc Task

The Timer Service Task manages software timers in FreeRTOS. It is created on scheduler start and used by FreeRTOS kernel.

9.1.6 Idle Task

The Idle Task is a low-priority task that runs when no other tasks are ready to execute in FreeRTOS. It handles system housekeeping functions like freeing memory from deleted tasks and can enter low-power modes to save energy. Users can also add custom background operations via an idle hook function.

9.2 Application specific Threads

9.2.1 Listener Task

The Listener Task is running only when Listener application is started in CLI build. It runs Listener application to manage exchange of data between DWT UWB drivers and the host machine. For more detailed information about application, please refer to *UWB sniffer: LISTENER*.

9.2.2 LLHW MCPS Task

LLHW MCPS Task is used to handle signals and callbacks between LLHW and MAC layer. Task is created on *llhw_mcps_init()* call and used internally by LLHW and UWB stack. Typically, this task is running when INITF/RESPF application is started in CLI build or from the startup of the UCI build.

9.2.3 Qworkqueue Task

The Qworkqueue is a task holding a queue which manages work that needs to be performed outside the context of the interrupt service routines. Qworkqueue API can be found in **QOSAL** library. During the SDK runtime, multiple qworkqueues are created when UWB stack is used (when INITF/RESPF application is started in CLI build or from the startup of UCI build). They are used internally by UWB stack e.g. to handle UCI data to be reported from stack to the host.

9.2.4 UCI task

The UCI Task is responsible for opening UCI backends and UCI server inside the uwb-stack at the startup. During the runtime, its primary role is to transfer UCI messages from UART/USB interface to uwb-stack. This task is running from the startup of the firmware and it operates only in the UCI build.

9.3 Threads stack usage

9.3.1 FreeRTOS

The table below shows the threads stack usage of the tasks used in FreeRTOS in two different cases:

- Idle state (no applications running)

THREAD NAME	Stack usage
Control	968/2048
IDLE	164/512
Flush	184/512
Logger	100/768
Default	172/4304
Tmr Svc	156/512
Total HEAP	51200
Current HEAP used	8824
Max HEAP used	8824

- INITF application is running

THREAD NAME	Stack usage
Control	968/2048
IDLE	164/512
FLush	220/512
Logger	112/768
qworkqueue	212/3072
llhw_mcps	1200/2052
qworkqueue	2064/3072
Tmr Svc	156/512
Default	3300/4304
Total HEAP	51200
Current HEAP used	20000
Max HEAP used	21440

10 SDK SW Architecture

10.1 Overview

The SDK platform supports FreeRTOS on ARM Cortex M4. The sections below discuss the architecture, structure and workflow of the software. This should enable the developer to understand the philosophy and be able to add functionality or port the project to other platforms if needed.

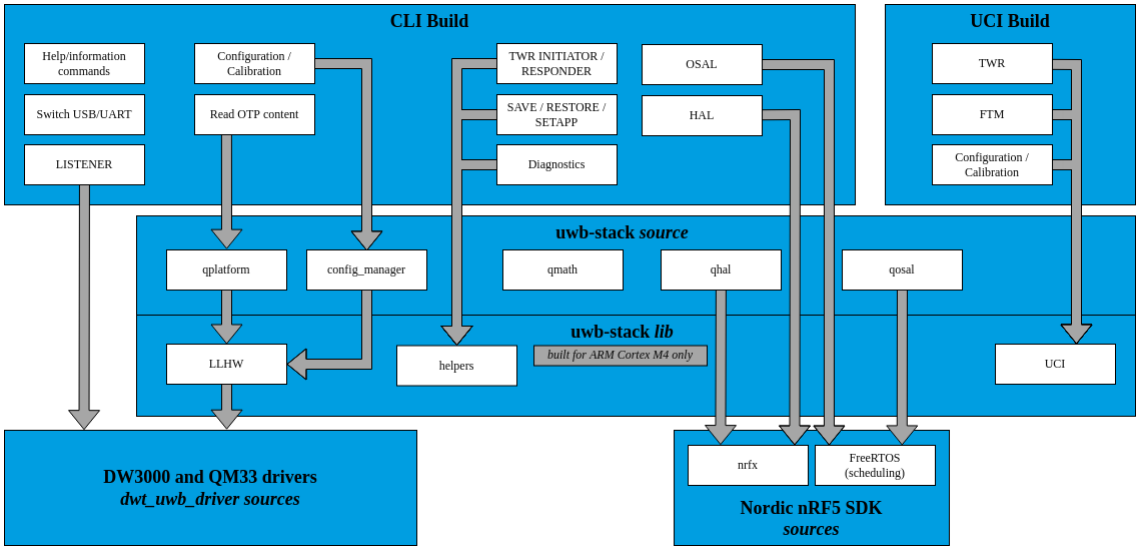

Fig. 10.1: Overview of FreeRTOS based architecture.

The figure above shows the layered structure: “Top-level applications”, “FreeRTOS”, “UWB Stack”, “HAL” and “SDK BSP”.

10.2 Linker sections

The SDK relies on linker sections to automatically register applications, drivers, NVM configurations, etc… This section defines how to modify your linker section to handle this. The linker description file (ld file) is located in your project folder. See section *Projects folder* for details.

10.2.1 Flash sections

10.2.1.1 ISR Vectors section

This section contains the traditional ISR vector table. It starts with a vector array defined with symbol .isr_vector. It is usually located at the Flash entry address.

10.2.1.2 NVM section

This section is a placeholder for storing configuration which will not be lost between a power cycle.

It starts with symbol `__fconfig_start` and ends with symbol `__fconfig_end`.

It is mandatory to ensure that the `__fconfig_start` is aligned at the start of a flash page and that a full page is reserved for this purpose.

Only one variable will be explicitly placed in this section, defined like this:

```
__attribute__((section(".fconfig"))) const uint32_t DummyConfig[1]  = {0xDEADBEEF};
```

Its role is to guarantee that the checksum will be incorrect at first powerup after download, to force a factory default to take place.

See section *NVM configuration* for details about NVM save and restore mechanism.

10.2.1.3 Applications and Commands section

This series of sections will determine which applications are available for the current build.

They are placed in a way that the FW browses them and displays them in the correct order when executing HELP command.

See section *HELP* for details.

10.2.1.4 Applications section

This series of sections will determine which applications are available for the current build.

10.2.1.5 Applications NVM Configuration section

This series of sections will determine which applications are available for the current build.

The first sub-section will handle all the commands which are allowed to execute at any time.

It starts with symbol `__known_commands_start`.

It contains the default function defined with symbol `.known_commands_anytime`.

The second sub-section will handle all the commands which are considered as Applications.

It starts with symbol `__known_commands_app_start`.

It contains the default function defined with symbol `.known_commands_app`.

The third sub-section will handle all the sub-commands of individual applications.

It starts with symbol `__known_app_subcommands_start`.

It contains the default function defined with symbol `.known_app_subcommands`.

The fourth sub-section will handle all the commands that can only be ran if we are in idle state (meaning, no application running).

It starts with symbol `__known_commands_ilde_start`.

It contains the default function defined with symbol `.known_commands_ilde`.

The fifth sub-section will handle all the commands considered service commands.

It starts with symbol `__known_commands_service_start`.

It contains the default function defined with symbol `.known_commands_service`.

Lastly, the symbol `__known_commands_end` will mark the end of this section.

10.2.1.6 Restoring default Configuration section

This section is a placeholder for storing configuration default functions.

It starts with symbol `__config_entry_start` and ends with symbol `__config_entry_end`.

It contains the default function defined with symbol `.config_entry`.

When the user will call RESTORE command (see *RESTORE*) or if the version is invalid, then a factory default will be restored, including the calibration values.

To do so, each function listed in this section will be executed.

For example, for FiRa application, the function below will be executed.

```
static void restore_fira_default_config(void)
{
    memcpy(&fira_config_ram, &fira_config_flash_default, sizeof(fira_config_ram));
}

__attribute__((section(".config_entry"))) const void (*p_restore_fira_default_config)(void) =␣
↪(const void *)&restore_fira_default_config;
```

10.2.2 RAM sections

10.2.2.1 Applications NVM Configuration section

This section is a placeholder for every parameter an application wants to store into NVM.

It starts with symbol `__rconfig_start` and ends with symbol `__rconfig_end`.

One structure uses `.rconfig`, which contains all parameter and attributes to run the application:

```
static dwt_app_config_t dwt_app_config_ram __attribute__((section(".rconfig"))) = {0};
```

See section *NVM configuration* for details about NVM save and restore mechanism.

10.3 NVM configuration

10.3.1 Overview

- The way the configuration is handled inside the SDK is done through linker sections.
- All the variables that have to be stored in NVM are stored in a consecutive RAM section.
- This RAM section is copied as a unique block inside NVM.
- At Power-up, the configuration is loaded from NVM and copied inside this RAM section.

- Sending the SAVE command will write the RAM configuration block into Flash.
- Sending RESTORE command will restore the default configuration along with the default calibration values, please check *Calibration important notes*.

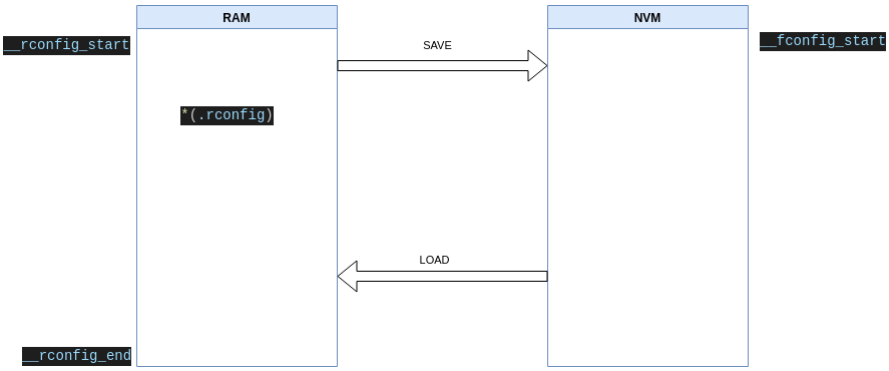

Fig. 10.2: NVM configuration

10.3.2 Initialization sequence during powerup

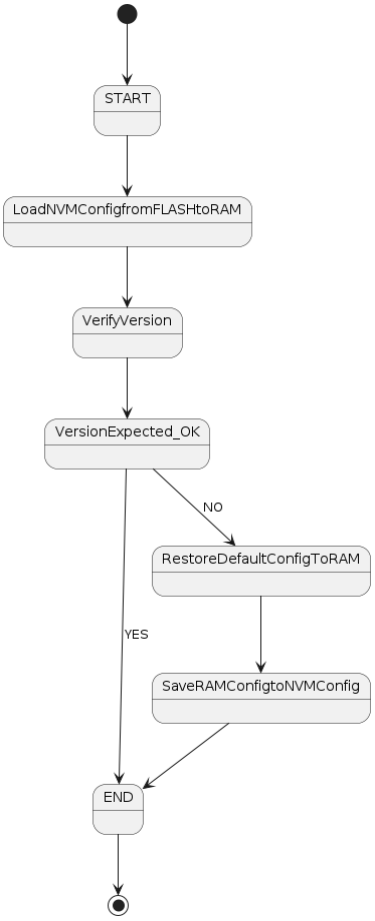

Fig. 10.3: NVM boot sequence

10.4 Folders structure

The table below shows how the source code is distributed among the different folders.

```
root
├── Libs                        Components provided as libraries
│   ├── dwt_uwb_drivers         Libraries, UWB chip driver layer (sources)
│   ├── uwbstack                Libraries, MAC layer (sources)
│   └── uwbstack_libs           Libraries, MAC layer (static libraries)
├── LICENSES                    Software Licenses
├── Projects                    Contains different projects grouped by OS and target
│   └── DW3_QM33_SDK
├── SDK_BSP                     Board support package provided by MCU manufacturer
│   └── Nordic                  Nordic BSP folder
└── Src
    ├── AppConfig              Load/Save/Restore apps configuration
    ├── Apps                   Implementation of the apps
    ├── Boards                 API for the board's specific implementation
    ├── Comm                   Projects Communication resources
    ├── EventManager           Event manager functionalities implementation
    ├── HAL                    Hardware Abstraction Layer
    │   ├── Inc                HAL API header files
    │   └── Src                HAL API source files
    ├── Helpers                Generic Helpers (JSON, CRC, DBG, ...)
    ├── OS                     OS
    ├── OSAL                   OS Abstraction Layer
    ├── SecureComponent        Cryptography implementation for uwbstack/fira_crypto/fira_random
    └── UWB                    UWB project's utilities implementation
```

10.5 Layers

10.5.1 AppConfig API

This folder contains what is needed to store non-volatile configuration into Non-Volatile Memory (NVM).

The way to save data to NVM is Board dependent (what flash memory is used).

For this reason, we implemented this as standalone folder to offer the required flexibility.

A user can then adapt the code to match their requirements.

The key file is located inside *AppConfig/{MyBoard}/config.c*.

Please also see the *NVM configuration* section on NVM configuration for further details.

10.5.1.1 API

The API header file is located inside *AppConfig/Inc/appConfig.h*

It contains three functions:

```
void load_bssConfig(void)
```

This function will load config from FLASH to RAM.

```
void restore_bssConfig(void)
```

This function will restore default config to RAM and save it back to FLASH.

```
error_e save_bssConfig(void)
```

This function will save the RAM config to FLASH.

Please also see *NVM configuration*.

10.5.1.2 Linker symbols used

```
extern uint8_t __config_entry_start[];
extern uint8_t __config_entry_end[];
extern uint8_t __fconfig_start[];
extern uint8_t __rconfig_start[];
extern uint8_t __rconfig_end[];
```

Please also see *Linker sections*.

10.5.2 Applications layer (Apps)

10.5.2.1 common folder

See also *Always running Threads*

10.5.2.2 FiRa demo application: INITF / RESPF

INITF (Initiator FiRa) is an application that sets the unit as initiator in a FiRa two-way ranging session.

RESPF (Responder FiRa) is an application that sets the unit as responder in a FiRa two-way ranging session.

10.5.2.3 UWB sniffer: LISTENER

LISTENER is an application that sets the unit into receive mode and it reports any received data.

10.5.3 Board Abstraction Layer (Boards)

Board Abstraction Layer (BAL) allows to customize the middleware initialization and logic for the target. Each board supported by SDK has its dedicated folder that contains modules which implement the BAL API.

Currently, the BAL layer is implemented and defined in two locations:

- **Board Components** `./Src/Boards` - implementations and customization of generic part of BAL for specific targets,
- **Qplatform Component** `./Libs/uwb-stack/libs/qplatform` - implementations of the uwb-stack specific part of BAL for specific targets.

The **Board Component** implements the generic platform initialization logic and provides the means for customization of the generic BAL related part (as well as customization of the HAL).

The **Qplatform Component** implements the uwb-stack initialization related to platform, so it provides the means for customization of the uwb-stack related part of BAL.

If the implementation for the BAL for the given platform does not already exist, the implementation is required to be created. In this case, please refer to *Porting BAL*.

10.5.3.1 Board Component

10.5.3.1.1 Files

```
root
└── Src
    └── Boards                                  Boards Abstraction Layer
        ├── Inc                                 HAL API header files
        │   ├── int_priority.h                  Interrupt priorities definition
        │   └── thisBoard.h                     Board API header file
        └── Src                                 Boards source files
            ├── Common                          Common code for all board
            │   ├── helper_platform_l1_config.c Common helpers to develop platform L1 Config
            │   └── helper_platform_l1_config.h
            ├── Common_NRF                      Common code for NRF board
            │   ├── nrf52_board.c               Common Helpers for nrf52 boards
            │   └── nrf52_board.h
            ├── Common_OTP                      Common OTP code
            │   └── OTP_config.h
            ├── <target>                        <target> specific code
            │   ├── CMakeLists.txt
            │   ├── Common
            │   │   └── custom_board.h          BAL customization header
            │   ├── FreeRTOS
            │   │   └── <target>.c              <target> specific BAL implementation
            │   └── platform_l1_config.c        <target> specific l1 config ops
            │
            └── ...
```

10.5.3.1.2 API definition

thisboard.h

The main() function will call two key functions during the initialization phase:

```
void BoardInit(void);
void board_interface_init(void);
```

custom_board.h

This file is a key file in target SDK mechanism.

It is used to configure the target SDK the way we need for our project, such as pinout assignment and other features.

int_priority.h

This file contains the definitions of thread priorities.

Other important files

Files used to customize a particular target:

10.5.3.2 Qplatform Component

For the detailed description of the **Qplatform component** please refer to *Documentation/uwb-qplatform-api.pdf*

The customization of the **Qplatform components** is done by means of: `Firmware/Projects/DW3_QM33_SDK/<OS>/<target>/ProjectDefinition/uwb_stack_llhw.cmake` (ProjectDefinition target).

Below is an example table of the components customized by ProjectDefinition target in `uwb_stack_llhw.cmake` file for nRF52840DK board.

LLHW drv definitions
Nordic SDK definitions SPI
GPIO
CLOCK
ARM SYSTICK
TIMER
DTU Base timer
UWB stack Idle timer
HAL timer
HAL FS timer
LOG timestamp Timer
RTC
RTC conf.
Watchdog
NMV Flash Controller
USB Controller

10.5.4 Hardware Abstraction Layer (HAL)

HAL (Hardware Abstraction Layer) provides all the services needed by the application layer so the access to hardware resources is agnostic.

10.5.4.1 Overview

The Hardware Abstraction Layer is designed to make the application adaptable to any hardware it may be running on. All the possible hardware peripherals have their own HAL files.

In general, project defines its own *HAL Layer* implementation, but a part of the HAL Layer implementation is delegated to QHAL (Qorvo HAL), provided by uwbstack sublibrary `Firmware/Libs/uwb-stack/libs/qhal` (refer to `Documentation/uwb-qhal-api.pdf`). In other words, the *HAL Layer* in SDK is covered by two implementations of HAL: HAL and QHAL. The generic part of functionalities of the *HAL Layer* for the platform are covered by the HAL, the specific part of the *HAL Layer*, for components provided by uwb-stack, is covered by the QHAL.

Note: We are going to use following terminology:

HAL - *part of HAL Layer defined in* `Firmware/Src/HAL/Src`

QHAL - *part of HAL Layer defined in* `Firmware/Libs/uwb-stack/libs/qhal`

HAL Layer = HAL + QHAL

- Files location

```
root
└── Src
    └── HAL                    Hardware Abstraction Layer
        ├── Inc                HAL API header files used
        └── Src                HAL API source files to adapt
```

```
root
└── Libs
    └── uwb-stack
        └── libs
            └── qhal       Qorvo Hardware Abstraction Layer
                ├── Inc    QHAL API header files used
                └── Src    QHAL API source files to adapt
```

We provide *HAL Layer* implementation for nrf52 boards family.

If a use case is not covered in the supported list, then refer to *Porting HAL* for guidance on how to do the porting.

10.5.4.2 HAL GPIO

10.5.4.2.1 Files

```
root
└── Src
    └── HAL                    Hardware Abstraction Layer
        ├── Inc                HAL API header files
        │   └── HAL_gpio.h     HAL GPIO API header file
        └── Src                HAL API source files
            └── target         HAL target implementation
                └── HAL_gpio.c HAL GPIO implementation for target
```

10.5.4.2.2 API definition

```
struct hal_gpio_s
{
   void (*init)(int, char *, int, int);
   void (*on)(int);
   void (*off)(int);
   void (*toggle)(int);
};

extern const struct hal_gpio_s Gpio;
```

- `void (*init)(int, char *, int, int);`

  Initialize GPIO pin

  Parameters:

  `int`: GPIO number

  `char *`: GPIO label: GPIO port for example

  `int`: GPIO pin

  `int`: GPIO configuration flags

Return: none

- `void (*on)(int);`

  Set GPIO pin

  Parameters: `int` GPIO number

  Return: none

- `void (*off)(int);`

  Unset GPIO pin

  Parameters: `int` GPIO number

  Return: none

- `void (*toggle)(int);`

  Toggle GPIO pin

  Parameters: `int` GPIO number

  Return: none

```
void hw_debug_D0(void);
void hw_debug_D1(void);
```

Debug GPIO functions: can be used to control LEDs

Parameters: none

Return: none

10.5.4.3 HAL BUTTON

Note: This API is using a legacy style and will be refactored in a future release

10.5.4.3.1 Files

```
root
└── Src
    └── HAL                    Hardware Abstraction Layer
        ├── Inc                HAL API header files
        │   └── HAL_button.h     HAL button API header file
        └── Src                HAL API source files
            └── nrfx           HAL Nordic implementation
                └── HAL_button.c  HAL button implementation for Nordic
```

10.5.4.3.2 API definition

```
bool isButtonPressed(void);
```

Check if the button is pressed

Parameters: none

Return: true if the button is pressed

```
void configure_button(void);
```

Configure button input

Parameters: none

Return: none

10.5.4.4 HAL LED

All the necessary APIs to initialize, switch ON, OFF, or TOGGLE and LED

10.5.4.4.1 Files

```
root
└── Src
    └── HAL                     Hardware Abstraction Layer
        ├── Inc                 HAL API header files
        │   └── HAL_led.h       HAL LEDs API header file
        └── Src                 HAL API source files
            └── nrfx            HAL Nordic implementation
                └── HAL_led.c   HAL LEDs implementation for Nordic
```

10.5.4.4.2 API definition

```
struct hal_led_s
{
   void (*init)(int, char *, int, int);
   void (*on)(int);
   void (*off)(int);
   void (*toggle)(int);
};

extern const struct hal_led_s Led;
```

- `void (*init)(int, char *, int, int);`

  Initialize LED

  Parameters:

  `int`: Led index

  `char *`: Led name

  `int`: Pin number for the LED

  `int`: Configuration flags

Return: none

- `void (*on)(int);`

  Set LED

  Parameters: `int` Led index

  Return: none

- `void (*off)(int);`

  Unset LED

  Parameters: `int` Led index

  Return: none

- `void (*toggle)(int);`

  Toggle LED

  Parameters: `int` Led index

  Return: none

10.5.4.5 HAL UART

Use this API when your design is using real UART to interface to the host.

When your design is USB Virtual Com based, use HAL USB.

Note: This API is using a legacy style and will be refactored in a future release

10.5.4.5.1 Files

```
root
└── Src
    └── HAL                         Hardware Abstraction Layer
        ├── Inc                     HAL API header files
        │   └── HAL_uart.h          HAL UART API header file
        └── Src                     HAL API source files
            └── nrfx                HAL Nordic implementation
                └── HAL_uart.c      HAL UART implementation for Nordic
```

10.5.4.5.2 API definition

```
bool IsUartDown(void);
```

Get UART power state

Parameters: none

Return: `TRUE` if the UART is in low power mode

```
void SetUartDown(bool val);
```

Change UART low-power state

Parameters: power state

Return: none

```
void deca_uart_close(void);
```

Flush UART buffers and close it

Parameters: none

Return: none

```
void deca_uart_init(void);
```

Initialize UART as the main interface (chosen setup is 115200, 8, N, 1, no Hardware flow control)

Parameters: none

Return: none

```
void deca_uart_receive(void);
```

Receive UART data and put it in a buffer

Parameters: none

Return: none

```
int deca_uart_transmit(uint8_t *ptr, uint16_t sz);
```

Send data over UART

Parameters:

- `uint8_t *ptr`: pointer to the buffer to send
- `uint16_t sz`: size of the data to send

Return: error status

```
int deca_uart_putc(uint8_t car);
```

Send one character over UART

Parameters: character to send

Return: error status

```
void deca_uart_flush(void);
```

Flush UART buffer

Parameters: none

Return: none

10.5.4.6 HAL USB

10.5.4.6.1 Files

```
root
└── Src
    └── HAL                     Hardware Abstraction Layer
        ├── Inc                 HAL API header files
        │   └── HAL_usb.h       HAL USB API header file
        └── Src                 HAL API source files
            └── target          HAL target implementation
                └── HAL_usb.c   HAL USB implementation for target
```

10.5.4.6.2 API definition

```
struct hal_usb_s
{
   void (*init)(void(*rx_callback)(uint8_t *ptr, size_t len));
   void (*deinit)(void);
   bool (*transmit)(uint8_t *tx_buffer, int size);
   void (*receive)(void);
   void (*update)(void);
   bool (*isTxBufferEmpty)(void);
};
extern const struct hal_usb_s Usb;
```

- `void (*init)(void(*rx_callback)(uint8_t *ptr, size_t len));`

  USB initialization

  Parameters:

  `void(*rx_callback)(uint8_t *ptr, size_t len)`: Optional USB reception callback

  Return: none

- `void (*deinit)(void);`

  USB deinitialization

  Parameters: none

  Return: none

- `bool (*transmit)(uint8_t *tx_buffer, int size);`

  USB send data

  Parameters:

  `uint8_t *tx_buffer`: buffer containing data to send

  `int size`: size of the data to send

  Return: none

- `void (*receive)(void);`

  USB receive data

  Parameters: none

  Return: none

- `void (*update)(void);`

  USB update state machine

  Parameters: none

  Return: none

- `bool (*isTxBufferEmpty)(void);`

  Check if there is nothing to send/sending

  Parameters: none

  Return: `TRUE` if tx buffer empty

10.5.4.7 HAL SPI

This SPI Hardware Abstraction layer will abstract SPI accesses to the UWB transceiver.

10.5.4.7.1 Files

```
root
└── Src
    └── HAL                     Hardware Abstraction Layer
        ├── Inc                 HAL API header files
        │   └── HAL_SPI.h       HAL SPI API header file
        └── Src                 HAL API source files
            └── target          HAL target implementation
                └── HAL_SPI.c   HAL SPI implementation for Nordic
```

10.5.4.7.2 API definition

10.5.4.7.2.1 SPI functions API

```
struct spi_s
{
    void (*cs_low)(void *handler);
    void (*cs_high)(void *handler);
    void (*slow_rate)(void *handler);
    void (*fast_rate)(void *handler);
    int (*read)(void *handler, uint16_t headerLength, const uint8_t *headerBuffer, uint16_t␣
↪readlength, uint8_t *readBuffer);
    int (*write)(void *handler, uint16_t headerLength, const uint8_t *headerBuffer, uint16_t␣
↪readlength, const uint8_t *readBuffer);
    int (*read_write)(void *handler, uint8_t *readBuffer, uint16_t readlength, uint8_t␣
↪*writebuffer, uint16_t writeLength);
    int (*write_with_crc)(void *handler, uint16_t headerLength, const uint8_t *headerBuffer,␣
↪uint16_t bodyLength, const uint8_t *bodyBuffer, uint8_t crc8);
    void *handler;
};
typedef struct spi_s spi_t;
```

- `void (*cs_low)(void *handler);`

Unset Chip Select pin

Parameters: SPI handler

Return: none

- `void (*cs_high)(void *handler);`

  Set Chip Select pin

  Parameters: SPI handler

  Return: none

- `void (*slow_rate)(void *handler);`

  Deinit SPI instance Set SPI handler frequency to slow (frequency should be defined in init_spi) Init SPI instance

  Parameters: SPI handler

  Return: none

- `void (*fast_rate)(void *handler);`

  Deinit SPI instance Set SPI handler frequency to fast (frequency should be defined in init_spi) Init SPI instance GPIO configuration may need to be changed to support increased frequency

  Parameters: SPI handler

  Return: none

- `int (*read)(void *handler, uint16_t headerLength, const uint8_t *headerBuffer, uint16_t readlength, uint8_t *readBuffer);`

  Unset Chip Select pin Read SPI frame from instance buffer Set Chip Select pin

  Parameters:

  `void *handler`: SPI handler

  `uint16_t headerLength`: length of the header

  `const uint8_t *headerBuffer`: reception buffer of the header

  `uint16_t readlength`: length of the frame without header

  `uint8_t *readBuffer`: reception buffer of the frame

  Return: error code

- `int (*write)(void *handler, uint16_t headerLength, const uint8_t *headerBuffer, uint16_t readlength, const uint8_t *readBuffer);`

  Unset Chip Select pin Write frame to SPI instant buffer Set Chip Select pin

  Parameters:

  `void *handler`: SPI handler

  `uint16_t headerLength`: length of the header

  `const uint8_t *headerBuffer`: dispatch buffer of the header to send

  `uint16_t readlength`: length of the frame without header

  `const uint8_t *readBuffer`: dispatch buffer of the frame to send

  Return: error code

- `int (*read_write)(void *handler, uint8_t *readBuffer, uint16_t readlength, uint8_t *writebuffer, uint16_t writeLength);`

Unset Chip Select pin Read and write SPI frame Set Chip Select pin

Parameters:

void *handler: SPI handler

uint8_t *readBuffer: reception buffer

uint16_t readlength: length of the frame to receive

uint8_t *writebuffer: dispatch buffer

uint16_t writeLength: length of the frame to send

Return: error code

- **int** (*write_with_crc)(**void** *handler, **uint16_t** headerLength, **const** **uint8_t** *headerBuffer, **uint16_t** bodyLength, **const** **uint8_t** *bodyBuffer, **uint8_t** crc8);

Note: This function is unused

Write SPI frame with CRC

Parameters:

void *handler: SPI handler

uint16_t headerLength: length of the header

const **uint8_t** *headerBuffer: dispatch buffer of the header to send

uint16_t bodyLength: length of the frame without header

const **uint8_t** *bodyBuffer: dispatch buffer of the frame to send

uint8_t crc8: calculated CRC

Return: error code

- **void** *handler;

  SPI handler

10.5.4.7.2.2 SPI GPIOs definition

```
struct spi_port_config_s
{
    uint32_t idx;
    uint32_t cs;
    uint32_t clk;
    uint32_t mosi;
    uint32_t miso;
    uint32_t min_freq;
    uint32_t max_freq;
};
typedef struct spi_port_config_s spi_port_config_t;
```

- idx

  Index of the SPI peripheral (example SPI0, SPI1, SPI2, ...)

- cs, clk, mosi, miso

  Index of the GPIO used for the CS, CLK, MOSI and MISO (example cs=12 means CS pin is GPIO12)

- `min_freq`

  Low SPI clock speed frequency (kHz)

- `max_freq`

  High SPI clock speed frequency (kHz)

10.5.4.7.2.3 SPI initialization

```
const struct spi_s *init_spi(const spi_port_config_t *spi_port_cfg);
```

Initialization of SPI connected to UWB chip

Parameters: spi config (see above)

Return: spi structure configured

10.5.4.8 HAL RTC

10.5.4.8.1 Files

```
root
└── Src
    └── HAL                     Hardware Abstraction Layer
        ├── Inc                 HAL API header files
        │   └── HAL_rtc.h       HAL RTC API header file
        └── Src                 HAL API source files
            └── nrfx            HAL Nordic implementation
                └── HAL_RTC.c   HAL RTC implementation for Nordic
```

10.5.4.8.2 API definition

```
struct hal_rtc_s
{
   void (*init)(void);
   void (*deinit)(void);
   void (*enableIRQ)(void);
   void (*disableIRQ)(void);
   void (*setPriorityIRQ)(void);
   uint32_t (*getTimestamp)(void);
   uint32_t (*getTimeElapsed)(uint32_t start, uint32_t stop);
   void (*reload)(uint32_t time);
   void (*configureWakeup_ms)(uint32_t period_ms);
   void (*configureWakeup_ns)(uint32_t period_ns);
   float (*getWakeupResolution_ns)(void);
   void (*setCallback)(void (*cb)(void));
};
extern const struct hal_rtc_s Rtc;
```

- `void (*init)(void);`

  Initialization and enabling of the RTC driver instance

  Parameters: none

Return: none

- `void (*deinit)(void);`

  Disable and deinitialize RTC timer

  Parameters: none

  Return: none

- `void (*enableIRQ)(void);`

  Enable RTC interrupt

  Parameters: none

  Return: none

- `void (*disableIRQ)(void);`

  Disable RTC interrupt

  Parameters: none

  Return: none

- `void (*setPriorityIRQ)(void);`

  Set RTC Wakeup timer with a high priority

  Parameters: none

  Return: none

- `uint32_t (*getTimestamp)(void);`

  Get RTC current counter value

  Parameters: none

  Return: RTC counter value

- `uint32_t (*getTimeElapsed)(uint32_t start, uint32_t stop);`

  Returns elapsed time

  Parameters:

  `start`: Start date

  `stop`: Stop date

  Return: Elapsed time

- `void (*reload)(uint32_t time);`

  Reload RTC registers after interrupt to prepare next compare match event

  Parameters: Absolute value to be set in the compare register

  Return: none

- `void (*configureWakeup_ms)(uint32_t period_ms);`

  Setup RTC Wakeup timer period_ms is awaiting time in ms

  Parameters: Period in ms

  Return: none

- `void (*configureWakeup_ns)(uint32_t period_ns);`

Setup RTC Wakeup timer period_ms is awaiting time in ns

Parameters: Period in ns

Return: none

- `float (*getWakeupResolution_ns)(void);`

  Get WKUP timer resolution

  Parameters: none

  Return: Resolution in ns

- `void (*setCallback)(void (*cb)(void));`

  Set RTC callback

  Parameters: callback

  Return: none

10.5.4.9 HAL TIMER

10.5.4.9.1 Files

```
root
└── Src
    └── HAL                     Hardware Abstraction Layer
        ├── Inc                 HAL API header files
        │   └── HAL_timer.h     HAL TIMER API header file
        └── Src                 HAL API source files
            └── nrfx            HAL Nordic implementation
                └── HAL_timer.c HAL TIMER implementation for Nordic
```

10.5.4.9.2 API definition

10.5.4.9.2.1 Fast sleep timer

```
struct hal_fs_timer_s
{
   /* timer instance */
   void *timer;

   /* timer external ISR */
   void (*timerIsr)(void *dwchip);

   /* @brief Initialization/uninit of the Fast Sleep Timer
   *        cb - is the "timerIsr"
   */
   void (*init)(void *self, void *dwchip, void (*cb)(void *));

   /* @brief De-Initialization/uninit of the Fast Sleep Timer */
   void (*deinit)(void *self);

   /* @brief  start the timer
   *         int_en    - True/False to generate an interrupt on counter compare event
```

(continues on next page)

(continued from previous page)

```
    *          time_us    - is time to wait in microseconds
    *          corr_tick - the number of timer's ticks it needs to start the counting from
    */
    void (*start)(void *self, bool int_en, uint32_t next_cc_us, int corr_tick);

    /* @brief stop the timer
    */
    void (*stop)(void *self);

    /* @brief returns the fast sleep timer current tick value
    * this will be the number of ticks since its last reset (since last CC event)
    */
    uint32_t const (*get_tick)(void *self);

    /* timer tick frequency */
int freq;
};

typedef struct hal_fs_timer_s hal_fs_timer_t;
```

The purpose of this timer is to fast wake the UWB chip from sleep.

- **void *timer;**
  Instance of the sleep timer. This timer is used to enable low-power sleep of UWB subsystem.

  For Nordic target, the timer 4 instance is used as below:

  ```
  static nrf_drv_timer_t fs_timer = NRF_DRV_TIMER_INSTANCE(4);
  ```

- **void (*timerIsr)(void *dwchip);**
  This is the callback which will be called by the hal_fs_timer. The function lp_timer_fira is used as a callback for the timer 4.

  The role of this callback is to handle the state of the UWB chip and set the next operational state.

- **void (*init)(void *self, void *dwchip, void (*cb)(void *));**
  Initialization of the fast Sleep Timer driver instance. This timer is:

  – Running on 1 MHz.

  – in continuous mode with counter compare events.

```
htimer->init(htimer, dw, lp_timer_fira);
```

- **void (*deinit)(void *self);**
  De-Initialization of the fast Sleep Timer driver instance.

- **void (*start)(void *self, bool int_en, uint32_t next_cc_us, int corr_tick);**
  Start the timer.

  Parameters:

  bool `int_en`: True/False to generate an interrupt on counter compare event.

  uint32_t `next_cc_us`: Time to wait in microseconds.

  int `corr_tick`: The number of timer's ticks it needs to start the counting from

  Return: none

- **void (*stop)(void *self);**
  Stop the timer.

- `uint32_t const (*get_tick)(void *self);`
  Returns the fast sleep timer current tick value. This will be the number of ticks since its last reset (since last Counter Compare event).
- `int freq;`
  Timer tick frequency (1 MHz).

10.5.4.9.2.2 General purpose timer

```
struct hal_timer_s
{
   uint32_t (*init)(void);
   void (*start)(volatile uint32_t *p_timestamp);
   bool (*check)(uint32_t timestamp, uint32_t time);
};

extern const struct hal_timer_s Timer;
```

- `uint32_t (*init)(void);`

  Initialize timer using interrupt to create a tick each one millisecond

  Parameters: none

  Return: error code

- `void (*start)(volatile uint32_t *p_timestamp);`

  Save system timestamp

  Parameters:

  `volatile uint32_t *p_timestamp`: pointer on current system timestamp

  Return: none

- `bool (*check)(uint32_t timestamp, uint32_t time);`

  Check if time from current timestamp over expectation

  Parameters:

  `uint32_t timestamp`: Current timestamp

  `uint32_t time`: Time expectation

  Return:
  `True` if time is over `False` if time is not over yet

10.5.4.10 HAL WATCHDOG

10.5.4.10.1 Files

```
root
└── Src
    └── HAL                      Hardware Abstraction Layer
        ├── Inc                  HAL API header files
        │   └── HAL_watchdog.h   HAL watchdog API header file
        └── Src                  HAL API source files
            └── nrfx             HAL Nordic implementation
                └── HAL_watchdog.c   HAL watchdog implementation for Nordic
```

10.5.4.10.2 API definition

```
struct hal_watchdog_s
{
    void (*init)(int ms);
    void (*refresh)(void);
};
extern const struct hal_watchdog_s Watchdog;
```

- `void (*init)(int ms);`

  Initialize watchdog

  Parameters: Reload time in ms

  Return: none

- `void (*refresh)(void);`

  Refresh/reload watchdog

  Parameters: none

  Return: none

10.5.4.11 HAL UWB

10.5.4.11.1 Files

```
root
└── Src
    └── HAL                     Hardware Abstraction Layer
        ├── Inc                 HAL API header files
        │   └── HAL_uwb.h       HAL UWB API header file
        └── Src                 HAL API source files
            └── nrfx            HAL Nordic implementation
                └── HAL_DW3000.c  HAL UWB implementation for Nordic
```

10.5.4.11.2 API definition

```
struct hal_uwb_s
{
   void (*init)(void);              // init MCU I/O to the DW chip
   int (*probe)(void);              // probe the driver

   void (*irq_init)(void);          // init MCU IRQ line
   void (*enableIRQ)(void);         // enable IRQ from UWBS
   void (*disableIRQ)(void);        // disable IRQ from UWBS

   void (*reset)(void);             // HW reset of UWBS, Note: this will update the sleep status

   // UWB various wakeup mechanisms
   void (*wakeup_start)(void);      // start waking up UWBS, Note: this will update the sleep␣
↪status
   void (*wakeup_end)(void);        // stop waking up UWBS
```

(continues on next page)

(continued from previous page)

```
    void (*wakeup_fast)(void);      // wakeup UWBS without final pause, Note: this will update␣
↪the sleep status
    void (*wakeup_with_io)(void);   // wakeup UWBS with guarantee. Note: this will update the␣
↪sleep status

    // UWB sleeping
    sleep_status_t vsleep_status;
    void (*sleep_status_set)(sleep_status_t); // set local sleep status to the new value. Avoid␣
↪to use this fn() directly.
    sleep_status_t (*sleep_status_get)(void); // get local UWB sleeping status
    void (*sleep_config)(void);               // configure UWB for sleeping
    void (*sleep_enter)(void);                // enter deep sleep if not already. Note: this␣
↪will update the status

    sleep_mode_t vsleep_mode;
    void (*sleep_mode_set)(sleep_mode_t);     // specify in which mode UWB can sleep NONE/IRQ/APP
    sleep_mode_t (*sleep_mode_get)(void);     // get the current UWB can sleep mode

    // MCU sleeping
    void (*mcu_sleep_config)(void); // enable MCU-specific init features
    void (*mcu_suspend_clk)(void);  // turn off MCU clock XTAL
    void (*mcu_resume_clk)(void);   // turn on MCU clock XTAL

    aoa_enable_t (*is_aoa)(void);   // chip + PCB design both support AOA

    // SIP support
    bool (*is_sip)(void);           // check if it is a SIP device
    void (*sip_configure)(struct uwbs_sip_config_s *sip_cfg);    // configuring SIP device

    struct dw_s *uwbs;              // UWB Subsystem connection: SPI, I/O and Driver

    void (*stop_all_uwb)(void);
    error_e (*disable_irq_and_reset)(int);
    void (*deinit_callback)(void);
};

extern struct hal_uwb_s hal_uwb;
```

- `void (*init)(void);`

  Initialize GPIO and SPI to communicate with UWB chip

  Parameters: none

  Return: none

- `void (*probe)(void);`

  Select drivers adapted to the UWB chip

  Parameters: none

  Return: none

- `void (*irq_init)(void);`

  Initialize the UWB IRQ of the MCU to process the events

  Parameters: none

  Return: none

- `void (*enableIRQ)(void);`
  Enable the UWB IRQ
  Parameters: none
  Return: none
- `void (*disableIRQ)(void);`
  Disable the UWB IRQ
  Parameters: none
  Return: none
- `void (*reset)(void);`
  Reset the UWB chip by pulling DW_RESET pin low
  Parameters: none
  Return: none
- `void (*wakeup_start)(void);`
  Wake up UWB chip (hold Chip Select low)
  Parameters: none
  Return: none
- `void (*wakeup_end)(void);`
  Stop talking with UWB chip (hold Chip Select high)
  Parameters: none
  Return: none
- `void (*wakeup_fast)(void);`
  Sequence to wake up UWB chip:
  wakeup_start 200µs wakeup_end
  Parameters: none
  Return: none
- `void (*wakeup_with_io)(void);`
  Sequence to wake up UWB chip with guarantee:
  wakeup_start wait 200µs wakeup_end wait 1500µs
  Parameters: none
  Return: none
- `sleep_status_t vsleep_status;`
  UWB_IS_NOT_SLEEPING UWB_IS_SLEEPING
- `void (*sleep_status_set)(sleep_status_t);`
  Set local sleep status (this should be used directly in the other functions)
  Parameters: `sleep_status_t`: status to set (SLEEPING / NOT SLEEPING)
  Return: none
- `sleep_status_t (*sleep_status_get)(void);`

Get local sleep status

Parameters: none

Return: `sleep_status_t`: status (SLEEPING / NOT SLEEPING)

- `void (*sleep_config)(void);`

  Configure chip sleep mode and wakeup possibility

  Parameters: none

  Return: none

- `void (*sleep_enter)(void);`

  Enter chip in sleep

  Parameters: none

  Return: none

- `sleep_mode_t vsleep_mode;`

  UWB_CANNOT_SLEEP UWB_CAN_SLEEP_IN_IRQ: chip can be put to sleep immediately after IRQ UWB_CAN_SLEEP_IN_APP: chip can be put to sleep by APP

- `void (*sleep_mode_set)(sleep_mode_t);`

  Specify in which mode the chip can sleep NONE/IRQ/APP

  Parameters: `sleep_mode_t`: mode selected

  Return: none

- `sleep_mode_t (*sleep_mode_get)(void);`

  Get the mode where the chip can sleep

  Parameters: none

  Return: mode where the chip can sleep

- `void (*mcu_sleep_config)(void);`

  Enable MCU-specific sleep configuration

  Parameters: none

  Return: none

- `void (*mcu_suspend_clk)(void);`

  Turn off MCU clock XTAL

  Parameters: none

  Return: none

- `void (*mcu_resume_clk)(void);`

  Turn on MCU clock XTAL

  Parameters: none

  Return: none

- `aoa_enable_t (*is_aoa)(void);`

  To know if chip and PCB support AoA

  Parameters: none

  Return: AOA_ERROR, AOA_ENABLED or AOA_DISABLED

- `struct dw_s *uwbs;`

  UWB Subsystem connection: SPI, I/O and Driver

- `void (*stop_all_uwb)(void);`

  Disable IRQ, reset chip and deinitialize GPIO, SPI and callbacks

  Parameters: none

  Return: none

- `error_e (*disable_irq_and_reset)(int);`

  Disable IRQ and reset chip

  Parameters: none

  Return: none

- `void (*deinit_callback)(void);`

  Deinitialize callbacks

  Parameters: none

  Return: none

10.5.4.12 HAL ERROR

10.5.4.12.1 Files

```
root
└── Src
    └── HAL                     Hardware Abstraction Layer
        ├── Inc                 HAL API header files
        │   └── HAL_error.h     HAL ERROR API header file
        └── Src                 HAL API source files
            └── nrfx            HAL Nordic implementation
                └── HAL_error.c HAL ERROR implementation for Nordic
```

10.5.4.12.2 API definition

```
void error_handler(int block, error_e err);
```

Register and indicate errors

Parameters:

`int block`: error handler will be blocking forever if `TRUE`

`error_e err`: error type

Return: none

```
error_e get_lastErrorCode(void);
```

Get last indicated error

Parameters: none

Return: last raised error code

10.5.4.13 HAL SLEEP

This API provides sleep functionalities.

Note: This API is using a legacy style and will be refactored in a future release

10.5.4.13.1 Files

```
root
└── Src
    └── HAL                     Hardware Abstraction Layer
        ├── Inc                 HAL API header files
        │   └── HAL_sleep.h     HAL SLEEP API header file
        └── Src                 HAL API source files
            └── nrfx            HAL Nordic implementation
                └── HAL_sleep.c HAL SLEEP implementation for Nordic
```

10.5.4.13.2 API definition

```
void deca_sleep(unsigned int time_ms);
```

Delay execution for specified time in ms

Parameters: time in ms of the delay

Return: none

```
void deca_usleep(unsigned long time_us);
void usleep(unsigned long time_us);
```

Note: deca_usleep() and usleep() are identical to support legacy application

Delay execution for specified time in µs

Parameters: time in µs of the delay

Return: none

10.5.4.14 HAL LOCK

All the necessary APIs to lock or unlock handle.

10.5.4.14.1 Files

```
root
└── Src
    └── HAL                     Hardware Abstraction Layer
        └── Inc                 HAL API header files
            └── HAL_lock.h      HAL LEDs API header file
```

10.5.4.14.2 API definition

```
QHAL_LOCK(__HANDLE__)
```

Macro to lock handle

Parameters: handle

```
QHAL_UNLOCK(__HANDLE__)
```

Macro to unlock handle

Parameters: handle

10.5.5 OSAL folder

OSAL stands for OS Abstraction Layer.

OSAL folder contains the source code to abstract the OS used in a project:

Project uses QOSAL (Qorvo OSAL) as OSAL Layer.

QOSAL is provided by Qorvo UWB Stack's sublibrary `Firmware/Libs/uwbstack/libs/qosal` (refer to `Documentation/uwb-qosal-api.pdf`).

10.5.6 Projects folder

Projects folder contains the SDK project files.

10.5.6.1 Files

```
Projects/
├── Common
│   ├── cmakefiles
│   │   ├── arm-none-eabi-gcc.cmake
│   │   ├── dwt_uwbdriver.cmake                 Build script for dwt driver
│   │   └── uwbstack.cmake                      Build script for uwbstack
│   └── scripts
│       └── CreateTargetCommon.py               Generic create target script
└── DW3_QM33_SDK
    └── FreeRTOS
        ├── DW3_QM33_SDK-FreeRTOS-Common
        │   ├── cmakefiles
        │   │   └── DW3_QM33_SDK-FreeRTOS.cmake     Main project cmake script
        │   ├── hooks.c                             Interrupt function hooks
```

(continues on next page)

(continued from previous page)

```
        ├── main_cli.c                      CLI main entry
        └── main_uci.c                      UCI main entry
    ├── nRF52840DK
        ├── CMakeLists.txt                  Generic Project Cmake script
        ├── CreateTarget.py                 Specific create target script
        ├── isr_hs.c
        ├── nRF52840.ld                     Project linker script
        ├── project_CLI.cmake               Specific CLI application build script
        ├── project_UCI.cmake               Specific UCI application build script
        ├── project_common.cmake            Project customization cmake script
        └── ProjectDefinition
            ├── CMakeLists.txt              ProjectDefinition module build script
            ├── FreeRTOSConfig.h            Project FreeRTOS customization header
            ├── ProjectName.c               Name of project
            ├── sdk_config.h                Project SDK customization script
            └── uwb_stack_llhw.cmake        uwbstack build customization script
    ├── DWM3001CDK
        ...
    └── Type2AB_EVB
        ...
```

The Project's customizing details, due to supported platform, are defined in `Project/<SDK>/<OS>/<BOARD>/project_common.cmake`.

Below is the table of most significant parameters controlled by project_common.cmake:

MY_BOARD	board type
MY_OS	chosen OS
MY_PROJECT	project chosen by name
MY_HAL	chosen HAL implementation
MY_BSP	chosen BSP SDK
MY_CPU	platform MCU type
MY_LD_FILE	path to linker script
PROJECT_ARCH	architecture of MCU
PROJECT_FP	type of chosen floating point ABI
PROJECT_FPU	type of chosen floating point architecture
USE_CRYPTO	control whether the encryption engine is on or off
USE_DRV_DW3xxx	control type of supported transceiver

See also *Build common definitions* section in *CMake flags customization*.

For details of the porting process of Project folder refer to *Porting Project*.

10.5.7 SDK BSP Folder

SDK BSP folder includes the Board Support Package (drivers) needed for the different targets supported.

Currently Project natively support NORDIC_SDK_17_1_0.

10.5.7.1 Files

```
SDK_BSP
└── Nordic
    └── NORDIC_SDK_17_1_0
```

11 Customizing the firmware

11.1 CMake flags customization

11.1.1 Project CMake Configuration

Each target within the project is defined by a set of four CMake files:

1. **DW3_QM33_SDK-FreeRTOS.cmake**: this file contains configurations that apply to all board types.
   - *Firmware/Projects/DW3_QM33_SDK/FreeRTOS/DW3_QM33_SDK-FreeRTOS-Common/cmakefiles/DW3_QM33_SDK-FreeRTOS.cmake*
2. **project_common.cmake** this file includes settings that are specific to a particular board but are shared across different build types (CLI/UCI).
   - *Firmware/Projects/DW3_QM33_SDK/FreeRTOS/<board_name>/project_common.cmake*
3. **project_CLI.cmake** / **project_UCI.cmake**: these files contain configurations that are specific to each build type (CLI or UCI).
   - *Firmware/Projects/DW3_QM33_SDK/FreeRTOS/<board_name>/project_CLI.cmake*
   - *Firmware/Projects/DW3_QM33_SDK/FreeRTOS/<board_name>/project_UCI.cmake*

You can configure various aspects of the project by setting CMake flags in the mentioned configuration files. In sections below, you can find an overview of some key flags that you might adjust.

11.1.1.1 Board common definitions

Inside **DW3_QM33_SDK-FreeRTOS.cmake** file you can set following configurations:

- The flag below turns on generation of the *compile_commands.json* file, which can be useful for many IDEs to enhance their IntelliSense engines. If you do not use this file, you can disable the flag below.

```
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
```

- The flag below includes QOSAL implementation for FreeRTOS. If you port SDK to a different OS, it is crucial to change this flag (refer to *Porting to other OS*).

```
set(CONFIG_QOSAL_IMPL_FREERTOS 1)
```

- Configs below are specific to L1 and calibration, we recommend leaving them unchanged.

```
# set l1 config needs (used on Src/UWB/CMakelists.txt)
set(CONFIG_L1_CONFIG_CUSTOM_DEFAULT_USE_DEDICATED_SECTION ON)
set(CONFIG_SECURE_PARTITIONS_UWB_L1_CONFIG_SHA256_SIZE 32)
set(CONFIG_SECURE_PARTITIONS_UWB_L1_CONFIG_SIZE 4096)
```

(continues on next page)

(continued from previous page)

```
# Linker calib flags
set(CMAKE_EXE_LINKER_FLAGS
    "${CMAKE_EXE_LINKER_FLAGS} \
-Wl,--defsym=CONFIG_SECURE_PARTITIONS_UWB_L1_CONFIG_SHA256_SIZE=${CONFIG_SECURE_PARTITIONS_
↪UWB_L1_CONFIG_SHA256_SIZE}"
)

set(CMAKE_EXE_LINKER_FLAGS
    "${CMAKE_EXE_LINKER_FLAGS} \
-Wl,--defsym=CONFIG_SECURE_PARTITIONS_UWB_L1_CONFIG_SIZE=${CONFIG_SECURE_PARTITIONS_UWB_L1_
↪CONFIG_SIZE}"
)
```

- Below you can find a set of flags for C compiler, assembler and linker. You can change here e.g. floating point unit architecture, standard of compiler, level of debug information and many more. This is also the best place to add your own compiler/assembler/linker flags.

```
set(CMAKE_C_FLAGS
    "-mcpu=cortex-${PROJECT_ARCH}\
        -mfpu=${PROJECT_FPU}\
        -mfloat-abi=${PROJECT_FP}\
        -mthumb\
        -specs=nano.specs\
        -specs=nosys.specs\
        -fdata-sections\
        -ffunction-sections\
        -std=c11\
        -Wall\
        -g3\
        -D${MY_CPU}\
        ${CMAKE_CUSTOM_C_FLAGS}"
)

set(CMAKE_ASM_FLAGS
    "-mcpu=cortex-${PROJECT_ARCH}\
        -mfpu=${PROJECT_FPU}\
        -mfloat-abi=${PROJECT_FP}\
        -mthumb\
        -fdata-sections\
        -ffunction-sections\
        -Wall\
        -g3\
        -D${MY_CPU}\
        -D__STACK_SIZE=2048\
        -D__HEAP_SIZE=0\
        ${CMAKE_CUSTOM_C_FLAGS}"
)

set(CMAKE_EXE_LINKER_FLAGS
    "${CMAKE_EXE_LINKER_FLAGS} -u _printf_float -Wl,--gc-sections,-T${CMAKE_SOURCE_DIR}/$
↪{MY_LD_FILE},-Map=${PROJECT_NAME}.map -Wl,--print-memory-usage"
)
```

- Below you can find HAL specific flags, for more information please refer to *Porting Platform*.

```
# Set qhal needs
set(CONFIG_QHAL_IMPL_NRFX ON)
set(CONFIG_QPLATFORM_IMPL_QM33_QHAL_NON_ZEPHYR ON)
set(CONFIG_QHAL_MAX_GPIO_CALLBACKS 6)
```

11.1.1.2 Build common definitions

Inside **project_common.cmake** file you can set following configurations:

- The project is built with *-O3* flag to enable maximum level of optimization. For debugging purposes, it can be lowered to e.g. *-Og*, however the code execution may not meet the timing requirements of UWB-STACK, hence the ranging may not be successful.

```
add_definitions(-O3)
```

- The following flag enables the usage of crypto library, which is used for STS calculation. We recommend leaving it turned on unless your design does not require encrypting functionalities.

```
set(USE_CRYPTO 1)
```

- The flag below is essential for the correct operation of the UWB-STACK, it shall not be changed.

```
set(CONFIG_QM33 1)
```

- Flags below are used for proper driver choice (also used in LLHW and UWB-STACK libraries). If your project design is limited to one of DW family, the other flag can be turned off to limit the memory consumption.

```
set(USE_DRV_DW3000 1)
set(USE_DRV_DW3720 1)
```

- The flag below manages HW protection of CPU core and memory. During the development phase, we advise not to alter this flag to maintain an effective debugging process. Modifying this flag could result in the nRF52 CPU locking the programming and debugging ports, until the full erase of the chip is performed.

  The HW protection management is implemented in `Firmware/Src/Boards/Src/Common_NRF/nrf52_board.c` explicitly for the nRF52 SoC because some of this SoC revisions may have the HW protection enabled by default. This implementation needs to be ported if you want to keep the HW protection management when using a different SoC/CPU.

```
set(CONFIG_SOC_PROTECT 0)
```

- The flag below enables the logging system. When it is not needed, disabling this flag will result in saving some CPU resources (OS task, RTC timer, RTT block, memory).

```
set(CONFIG_LOG ON)
```

- When logs are turned on, you can adjust filtering to print more or less logs:

```
# 0=NONE 1=ERROR 2=WARNING 3=INFO 4=DEBUG
set(CONFIG_QLOG_LEVEL 3)
```

11.1.1.3 CLI specific definitions

The key difference between the UCI and CLI builds is their method of communication and control. The UCI build leverages the UCI protocol, requiring an external host to configure and initiate operations on the device. In contrast, the CLI build interacts with the UWB-STACK directly via the helpers API, which allows for autonomous operation without the need for an external interface, making it ideal for standalone devices such as tags.

Inside **project_CLI.cmake** file you can set following configurations:

- If you wish to control device via UCI, use enable the *USE_UCI* flag. Otherwise, use the *USE_CLI* flag.

```
set(USE_CLI 1)
set(USE_UCI 0)
```

- The Listener is an example of non-FiRa application. It uses DWT UWB Drivers directly, without the UWB-STACK. If you want to disable it from the CLI build, turn off the flag below.

```
set(USE_LISTENER 1)
```

- If you do not want to use FiRa specific functionalities in CLI build, you can disable the flag below. However, it will not affect memory footprint of UWB-STACK, this library is always built with FiRa region.

```
set(USE_FIRA 1)
```

11.1.1.4 UCI specific definitions

Inside **project_UCI.cmake** file you can set following configurations:

- If you do not need PCTT functionalities (UWB tests), you can disable flags below:

```
set(USE_PCTT 1)
set(USE_FTM_UCI 1)
```

- Flags below are crucial for the system when UCI is used, we advise not to disable them.

```
set(USE_CONF_MANAGER_UCI 1)
set(USE_MAC_CALIB_UCI 1)
```

11.1.2 UWB-STACK CMake Configuration

UWB-STACK is configured inside CMake file:

- *Firmware/Projects/Common/cmakefiles/uwbstack.cmake*

Use mentioned file to set following configurations:

- The UWB-STACK library is delivered in two versions - FiRa only and full (FiRa + PCTT). If you do not require the PCTT functionalities in the UCI build, you can change condition below to include only *Firmware/Libs/uwbstack_libs/delivery/full/Release*. This will decrease the memory footprint of your design.

```
if(USE_UCI OR USE_PCTT)
    message("Using full uwbstack")
    set(UWBSTACK_LIBS_PATH
      "${CMAKE_CURRENT_LIST_DIR}/../../../${LIBS_PATH}/uwbstack_libs/delivery/full/Release"
    )
else()
    message("Using Fira only uwbstack")
```

(continues on next page)

(continued from previous page)

```
    set(UWBSTACK_LIBS_PATH
        "${CMAKE_CURRENT_LIST_DIR}/../../../${LIBS_PATH}/uwbstack_libs/delivery/fira/
↪Release"
    )
endif()
```

11.1.3 DWT UWB Drivers CMake Configuration

DWT UWB Drivers library is configured inside CMake file:

- *Firmware/Projects/Common/cmakefiles/dwt_uwbdriver.cmake*.

We recommend retaining the default settings.

11.2 Adding a new CLI command

11.2.1 CLI Overview

11.2.1.1 Command parser

Core functionality of CLI is placed inside the command parser module, which is implemented in `command_parser()` function of *Firmware/Src/Apps/Src/common/cmd/cmd.c* file. It is a first component which receives command from the host terminal and performs initial parsing of the command.

After the command is parsed and successfully recognized, the parser calls the function associated with this command when mode of operation condition is met (see *Mode of operation*).

Command can be extended with parameters that can be categorized into three distinct groups:

- **single parameter** - only the first number after command string is recognized as parameter and it is passed via the **val** parameter to the command's function (e.g. DIAG 1).
- **multiple parameters** - multiple parameters can be added after the command string, whole string is passed to the command's function, and the function is responsible for further parsing (e.g. INITF -CHAN=5 -BLOCK=200).
- **JSON parameter** - whole command along with parameters are passed to the command's function as a cJSON object, command's function is responsible for further parsing (currently no example in the SDK).

Parsing of a JSON commands is performed by the cJSON library, for more details please refer to cJSON GitHub[18].

11.2.1.2 API description

11.2.1.2.1 Command function

To provide a unified interface for declaring a CLI command function, the *REG_FN* macro is used:

```
#define REG_FN(x) const char *x(char *text, int val, cJSON *params)
```

Parameters:

- **char *text**: The text provided by the CLI user.
- **int val**: A numeric value provided after the command (simple parameter).
- **cJSON *params**: Parameters in cJSON format (JSON parameter).

[18] https://github.com/DaveGamble/cJSON

- **return**: String to be printed after command is executed. Usually macro `CMD_FN_RET_OK` or `CMD_FN_RET_NOK` is used.

11.2.1.2.2 Command API

The command is specified by the **command_s** structure, refer to the description below.

```
struct command_s
{
    const char *name;
    const mode_e mode;
    REG_FN((*fn));
    const char *cmnt;
};
```

- **char *text**: Name of command which will be used in CLI. It is limited to 9 characters, can be modified with macro **MAX_CMD_LEN** in `Firmware/Src/Apps/Src/common/cmd/cmd.h`.
- **const mode_e mode**: see *Mode of operation*.
- **REG_FN((*fn))**: Function pointer which will be called for given command.
- **const char *cmnt**: Help string for your command which will be printed in the CLI.

11.2.1.2.3 Subcommand API

The command can be also used as a subcommand of an application. Then, it can be executed only when a correct application is running. The subcommand structure needs to be added to application's definition as described in *Implementation of the new subcommand*.

Subcommands of one application are grouped in a *subcommand_group_s* structure, refer to the description below.

```
struct subcommand_group_s
{
    const char *name;
    const struct command_s *subcommands;
    const uint8_t cnt;
};
```

- **const char *name**: Text to be printed in help command to delimit command group.
- **const struct command_s *subcommands**: Pointer to the array of subcommands.
- **const uint8_t cnt**: Number of subcommands in the array.

11.2.1.2.4 Mode of operation

To distinguish between command types within the CLI engine, each command includes a field indicating its mode of operation (**enum mode_e**). During command parsing, the parser checks this parameter to determine if the command can be executed in the current state of the system.

```
typedef enum
{
    mANY = 0,
    mIDLE = 1,
    mAPP = 2,
} mode_e;
```

There are three types of operation defined:

- **mANY** - a general-purpose command which can be executed always.
- **mIDLE** - a general-purpose command which can be executed only in the IDLE mode (when no application is running).
- **mAPP** - a command that starts an application and can only be executed in the IDLE mode. This mode of operation is also used for additional functionalities, such as subcommand resolution.

11.2.2 Implementation of the new command

Note: Please note the following substitution is used throughout this guide:

- Replace **<command_name>** with the name of your new command.

All general-purpose commands are placed inside *Firmware/Src/Apps/Src/common/cmd/cmd_fn.c*. We encourage you to define your new CLI commands in this file.

1. To declare your new command function, please follow this template:

```
REG_FN(f_<command_name>)
{
  // command main function
}
```

   If you want to provide more parameters than one into the command, the command function will be responsible for parsing. Whole command along with parameters is passed to the function as a C string via the **text** variable.

2. Add a help string for your command:

```
const char COMMENT_<command_name>[] = {"Text of your command help."};
```

3. Add your command definition in the proper command's array, e.g. if you define an anytime command, place it inside `known_commands_anytime_all`:

```
const struct command_s known_commands_anytime_all[] __attribute__((section(".known_
↪commands_anytime"))) = {
    // ... (existing command definitions)
    {"<command_name>", mANY, f_<command_name>, COMMENT_<command_name>},
};
```

4. After building the firmware, the new command should be ready to use and visible when using the **HELP** command.

11.2.3 Implementation of the new subcommand

Note: Please note the following substitution is used throughout this guide:

- Replace **<command_name>** with the name of your new command.
- Replace **<application_name>** with the name of your application.

Note: For reference, you can take as an example the **LSTAT** subcommand of **LISTENER** application, which is implemented in `Firmware/Src/Apps/Src/listener/listener_fn.c` and `Firmware/Src/Apps/Src/listener/task_listener.c`.

1. Repeat steps 1. and 2. from *Implementation of the new command*.
2. Define your command in a new array and place it inside `.known_app_subcommands` section.

```
const struct command_s subknown_commands_<application_name>[] __attribute__((section(
↪".known_app_subcommands"))) = {
    {"<command_name>", mAPP, f_<command_name>, COMMENT_<command_name>},
};
```

3. Add subcommand to the proper application:

```
const struct subcommand_group_s subcommands = {"<application_name> Options", &known_
↪subcommands_<application_name>, 1};
```

4. After building the firmware, the new command should be ready to use and visible when using the **HELP** command both in IDLE mode and during runtime of the corresponding application.

11.3 Porting Platform

11.3.1 Porting HAL

11.3.1.1 Introduction

As *HAL Layer* is the link with the hardware level, its implementation allows to support various boards and adapt to any new board.

For porting the entire project to a new platform, you have to adapt the entire *HAL Layer* (HAL and QHAL), which means to create an implementation for `Firmware/Src/HAL/Src` and `Firmware/Libs/uwb-stack/libs/qhal` for the platform, see *Architecture/Layers/HAL_Overview* for more details.

Note: Only Nordic MCUs using nRF SDK for the nRF52 Series SoCs with FreeRTOS are supported for now. HAL sources for Nordic can be found in `Firmware/Src/HAL/Src/nrfx`, QHAL sources for Nordic can be found in `Firmware/Libs/uwb-stack/libs/qhal/src/nrfx`, nRF5 SDK path is `Firmware/SDK_BSP/Nordic/NORDIC_SDK_17_1_0`.

If an MCU brand other than Nordic is used, the corresponding SDK will be needed. SDKs can be stored in the SDK_BSP folder.

11.3.1.2 Porting process

Note: For all steps it is recommended to copy the skeletal structure from existing implementations for already supported platforms.

To support any other platform, on HAL level, following steps have to be accomplished:

1. (If needed) create a new folder under `Firmware/SDK_BSP/<MCU_manufacturer>` and add framework for chosen MCU.

2. Create a new folder <board> under `Firmware/Src/HAL/Src/<board>` and `Firmware/Libs/uwb-stack/libs/qhal/src/<board>`, implement there all the functions defined by API of the *HAL Layer* (HAL and QHAL respectively):

   Note: For HAL api refer to *Architecture/Layers/HAL*

   For QHAL api refer to *Documentation/uwb-qhal-api.pdf*

3. Adapt CMake files for integrating the new platform:

   - `Firmware/Src/HAL/Src/<MCU>/CMakeLists`, HAL component is integrated with the project by means of CMake's library target *HAL*.
   - `Firmware/Libs/uwb-stack/libs/qhal/CMakeLists`, QHAL component is integrated with the project by means of CMake's library target *qhal*.

   Note: Customization of the Project due to *HAL Layer* is done at the level of Project definition *Architecture/Layers/Project*.

   - Some of customization may be required to be done in `Firmware/Projects/DW3_QM33_SDK/FreeRTOS/DW3_QM33_SDK-FreeRTOS-Common/cmakefiles/DW3_QM33_SDK-FreeRTOS.cmake` the generic common project CMake file, e.g. new parameter *CONFIG_QHAL_IMPL_XXX* for new *HAL Layer* implementation. Below is a table of QHAL configuration parameters that may require to be customized within the file.

CONFIG_QHAL_IMPL_NRFX	Enabling support of NRFX based platform in the QHAL.
CONFIG_QHAL_MAX_GPIO _CALLBACKS	Define max number of callbacks in the QHAL.

Note that a new parameter may need to be defined if a new SDK is added.

Note: There are two flavors of applications in the SDK:

- uwb-stack based FiRa applications (CLI: `INITF`, `RESPF`, UCI: uci),
- direct dwt_driver based applications (listener).

Depending on which flavor of application you want to port to a new platform, some part of the HAL layer may not need to be ported. In particular, for projects that contains only applications using uwb-stack and not interacting directly with the drivers, there is no need to port `Firmware/HAL/Src/<MCU>/HAL_DW3000.c` (e.g. listener).

11.3.2 Porting BAL

11.3.2.1 Introduction

To have the software fully working on a new board, adapting it to the board is necessary. Adaptation of the project's middleware to the new board is done by Board Abstraction Layer (BAL). For details see *Architecture/Layers/Board Abstraction Layer*.

11.3.2.2 Porting process

To port entire BAL layer, you have to:

1. Create the implementation of the **Board Component** for your target board.

   The implementation should be located at ./Src/Boards/Src/<target>:

   > **Note:** It is recommended to create a copy of the `Board Component` of an already existing target and adjust it for the new target board.

   - Create the implementation of the `Firmware/Src/Boards/Inc/thisBoard.h` interface at location `Firmware/Src/Boards/Src/<target>/<OS>/<target>.c`,
     - `peripherals_init(void)` must contain the initialization of the clock and watchdog.
     - `BoardInit(void)` must contain the pin initialization, call peripheral initialization and can be customized to flash LEDs.
   - Create the customization file `Firmware/Src/Boards/Src/<target>/Common/custom_board.h`,

     Below is an example table of the components customized by custom_board.h for `nRF52840DK` board.

UART
LED
HAL_DW3000
HAL_DW3000/SPI

   - Create the implementation of `l1_config_reset_to_default` procedure in `Firmware/Src/Boards/Src/<target>/platform_l1_config.c` (refer to *Documentation/uwb-l1-configuraion.pdf*).
   - Adjust the `Firmware/Src/Boards/Src/<target>/CMakeLists.txt` by customizing it for the current target board. The `Board Component` is integrated with project as CMake's library target `Board`.
2. Customize the **Qplatform component** for your target board.

   Since the `Qplatform component` implementation is generic for QM33, it only requires customization for the new target. The customization of `Qplatform component` is done by means of `ProjectDefinition` in `Firmware/Projects/DW3_QM33_SDK/<OS>/<target>/ProjecDefinition/uwb_stack_llhw.cmake`. Especially section `LLHW drv definitions`. For more details see description of **Qplatform Component** section in chapter *Board Abstraction Layer (Boards)*.

11.3.3 Porting to other OS

11.3.3.1 Introduction

Before adding a new OS to the SDK, we encourage you to familiarize yourself with how FreeRTOS is integrated into the SDK's structure along with **OSAL** layer by referring to *Projects folder* and *OSAL folder*.

The OSAL layer of the Project is delegated to the QOSAL (Qorvo Operating System Abstraction Layer). In other words, project uses/implements QOSAL as OSAL Layer.

11.3.3.2 QOSAL

11.3.3.2.1 Introduction

The key to adapting the SDK for your chosen OS is to port the QOSAL.

QOSAL API is defined in `Firmware/Libs/uwb-stack/libs/qosal/include`. Start by consulting the **UWB QOSAL API** document (`Documentation/uwb-qosal-api.pdf`), which provides a comprehensive API overview and porting instructions.

11.3.3.2.2 QOSAL implementation

We recommend using the FreeRTOS QOSAL implementation as a reference when porting to a different OS. Implementation of the QOSAL for FreeRTOS is placed inside following directories:

- `Firmware/Libs/uwb-stack/libs/qosal/src/freertos`
- `Firmware/Libs/uwb-stack/libs/qosal/include/freertos`

The QOSAL library directory includes a Zephyr implementation that you may use as a reference. However, be aware that this implementation has not been tested with the current SDK.

The QOSAL API and its implementation are designed to support the full functionality of the uwb-stack and code within this SDK. Bear in mind that you may need to extend the API to accommodate the particular needs of your OS and product design.

11.3.3.2.3 Functionalities description

API	FUNCTIONALITY
QASSERT	Assertion handling (no specific implementation for FreeRTOS)
QATOMIC	Wrapper for GCC atomic operations (no specific implementation for FreeRTOS)
QERR	Qorvo specific error codes and enum conversion
QIRQ	Enabling/disabling interrupts in the system
QLOG	Logging system (depends on **nrfx** library, implementation placed inside `Firmware/Libs/uwb-stack/libs/qhal/src/nrfx/qlog_impl.h`)
QMALLOC	Dynamic memory management
QMSQ_QUEUE	Message queues management
QMUTEX	Mutex management
QOS	High level OS functions, e.g. to start the scheduler
QPM	ACPI standard power management of the system (not supported for FreeRTOS in this SDK, please follow Zephyr implementation for design guides)
QPRIVATE	Private API implementation specific for FreeRTOS (used only inside QOSAL library)
QMPROFILING	Task and stack usage info
QRAND	Random number generation (currently not used in uwb-stack and this SDK)
QSEMAPHORE	Semaphores management
QSIGNAL	Management of signals to communicate between tasks
QTHREAD	Threads/tasks management
QTIME	System time and task sleep management
QMTOOLCHAIN	Toolchain specific definitions (no specific implementation for this SDK)
QTRACING	Printing MAC traces (not supported in this SDK)
QWORKQUEUE	Managing work queue - tasks to delegate low-priority work from ISRs or high-priority tasks

11.3.3.3 Porting

Note: Please note the following substitutions are used throughout this guide:

- Replace **<os_name>** with the name of the operating system you are using.
- Replace **<board_name>** with the name of your chosen development kit.

11.3.3.3.1 Add a new OS

Start the porting process by adding your OS as a CMake library named **OS** in the directory `Firmware/Src/OS/<os_name>/`. Refer to the FreeRTOS configuration in `Firmware/Src/OS/FreeRTOS/CMakeLists.txt` for an example.

11.3.3.3.2 Port the QOSAL

1. Provide your implementation of **QOSAL** in `Firmware/Libs/uwb-stack/libs/qosal/src/<os_name>/`, and `Firmware/Libs/uwb-stack/libs/qosal/include/<os_name>/`. Then add sources to `Firmware/Libs/uwb-stack/libs/qosal/CMakeLists.txt`.
2. Hide your QOSAL implementation under CMake flag `CONFIG_QOSAL_IMPL_<os_name>` inside `Firmware/Libs/uwb-stack/libs/qosal/CMakeLists.txt`.
3. Set up the **OSAL** library:
   - Copy `Firmware/Src/OSAL/Src/FreeRTOS/CMakeLists.txt` to `Firmware/Src/OSAL/Src/<os_name>/CMakeLists.txt`.
   - Modify **OSAL** library in `Firmware/Src/OSAL/Src/<os_name>/CMakeLists.txt` to suit your OS requirements.

Warning: Due to system constraints, uwb-stack library is precompiled with `Firmware/Libs/uwb-stack/libs/qosal/include/dummy`. This enforces the values defined by *QOSAL_IMPL_THREAD_MAX_NAME_LEN* and implementation of *QOSAL_IMPL_THREAD_STACK_DEFINE* from “dummy” implementation are used within the uwb-stack. Consequently, any changes made to these macros in the SDK will not affect the precompiled uwb-stack, which continues to rely on the “dummy” definitions.

Additionally, the uwb-stack uses “dummy” version of QLOG, so that logs produced by uwb-stack are not accessible. Despite this, the OS-specific implementation within the SDK will function correctly.

Warning: QLOG implementation is based on nrfx libraries and depends on QHAL, the implementation is placed inside `Firmware/Libs/uwb-stack/libs/qhal/src/nrfx/qlog_impl.h`.

4. Check if your OS design is aligned to macros *QOSAL_IMPL_THREAD_MAX_NAME_LEN* and *QOSAL_IMPL_THREAD_STACK_DEFINE* from `Firmware/Libs/uwb-stack/libs/qosal/include/dummy/qosal_impl.h`.
5. Check if QLOG implementation in `Firmware/Libs/uwb-stack/libs/qhal/src/nrfx/qlog_impl.h` is aligned to needs of your OS and system design.

11.3.3.3.3 Port the Project

Copy the `Firmware/Projects/DW3_QM33_SDK/FreeRTOS/` directory to `Firmware/Projects/DW3_QM33_SDK/<os_name>/`, substituting instances of **FreeRTOS** with your **<os_name>** in all subdirectories. Then, adjust the contents to fit your OS-specific configurations:

1. To compile SDK with your OS, you need to replace **CONFIG_QOSAL_IMPL_FREERTOS** with **CONFIG_QOSAL_IMPL_<os_name>** in `Firmware/Projects/DW3_QM33_SDK/<os_name>/DW3_QM33_SDK-<os_name>-Common/cmakefiles/DW3_QM33_SDK-<os_name>.cmake`.
2. Update **ProjectName** string in `Firmware/Projects/DW3_QM33_SDK/<os_name>/<board_name>/ProjectDefinition/ProjectName.c`.
3. There are also two FreeRTOS specific definitions in `Firmware/Projects/DW3_QM33_SDK/FreeRTOS/<board_name>/ProjectDefinition/uwb_stack_llhw.cmake` that you may need to update for your OS needs:
   - `NRF_LOG_TIMESTAMP_DEFAULT_FREQUENCY`,
   - `RTC1_ENABLED`.
4. Update FreeRTOS with **os_name** in Projects/DW3_QM33_SDK/<os_name>/<board_name>/CreateTarget.py.
5. Set the proper name of your OS in **MY_OS** flag in `Firmware/Projects/DW3_QM33_SDK/<os_name>/<board_name>/project_common.cmake`.
6. Align **CONFIG_QPLATFORM_IMPL_QM33_QHAL_NON_ZEPHYR** flag in `Firmware/Projects/DW3_QM33_SDK/FreeRTOS/DW3_QM33_SDK-FreeRTOS-Common/cmakefiles/DW3_QM33_SDK-FreeRTOS.cmake`. Set it to **OFF** when Zephyr OS is used.

11.3.3.3.4 Other alignments

1. Replace **FreeRTOS** in `Firmware/Projects/Common/scripts/CreateTargetCommon.py` with name defined by **MY_OS** cmake flag.
2. Check if priorities definitions in `Firmware/Src/Boards/Inc/Common/int_priority.h` are aligned with needs of your OS.
3. You can also remove FreeRTOS specific parts:
   - `Firmware/Projects/DW3_QM33_SDK/<os_name>/<board_name>/ProjectDefinition/FreeRTOSConfig.h`,
   - `Firmware/Projects/DW3_QM33_SDK/<os_name>/DW3_QM33_SDK-<os_name>-Common/hooks.c`
4. To keep VS Code integration, replace occurrences of **FreeRTOS** string in `Firmware/.vscode/tasks.json` and `Firmware/.vscode/launch.json` with name defined by **MY_OS** CMake flag.

11.3.4 Porting Project

11.3.4.1 Introduction

Adding support for the new platform requires porting the Project folder to new platform. For architectural details of Project folder refer to *Projects folder*.

Note: We describe the porting process with the assumption that the supported OS is FreeRTOS. For the other OS see *Porting to other OS*.

11.3.4.2 Porting process

For adding support for a new board to the Project folder, the following steps must be done:

1. Create a new instance of Project folder `Firmware/Project/DW3_QM33_SDK/FreeRTOS/<board>` for the board you are porting the project to.

   Note: It is recommended to create a copy of the already existing target (e.g. `Firmware/Project/DW3_QM33_SDK/FreeRTOS/nRF52840DK`) and adjust it for the new one being created.

2. Adjust files inside the folder:
   - `CreateTarget.py` - update the board variable (*board=<board_name>*),
   - `<board>.ld` - rename and adjust linker script for your platform,
   - `project_common.cmake` - update board-specific fields. For detailed description of the fields, see *Projects folder*.
   - `ProjectDefinition/ProjectName.c` - update the ProjectName description with the proper board name.
   - `uwb_stack_llhw.cmake` - customize the GPIO board pinouts for the board.

     Note: `uwb_stack_llhw.cmake` is a part of BAL layer which is responsible for customization of board specific details. See *Porting BAL*.

11.3.5 Porting to other Board

11.3.5.1 Introduction

For porting the project to the target board, these steps are to be done:

Note: Depending on the board family you want to port, some steps may not need to be done because the particular layers may already be implemented in the SDK.

1. Porting HAL Layer *Porting HAL* (if required)
2. Porting BAL Layer *Porting BAL* (if required)
3. Porting OS Layer *Porting to other OS* (if required)
4. Porting Project *Porting Project*

12 Calibration and Configuration

12.1 Overview

12.1.1 Introduction

The calibration and configuration system allows to push some settings to the device.

The calibration and configuration settings are used to specify:

- The antenna set used to transmit and receive frames
- The antenna set and the frame segment used to compute the Angle of Arrivals (AoAs)
- The lookup table matching the mounted antenna to translate measured Phase Difference of Arrival (PDoA) to AoA
- The TX power for different transmission phases, the antenna delay, ...

The calibration and configuration are based on a Key/Value mechanism. It is possible to set the calibration and configuration through UCI or through CLI build.

The calibration and configuration are contained inside a JSON file, which is parsed and sent to the device through UCI, using the UCI version of the binary. The CLI version allows to set or get calibration values manually.

12.1.1.1 Calibration important notes

Note: Configuration and calibration settings are non-volatile, they are retained after power cycle or firmware update (without Non-Volatile Memory erase).

Warning: If you are not using a QM33120WDK1, DWM3001CDK or Type2AB EVB (e.g. DWM3000EVB), you must flash the calibration before starting a ranging session.

Warning: Using the default calibration leads to decreased ranging accuracy. The device uses the default settings under one of the following circumstances:

- when you receive a new development kit board,
- when whole FLASH / NVM is erased,
- when you use the RESTORE CLI command,
- when you use reset_calibration python script.

In any of these situations, calibrate your board manually before starting the ranging session. A step-by-step guide can be found in the chapter *Distance and Angle Calibration*.

The compatibility of the calibration data from one SDK version to another is not guaranteed and ranging may be not possible at all.

For instructions, please refer to the *Pushing calibration to a device* section.

12.1.2 Understanding the Calibration and Configuration

12.1.2.1 TxPower

The Calibration allows to define up to 8 reference frames. Each of these reference frames is configured with a set of properties (for instance, payload length, frame type, power index...).

When a frame is transmitted by the chip, the firmware will select the nearest corresponding reference frame and use the transmit power of this reference frame as a working base. This transmit power is then adjusted according to the characteristics of the frame to be sent in order to optimize the transmit power while staying within the regulation requirements.

The corresponding power index calibration key allows to change the transmit power of the frame and to stay compliant with the FCC requirements.

12.1.2.1.1 Low Noise Amplifier

Note: Please note that the LNA is not part of the provided development kit.

A dedicated GPIO on the chip is activated (goes high) when the chip is in receive mode.

QM33100 and DW3000 have an internal LNA but it is possible to enhance reception performance furthermore by integrating a LNA to the front end of your design. To enable the LNA, adjust the relevant configuration key ant<K>.lna.

Once the LNA key is enabled, a dedicated GPIO on the UWB chip is activated (goes high) when the chip is in a receive mode to activate the LNA.

This GPIO depends on the UWB chip, following this table:

Table 12.1: LNA GPIO

Chip	Signal name
DW3xx0	GPIO6 / EXTRXE[19]
QM331x0	GPIO5 / EXTRXE

We recommend the UWB LNA QM14068[20]

For more information, check the UWB chip datasheet and the Application Note APS304: Increasing Range Using an External LNA DW3000 / QM33100[21]

[19] EXTRXE - External Receiver Enable
[20] https://store.qorvo.com/products/detail/qm14068-qorvo/695711/
[21] https://www.qorvo.com/products/d/da009232

12.1.2.1.2 Power Amplifier

Note: Please note that the PA is not part of the provided development kit.

Enhancing transmission performance is possible by integrating a PA into your design. To enable the PA, adjust the relevant configuration key ant<K>.pa.

Once the PA key is enabled, a dedicated GPIO on the UWB chip is activated (goes high) when the chip is in a transmit mode to activate the PA.

This GPIO depends on the UWB chip, following this table:

Table 12.2: PA GPIO

Chip	Signal name
DW3xx0	GPIO5 / EXTTXE[22]
QM331x0	GPIO4 / EXTTXE

We recommend the UWB PA QM14070[23]

For more information, check the UWB chip datasheet.

12.1.2.2 Calibration

Depending on the board and the antenna design, it is possible to set calibration data which are applied to the chip, like the TxPower, the antenna delay, the PDoA offset, ...

12.2 Pushing calibration to a device

12.2.1 Pushing the calibration over UCI

The *uwb-qorvo-tools* package, located in *Tools/uwb-qorvo-tools*, provides the *load_cal* script. This script can be utilized to load calibration values from a JSON file into the device through UCI.

Note: For more detailed information about *load_cal*, including command options and descriptions, as well as some default calibration files for supported targets, refer to the *uwb-qorvo-tools* documentation.

12.2.2 Pushing the calibration over GUI

The Qorvo One TWR GUI allows for loading calibration into the device using the *Device Configuration popup*.

Default Calibration and Configuration

The GUI uses default calibration and configuration data encapsulated within JSON files. To load default configuration:

1. Click *Reset all Configuration* button and wait for the finish.
2. Click the *Save* button.

To extract and analyze the current calibration:

1. Click the *Export Calibration* button and choose your desired path for the JSON file.

[22] EXTTXE - External Transmit Enable
[23] https://store.qorvo.com/products/detail/qm14070-qorvo/705311/

2. Click the *Save* button.

Customizing Calibration Files

The exported default configuration JSON files can be customized to meet specific requirements. To use a customized calibration file:

1. Modify the exported JSON file as needed.
2. Click the *Import Calibration* button and import the customized JSON calibration file.
3. Click the *Save* button.

By following these steps, you can tailor the calibration settings to fit your specific requirements.

12.3 Calibration and Configuration keys dictionary

For details about L1 Configuration and Calibration keys, please refer to the `uwb-l1-configuration` document.

12.4 Erase calibration

To apply a new calibration, it is recommended to reset the already existing calibration of the UWB device.

It is also possible to fully erase it if needed.

12.4.1 How to reset calibration

After following Prerequisites chapter from uwb-qorvo-tools documentation, use reset_calibration.py script (under scripts/generic/device/reset_calibration) to reset the calibration to default one.

More details about this script can be found in uwb-qorvo-tools documentation.

Warning: If you are not using a QM33120WDK1, DWM3001CDK or Type2AB EVB (e.g. DWM3000EVB), you must flash the calibration before starting a ranging session.

Warning: Using the default calibration leads to decreased ranging accuracy (see section *Calibration important notes* for more details).

12.4.2 How to erase calibration

Erasing the calibration is supported using both embedded J-link (J-Link OB) as well as external J-Link, depending on the board that is being used.

Steps to erase calibration are identical to those described in section flashing. Instead of flashing, erase the device using the “Erase chip” button, then reflash the firmware.

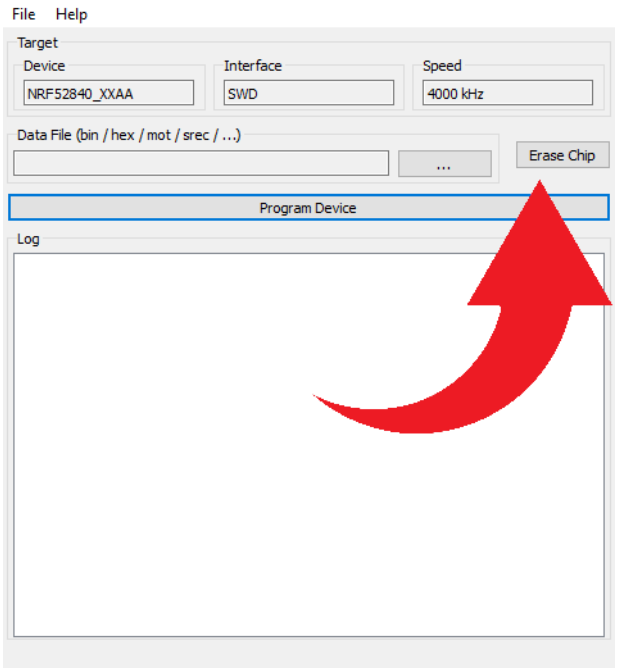

Fig. 12.1: Erase chip button

Warning: If you are not using a QM33120WDK1, DWM3001CDK or Type2AB EVB (e.g. DWM3000EVB), you must flash the calibration before starting a ranging session.

12.5 Distance and Angle Calibration

Warning: For any production tests and end product calibration, please refer to application note APS312 Production Tests for DW3000-Based Products[24].

For evaluation, measurements over the air can already provide good performances by following simple steps.

Note: It is possible to automatically calibrate the distance by using the GUI, please check *Calibration* in the GUI chapter for more details.

[24] https://www.qorvo.com/products/p/QM33120W#documents

12.5.1 Distance

You can calibrate the distance manually by tweaking the antenna delay calibration key `ant<x>.ch<y>.ant_delay` (with `x` corresponding to the antenna path used and `y` the channel used). See *Calibration and Configuration keys dictionary* for more details.

This calibration is stored in NVM so you can perform it only once.

Here are the steps to follow in order to calibrate the antenna delay using CLI and UCI builds:

12.5.1.1 CLI

- Put a **known distance** between your two devices. It is recommended to place them approximately two meters apart.
- Start a Two-Way Ranging session, using `INITF` and `RESPF` commands (see *INITF/RESPF command* for more details).
- Stop the ranging on the device you want to calibrate.
- Adjust its antenna delay, using `CALKEY` command (see *IDLE time Commands* for more details).
- Restart the Two-Way Ranging session.
- Repeat the last three steps until reported distance corresponds to real measured distance.

12.5.1.2 UCI

- Put a **known distance** between your two devices. It is recommended to place them approximately two meters apart.
- Start a Two-Way Ranging session, using `run_fira_twr` script on both devices. It is recommended to use `-stat` parameter to get the average distance at the end of the script (see *Tools/uwb-qorvo-tools/UWB-Qorvo-Tools-guide.pdf* for more information).
- Wait for the script to finish.
- On the device you want to calibrate, adjust the antenna delay, using `set_cal` script (see *Tools/uwb-qorvo-tools/UWB-Qorvo-Tools-guide.pdf* for more information).
- Restart the Two-Way Ranging session.
- Repeat the last three steps until reported distance corresponds to real measured distance.

12.5.2 Angle

You can calibrate the angle manually by tweaking the PDoA (Phase Difference of Arrival) offset calibration key `ant_pair<x>.ch<y>.pdoa.offset` (with `x` corresponding to the antenna pair used and `y` the channel used). See *Calibration and Configuration keys dictionary* for more details.

This calibration is stored in NVM so you can perform it only once.

Here are the steps to follow in order to calibrate the PDoA offset using CLI and UCI builds:

12.5.2.1 CLI

- Put a known angle of **0 degree** between your two devices.
- Start a Two-Way Ranging session, using `INITF` and `RESPF` commands (see *INITF/RESPF command* for more details). If only one device is AoA-capable, use this device to run `INITF`.
- Stop the ranging on the device you want to calibrate.
- Adjust its PDoA offset, using `CALKEY` command (see *IDLE time Commands* for more details).
- Restart the Two-Way Ranging session.
- Repeat the last three steps until reported angle corresponds to known angle (0 degree).

12.5.2.2 UCI

- Put a known angle of **0 degree** between your two devices.
- Start a Two-Way Ranging session, using `run_fira_twr` script on both devices. It is recommended to use `-stat` parameter to get the average angle at the end of the script (see *Tools/uwb-qorvo-tools/UWB-Qorvo-Tools-guide.pdf* for more information). If only one device is AoA-capable, remember to set this device as an initiator.
- Wait for the script to finish.
- On the device you want to calibrate, adjust the PDoA offset, using `set_cal` script (see *Tools/uwb-qorvo-tools/UWB-Qorvo-Tools-guide.pdf* for more information).
- Restart the Two-Way Ranging session.
- Repeat the last three steps until reported angle corresponds to known angle (0 degree).

13 OTP Memory Map

The DW3000 and QM33 UWB chips series contain an on-chip One-Time Programmable (OTP) memory.

This memory can be used to store calibration data such as TX power level and crystal initial frequency error adjustment.

The presented OTP memory map contains values calibrated after production of development kits they do not apply to individual chips.

OTP memory locations are defined in the following tables.

- OTP memory locations are each 32-bits wide.
- OTP addresses are word addresses, so each increment of address specifies a different 32-bit word.
- Memory allocation depends on the OTP Revision written at the address 0x1F.

13.1 OTP Revision 1

Written to OTP

Calibrated and written to OTP

OTP Address	Size (Used Bytes)	Byte_3	Byte_2	Byte_1	Byte_0	Programmed by
0x000	4	64 bit EUID				Customer
0x001	4					
0x002	4	Alternative 64 bit EUID (Selected via reg/SR register)				Customer
0x003	4					
0x004	4	LDO Tune				IC Prod. Test
0x005	4					
0x006	4	{ "0001.0000.0001", "CHIP ID 5 nibbles (20 bits)" }				IC Prod. Test
0x007	4	{ "0001", "LOT ID - 7 nibbles (28 bits)" }				IC Prod. Test
0x008	3		VBAT @ 3.0 V	VBAT @ 3.62 V	VBAT @ 1.62 V	IC Prod. Test
0x009	1				Temp @ 22±2° C	IC Prod. Test
0x00A	4	Bias Tune				IC Prod. Test
0x00B	4	Antenna Delay - Rx RF Loop		Antenna Delay - Tx RF Loop		IC Prod. Test
0x00C	4	PDoA Iso. Ch9 RF2 → RF1	PDoA Iso. Ch9 RF1 → RF2	PDoA Iso. Ch5 RF2 → RF1	PDoA Iso. Ch5 RF1 → RF2	IC Prod. Test
0x00D	4	W.S. Lot ID [3]	W.S. Lot ID [2]	W.S. Lot ID [1]	W.S. Lot ID [0]	IC Prod. Test
0x00E	2			W.S. Lot ID [5]	W.S. Lot ID [4]	IC Prod. Test
0x00F	3		W.S. Wafer no.	W.S. Y Loc	W.S. X Loc	IC Prod. Test
0x010	4	Ch5 Tx Power Level - PRF16				Customer
0x011	4	Ch5 Tx Power Level - PRF64				Customer
0x012	4	Ch9 Tx Power Level - PRF16				Customer
0x013	4	Ch9 Tx Power Level - PRF64				Customer
0x014	4					Customer
0x015	4					Customer
0x016	4					Customer
0x017	4					Customer
0x018	4	Ch5_PGCNT		Ch9_PGCNT		Customer
0x019	4					Customer
0x01A	4	CH5 Rx Antenna Delay - PRF64		CH5 Tx Antenna Delay - PRF64		Customer
0x01B	4	CH5 Rx Antenna Delay - PRF16		CH5 Tx Antenna Delay - PRF16		Customer
0x01C	4	CH9 Rx Antenna Delay - PRF64		CH9 Tx Antenna Delay - PRF64		Customer
0x01D	4	CH9 Rx Antenna Delay - PRF16		CH9 Tx Antenna Delay - PRF16		Customer
0x01E	1	Frame Duration - us			Xtal_Trim [6:0]	Customer
0x01F	1	Platform ID		Cal Rev	OTP Rev.	Customer
0x020	4	Rx_Tune_Cal: DGS_CFG0				IC Prod. Test
0x021	4	Rx_Tune_Cal: DGS_CFG1				IC Prod. Test
0x022	4	Rx_Tune_Cal: DGS_CFG2				IC Prod. Test
0x023	4	Rx_Tune_Cal: DGS_CFG3				IC Prod. Test
0x024	4	Rx_Tune_Cal: DGS_CFG4				IC Prod. Test
0x025	4	Rx_Tune_Cal: DGS_CFG5				IC Prod. Test
0x026	4	Rx_Tune_Cal: DGS_CFG6				IC Prod. Test
0x027	4	Rx_Tune_Cal: DGC_LUT_0 - CH5				IC Prod. Test
0x028	4	Rx_Tune_Cal: DGC_LUT_1 - CH5				IC Prod. Test
0x029	4	Rx_Tune_Cal: DGC_LUT_2 - CH5				IC Prod. Test
0x02A	4	Rx_Tune_Cal: DGC_LUT_3 - CH5				IC Prod. Test
0x02B	4	Rx_Tune_Cal: DGC_LUT_4 - CH5				IC Prod. Test
0x02C	4	Rx_Tune_Cal: DGC_LUT_5 - CH5				IC Prod. Test
0x02D	4	Rx_Tune_Cal: DGC_LUT_6 - CH5				IC Prod. Test
0x02E	4	Rx_Tune_Cal: DGC_LUT_0 - CH9				IC Prod. Test
0x02F	4	Rx_Tune_Cal: DGC_LUT_1 - CH9				IC Prod. Test
0x030	4	Rx_Tune_Cal: DGC_LUT_2 - CH9				IC Prod. Test
0x031	4	Rx_Tune_Cal: DGC_LUT_3 - CH9				IC Prod. Test
0x032	4	Rx_Tune_Cal: DGC_LUT_4 - CH9				IC Prod. Test
0x033	4	Rx_Tune_Cal: DGC_LUT_5 - CH9				IC Prod. Test
0x034	4	Rx_Tune_Cal: DGC_LUT_6 - CH9				IC Prod. Test
0x035	4	PLL_Lock_Code				IC Prod. Test
0x036 - 0x05F	4	Unallocated				Customer
0x060	4	QSR Register (Special function register)				Reserved
0x061	1				Q_RR Register	Reserved
0x062 - 0x077	4	Unallocated				Customer
0x078	4	AES_Key [127:96] (Big endian order)				Customer
0x079	4	AES_Key [95:64] (Big endian order)				Customer
0x07A	4	AES_Key [63:32] (Big endian order)				Customer
0x07B	4	AES_Key [31:0] (Big endian order)				Customer
0x07C	4	AES_Key [255:224] (Big endian order)				Customer
0x07D	4	AES_Key [223:192] (Big endian order)				Customer
0x07E	4	AES_Key [191:160] (Big endian order)				Customer
0x07F	4	AES_Key [159:128] (Big endian order)				Customer

Fig. 13.1: OTP Revision 1

Description of Customer part of the OTP:

- `0x010`: Not filled after production
- `0x011`: FCC compliant Tx Power value using channel 5
- `0x012`: Not filled after production
- `0x013`: FCC compliant Tx Power value using channel 9
- `0x018`: `PGCOUNT` (pulse generator count) value used during production tests
- `0x01A`: Antenna delays for channel 5 with STS segment length = 64
- `0x01C`: Antenna delays for channel 9 with STS segment length = 64
- `0x01E`: Frame duration used during the production tests (in µs) | Crytal trimming to fine tune UWB central frequency
- `0x01F`: `Platform ID`: unique identifier to distinguish hardware platform | `Cal Rev`: production test version | OTP Revision: organization of OTP memory
- `0x036` - `0x05F`: Free to use
- `0x062` - `0x077`: Free to use
- `0x078` - `0x07F` is recommended for AES key, if needed

14 Licenses in this SDK

Read the header of each file to know more about the license concerning this file.

Following licenses are used directly in the SDK:

- **Qorvo licenses** with identifiers **LicenseRef-Qorvo-1**, **LicenseRef-Qorvo-2**: majority of the source code (*Firmware/LICENSES/* directory).
- **Nordic Semiconductor ASA license**: *sdk_config.h* files (header of respective file).
- **CodeSourcery Inc license**: nRF linker scripts (header of respective file).
- **sha256 license**: sha256 library (*Firmware/Libs/uwb-stack/config_manager/src/sha256.c*)

Following licenses are included from nRF5 SDK:

- **nRF5 SDK license**: The majority of the source code included in the nRF5 SDK (*Firmware/SDK_BSP/Nordic/NORDIC_SDK_17_1_0/documentation/nRF5_Nordic_license.txt*).
- **nRF5 Garmin Canada license**: Files released by Garmin Canada included in the nRF5 SDK (*nRF5_Garmin_Canada_license.txt*).
- **SoftDevice license**: The SoftDevice and its headers

  (*Firmware/SDK_BSP/Nordic/NORDIC_SDK_17_1_0/components/softdevice/sxxx/doc/<license file>*).
- **ARM 3-clause BSD license**: CMSIS and system files

  (*Firmware/SDK_BSP/Nordic/NORDIC_SDK_17_1_0/components/toolchain*).
- **FreeRTOS license**: FreeRTOS configuration files (*FreeRTOSConfig.h* in the examples that show how to use FreeRTOS).
- **Third-party licenses**: All third-party code contained in *Firmware/SDK_BSP/Nordic/NORDIC_SDK_17_1_0/external/* (respective licenses included in each of the imported projects).

The provided HEX files were compiled using the projects located in the folders. For license and copyright information, see the individual .c and .h files that are included in the projects.

15 Contact Information

For the latest specifications, additional product information, worldwide sales and distribution locations:
Web: www.qorvo.com
Tel: 1-844-890-8163
Email: customer.support@qorvo.com

16 Important Notice

The information contained in this Data Sheet and any associated documents (“Data Sheet Information”) is believed to be reliable; however, Qorvo makes no warranties regarding the Data Sheet Information and assumes no responsibility or liability whatsoever for the use of said information. All Data Sheet Information is subject to change without notice. Customers should obtain and verify the latest relevant Data Sheet Information before placing orders for Qorvo® products. Data Sheet Information or the use thereof does not grant, explicitly, implicitly or otherwise any rights or licenses to any third party with respect to patents or any other intellectual property whether with regard to such Data Sheet Information itself or anything described by such information.

DATA SHEET INFORMATION DOES NOT CONSTITUTE A WARRANTY WITH RESPECT TO THE PRODUCTS DESCRIBED HEREIN, AND QORVO HEREBY DISCLAIMS ANY AND ALL WARRANTIES WITH RESPECT TO SUCH PRODUCTS WHETHER EXPRESS OR IMPLIED BY LAW, COURSE OF DEALING, COURSE OF PERFORMANCE, USAGE OF TRADE OR OTHERWISE, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Without limiting the generality of the foregoing, Qorvo® products are not warranted or authorized for use as critical components in medical, life-saving, or life-sustaining applications, or other applications where a failure would reasonably be expected to cause severe personal injury or death. Applications described in the Data Sheet Information are for illustrative purposes only. Customers are responsible for validating that a particular product described in the Data Sheet Information is suitable for use in a particular application.

FiRa, FiRa Consortium, the FiRa logo, the FiRa Certified logo, and FiRa tagline are trademarks or registered trademarks of FiRa Consortium or its licensor(s)/ supplier(s) in the US and other countries and may not be used without permission. All other trademarks, service marks, and product or service names are trademarks or registered trademarks of their respective owners.