Instituto Tecnológico
y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación del 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática
Maestría en Diseño Electrónico

A Practical Start with Zephyr RTOS

TRABAJO RECEPCIONAL que para obtener el **GRADO** de
MAESTRO EN DISEÑO ELECTRÓNICO

Presenta: **EARL ORLANDO RAMÍREZ SÁNCHEZ**
Director: **OMAR HUMBERTO LONGORIA GÁNDARA**
Codirector: **JOSÉ LUIS PIZANO ESCALANTE**

Tlaquepaque, Jalisco. 2 de diciembre de 2024.

To my family, for their unconditional love, their support every step of the way, for being my constant source of inspiration, and for understanding when I had to take time away from them to focus on my studies.

To my teachers, for sharing their knowledge, fostering my curiosity, and guiding me with patience and dedication.

To my advisors, for their patience, guidance, trust, and for teaching me to overcome challenges through effort and determination.

This work and this master’s degree are the result of the support, encouragement, and inspiration I have received from all of you. With heartfelt gratitude and admiration, I dedicate this accomplishment to you.

Summary

The field of embedded systems has grown significantly in recent years, driving increased demand for advanced Real-Time Operating Systems (RTOS). These systems manage complex tasks, ensure fast responses, and optimize resource usage. Zephyr RTOS stands out as an open-source solution with solid community support and compatibility across various hardware platforms. Supported by the Linux Foundation, its modular design makes it an appealing choice for scalable and customizable solutions.

However, new users often face challenges when configuring it for hardware that lacks native support, especially when working with Devicetree structures and Kconfig files. While powerful, these components require a solid understanding of system configuration and hardware integration.

This case study provides a practical guide to reduce the learning curve and help developers adapt Zephyr to their projects. It covers critical topics such as adding hardware support for the NXP FRDM-K66F board and a Liquid Crystal Display (LCD) with the PCD8544 controller, handling interrupts, managing multithreading and synchronization, and integrating external devices.

Contents

1. Introduction

The field of embedded systems is quickly evolving, with more applications requiring multitasking, real-time processing, and efficient resource use. An RTOS provides the architecture to manage these complexities by breaking operations into threads, prioritizing tasks, and handling interrupts [1].

Among modern options, Zephyr stands out for its flexibility, modularity, and strong community support [2]. Backed by the Linux Foundation [3], its open-source nature makes it an appealing and reliable choice for developers.

This chapter introduces the motivation and scope of this research, providing an overview of the challenges faced when adapting Zephyr RTOS to unsupported hardware platforms. It outlines the problem statement, the general and specific objectives of the study, related works that inspired the approach, and the contributions.

1.1. Problem Statement

Despite its many advantages, Zephyr presents challenges when adapted to unsupported hardware. This is particularly true for those unfamiliar with its configuration tools, such as Kconfig and Devicetree, which are essential for defining how the system interacts with hardware. Understanding how these tools connect the build system to the hardware configuration requires a considerable learning curve.

Furthermore, the need for targeted resources and streamlined guides for new users significantly impacts the onboarding experience, creating a barrier to entry for those aiming to explore its capabilities. This gap increases the need for case studies and practical examples that facilitate an intuitive understanding of Zephyr's adaptation process. Without such resources, developers may face extended development times, reduced productivity, and delayed project delivery, limiting a broader adoption for unsupported hardware platforms.

Adopting Zephyr for established industrial products presents significant challenges, primarily due to rigid architectures optimized for specific applications and legacy codebases often built on proprietary or legacy standards. Integration may require substantial efforts, such as redesigning hardware-specific drivers or adapting proprietary communication protocols. Additionally, industrial sectors typically require rigorous revalidation and certification processes, such as compliance with safety-critical standards, which increase the development time and costs when transitioning to a new RTOS.

This research provides a detailed case study focused on extending support to the NXP FRDM-K66F board and the PCD8544 LCD controller.

1.2. General Objective

This case study aims to simplify Zephyr adoption for new users by reducing the learning curve and providing practical demonstrations of hardware adaptation and multitasking features. It explains how to adapt Zephyr to hardware that is not natively supported, demonstrates how to create drivers for new devices by following the official guidelines, provides examples of using peripheral interrupts to handle communication with external devices, and explores multitasking features, including task scheduling and the use of synchronization mechanisms to manage complex operations.

1.3. Specific Objectives

The specific objectives of this research are as follows:

- Demonstrate how to adapt Zephyr RTOS to new hardware using the NXP FRDM-K66F board as an example.
- Develop and integrate a driver for the PCD8544 LCD controller into the Zephyr environment, leveraging its Serial Peripheral Interface (SPI) and General-Purpose Input/Output (GPIO) peripheral drivers.
- Illustrate the configuration and handling of GPIO interrupts to manage hardware events, such as button presses.

- Explore Zephyr's multitasking capabilities by implementing task scheduling and synchronization mechanisms, including semaphores, message queues, mutexes, atomic variables, and events.

1.4. Related Works

A notable example of research that inspired this study is the collaborative effort between Antmicro and Silicon Labs, which focused on enabling Zephyr support for the EFR32 Micro-Controller Unit (MCU) family [4]. This work highlights a systematic approach to integrating new hardware. The process addressed complex challenges such as resolving Hardware Abstraction Layer (HAL) co-dependencies, ensuring licensing compatibility, and modernizing outdated codebases to align with its evolving Application Programming Interface (API) standards. Its structured methodology and lessons learned align with the objectives of this case study of extending hardware support through practical examples, including driver adaptation, interrupt handling, and multitasking for the NXP FRDM-K66F board and the PCD8544 LCD controller.

Another relevant work is the tutorial on mastering Zephyr driver development [5]. It provides a comprehensive guide to understanding its modular architecture and best practices for driver development. The tutorial elaborates on using the main tools of Zephyr: Devicetree and Kconfig, which are used to define hardware interactions, design custom APIs, and develop out-of-tree drivers. The proposals in [5] emphasize the importance of modular design and abstraction layers in the driver framework through hands-on examples, such as implementing a sensor driver, which aligns with the methodology of adapting unsupported hardware of this work.

The official Zephyr Project website offers a step-by-step guide to driver development [6], emphasizing the importance of adhering to best practices for efficient hardware integration.

1.5. Contributions

The contributions of this study include:

- Provide a practical framework for adapting Zephyr to unsupported hardware platforms, using the NXP FRDM-K66F board as a case study. It details the configuration of critical components like Kconfig and Devicetree, offering a reusable approach for similar tasks.

- Adapt a bare-metal driver for the PCD8544 LCD controller to operate within the Zephyr environment. This example serves as a guide for developers seeking to integrate custom hardware components.
- Demonstrate the configuration and handling of GPIO interrupts, showcasing how to trigger an Interrupt Service Routine (ISR) in response to external signals such as button presses.
- Explore the multitasking features, presenting examples of thread creation, task scheduling, and synchronization mechanisms to manage concurrent operations effectively.
- Present a complete practical implementation to illustrate the real-world challenges and solutions for integrating new hardware and software.

2. Materials

The success of any embedded systems project depends on selecting the right combination of hardware and software tools. This chapter introduces the critical materials used in this study, highlighting their roles and interconnections. It begins with an overview of the NXP Kinetis MK66F microcontroller and the Freedom FRDM-K66F evaluation board, which form the hardware foundation. Next, it explores the PCD8544 LCD controller, integrated as a custom peripheral in this research. Additionally, the chapter details essential software tools and frameworks for development, including the Zephyr Software Development Kit (SDK), West meta-tool, and the CMake build system, as well as the Devicetree and Kconfig systems, indispensable for hardware abstraction and configuration in Zephyr-based projects. Finally, it outlines the project structure and build system, emphasizing how these components integrate to create a cohesive development environment.

2.1. NXP Kinetis MK66F Microcontroller

The NXP Kinetis MK66F microcontroller is part of the Kinetis K6x family, built on the ARM Cortex-M4F core [7]. It runs at a maximum clock speed of 180 MHz and features a floating-point unit (FPU), which helps with complex calculations and is especially useful for signal-processing tasks.

The microcontroller includes 2 MB of flash memory and 256 KB of SRAM and supports many communication protocols, such as Inter-Integrated Circuit (I2C), SPI, and Universal Asynchronous Receiver/Transmitter (UART).

2.2. Freedom FRDM-K66F Board

The Freedom FRDM-K66F [8] is an evaluation board designed to showcase the capabilities of the NXP MK66F microcontroller. It is a reliable and cost-effective platform for developing and testing embedded applications. The board includes built-in components that simplify development, including a Red-Green-Blue Light-Emitting Diode (RGB LED) and buttons, an Open Serial Debug Adapter (OpenSDA), an accelerometer and a magnetometer, and pin headers compatible with many Arduino shields to connect additional components, such as displays, sensors, or communication devices without complex wiring or hardware modifications.

Fig. 2-1 NXP Freedom FRDM-K66F board [8].

2.3. LCD with the PCD8544 Controller

The PCD8544 is a low-power LCD controller often used in small graphic displays like older Nokia phones [9]. It controls a 48x84-pixel screen and communicates with the microcontroller using SPI.

This LCD controller is known for its low power consumption, making it suitable for battery-operated devices. It works with voltages between 2.7V and 3.3V, which makes it easy to use with most modern microcontrollers, including the NXP MK66F. It has a built-in character generator, so it can display simple text without creating custom graphics. It can use its Random-Access Memory (RAM) to show custom images or bitmaps for more advanced displays.

The required signals to interface this LCD to a microcontroller are as follows:

- *CLK* (Clock): SPI clock to synchronize data transfer.
- *DIN* (Data Input): SPI data input to transfer data from the microcontroller.
- *CE* (Chip Enable): SPI chip select to activate this device for communication.
- *DC* (Data/Command): GPIO to distinguish whether the SPI input is data or command.
- *RST* (Reset): GPIO to reset the display during initialization.
- *BL* (Backlight): GPIO or Pulse-Width Modulator (PWM) to adjust the backlight brightness.
- *VCC* (Power): Power supply input (2.7V to 3.3V).
- *GND* (Ground): Common ground connection.

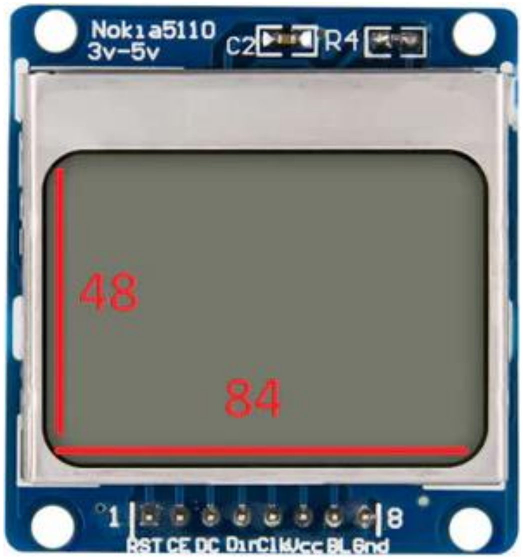

Fig. 2-2 48x84 LCD with the PCD8544 controller, used in old phones like the Nokia 5110 [10].

2.4. Zephyr RTOS

Zephyr is a lightweight RTOS designed for embedded devices. It offers modular architecture and comprehensive hardware abstraction and supports multitasking, driver development, and efficient peripheral handling, making it ideal for resource-constrained systems.

Zephyr is an attractive choice in the RTOS landscape, offering modern features and advantages. As an open-source project, it ensures transparency and continuous improvement, and the Linux Foundation and major technology companies support it. Its flexibility and modular architecture allow developers to customize it to suit a wide range of use cases, from lightweight IoT devices to complex applications. Zephyr supports multiple architectures, including ARM, RISC-V, and x86, and is compatible with over 750 development boards [11], providing extensive hardware versatility. Its secure and robust kernel includes notable features like memory protection, fault handling, multiple scheduling algorithms, and integrated cryptographic libraries, making it an ideal choice for real-time, secure, and scalable embedded systems [12].

Fig. 2-3 Zephyr RTOS logo [11].

2.4.1 Zephyr SDK

The first step in setting up Zephyr RTOS is to install the Zephyr SDK, which includes essential tools such as compilers, linkers, and debugging utilities. The official Zephyr Getting Started Guide [12] outlines the installation process and the prerequisites.

2.4.2 West Meta-Tool

The West meta-tool is essential for managing Zephyr-based projects [13]. It is a project management [14] and a build tool [15]. It is a wrapper for various commands, simplifying tasks such as initializing projects, fetching code, and building applications.

To get started, use the *west init* command to initialize a workspace where the project and its dependencies are stored. Then, run *west update* to download and set up all the necessary modules, ensuring that the workspace includes the latest versions of libraries and components.

The *west build* command compiles the project and simplifies the process of specifying board types, source files, and build directories. For instance, the board can be specified with the --*board (-b)* option: *west build -b frdm_k66f*. The --*pristine (-p) always* option ensures that the build directory is cleaned and recreated: *west build -p always -b frdm_k66f*.

The *west flash* command can then program the compiled firmware to the target board. It directly interfaces with supported debuggers, streamlining the flashing process without needing separate tools. Likewise, the *west debug* command allows for the GNU debugger (GDB) interface.

2.4.3 Build System (CMake)

CMake is the primary tool for building applications with the Zephyr kernel [16]. It divides the build process into two stages: the configuration phase and the build phase. The configuration phase processes *CMakeLists.txt* scripts to set up a project model and generate platform-specific build scripts. The build phase compiles and links the source code into final output files.

a) Configuration Phase

The configuration phase begins when the *cmake* command is run, specifying the source application directory and the target board. During this phase, CMake processes the main *CMakeLists.txt* file in the project directory and references other *CMakeLists.txt files* throughout the Zephyr build tree. This structured processing builds a complete project model.

CMake collects information from the board and application-specific configurations, creating build scripts that define the steps for compiling and linking the application. These scripts, stored in the build directory, guide the build phase. Additionally, CMake may generate preprocessor macros and configuration headers needed for compilation.

b) Build Phase

In the build phase, the generated build scripts compile and link the source code, producing the final output files (e.g., *zephyr.elf*, *zephyr.hex*) ready to flash onto the target board. The build system recompiles only modified parts, saving time. However, specific changes may require re-running the configuration phase before building.

CMake uses targets to organize the build process. The app target is typically used for application source files. A basic *CMakeLists.txt* file should include the following commands and can be updated to include more source files.

```
cmake_minimum_required(VERSION 3.20.0)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(my_zephyr_app)

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

2.4.4 Devicetree

The Devicetree describes the hardware in a structured and readable way [17]. It consists of two parts: Devicetree Source (DTS) files and bindings, which help the build system generate header files that reference and access the hardware during build time. The device tree helps the system know what hardware is available and how to set it up.

a) Devicetree Sources

The DTS files are hierarchical text files that describe hardware components and their interconnections [18]. Each hardware part is represented as a "node" in the tree, which can have child nodes showing related components. Each node has properties that provide details about the hardware, such as its address or configuration, enabling hardware settings to be defined without changing the source code.

The struct device holds vital information about each device, including its name and configuration. Initialization levels determine when drivers are set up during the boot process. Drivers without dependencies are loaded early, while those that rely on other drivers are initialized later, ensuring devices start in the correct order.

In the following example, an I2C device is at address 0x40003000 with a register size 0x1000. The compatible property indicates the type of device, allowing the system to choose the correct driver.

```
/i2c@40003000 {
    compatible = "mydevice,i2c";
    reg = <0x40003000 0x1000>;
};
```

b) Devicetree Overlay

Devicetree overlay (**.overlay*) files extend the base configuration to adjust for specific applications [19]. These files are stored in the project directory and help customize the board without modifying the original DTS files.

The following example updates the I2C1 clock frequency to 100 KHz, regardless of the configuration in the original DTS files.

```
&i2c1 {
    status = "okay";
    clock-frequency = <100000>;
};
```

c) Devicetree Bindings

While DTS files describe the hardware, bindings define how these descriptions should be interpreted by providing metadata that tells the build system how to interpret each node [20]. The bindings are YAML files that specify property types, constraints, and the semantics of node properties.

The following binding example matches the previous DTS example because the compatible properties are identical.

```
compatible: "mydevice,i2c"

properties:
  reg:
    type: array
    required: true
```

2.4.5 Configuration System (Kconfig)

During the building process, the Zephyr configuration system adapts the kernel and its subsystems to specific application and platform needs [21]. This method, also used by the Linux kernel, allows developers to set up configurations without modifying source code. Kconfig options, called symbols, are defined in Kconfig files to describe dependencies between symbols and ensure valid configurations. Symbols can be grouped into menus and sub-menus to keep the configuration organized, improving usability in interactive interfaces.

a) Kconfig Files and Tools

Kconfig files organize configuration options, arranging various settings into menus, making them easier to find and modify. During the building process, Kconfig generates a file called *autoconf.h*, which includes macros for conditional compilation. This helps to remove unused code, resulting in a smaller and more efficient final application. Project-specific configurations are stored in the *prj.conf* file.

Developers can add specific lines to the configuration file to enable certain features. For example, the following line would activate floating-point support.

```
CONFIG_FPU=y
```

Although it is possible to edit the *zephyr/.config* file manually, Zephyr offers tools that simplify this process.

The *menuconfig* tool displays options in a text-based menu format in the terminal.

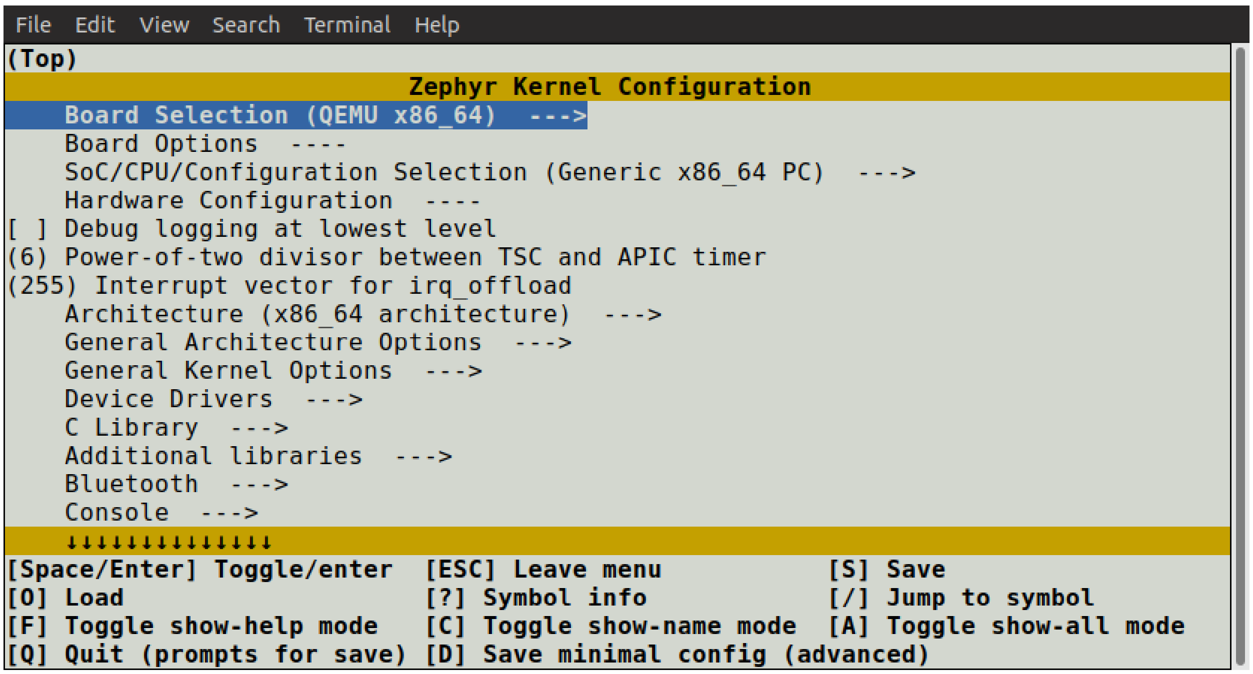

Fig. 2-4 Zephyr's menuconfig tool to configure the Kconfig [22].

The *guiconfig* provides a graphical way to adjust the kernel configurations. Both tools allow for quick configuration modification and understanding of how different settings are connected.

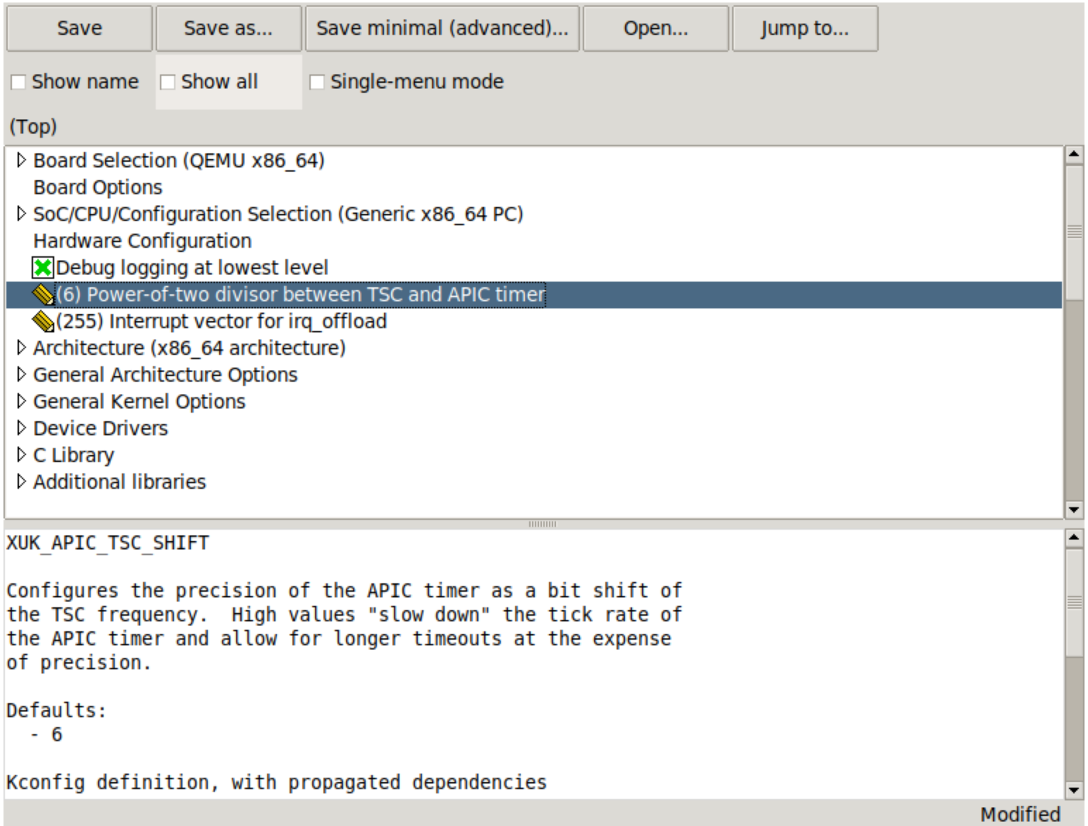

Fig. 2-5 Zephyr's guiconfig tool to configure the Kconfig [22].

2.4.6 Project Structure and Build System

The project structure organizes essential files and configurations, providing a clear layout for development and hardware customization.

The following file structure represents a basic project.

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

- *CMakeLists.txt*: The entry point for the build system to link the project to the CMake infrastructure.
- *app.overlay*: An optional Devicetree overlay for hardware customization.
- *prj.conf*: The primary Kconfig fragment to define configuration symbols.
- *src/*: Directory containing source code.

Zephyr also looks for Devicetree and Kconfig files in the following predefined paths during the build process.

- Devicetree paths:
    - *<ZEPHYR_BASE>/boards/<ARCH>/<BOARD>/*
    - *<ZEPHYR_BASE>/dts/*
    - *<APP_DIR>/boards/*
    - *<APP_DIR>/dts/*
- Kconfig paths:
    - *<ZEPHYR_BASE>/boards/<ARCH>/<BOARD>/*
    - *<APP_DIR>/*

3. Methods

This chapter outlines the methodology for adapting and implementing embedded system components and features for the FRDM-K66F platform and associated peripherals. It describes the steps to extend board support, integrate supported and custom peripherals, and implement application functionality with real-time multitasking.

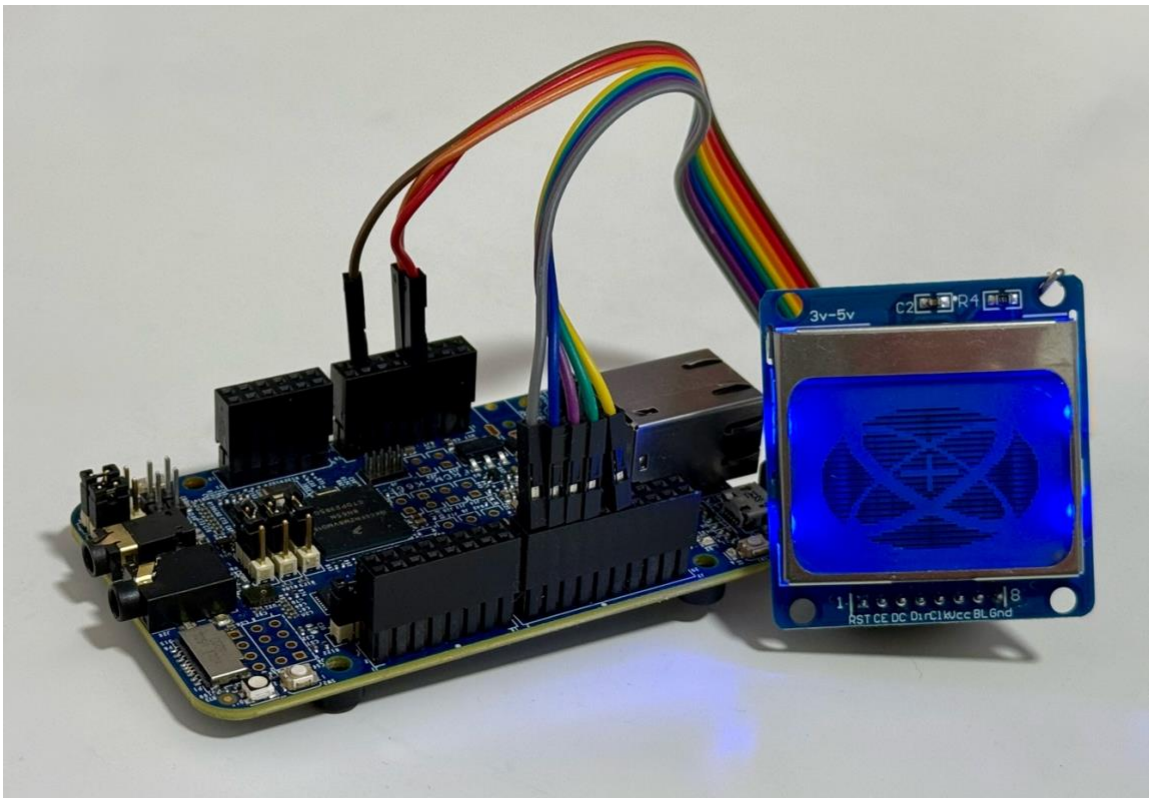

Fig. 3-1 An LCD Connected to the FRDM-K66F Board.

3.1. Extending Board Support

Adding support for the NXP FRDM-K66F board involves creating a dedicated directory within the project structure and populating it with configuration and metadata files as outlined in the Zephyr Board Porting Guide [23]. These files integrate the board into the Zephyr build system and define its hardware features. The FRDM-K66F implementation leverages similarities with the existing support for the FRDM-K64F board.

A directory containing mandatory and optional files to define the board is created under *zephyrproject/zephyr/boards/nxp/frdm_k66f/*.

File name	Purpose	Mandatory or optional
board.yml	Defines high-level metadata about the board, such as the name, vendor, and associated System-on-Chip (SoC).	Mandatory
frdm_k66f.dts	The primary hardware description file, which specifies the SoC, memory layout, peripherals, and connectors.	Mandatory
Kconfig.frdm_k66f	Declares the SoC and board-level configuration options to link the board to its corresponding SoC.	Mandatory
Kconfig.defconfig	Defines default configurations for peripherals.	Optional
frdm_k66f_defconfig	Specifies board-specific settings for system clock, GPIO, and other high-level features.	Optional
frdm_k66f.yaml	Metadata describing board capabilities like peripherals and memory sizes for Zephyr automated testing.	Optional
frdm_k66f-pinctrl.dtsi	Defines pin multiplexing for peripherals.	Optional
board.cmake	Adds flashing and debugging support for tools like J-Link, LinkServer, and PyOCD.	Optional
CMakeLists.txt	Allows adding additional build sources if needed.	Optional

3.1.1 Board Metadata

The *board.yml* file serves as the board's high-level descriptor. It consolidates vital information such as the board name, vendor, and associated SoC, forming a bridge between the board and the tools that interact with it. This file enables Zephyr to identify and integrate the board seamlessly into its build and testing processes.

```
board:
  name: frdm_k66f
  vendor: nxp
  socs:
    - name: mk66f18
```

3.1.2 Hardware Description

The *frdm_k66f.dts* file is an essential part of the board configuration process, serving as the central source for defining the board hardware in the Devicetree format. This file offers a comprehensive hardware description, including the memory layout, clock configurations, pin assignments, and peripheral definitions. Zephyr can effectively interpret and communicate with the board's hardware components during runtime by abstracting these details into a single source, facilitating the development of portable applications by eliminating the need to hard-code hardware-specific information in the source code.

The following code snippet shows the portion configuring the RGB LED using PWM (Pulse Width Modulation) to adjust its brightness programmatically. Each LED is assigned to a FlexTimer Module (FTM), a channel number, and specific PWM settings such as frequency and polarity in this configuration. The aliases allow applications to interact with the LEDs via logical names. They abstract low-level details such as pin mappings and frequencies to enhance portability across boards.

```
/dts-v1/;

#include <nxp/nxp_k6x.dtsi>
#include <zephyr/dt-bindings/pwm/pwm.h>
#include "frdm_k66f-pinctrl.dtsi"
#include <zephyr/dt-bindings/input/input-event-codes.h>
/ {
	model = "NXP Freedom MK66F board";
	compatible = "nxp,mk66f18", "nxp,k66f", "nxp,k6x";

	aliases {
		green-pwm-led = &green_pwm_led;
		blue-pwm-led = &blue_pwm_led;
		red-pwm-led = &red_pwm_led;
	};

	pwmleds {
		compatible = "pwm-leds";
		green_pwm_led: green_pwm_led {
			pwms = <&ftm3 1 15625000 PWM_POLARITY_INVERTED>;
			label = "User PWM LD1";
		};
		red_pwm_led: red_pwm_led {
			pwms = <&ftm3 5 15625000 PWM_POLARITY_INVERTED>;
			label = "User PWM LD2";
		};
		blue_pwm_led: blue_pwm_led {
			pwms = <&ftm2 1 15625000 PWM_POLARITY_INVERTED>;
			label = "User PWM LD3";
			};
	};
};
```

3.1.3 Configuration Options

The *Kconfig.frdm_k66f* file links the board and its corresponding SoC configuration to ensure hardware and software build system compatibility.

```
config BOARD_FRDM_K66F
    select SOC_MK66F18
    select SOC_PART_NUMBER_MK66FN2M0VMD18
```

3.1.4 Default Hardware Configuration

The *Kconfig.defconfig* file sets default configurations for the board hardware. It defines essential parameters such as the external oscillator frequency, system clock settings, and default states for UART and I2C drivers to provide a functional application baseline.

```
if BOARD_FRDM_K66F
config OSC_XTAL0_FREQ
    default 12000000
config MCG_PRDIV0
    default 0x0
config MCG_VDIV0
    default 0x4 # Set to 180MHz clock
config UART_MCUX
    default y
    depends on SERIAL
config I2C_MCUX
    default y
    depends on I2C
endif # BOARD_FRDM_K66F
```

3.1.5 Core Board Settings

The *frdm_k66f_defconfig* file defines configurations for the board, such as setting the system clock and enabling core features such as GPIO, the Memory Protection Unit (MPU), I2C, and PWM drivers. It also activates RTOS services like the system console, ensuring the board is ready for development with reliable, pre-defined defaults that simplify application-specific work.

```
CONFIG_CONSOLE=y
CONFIG_UART_CONSOLE=y
CONFIG_SERIAL=y
CONFIG_GPIO=y
CONFIG_PINCTRL=y
CONFIG_SYS_CLOCK_HW_CYCLES_PER_SEC=180000000
CONFIG_ARM_MPU=y
CONFIG_HW_STACK_PROTECTION=y
CONFIG_BUILD_OUTPUT_HEX=y
CONFIG_PWM=y
CONFIG_I2C=y
```

3.1.6 Automated Testing Metadata

The *frdm_k66f.yaml* file provides metadata in a format valid for automated testing and other tools. It describes the board's capabilities, including memory specifications (256 KB RAM and 2048 KB Flash) and supported peripherals such as GPIO, I2C, SPI, and PWM, which is critical for Zephyr's automated tools, such as Twister [24].

```
identifier: frdm_k66f
name: NXP FRDM-K66F
type: mcu
arch: arm
ram: 256
flash: 2048
toolchain:
  - zephyr
  - gnuarmemb
supported:
  - arduino_gpio
  - arduino_i2c
  - arduino_serial
  - arduino_spi
  - gpio
  - i2c
vendor: nxp
```

3.1.7 Pin Multiplexing Configuration

The *frdm_k66f-pinctrl.dtsi* file specifies the physical pin configurations for the hardware components defined in the *frdm_k66f.dts* file. While the *frdm_k66f.dts* describes logical features, such as the RGB LED controlled via PWM and its associated peripherals, this one specifies the physical pins to implement these functionalities.

This separation simplifies adapting the configuration for different hardware revisions or custom board variants, as only the pin control file needs modification, leaving the rest of the system unaffected.

The code snippet below shows how the hardware pin mapping aligns with the definitions in *frdm_k66f.dts*. It specifies the connection between the board's physical pins and the logical channels used for controlling the RGB LEDs. Additionally, this file specifies the electrical properties of the pins, such as drive strength and slew rate.

```
#include <nxp/kinetis/MK66FN2M0VMD18-pinctrl.h>

&pinctrl {
	ftm3_default: ftm3_default {
		group0 {
			pinmux = <FTM3_CH1_PTE6>,
				<FTM3_CH4_PTC8>,
				<FTM3_CH5_PTC9>;
			drive-strength = "low";
			slew-rate = "fast";
		};
	};
};
```

3.1.8 Debugging and Flashing Support

The *board.cmake* file integrates debugging and flashing tools. It configures tools like J-Link, LinkServer, and PyOCD, specifying the microcontroller model (MK66FN2M0xxx18) to ensure accurate communication. This file also includes scripts from Zephyr's base directory, enabling seamless integration with Zephyr's build and debugging systems. The configurations support developers in efficiently flashing and debugging applications on the FRDM-K66F.

```
board_runner_args(jlink "--device=MK66FN2M0xxx18")
board_runner_args(linkserver  "--device=MK66FN2M0xxx18:FRDM-K66F")
board_runner_args(pyocd "--target=k66f")

include(${ZEPHYR_BASE}/boards/common/linkserver.board.cmake)
include(${ZEPHYR_BASE}/boards/common/pyocd.board.cmake)
```

3.1.9 Custom Build Sources

The *CMakeLists.txt* file includes additional build sources specific to the board, such as custom drivers or application-specific modules. It integrates directly with Zephyr's build system, ensuring these sources are correctly compiled and linked.

3.2. Instantiating Supported Peripherals

The FXOS8700 sensor, which integrates an accelerometer and a magnetometer, is natively supported by Zephyr RTOS. Its support is derived from the FRDM-K64F board and extends to the FRDM-K66F with minimal additional configurations required to enable its use in an application.

3.2.1 Devicetree Configuration

The FXOS8700 sensor is defined in the *frdm_k66f.dts* file, which specifies the necessary hardware details for Zephyr to initialize and interact with the sensor. This includes activating the *I2C0* peripheral required for communication and linking the device node to the driver.

The compatible property is set to *"nxp,fxos8700"*, which the build system uses to associate the sensor node with the driver defined in *zephyr/drivers/sensor/nxp/fxos8700/fxos8700.c*. The driver defines the macro *DT_DRV_COMPAT* as *nxp_fxos8700*, ensuring the appropriate code is linked during the build process. Following the manufacturer's datasheet, the sensor's I2C address is specified in the reg property as 0x1D [25].

Aliases are defined to simplify application-level access to the sensor, allowing it to be referenced as *magn0* for the magnetometer and *accel0* for the accelerometer.

```
aliases {
    magn0 = &fxos8700;
    accel0 = &fxos8700;
};

arduino_i2c: &i2c0 {
    status = "okay";
    pinctrl-0 = <&i2c0_default>;
    pinctrl-names = "default";

    fxos8700: fxos8700@1d {
        compatible = "nxp,fxos8700";
        reg = <0x1d>;
        int1-gpios = <&gpioc 17 GPIO_ACTIVE_LOW>;
        int2-gpios = <&gpioc 13 GPIO_ACTIVE_LOW>;
    };
};
```

3.2.2 Kconfig Configuration

Enabling the FXOS8700 sensor requires setting specific Kconfig options at both the board and project levels. At the board level, the *frdm_k66f_defconfig* file includes the following configuration to ensure that the I2C peripheral is enabled.

```
CONFIG_I2C=y
```

Additionally, the *Kconfig.defconfig* file defines the default behavior for the I2C driver.

```
config I2C_MCUX
     default y
     depends on I2C
```

At the project level, the *prj.conf* file includes configurations for the FXOS8700 to enable hybrid mode for simultaneous accelerometer and magnetometer use, add support for temperature readings, and configure the sensor to use its thread for handling events. Floating-point number support is also activated to format and print sensor data in the console.

```
CONFIG_FXOS8700_MODE_HYBRID=y
CONFIG_FXOS8700_TEMP=y
CONFIG_FXOS8700_TRIGGER_OWN_THREAD=y
CONFIG_CBPRINTF_FP_SUPPORT=y
```

3.2.3 Usage in the Application

Using a peripheral in the application involves binding the device to the static configuration, fetching data from it, and processing it for application-specific purposes.

a) Device Binding

The sensor is instantiated using the *DEVICE_DT_GET_ONE* macro, which retrieves the device handle based on the Devicetree configuration. Before using it, it’s necessary to ensure proper initialization with the *device_is_ready* function.

```
#include <zephyr/drivers/sensor.h>

const struct device *const dev = DEVICE_DT_GET_ONE(nxp_fxos8700);
if (!device_is_ready(dev)) {
    printk("Device %s is not ready\n", dev->name);
    return;
}
```

b) Data Fetch

Data is fetched into the *dev* structure from the sensor using the *sensor_sample_fetch* function. The accelerometer, magnetometer, and temperature data are then extracted using *sensor_channel_get*.

```
struct sensor_value accel[3];
struct sensor_value magn[3];
struct sensor_value temp;

while (1) {
    if (sensor_sample_fetch(dev)) {
        printk("Failed to fetch sensor data\n");
        return;
    }

    sensor_channel_get(dev, SENSOR_CHAN_ACCEL_XYZ, accel);
    sensor_channel_get(dev, SENSOR_CHAN_MAGN_XYZ, magn);
#ifdef CONFIG_FXOS8700_TEMP
    sensor_channel_get(dev, SENSOR_CHAN_DIE_TEMP, &temp);
#endif
}
```

c) Data Processing

The fetched data is converted to a format suitable for further processing with the *sensor_value_to_double* function to convert the raw sensor values into double-precision floating-point numbers.

```
double accel_x = sensor_value_to_double(&accel[0]);
double accel_y = sensor_value_to_double(&accel[1]);
double accel_z = sensor_value_to_double(&accel[2]);
double magn_x = sensor_value_to_double(&magn[0]);
double magn_y = sensor_value_to_double(&magn[1]);
double magn_z = sensor_value_to_double(&magn[2]);
#ifdef CONFIG_FXOS8700_TEMP
double temperature = sensor_value_to_double(&temp);
#endif
```

3.3. Extending Device Support

The LCD, powered by the PCD8544 controller, is integrated into the application to provide a graphical interface. Although Zephyr does not natively support this device, support was added through custom configurations and the driver implementation.

3.3.1 Devicetree Configuration

The LCD is defined in the project folder, in the *boards/frdm_k66f.overlay* file, which specifies the SPI and GPIO configurations required for communication and control. The compatible property is set to *"nokia,lcd-5110"*, linking the device to the custom binding defined in the *dts/bindings/nokia,lcd-5110.yaml* file. This binding ensures the system recognizes the SPI device and its GPIO properties, such as Data/Command, Reset, and Backlight pins.

```yaml
description: |
  Binding for Nokia 5110 LCD display using the PCD8544 controller.

compatible: "nokia,lcd-5110"

include:
  - spi-device.yaml

properties:
  dc-gpios:
    type: phandle-array
    required: true
    description: Data/Command control GPIO pin.
  reset-gpios:
    type: phandle-array
    required: true
    description: Reset GPIO pin.
  backlight-gpios:
    type: phandle-array
    required: false
    description: Backlight pin.
  label:
    type: string
    required: false
    description: Device label.
```

The corresponding pin mapping for the *SPI0* is provided in *frdm_k66f-pinctrl.dtsi*, under the *spi0_default* group. Additional GPIO configurations can be added in this file depending on the application needs.

```
spi0_default: spi0_default {
     group0 {
          pinmux = <SPI0_PCS0_PTD0>,
               <SPI0_SCK_PTD1>,
               <SPI0_SOUT_PTD2>,
               <SPI0_SIN_PTD3>;
          drive-strength = “low”;
          slew-rate = “fast”;
     };
};
```

Meanwhile, the GPIO pins to control the Data/Command (DC), Reset, and optional Backlight signals are defined in the *boards/frdm_k66f.overlay* file, which references the previously defined *spi0_default*.

```
&spi0 {
    status = “okay”;
    pinctrl-0 = <&spi0_default>;
    pinctrl-names = “default”;
    lcd5110: lcd5110@0 {
        compatible = “nokia,lcd-5110”;
        reg = <0>; // Use hardware CS0
        spi-max-frequency = <500000>;
        dc-gpios = <&gpiob 19 GPIO_ACTIVE_HIGH>; // Data/Command pin
        reset-gpios = <&gpiob 18 GPIO_ACTIVE_HIGH>; // Reset pin
        backlight-gpios = <&gpioa 6 GPIO_ACTIVE_HIGH>; // Backlight
        label = “LCD_5110”;
    };
};
```

3.3.2 Kconfig Configuration

Regarding the Kconfig, the required configurations to enable the drivers for SPI communication and GPIO control are in the *prj.conf* file.

```
CONFIG_SPI=y
CONFIG_GPIO=y
```

3.3.3 Wiring Description

The LCD is connected to the microcontroller through SPI and GPIO signals, which are wired as follows.

Peripheral	Pin	Port
SPI	Clock (CLK)	PTD1
SPI	Data Input (DIN)	PTD2
SPI	Chip Enable (CE)	PTD0
GPIO	Data/Command (DC)	PTB19
GPIO	Reset (RST)	PTB18
GPIO	Backlight (BL)	PTA6

3.3.4 Usage in the Application

The application interacts with the LCD using the SPI and GPIO drivers.

a) Device Binding

The application interacts with the LCD using the SPI and GPIO drivers provided by Zephyr. Device bindings for SPI and GPIO are retrieved using the *SPI_DT_SPEC_INST_GET* and *GPIO_DT_SPEC_INST_GET* macros, respectively. These macros link the application to the corresponding hardware definitions in the Devicetree.

```
#include <zephyr/drivers/spi.h>
#include <zephyr/drivers/gpio.h>
#define DT_DRV_COMPAT nokia_lcd_5110
#define SPI_OP (SPI_WORD_SET(8) | SPI_TRANSFER_MSB)

static const struct spi_dt_spec lcd_spi = SPI_DT_SPEC_INST_GET(0,
SPI_OP, 0);
static const struct gpio_dt_spec dc_gpio = GPIO_DT_SPEC_INST_GET(0,
dc_gpios);
static const struct gpio_dt_spec reset_gpio = GPIO_DT_SPEC_INST_GET(0,
reset_gpios);
static const struct gpio_dt_spec backlight_gpio =
GPIO_DT_SPEC_INST_GET(0, backlight_gpios);
```

Before using these devices, the application ensures they were properly initialized with the *device_is_ready* function.

```
if (!spi_is_ready_dt(&lcd_spi)) {
    printk("SPI device not ready\n");
    return;
}
if (!device_is_ready(dc_gpio.port)) {
    printk("DC GPIO device not ready\n");
    return;
}
if (!device_is_ready(reset_gpio.port)) {
    printk("Reset GPIO device not ready\n");
    return;
}
if (!device_is_ready(backlight_gpio.port)) {
    printk("Backlight GPIO device not ready\n");
    return;
}
```

b) Device Usage

Data transmission is handled by the *LCD_write_byte* function, which uses SPI to send data or commands to the display. The function appropriately sets the Data/Command (DC) pin before initiating the SPI transfer.

```
void LCD_nokia_write_byte(uint8_t data_or_command, uint8_t data)
{
    int ret;
    struct spi_buf tx_buf;
    struct spi_buf_set tx_bufs;

    gpio_pin_set_dt(&dc_gpio, data_or_command == LCD_DATA ? 1 : 0);

    tx_buf.buf = &data;
    tx_buf.len = 1;
    tx_bufs.buffers = &tx_buf;
    tx_bufs.count = 1;

    ret = spi_write_dt(&lcd_spi, &tx_bufs);
    if (ret < 0) {
        printk("Failed to write to SPI device: %d\n", ret);
    }
}
```

The initialization process involves configuring the GPIO pins and sending a series of commands, such as setting the instruction set, contrast, bias mode, and display mode, over SPI to set up the display.

```
// Configure GPIOs
ret = gpio_pin_configure_dt(&dc_gpio, GPIO_OUTPUT);
if (ret < 0) {
    printk("Failed to configure DC GPIO\n");
    return;
}

ret = gpio_pin_configure_dt(&reset_gpio, GPIO_OUTPUT);
if (ret < 0) {
    printk("Failed to configure Reset GPIO\n");
    return;
}

ret = gpio_pin_configure_dt(&backlight_gpio, GPIO_OUTPUT);
if (ret < 0) {
    printk("Failed to configure Backlight GPIO\n");
    return;
}

// Turn on the backlight
gpio_pin_set_dt(&backlight_gpio, 1);

// Reset the LCD
gpio_pin_set_dt(&reset_gpio, 0);
k_sleep(K_MSEC(10));
gpio_pin_set_dt(&reset_gpio, 1);

// Initialization commands
LCD_nokia_write_byte(LCD_CMD, 0x21); // Extended instruction set
LCD_nokia_write_byte(LCD_CMD, 0xB1); // Set LCD Vop (Contrast)
LCD_nokia_write_byte(LCD_CMD, 0x04); // Set Temperature Coefficient
LCD_nokia_write_byte(LCD_CMD, 0x14); // LCD Bias Mode 1:48

LCD_nokia_write_byte(LCD_CMD, 0x20); // Basic instruction set
LCD_nokia_write_byte(LCD_CMD, 0x0C); // Normal display mode

printk("LCD 5110 initialization done\n");
```

Once initialized, the display can perform various operations, providing a foundation for more complex graphical tasks within the application. The *LCD_goto_xy* function sets the cursor to a specific column and row by sending the corresponding commands over SPI.

```
void LCD_nokia_goto_xy(uint8_t x, uint8_t y)
{
    LCD_nokia_write_byte(LCD_CMD, 0x80 | x); // Column
    LCD_nokia_write_byte(LCD_CMD, 0x40 | y); // Row
}
```

The *LCD_send_char* function uses a predefined ASCII font table to display individual characters, each represented by a series of 5 vertical bytes sent to the display sequentially.

```
void LCD_nokia_send_char(uint8_t character)
{
    uint16_t index = 0;
    LCD_nokia_write_byte(LCD_DATA, 0x00); // Blank vertical line
    for (index = 0; index < 5; index++) {
        LCD_nokia_write_byte(LCD_DATA, ASCII[character-0x20][index]);
    }
    LCD_nokia_write_byte(LCD_DATA, 0x00); // Blank vertical line
}
```

The *LCD_bitmap* function takes an array representing the graphical data and writes it to the display memory to display a complete bitmap.

```
void LCD_nokia_bitmap(const uint8_t bitmap[])
{
    uint16_t index = 0;
    for (index = 0; index < (LCD_X * LCD_Y / 8); index++) {
        LCD_nokia_write_byte(LCD_DATA, bitmap[index]);
    }
}
```

The *LCD_clear* function clears the display by writing zeroes to all the pixels in the display memory, effectively turning them off.

```
void LCD_nokia_clear(void)
{
    uint16_t index = 0;
    for (index = 0; index < (LCD_X * LCD_Y / 8); index++) {
        LCD_nokia_write_byte(LCD_DATA, 0x00);
    }
    LCD_nokia_goto_xy(0, 0); // Return to home position after clearing
}
```

3.3.5 Multi-thread Support

The system manages its operations using six threads created and initialized during the system setup. Thread configuration is handled using Zephyr's threading APIs [26].

The following code snippet demonstrates how build-time threads are defined using the *K_THREAD_DEFINE* macro, which statically configures thread properties such as stack size, priority, entry function, and optional arguments.

```
#include <zephyr/kernel.h>

#define STACKSIZE                  (1024)

#define PRIORITY_SENSOR_SAMPLING   (4)
#define PRIORITY_GESTURE_DETECTION (5)
#define PRIORITY_LED_CONTROL       (6)
#define PRIORITY_TERMINAL_LOGGING  (7)
#define PRIORITY_LCD_DISPLAY       (8)
#define PRIORITY_BUTTON            (9)

K_THREAD_DEFINE(sensor_sampling_id, STACKSIZE, sensor_sampling_thread,
NULL, NULL, NULL, PRIORITY_SENSOR_SAMPLING, 0, 0);

K_THREAD_DEFINE(gesture_detection_id, STACKSIZE,
gesture_detection_thread, NULL, NULL, NULL,
PRIORITY_GESTURE_DETECTION, 0, 0);

K_THREAD_DEFINE(led_control_id, STACKSIZE, led_control_thread, NULL,
NULL, NULL, PRIORITY_LED_CONTROL, 0, 0);

K_THREAD_DEFINE(lcd_display_id, STACKSIZE, lcd_display_thread, NULL,
NULL, NULL, PRIORITY_LCD_DISPLAY, 0, 0);

K_THREAD_DEFINE(terminal_logging_id, STACKSIZE,
terminal_logging_thread, NULL, NULL, NULL, PRIORITY_TERMINAL_LOGGING,
0, 0);

K_THREAD_DEFINE(button_id, STACKSIZE, button_thread, NULL, NULL, NULL,
PRIORITY_BUTTON, 0, 0);
```

Runtime threads can be dynamically created using the *k_thread_create* API, which allows for greater flexibility, such as creating threads in response to specific application events or conditions [27].

```
#include <zephyr/kernel.h>

struct k_thread dynamic_thread;
K_THREAD_STACK_DEFINE(dynamic_stack, STACKSIZE);

void start_dynamic_thread() {
    k_tid_t thread_id = k_thread_create(&dynamic_thread,
dynamic_stack, STACKSIZE, dynamic_thread_function, NULL, NULL, NULL,
PRIORITY_SENSOR_SAMPLING, 0, K_NO_WAIT);
}
```

3.3.6 Inter-processor Communication

The application relies on several synchronization mechanisms to coordinate operations between threads and manage shared resources efficiently. These mechanisms include semaphores, mutexes, message queues, events, and atomic variables.

```
#include <zephyr/kernel.h>
#include <zephyr/sys/atomic.h>

struct k_sem data_ready_sem;
struct k_mutex data_mutex;
struct k_msgq sensor_msgq;
struct k_event gesture_event;
atomic_t mode;
/* Buffer for message queue */
static float sensor_msgq_buffer[10 * sizeof(float[6])];

void shared_data_init(void)
{
    k_sem_init(&data_ready_sem, 0, 1);
    k_mutex_init(&data_mutex);
    k_msgq_init(&sensor_msgq, (char *)sensor_msgq_buffer,
sizeof(float[6]), 10);
    k_event_init(&gesture_event);
    atomic_set(&mode, e_VisualizationMode_SplashScreen);
}
```

a) Semaphore

A semaphore signals the availability or readiness of resources or tasks. It operates as a signaling mechanism, allowing one thread to notify others that an event has occurred. Semaphores are initialized using *k_sem_init*, where the initial count specifies the number of available signals, and the maximum count limits the number of signals that can be queued.

```
struct k_sem data_ready_sem;

void shared_data_init(void) {
    k_sem_init(&data_ready_sem, 0, 1); // Initial count is 0, max is 1
}
```

After updating shared data, the sensor sampling thread gives (*k_sem_give*) the semaphore, signaling other threads that new data is available. Threads waiting for this event call *k_sem_take* to proceed.

```
void sensor_sampling_thread(void) {
    while (1) {
        // Simulate data acquisition
        update_shared_data();
        k_sem_give(&data_ready_sem); // Signal data availability
        k_msleep(50);
    }
}

void gesture_detection_thread(void) {
    while (1) {
        k_sem_take(&data_ready_sem, K_FOREVER); // Wait for data
        process_shared_data();
    }
}
```

b) Mutex

A mutex ensures exclusive access to shared resources, such as memory buffers, preventing data corruption due to concurrent access by multiple threads. Mutexes are initialized using *k_mutex_init*.

```
struct k_mutex data_mutex;

void shared_data_init(void) {
    k_mutex_init(&data_mutex);
}
```

Threads lock the mutex using *k_mutex_lock* before accessing data and unlock it using *k_mutex_unlock* afterward to guarantee that only one accesses the resource at a time.

```
void lcd_display_thread(void) {
    while (1) {
        k_mutex_lock(&data_mutex, K_FOREVER); // Lock the mutex
        read_and_update_display();
        k_mutex_unlock(&data_mutex); // Unlock the mutex
        k_msleep(100);
    }
}
```

c) Message queue

Message queues allow threads to exchange structured data asynchronously, are initialized using *k_msgq_init*, and a buffer is provided to store messages.

```
#define MSGQ_MAX_MSGS 10
#define MSGQ_MSG_SIZE sizeof(float[6])
struct k_msgq sensor_msgq;
float sensor_msgq_buffer[MSGQ_MAX_MSGS * MSGQ_MSG_SIZE];

void shared_data_init(void) {
    k_msgq_init(&sensor_msgq, (char *)sensor_msgq_buffer,
MSGQ_MSG_SIZE, MSGQ_MAX_MSGS);
}
```

The producer thread enqueues messages using *k_msgq_put*, while the consumer thread dequeues them using *k_msgq_get*.

```
void sensor_sampling_thread(void) {
    float sensor_data[6];
    while (1) {
        acquire_sensor_data(sensor_data);
        k_msgq_put(&sensor_msgq, sensor_data, K_NO_WAIT);
        k_msleep(50);
    }
}
void terminal_logging_thread(void) {
    float sensor_data[6];
    while (1) {
        k_msgq_get(&sensor_msgq, sensor_data, K_FOREVER);
        log_sensor_data(sensor_data);
    }
}
```

d) Event

Events are used to broadcast status updates or conditions to multiple threads. They provide a flexible mechanism for signaling complex states. Events are initialized using *k_event_init*.

```
struct k_event gesture_event;

void shared_data_init(void) {
    k_event_init(&gesture_event);
}
```

The threads use *k_event_post* to set event bits.

```
void gesture_detection_thread(void) {
    while (1) {
        detect_gesture();
        k_event_post(&gesture_event, GESTURE_DETECTED_FLAG);
        k_msleep(200);
    }
}
```

The threads can wait for specific event conditions with the function *k_event_wait*, which suspends the task until the event is posted.

```
void led_control_thread(void) {
    while (1) {
        k_event_wait(&gesture_event, GESTURE_DETECTED_FLAG, false,
K_FOREVER);
        respond_to_gesture();
    }
}
```

e) Atomic variable

Atomic variables allow thread-safe and low-overhead updates to single variables without requiring mutexes. They are ideal for simple state management. Atomic variables are initialized using *atomic_set*.

```
atomic_t mode;

void shared_data_init(void) {
    atomic_set(&mode, e_VisualizationMode_SplashScreen);
}
```

Atomic variables are manipulated using atomic operations such as *atomic_get*, *atomic_set*, and *atomic_cas* (compare and set).

The following example shows how the atomic variable *mode* is atomically incremented in the ISR when a button is pressed.

```
void button_pressed(const struct device *dev, struct gpio_callback
*cb, uint32_t pins) {
    int current_mode = atomic_get(&mode);
    current_mode = (current_mode + 1) % e_VisualizationMode_MAX;
    atomic_set(&mode, current_mode);
    printk("Mode changed to %d\n", current_mode);
}
```

Meanwhile, the LCD thread uses the atomic variable to decide the visualization mode to show.

```
void lcd_display_thread(void) {
    while (1) {
        int mode = atomic_get(&mode);
        render_mode(mode);
        k_msleep(100);
    }
}
```

3.3.7 Hardware Interrupts

Hardware interrupts in Zephyr provide an efficient and responsive mechanism for handling asynchronous events triggered by hardware [28]. An ISR is a function that executes when an interrupt is signaled. ISRs in Zephyr are designed to run with minimal latency, ensuring that hardware events are handled promptly.

An ISR is associated with an interrupt request (IRQ) signal, a priority level, and an interrupt vector table entry. When an interrupt occurs, the hardware triggers an interrupt signal corresponding to a specific IRQ number, then the CPU uses the interrupt vector table to find the ISR associated with the IRQ, and finally, the ISR executes and handles the interrupt. ISRs are executed within the kernel’s interrupt context, with a dedicated stack separate from thread stacks.

a) Configuring GPIO Interrupts

GPIO interrupts are essential for handling external events such as button presses. Zephyr simplifies the process by abstracting the underlying complexity, allowing straightforward configuration and management of GPIO-based interrupts.

The GPIO pin is first defined and associated with its hardware descriptor using the *GPIO_DT_SPEC_GET_OR* macro. This binds the GPIO pin to its corresponding Devicetree configuration. Before proceeding, the associated GPIO device is checked for readiness using the *device_is_ready* function. Once verified, the pin is configured as an input using *gpio_pin_configure_dt*, just as it would be for non-interrupt-related use.

```
#include <zephyr/drivers/gpio.h>

#define SW0_NODE DT_ALIAS(sw0)

static const struct gpio_dt_spec button =
GPIO_DT_SPEC_GET_OR(SW0_NODE, gpios, {0});

void button_thread(void)
{
    int ret = 0;
    if (!device_is_ready(button.port)) {
        printk("Error: Button devices is not ready\n");
        return;
    }

    ret = gpio_pin_configure_dt(&button, GPIO_INPUT);
    if (ret != 0) {
        printk("Error %d: Failed to configure button\n", ret);
        return;
    }

    ret = gpio_pin_interrupt_configure_dt(&button,
GPIO_INT_EDGE_TO_ACTIVE);
    if (ret != 0) {
        printk("Error %d: Failed to configure button interrupt\n",
ret);

        return;
    }
    ...
}
```

A callback function is implemented to handle interrupts triggered by the GPIO pin. The callback must follow the prototype defined by *gpio_callback*. This function acts as the ISR and is invoked whenever the interrupt condition (e.g., a rising edge on the pin) is met.

The ISR is registered using the *gpio_init_callback* and *gpio_add_callback* functions. These ensure the specified callback is linked to the interrupt event on the target GPIO pin.

```
static struct gpio_callback button_cb_data;
static void button_pressed(const struct device *dev, struct
gpio_callback *cb, uint32_t pins)
{
    int current_mode = atomic_get(&mode);
    current_mode = (current_mode + 1) % e_VisualizationMode_MAX;
    atomic_set(&mode, current_mode);
}

void button_thread(void)
{
    gpio_init_callback(&button_cb_data, button_pressed,
BIT(button.pin));
    gpio_add_callback(button.port, &button_cb_data);
    while (1) { k_msleep(50); }
}
```

b) Nested Interrupt Handling

Zephyr supports nested interrupts, allowing higher-priority ISRs to preempt lower-priority ones. This feature ensures critical interrupts are serviced with minimal delay. To enable nested interrupts, use the *CONFIG_MULTI_LEVEL_INTERRUPTS* option.

c) Zero-Latency Interrupts

For ultra-low-latency requirements, Zephyr offers zero-latency interrupts using the *CONFIG_ZERO_LATENCY_IRQS* option. These interrupts bypass the kernel's interrupt locking mechanism and execute directly.

```
#define BUTTON_IRQ    DT_IRQN(DT_ALIAS(sw0))
ISR_DIRECT_DECLARE(button_isr) {
    ISR_DIRECT_PM();  // Inform the kernel about power management
    return 1;         // Indicate a scheduling decision may be needed
}

void configure_zero_latency_interrupt() {
    IRQ_DIRECT_CONNECT(BUTTON_IRQ, BUTTON_PRIO, button_isr,
IRQ_ZERO_LATENCY);
    irq_enable(BUTTON_IRQ);
}
```

d) Preventing Interrupts

Threads can temporarily prevent interrupts by locking IRQs using *irq_lock*. This is useful for critical sections where interrupts could cause inconsistent states.

```
unsigned int key = irq_lock();
// Perform critical section
irq_unlock(key);
```

Alternatively, specific IRQs can be disabled.

```
irq_disable(BUTTON_IRQ);
// Perform critical operation
irq_enable(BUTTON_IRQ);
```

e) Offloading ISR Work to Threads

ISRs should offload tasks to threads using kernel objects like semaphores, FIFOs, or work queues for long-running operations. This reduces ISR execution time and maintains system responsiveness. The following code snippet is an example of ISR work offloading to a thread using a semaphore.

```
struct k_sem isr_semaphore;

void button_isr(void *arg) {
    k_sem_give(&isr_semaphore);  // Signal the thread
}

void button_thread() {
    while (1) {
        k_sem_take(&isr_semaphore, K_FOREVER);
        printk("Processing ISR in thread\n");
    }
}
```

4. Implementation Results

The configurations outlined in the Methods chapter establish a functional baseline comprising the necessary peripherals and the multithreaded application framework. This setup ensures safe data sharing between peripherals and threads while enabling user interaction via the button, accelerometer, and LCD. As detailed in this chapter, these foundational configurations support the development of an application with multiple visualization modes. Refer to Appendix A for the complete source code of the threads discussed in this chapter.

4.1. Multithreaded Application

The application is built around six threads designed for a specific purpose and coordinated through shared data and synchronization mechanisms.

4.1.1 Main Function

The *main* function is defined in the *main.c* file, defines the six threads used for the application, and initializes the shared data used across threads, defined in the *shared_data.h* and *shared_data.c* files.

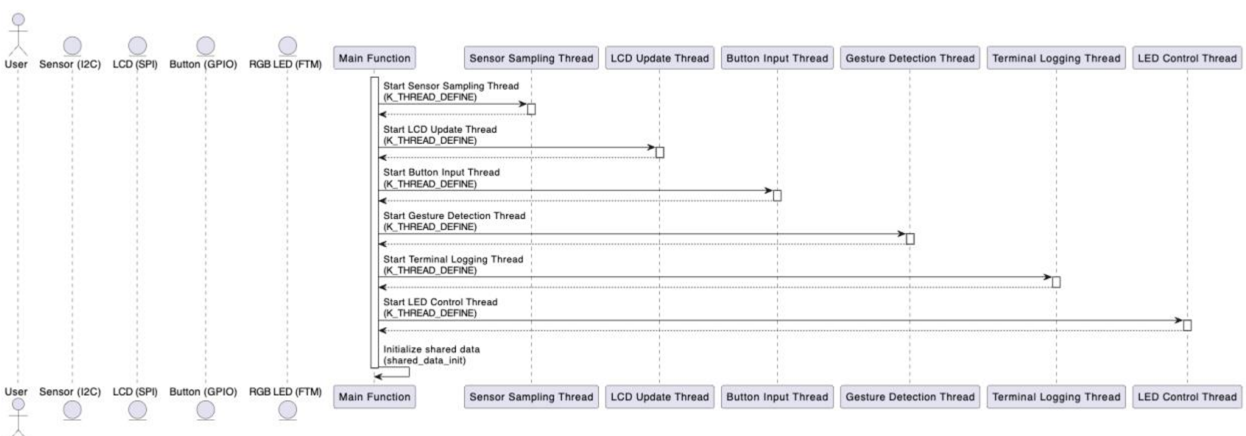

Fig. 4-1 Main Function Sequence Diagram.

4.1.2 Sensor Sampling Thread

The sensor sampling (*sensor_sampling*) thread is implemented in the *sensor_sampling.h* and *sensor_sampling.c* files. It runs every 50 milliseconds to gather real-time data from the accelerometer and magnetometer using I2C. It converts it to a floating-point format to update the shared memory buffers protected by the *data_mutex* mutex. This thread signals other threads via the *data_ready_sem* semaphore and enqueues sensor readings into the *sensor_msgq* message queue.

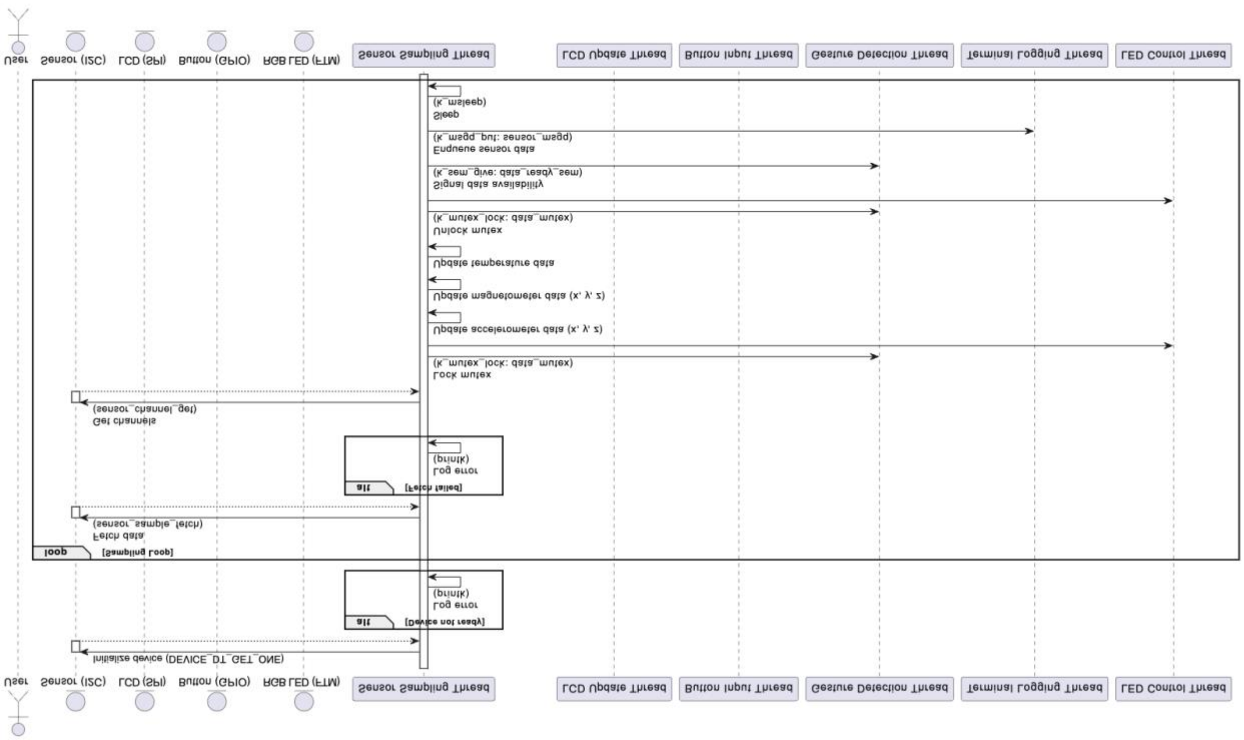

Fig. 4-2 Sensor Sampling Thread Sequence Diagram.

4.1.3 LCD Update Thread

The LCD update (*lcd_display*) thread is implemented across the *lcd_display.h* and *lcd_display.c* files and uses the driver from the *LCD5110/LCD_nokia.h*, *LCD5110/LCD_nokia.c*, *LCD5110/LCD_nokia_images.h*, and *LCD5110/LCD_nokia_images.c* files.

The screen content is updated every 100 ms based on the current visualization mode, managed by the *mode* atomic variable. The modes include a startup splash screen, numerical displays of accelerometer and magnetometer values, plots of data trends, a compass, a level bubble,

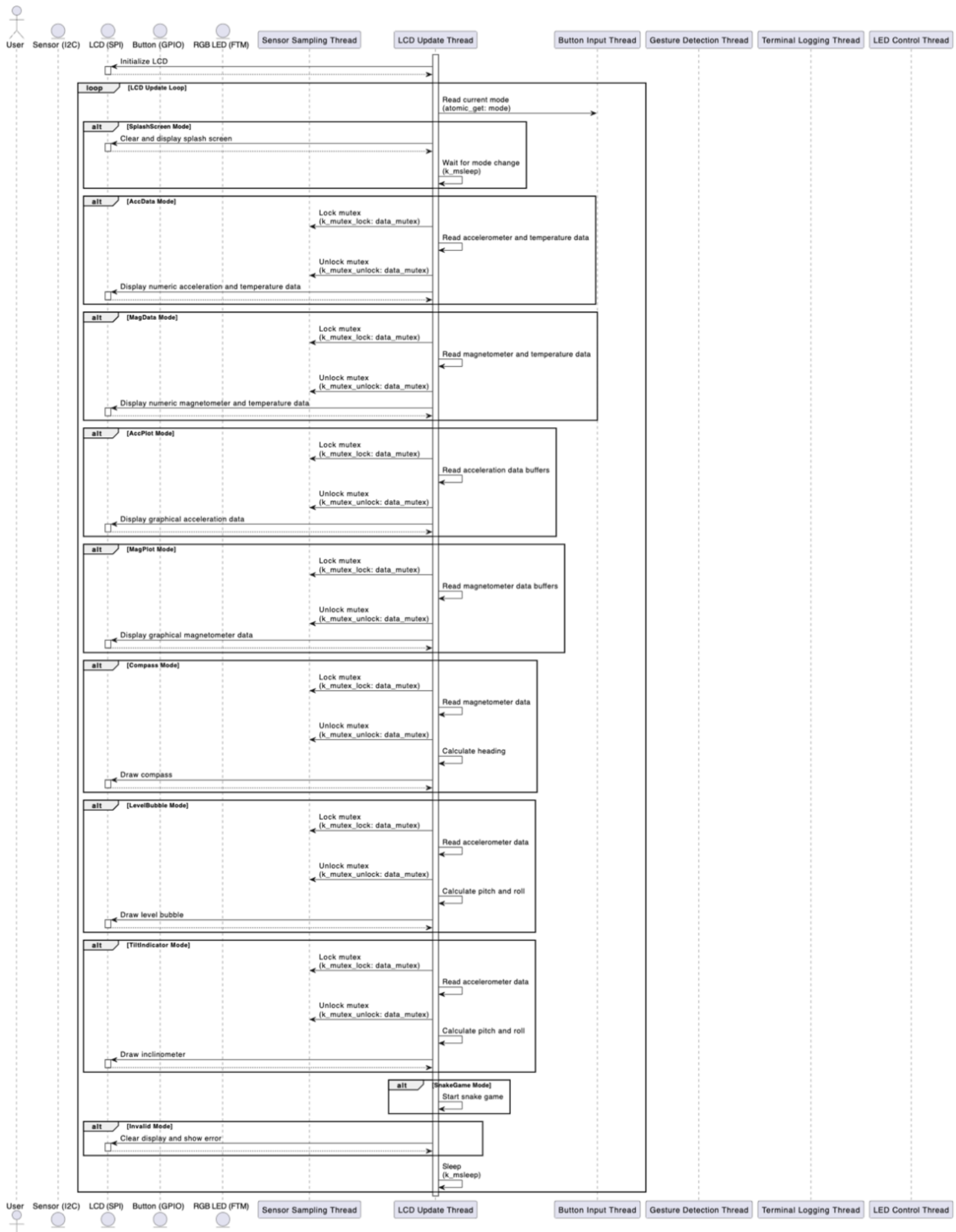

Fig. 4-3 LCD Update Thread Sequence Diagram.

an inclinometer, and a tilt-controlled snake game. The thread calls utility functions distributed across the files *utils.h* and *utils.c*, where the pitch, roll, and heading values are computed [29] using the data collected by the *sensor_sampling* thread, which is protected by the *data_mutex* mutex, as well as the *snake_game.h* and *snake_game.c* where the logic for the game is implemented.

a) Roll Computation

The roll is calculated as the angular rotation about the X-axis, typically using the *arctangent* function applied to the ratio of the Y and Z acceleration components. This value represents how much the device is tilted side-to-side, which helps visualize the level bubble and tilt indicator when combined with the pitch.

b) Pitch Computation

The pitch is derived from the angular rotation about the Y-axis and calculated using the *arctangent* of the ratio of the X and Z acceleration components. It indicates the degree of forward or backward tilt. Combined with the roll value, it helps visualize the level bubble and the tilt indicator.

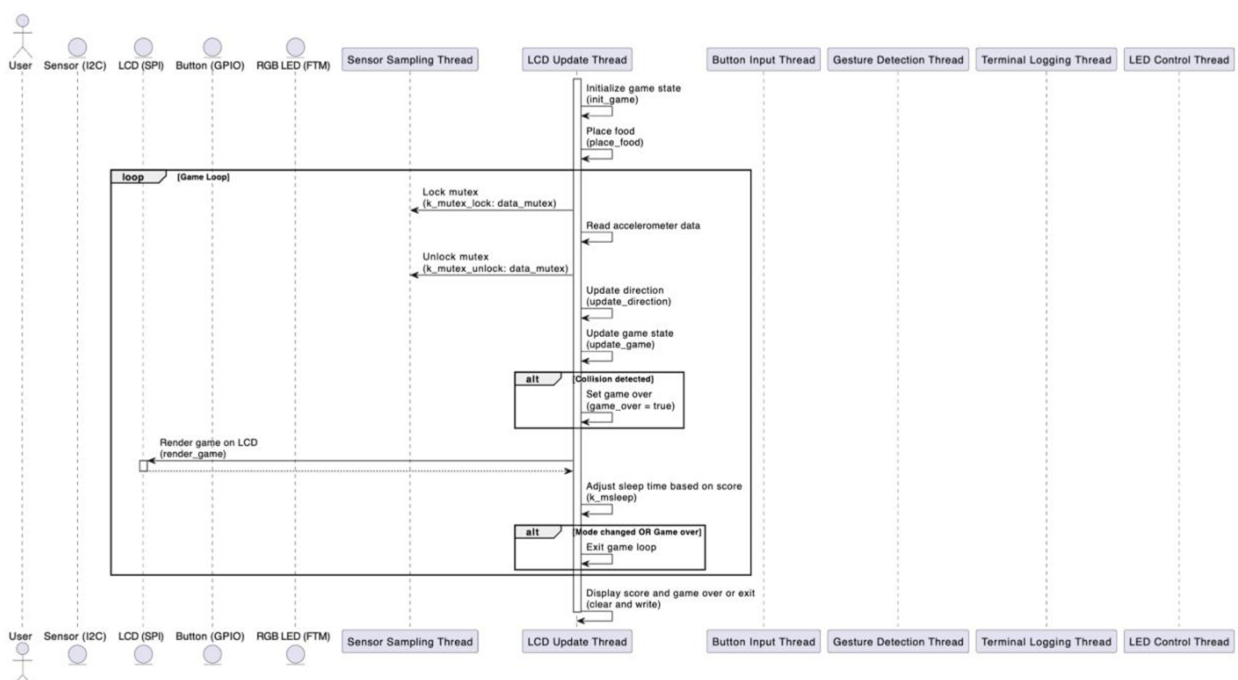

Fig. 4-4 Snake Game Sequence Diagram.

c) Heading Computation

The heading, also known as the magnetic heading or azimuth, is calculated using the magnetometer data in combination with the accelerometer to compensate for tilt. This involves computing the angle between the magnetic north and the horizontal component of the magnetic field using the *arctangent* function. The heading is adjusted based on the pitch and roll to ensure accuracy when the board is not perfectly level, which helps visualize the compass.

4.1.4 Button Input Thread

The button input (*button*) thread is implemented in the files *button.h* and *button.c*. It processes user inputs from a GPIO button via interrupts and updates the *mode* atomic variable to

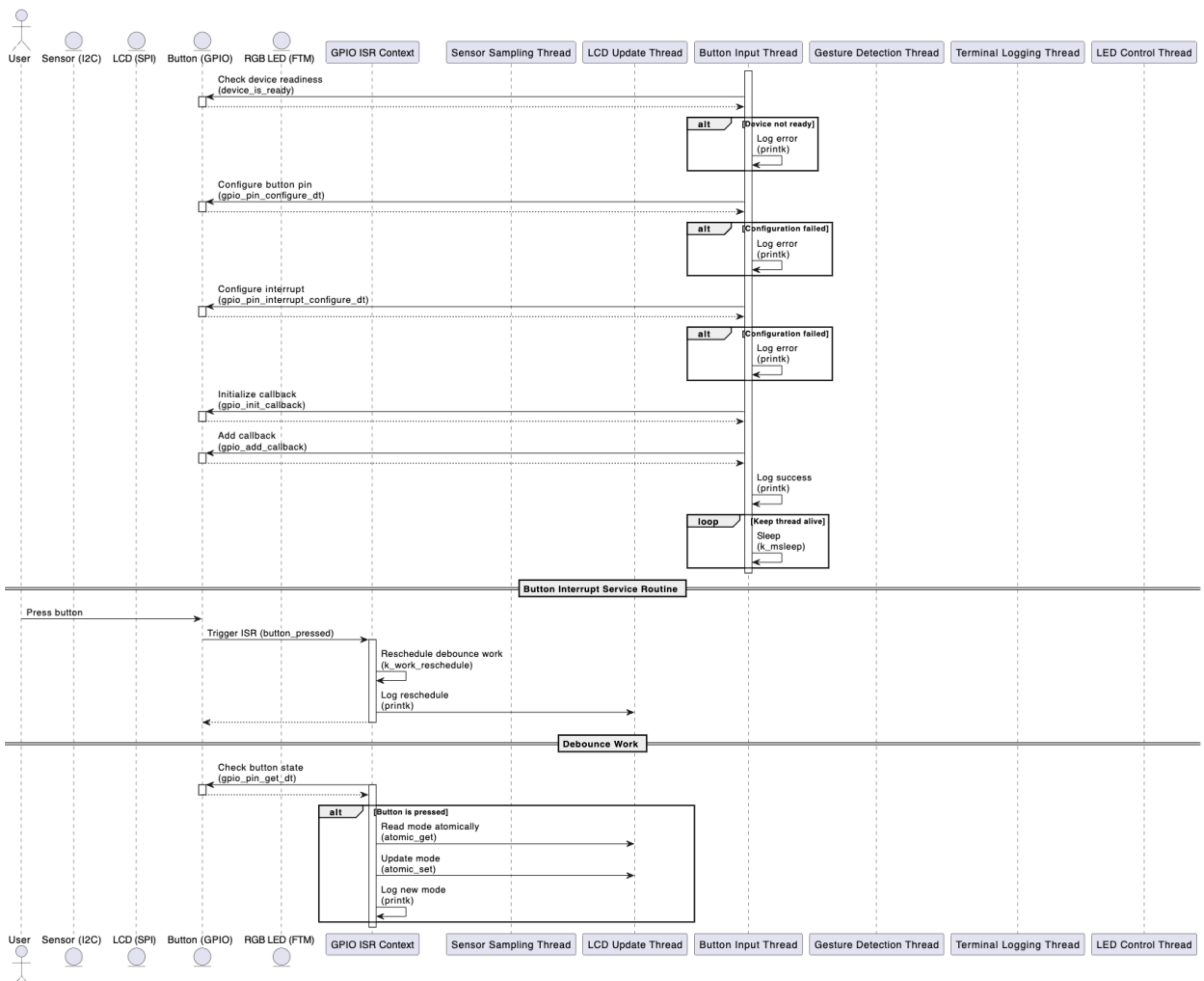

Fig. 4-5 Button Input Thread Sequence Diagram.

enable transitions between visualization modes. This variable is updated in the context of the GPIO ISR to handle the current visualization mode in this thread.

When an edge change is detected, a software-based debounce mechanism is applied. This involves starting a cooldown period timer, ensuring rapid, consecutive button presses are ignored [30]. Any additional edge changes restart the timer during the cooldown period, filtering out noise and unintended transitions. The button press is considered valid once the timer expires without further edge changes, and the visualization mode transitions accordingly.

4.1.5 Gesture Detection Thread

The gesture detection (*gesture_detection*) thread is implemented in the *gesture_detection.h* and *gesture_detection.c* files. It runs every 200 milliseconds but waits for accelerometer data to be available through the *data_ready_sem* semaphore. Once available, it is protected by the *data_mutex* mutex and analyzed to detect predefined gestures such as tilt, flip, and shake and post an event to the *gesture_event* event object.

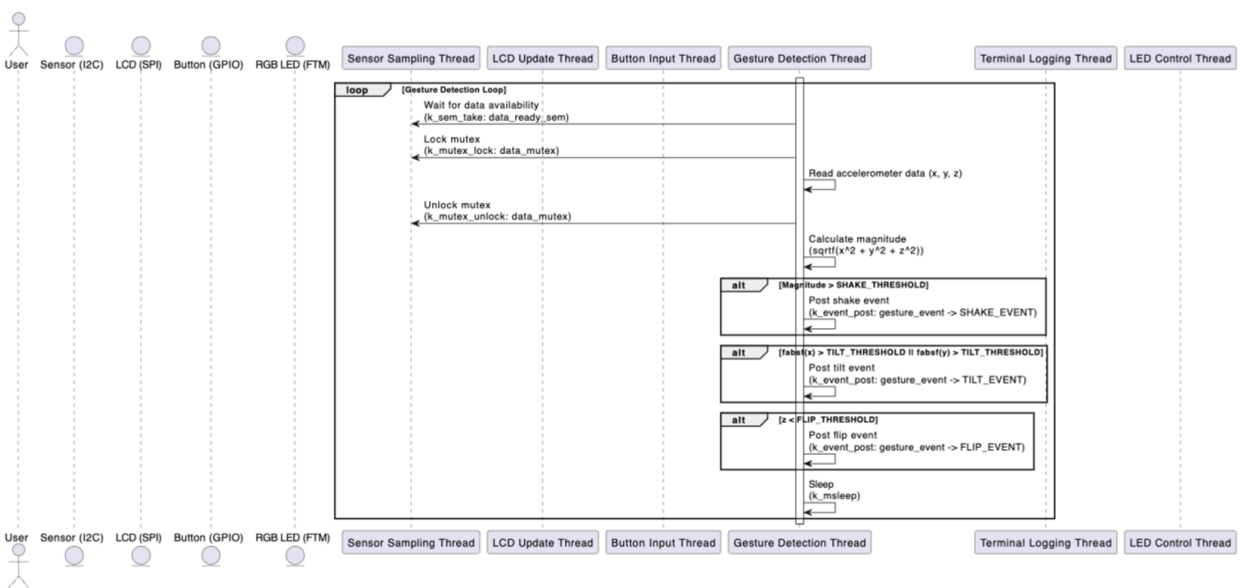

Fig. 4-6 Gesture Detection Thread Sequence Diagram.

4.1.6 Terminal Logging Thread

The terminal logging (*terminal_logging*) thread logs sensor data to the terminal every 200 milliseconds. It is implemented in the *terminal_logging.h* and *terminal_logging.c* files. It retrieves data packets from the *sensor_msgq* message queue and outputs them to the terminal.

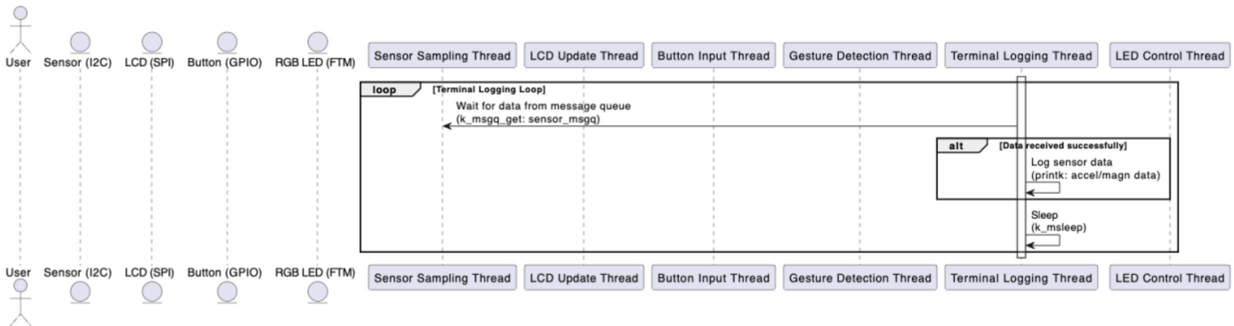

Fig. 4-7 Terminal Logging Thread Sequence Diagram.

A terminal can be opened to view the logged information with a UART baud rate of 115,200 baud.

Fig. 4-8 Terminal Logging Thread Output.

4.1.7 LED Control Thread

The LED control (*led_control*) thread is implemented in the *led_control.h* and *led_control.c* files. It runs every 200 milliseconds to provide visual feedback based on the most recently detected gesture using the RGB LED. The thread adjusts the LED's color and brightness based on events in the *gesture_event* object.

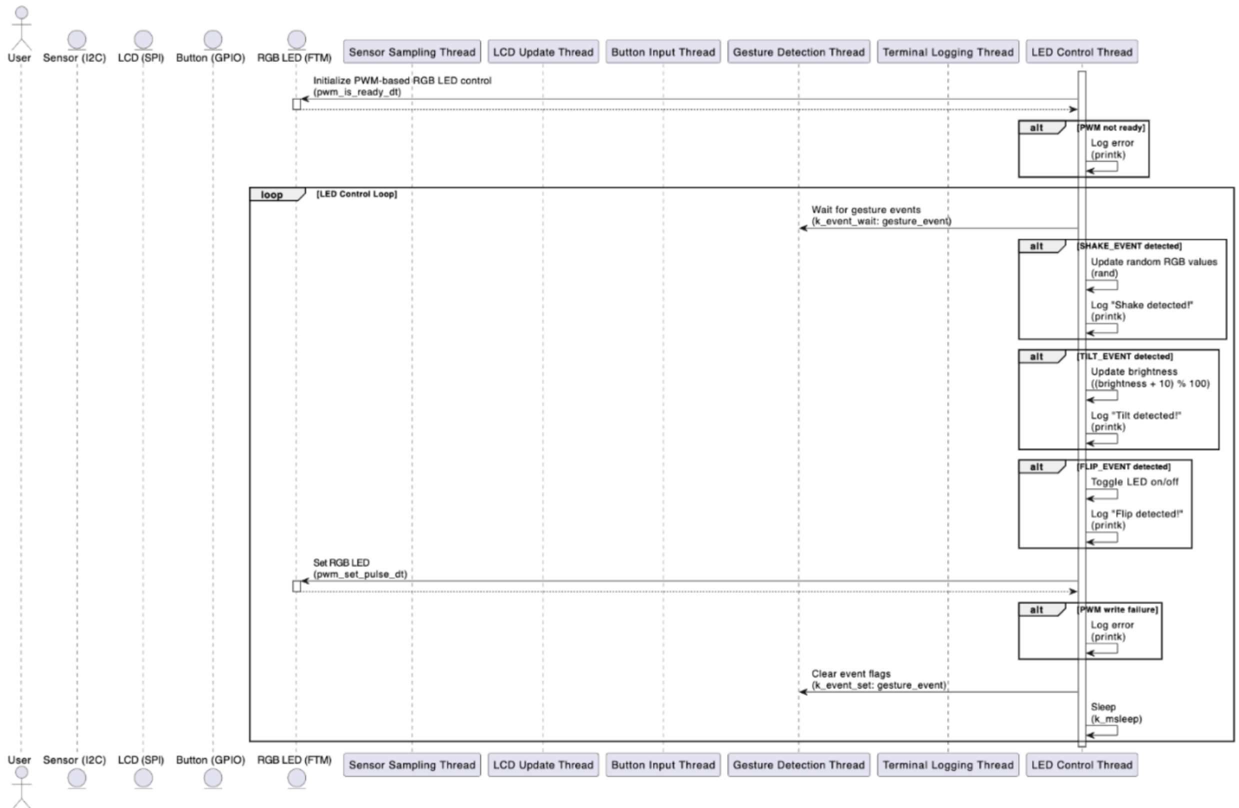

Fig. 4-9 LED Control Thread Sequence Diagram.

4.2. Visualization Modes

The application includes multiple visualization modes based on the user input to show data based on the accelerometer and magnetometer in LCD display. These modes are implemented in the LCD update thread and are managed by the *mode* atomic variable, which is updated based on user input from the button thread.

4.2.1 Splash Screen Mode

The splash screen mode (*e_VisualizationMode_SplashScreen*) introduces the application with a static startup image displayed on the LCD. This mode uses the *LCD_nokia_bitmap* function to load a predefined bitmap image, serving as a placeholder while initialization completes. The screen remains static until the user switches modes using a button press.

Fig. 4-10 Splash Screen Mode.

4.2.2 Numeric Display Mode

The numeric display modes (*e_VisualizationMode_AccData* and *e_VisualizationMode_MagData*) present real-time numerical values for the X, Y, and Z axes of acceleration, magnetic field data, and temperature. Mutex-protected access ensures the safe reading of shared sensor data. The *LCD_nokia_send_string* function formats and renders the text on the LCD.

Fig. 4-11 a) Acceleration Numeric Display Mode, b) Magnetometer Numeric Display Mode.

4.2.3 Graphical Display Mode

The graphical display modes (*e_VisualizationMode_AccPlot* and *e_VisualizationMode_MagPlot*) plot acceleration and magnetic field data over time. Historical data is stored in circular buffers, allowing the application to render line graphs using the *LCD_nokia_draw_line* function. This visualization helps identify trends and analyze sensor data behavior over time.

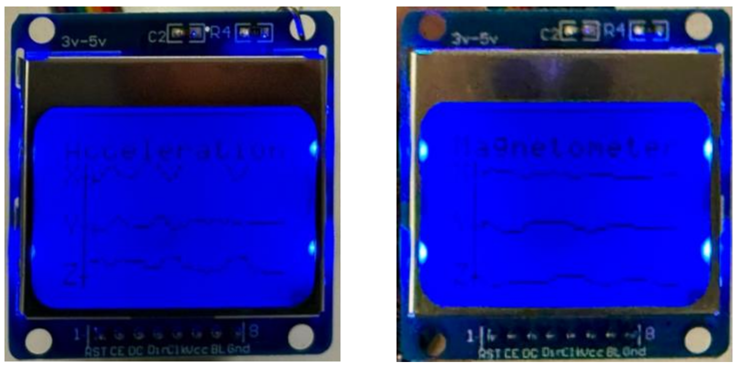

Fig. 4-12 a) Acceleration Graphical Display Mode, b) Magnetometer Graphical Display Mode.

4.2.4 Compass Mode

The compass mode (*e_VisualizationMode_Compass*) displays a compass, indicating the magnetic heading based on the magnetometer data. This data is represented graphically with a needle drawn using *LCD_nokia_draw_line*. This mode is ideal for navigation-related applications and offers a practical use of the magnetometer.

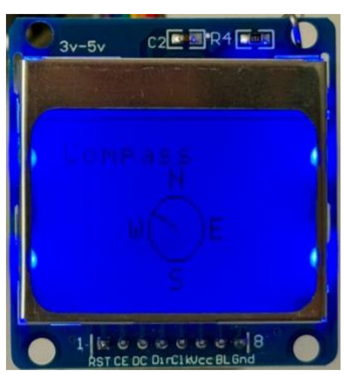

Fig. 4-13 Compass Mode.

4.2.5 Level Bubble Mode

The level bubble mode (*e_VisualizationMode_LevelBubble*) simulates a spirit level by showing the tilt along pitch and roll axes. It calculates these angles from accelerometer data and represents the tilt as a moving bubble within a rectangle for orientation calibration or leveling tasks.

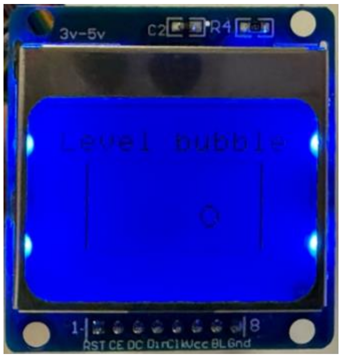

Fig. 4-14 Level Bubble Mode.

4.2.6 Tilt Indicator Mode

The tilt indicator mode (*e_VisualizationMode_TiltIndicator*) displays a visual indicator of the tilt in real-time as a dot that moves within a defined boundary to reflect the current pitch and roll angles.

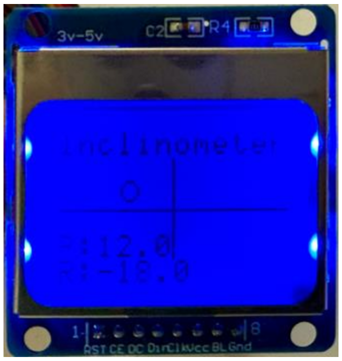

Fig. 4-15 Tilt Indicator Mode.

4.2.7 Snake Game Mode

The tilt-controlled snake game (*e_VisualizationMode_SnakeGame*) adds an element of entertainment. The classic snake game is controlled by tilting the device. The accelerometer data determines the snake's direction and the game updates dynamically with collision detection and food consumption logic. Upon exiting any game or mode, the application clears the LCD and transitions to the next cyclic mode, the splash screen.

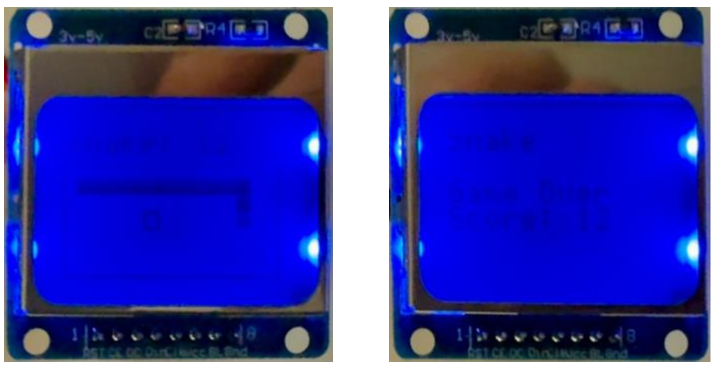

Fig. 4-16 a) Snake Game Mode, b) Snake Game Over Mode.

4.3. Debugging Integration with Visual Studio Code

Visual Studio Code (VS Code) is a widely popular Integrated Development Environment (IDE) known for its versatility. However, additional configuration is required to support debugging Zephyr RTOS applications.

Jonathan Beri offered a talk to demonstrate how developers can configure VS Code for embedded development by customizing configuration files such as *launch.json*, *settings.json*, and *tasks.json* or by creating a dedicated workspace file (e.g., *zephyr-macos.code-workspace*) within the project folder to centralize these settings [31].

These configurations unlock Zephyr RTOS debugging capabilities, such as compiling, flashing, and debugging within the VS Code environment. Detailed examples of these configurations are available in Appendix C.

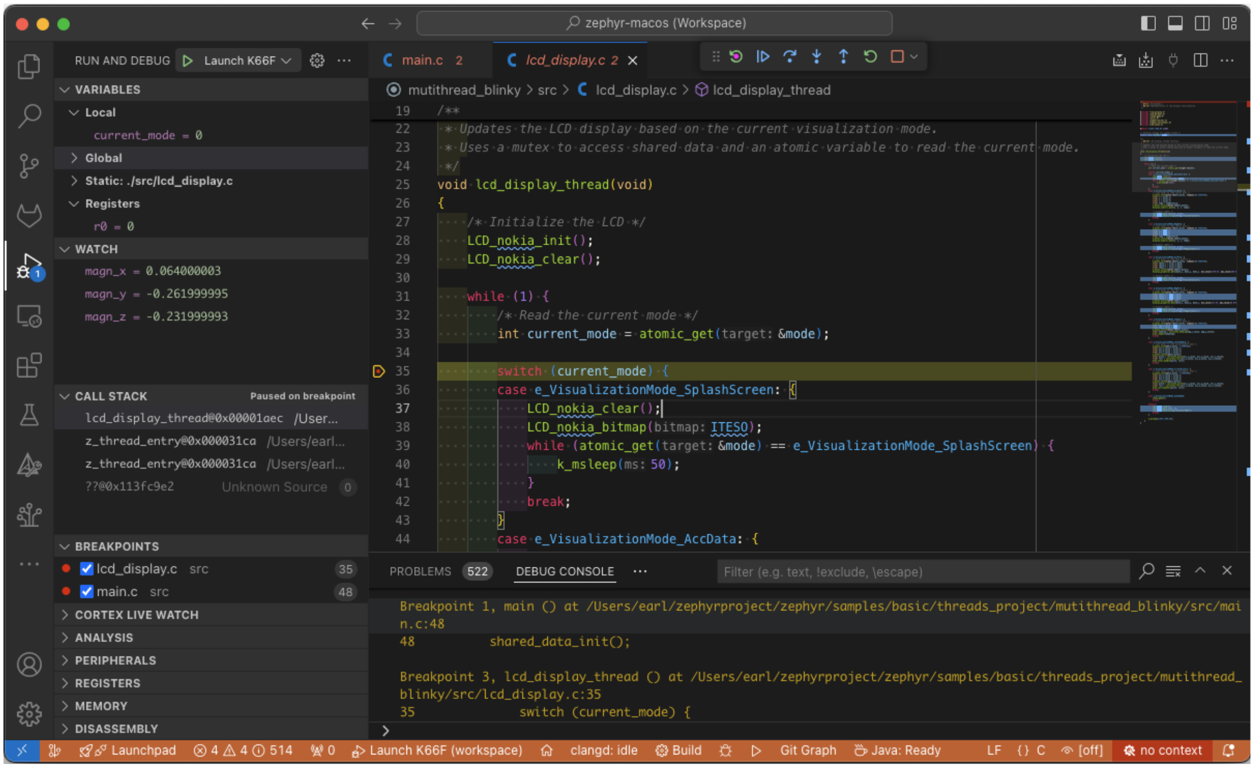

Fig. 4-17 Zephyr RTOS Debug Session in Visual Studio Code.

5. Analysis and Discussion

This research presents a practical case study to simplify the adoption of Zephyr RTOS for embedded systems developers. The study focuses on extending hardware support, integrating new drivers, and leveraging synchronization mechanisms for multithreaded applications. By adding support for the NXP FRDM-K66F board, implementing a custom driver for the PCD8544 LCD controller, and utilizing peripherals such as FTM, GPIO, SPI, and I2C, the research demonstrates the feasibility of adopting Zephyr RTOS. This chapter discusses the main learnings, challenges encountered, and areas for improvement.

5.1. Key Observations

This section summarizes notable findings and practical lessons learned during the development of the case study.

5.1.1 Extending Board Support

The modular architecture allows for easy adaptation to unsupported hardware, as demonstrated by the addition of support for the NXP FRDM-K66F board. Devicetree and Kconfig configurations have streamlined the hardware abstraction and integration from the NXP FRDM-K64F board, which the Zephyr Project officially supports. However, the reliance on configuration files from multiple locations within the Zephyr SDK and project directories led to confusion regarding which files needed to be configured, which required careful documentation review and debugging.

5.1.2 Developing Custom Drivers

The integration of the PCD8544 LCD controller demonstrated the flexibility in supporting new peripherals. Although the lack of native support for this LCD controller added complexity, it also highlighted the extensibility of the RTOS. Developing the custom driver did not require a thorough understanding of SPI and GPIO configurations.

5.1.3 Multitasking and Synchronization

The study successfully demonstrated multithreading capabilities and synchronization mechanisms. Shared resources, such as sensor data, were effectively managed using mutexes, while semaphores and message queues facilitated inter-thread communication.

5.1.4 Development Environment

Using VS Code as the primary development environment enhanced productivity after it was configured to debug the Zephyr RTOS-based project. Its debugging features, such as real-time variable inspection and breakpoint configuration, significantly reduced development time and improved error resolution, which aligns to make Zephyr accessible to new developers.

5.2. Challenges and Limitations

Despite its potential, adopting Zephyr RTOS presented several challenges described in this section.

5.2.1 Learning Curve

The Devicetree and Kconfig are powerful tools within the Zephyr environment but introduce a slow learning curve. The scattered nature of configuration files across multiple directories added complexity, making it challenging to determine the appropriate locations for specific settings. Additionally, the documentation needed more clarity in effectively explaining the available features and how to select them, further complicating the learning process and requiring reverse engineering to understand what must be configured to enable a desired kernel feature.

5.2.2 Hardware-Specific Customization

Adapting unsupported hardware requires extensive trial and error, particularly for low-level configurations such as pin multiplexing and interrupt setup. These tasks demanded a detailed understanding of both the RTOS and the hardware.

5.2.3 Debugging Challenges

While VS Code improved the debugging experience, accessing hardware-specific registers and troubleshooting peripheral issues were not seamlessly integrated, which added to the complexity of debugging low-level hardware interactions.

5.3. Future Directions

Building on the insights gained from this study, the following areas for improvement and further exploration are proposed to enhance developer experience with Zephyr RTOS:

- Creating detailed, step-by-step tutorials and guides focused on learning the core concepts of Zephyr RTOS would greatly benefit developers of all skill levels.
- Although graphical and automated tools for Devicetree and Kconfig already exist, their usability still needs improvement. Enhancing these tools with more intuitive interfaces and straightforward guidance would reduce reliance on documentation, making it easier for developers to understand and configure the system.
- Official support for widely used IDEs like VS Code would streamline development workflows. Improved debugging features, such as seamless integration for accessing peripheral registers and enhanced visualization of real-time data, would significantly ease troubleshooting and increase productivity.

Conclusions

This research demonstrated how Zephyr RTOS can be adapted to work with unsupported hardware. It focused on its modular design, multitasking abilities, and flexibility in adding custom peripherals. The structured steps in this work made it possible to achieve the project's goals successfully, demonstrating that Zephyr is a strong option for building real-time embedded systems. However, the adoption process was not without challenges, particularly for new users and those transitioning existing products to this technology.

Transitioning an existing product to Zephyr introduces additional complexity. Legacy architectures optimized for other RTOS or proprietary solutions often require significant refactoring, including redesigning hardware-specific drivers or adapting proprietary communication protocols to align with Zephyr's architecture and APIs. Moreover, adapting unsupported hardware involves trial and error, particularly for low-level configurations like pin multiplexing and interrupt setup, which demands an understanding of both the hardware and Zephyr. Industrial applications add further challenges, requiring rigorous revalidation and compliance with safety-critical standards and increasing development time and costs.

The learning curve associated with Zephyr's Devicetree and Kconfig mechanisms presented another significant challenge. Scattered configuration files made it difficult to determine where specific settings needed to be adjusted. Additionally, the documentation often lacked clarity, leaving developers to reverse-engineer solutions. Debugging low-level hardware interactions worsened these issues, as current tools provide limited access to hardware-specific registers and peripheral problems.

Despite these challenges, the modular design facilitates adding support for new hardware and developing custom drivers. Once the initial setup is complete, the system becomes more manageable. Its multitasking features, like semaphores, mutexes, and message queues, help make the system reliable and responsive in real-time applications. Once correctly configured, tools like VS Code enhance productivity by streamlining debugging and enabling build and flash commands to be executed directly from the IDE.

Improving the documentation and tools would significantly reduce the learning curve for developers of all levels. With its open-source nature and strong community support, Zephyr has the potential to become an even more flexible and scalable platform for embedded systems, encouraging broader adoption across diverse industries and solidifying its position as a leading solution for modern embedded applications.

Appendices

A. BOARD SUPPORT CONFIGURATION

This appendix describes the configurations to adapt Zephyr RTOS for the NXP FRDM-K66F board. It includes details about the Devicetree, Kconfig, and build system files that define the board's hardware features, system clock, and peripherals such as GPIO, I2C, and PWM.

The board support folder is accessible in the repository:

https://github.com/EarlOrlando/zephyr/tree/multithread_blinky/boards/nxp/frdm_k66f

B. APPLICATION SOURCE CODE

This appendix provides the source code for the multithreaded application developed for this research. It demonstrates Zephyr RTOS's multitasking capabilities and the integration of peripherals, including the FXOS8700 accelerometer/magnetometer and the PCD8544 LCD controller. The implementation includes threads for sensor sampling, LCD updates, gesture detection, LED control, terminal logging, and button input handling. Additional features include multiple visualization modes and a tilt-controlled Snake game.

The application source code is accessible in the repository:

https://github.com/EarlOrlando/zephyr/tree/multithread_blinky/samples/basic/mutithread_blinky

C. DEBUG CONFIGURATION

This appendix describes the setup for enabling debugging capabilities in VS Code. The project incorporates the *zephyr-macos.code-workspace* file to support debugging and development workflows. This configuration simplifies tasks such as building, flashing, and debugging the application.

The debug configuration file is accessible in the repository:

https://github.com/EarlOrlando/zephyr/blob/multithread_blinky/samples/basic/mutithread_blinky/zephyr-macos.code-workspace

Bibliography

[1] National Instruments, "What is a Real-Time Operating System (RTOS)?," 22 August 2024. [Online]. Available: https://www.ni.com/en/shop/data-acquisition-and-control/add-ons-for-data-acquisition-and-control/what-is-labview-real-time-module/what-is-a-real-time-operating-system--rtos--.html?srsltid=AfmBOoqM9PNTAyckr5PYsUNzwr3X_3L9MbUIZKD1CnDYx3g8_fKi_kk2. [Accessed 2 December 2024].

[2] Zephyr Project, "About the Zephyr Project," [Online]. Available: https://zephyrproject.org/learn-about/. [Accessed 2 December 2024].

[3] S. Bhartiya, "The Power of Zephyr RTOS," 14 December 2018. [Online]. Available: https://www.linuxfoundation.org/blog/blog/the-power-of-zephyr-rtos. [Accessed 2 December 2024].

[4] K. Gugala and A. Petterson, "Enabling the Silicon Labs EFR family in Zephyr – a case study from a port's lifecycle," Embedded Open Source Summit, 29 June 2023. [Online]. Available: https://www.youtube.com/watch?v=hgZmoFh47M4. [Accessed 2 December 2024].

[5] G. Marull-Paretas, "Tutorial: Mastering Zephyr Driver Development," Zephyr Project Developer Summit 2022, 8 June 2022. [Online]. Available: https://www.youtube.com/watch?v=o-f2qCd2AXo. [Accessed 2 December 2024].

[6] J. Wolff, "How to Build Drivers for Zephyr RTOS," 11 August 2020. [Online]. Available: https://www.zephyrproject.org/how-to-build-drivers-for-zephyr-rtos/. [Accessed 2 December 2024].

[7] NXP Semiconductors, "Kinetis K66-180 MHz, Dual High-Speed and Full-speed USBs, 2MB Flash Microcontrollers (MCUs) based on Arm Cortex-M4 Core," [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/k-series-arm-cortex-m4/k6x-ethernet/kinetis-k66-180-mhz-dual-high-speed-and-full-speed-usbs-2mb-flash-microcontrollers-mcus-based-on-arm-cortex-m4-core:. [Accessed 2 December 2022].

[8] NXP Semiconductors, "Freedom Development Platform for Kinetis K66, K65, and K26 MCUs," [Online]. Available: https://www.nxp.com/design/design-center/development-boards-and-designs/general-purpose-mcus/freedom-development-platform-for-kinetis-k66-k65-and-k26-mcus:FRDM-K66F. [Accessed 2 December 2024].

[9] Philips, "PCD8544," 12 April 1999. [Online]. Available: https://www.alldatasheet.com/datasheet-pdf/view/18170/PHILIPS/PCD8544.html. [Accessed 2 December 2024].

[10] R. Lozano, "LCD Nokia 5110 48x84 luz de fondo azul," 30 September 2020. [Online]. Available: https://www.taloselectronics.com/blogs/tutoriales/lcd-nokia-5110-48x84-luz-de-fondo-azul. [Accessed 2 December 2024].

[11] Zephyr Project Documentation, "Supported Boards and Shields," [Online]. Available: https://docs.zephyrproject.org/latest/boards/index.html. [Accessed 2 December 2024].

[12] Zephyr Project Documentation, "Why We Moved from FreeRTOS to Zephyr RTOS," 28 March 2023. [Online]. Available: https://zephyrproject.org/why-we-moved-from-freertos-to-zephyr-rtos/. [Accessed 2 December 2024].

[13] Zephyr Project Documentation, "Zephyr Branding," [Online]. Available: https://zephyrproject.org/branding/. [Accessed 2 December 2024].

[14] Zephyr Project Documentation, "Getting Started Guide," [Online]. Available: https://docs.zephyrproject.org/latest/develop/getting_started/index.html. [Accessed 2 December 2024].

[15] Zephyr Project Documentation, "West (Zephyr's Meta-Tool)," [Online]. Available: https://docs.zephyrproject.org/latest/develop/west/index.html. [Accessed 2 December 2024].

[16] Zephyr Project Documentation, "Built-in commands," [Online]. Available: https://docs.zephyrproject.org/latest/develop/west/built-in.html. [Accessed 2 December 2022].

[17] Zephyr Project Documentation, "Building, Flashing and Debugging," [Online]. Available: https://docs.zephyrproject.org/latest/develop/west/build-flash-debug.html. [Accessed 2 December 2024].

[18] Zephyr Project Documentation, "Build System (CMake)," [Online]. Available: https://docs.zephyrproject.org/latest/build/cmake/index.html. [Accessed 2 December 2024].

[19] Zephyr Project Documentation, "Devicetree," [Online]. Available: https://docs.zephyrproject.org/latest/build/dts/index.html. [Accessed 2 December 2024].

[20] Zephyr Project Documentation, "Syntax and structure," [Online]. Available: https://docs.zephyrproject.org/latest/build/dts/intro-syntax-structure.html. [Accessed 2 December 2022].

[21] Zephyr Project Documentation, "Devicetree HOWTOs," [Online]. Available: https://docs.zephyrproject.org/latest/build/dts/howtos.html. [Accessed 2 December 2024].

[22] Zephyr Project Documentation, "Devicetree bindings," [Online]. Available: https://docs.zephyrproject.org/latest/build/dts/bindings.html. [Accessed 2 December 2024].

[23] Zephyr Project Documentation, "Configuration System (Kconfig)," [Online]. Available: https://docs.zephyrproject.org/latest/build/kconfig/index.html. [Accessed 2 December 2024].

[24] Zephyr Project Documentation, "Interactive Kconfig interfaces," [Online]. Available: https://docs.zephyrproject.org/latest/build/kconfig/menuconfig.html. [Accessed 2 December 2024].

[25] Zephyr Project Documentation, "Board Porting Guide," [Online]. Available: https://docs.zephyrproject.org/latest/hardware/porting/board_porting.html. [Accessed 2 December 2024].

[26] Zephyr Project Documentation, "Test Runner (Twister)," [Online]. Available: https://docs.zephyrproject.org/latest/develop/test/twister.html. [Accessed 2 December 2024].

[27] NXP Semiconductors, "FXOS8700CQ," 25 April 2017. [Online]. Available: https://www.nxp.com/docs/en/data-sheet/FXOS8700CQ.pdf. [Accessed 2 December 2024].

[28] Zephyr Project Documentation, "Kernel," [Online]. Available: https://docs.zephyrproject.org/latest/kernel/index.html. [Accessed 2 December 2024].

[29] Zephyr Project Documentation, "Threads," [Online]. Available: https://docs.zephyrproject.org/latest/kernel/services/threads/index.html. [Accessed 2 December 2024].

[30] Zephyr Project Documentation, "Interrupts," [Online]. Available: https://docs.zephyrproject.org/latest/kernel/services/interrupts.html. [Accessed 2 December 2024].

[31] M. Pedley, "Tilt Sensing Using a Three-Axis Accelerometer," June 2013. [Online]. Available: https://www.nxp.com/docs/en/application-note/AN3461.pdf. [Accessed 2 December 2024].

[32] L. Ubieda, "How to Debounce Button Inputs in Real Time Operating Systems," 11 April 2023. [Online]. Available: https://zephyrproject.org/how-to-debounce-button-inputs-in-a-rtos/. [Accessed 2 December 2024].

[33] J. Beri , "Zephyr & Visual Studio Code: How to Develop Zephyr Apps with a Modern, Visual IDE," 12 July 2023. [Online]. Available: https://www.youtube.com/watch?v=IKNHPmG-Qxo. [Accessed 2 December 2024].