

Politecnico di Torino

Politecnico di Torino

Master's Degree in Electronic Engineering

A.a. 2025/2026

Graduation Session 09/2026

Development and Validation of a SoC-Based Control Unit for Neuroprosthetic Systems

Supervisor:
Prof. Paolo Motto Ros

Candidate:
Can Aksu

Abstract

Neuroprosthetic systems aim to establish bidirectional communication between the human nervous system and external assistive devices such as exoprostheses and exoskeletons. The NerveRepack project addresses this goal by developing a system in which neural signals are used to control the external mechatronic device, while sensory information can be transmitted back to the nervous system. An electronic platform is required to integrate the Wireless Power Transfer (WPT), Power Management System (PMS), and Data Communication subsystems that support this bidirectional architecture. In this context, a Zynq-7000 System-on-Chip (SoC)-based control unit was developed to provide a common embedded platform for controlling, monitoring, and testing the external system from Linux through the communication and control interfaces.

In the final implementation, the required hardware interfaces were distributed between the Processing System (PS) and the Programmable Logic (PL) according to the available peripherals. In the PS, two Inter-Integrated Circuit (I2C) controllers operating at 400 kHz and 100 kHz were configured along with a Serial Peripheral Interface (SPI) operating at 8 MHz, a Controller Area Network (CAN) interface, and a 32-bit General-Purpose Input/Output (GPIO) interface. In the PL, a third I2C interface operating at 1 MHz was implemented using an Advanced eXtensible Interface (AXI) IIC core connected to the processor through an AXI Interconnect.

A PetaLinux-based software environment was prepared to provide access to the implemented hardware. The work involved customizing the PetaLinux Board Support Package (BSP), including kernel and device tree configuration, followed by boot image generation and the development of user-space test applications written in C.

The interfaces were initially implemented on an AX7Z020B-based platform and were then verified and validated through Linux-based checks of interface availability and configuration, functional testing with external devices, loopback tests, and tests involving the WPT, PMS, and Data Communication subsystems. Following initial validation, the control system was migrated to the more compact Alinx AC7020C board, which uses the same Zynq-7000 SoC. Pin assignments were updated, and the bitstream and boot files were regenerated before the relevant validation activities were repeated on the new platform. The results demonstrate that the developed platform provides the necessary hardware and software infrastructure for the operation and testing of the main external subsystems of NerveRepack.

Table of Contents

List of Tables v

List of Figures vi

1 Introduction 1
- 1.1 Context and Motivation . . . 1
- 1.2 Thesis Objectives and Contributions . . . 2
- 1.3 Thesis Organization . . . 2

2 System Background and Requirements 4
- 2.1 NerveRepack System Architecture . . . 4
- 2.2 Control Unit Requirements . . . 5
- 2.3 Zynq-7000 SoC Architecture . . . 6
  - 2.3.1 Processing System . . . 7
  - 2.3.2 Programmable Logic . . . 8
  - 2.3.3 PS–PL Interconnection . . . 8
- 2.4 Communication and Control Interfaces . . . 9
  - 2.4.1 I2C and SMBus . . . 9
  - 2.4.2 SPI . . . 9
  - 2.4.3 CAN . . . 9
  - 2.4.4 GPIO . . . 10
- 2.5 Embedded Linux and PetaLinux . . . 10

3 Hardware Design and Implementation 11
- 3.1 Hardware Development Platforms . . . 11
  - 3.1.1 Initial AX7Z020B-Based Platform . . . 11
  - 3.1.2 AC7020C Platform and Migration . . . 13
- 3.2 Vivado Hardware Design . . . 14
  - 3.2.1 Zynq PS Configuration . . . 14
  - 3.2.2 Block Design and PL Integration . . . 15
- 3.3 Pin Assignments and I/O Constraints . . . 17

3.3.1 Initial AX7Z020B Pin Assignment . . . 17
3.3.2 AC7020C Pin Assignment . . . 18
3.4 Synthesis and Implementation . . . 19
3.5 Bitstream Generation and Hardware Export . . . 20

4 Embedded Linux Implementation and System Bring-Up 21
4.1 PetaLinux Development Environment and Project Setup . . . 21
4.2 Board Support Package Customization . . . 22
4.2.1 Linux Kernel and Root Filesystem Configuration . . . 22
4.2.2 Device Tree Configuration . . . 23
4.3 System Build and Deployment . . . 23
4.3.1 PetaLinux Build and Boot Image Generation . . . 24
4.3.2 SD Card Deployment and System Boot . . . 24
4.4 Verification of Interface Availability and Configuration . . . 24

5 Validation and System Integration 28
5.1 Validation Methodology . . . 28
5.2 Functional Interface Validation . . . 28
5.2.1 I2C Interface . . . 29
5.2.2 SPI Interface . . . 31
5.2.3 CAN Interface . . . 34
5.2.4 GPIO Interface . . . 35
5.3 Subsystem Validation . . . 37
5.3.1 Wireless Power Transfer Subsystem . . . 38
5.3.2 Power Management Subsystem . . . 40
5.3.3 Data Communication Subsystem . . . 44

6 Conclusion 49
6.1 Future Work . . . 50

Bibliography 51

List of Tables

3.1 External interface pin assignments on the initial AX7Z020B platform. 18
3.2 External interface pin assignments on the AC7020C platform. . . . 19
3.3 Post-implementation resource utilization for the two hardware platforms. 19

List of Figures

2.1 General architecture of the NerveRepack system [4]. 5
2.2 Architectural overview of the Zynq-7000 SoC [5]. 7

3.1 AC7Z020 core board used in the initial development platform [15]. . 12
3.2 AX7Z020B development board with the AC7Z020 core board [16] . 12
3.3 AC7020C board used for the final implementation of the control unit [17]. 13
3.4 Vivado block design of the control unit. 17

4.1 Linux-side verification of the I2C interfaces and their configured clock frequencies. 25
4.2 Linux-side verification of the SPI interface and its configuration. . . 26
4.3 Linux-side verification of the CAN interface. 26
4.4 Linux-side verification of the PS GPIO controller. 26

5.1 MP8859 device detection and register read/write operations on the `i2c-2` interface. 30
5.2 Zynq master terminal output during the full-duplex SPI test. 31
5.3 nRF52840 slave terminal output during the full-duplex SPI test. . . 31
5.4 Zynq master terminal output during the sequential packet transfer test. 32
5.5 nRF52840 slave terminal output during the sequential packet transfer test. 33
5.6 CAN interface configuration and internal loopback test results. . . . 35
5.7 Jumper connections and fixed-pattern loopback test results with GPIO[7:0] configured as outputs and GPIO[15:8] as inputs. 36
5.8 Fixed-pattern and walking bit loopback test results with GPIO[15:8] configured as outputs and GPIO[7:0] as inputs. 37
5.9 Overview of the control unit and subsystem hardware used in the WPT, PMS, and Data Communication validation tests. 38

5.10 Experimental setup for WPT subsystem validation, including the control platform, WPT hardware, and RX voltage measurement. . . 39
5.11 Real-time monitoring results during WPT subsystem validation. . . 40
5.12 Experimental setup for PMS validation. 41
5.13 Real-time battery, load, and supercapacitor monitoring during PMS validation. 42
5.14 Experimental setup for smart battery validation. 43
5.15 Real-time battery charging and external power status monitoring. . 43
5.16 Charging current setting and monitoring during PMM240 validation. 44
5.17 Experimental setup for bidirectional communication validation of the Data Communication subsystem. 45
5.18 Oscilloscope capture of bidirectional data transmission during validation of the Data Communication subsystem. 46
5.19 Wireshark capture showing the end of the first payload at 0x7F. . . 47
5.20 Wireshark capture showing the continuation of the byte sequence from 0x80 in the following payload. 48

Chapter 1

Introduction

1.1 Context and Motivation

Problems such as amputation or paralysis severely affect a person's independence and ability to perform daily activities. To mitigate this impact and improve the patient's quality of life, exoprostheses have been developed for amputees and exoskeletons for paralyzed patients. Exoprostheses and exoskeletons can restore some of the lost mobility, but controlling these devices is challenging. Many current solutions rely on residual muscle activity [1], providing the user with limited sensory information [2]. As a result, interaction with the assistive device may be less intuitive than controlling a biological limb.

To address this limitation, a bidirectional connection can be established between the nervous system and an external mechatronic device. Neural signals related to the patient's movement intention are received and used to control an exoskeleton or exoprosthesis. Conversely, data collected by sensors on the external device is converted into stimulation signals and then sent back to the nervous system. This is a closed-loop approach that combines motor control with sensory feedback. This method allows for a more direct interaction between the user and the assistive device [2, 3].

The NerveRepack project is based on this concept. Its aim is to develop an interface capable of bidirectional communication between the nervous system and an exoprosthesis or exoskeleton. The complete platform includes implantable electrodes and implantable electronics, as well as external modules for wireless power transfer, power management, data communication, and control [4].

For the reliable operation of the entire system, proper coordination of the electronic subsystems is essential. These modules use different control signals and communication interfaces.

This distributed structure, composed of different subsystems, requires a common

control unit. This platform must configure external modules, collect measurement and status information, manage control signals, and support data exchange with the implantable part. It must also support changes made during prototyping and system integration. Using independent control units for each module would make the setup more complex and complicate coordinated testing.

A SoC combining programmable logic and a processor meets the requirements for the proposed control unit. While the processor already includes various peripherals and communication interfaces, any missing interfaces can be implemented in the programmable logic according to the project requirements [5]. The processor can also run a Linux-based operating system, providing a software environment suitable for the subsequent development of the control unit [6].

In this context, the NerveRepack project required a common embedded controller to provide the communication and control interfaces needed for the operation and testing of the external system [4].

1.2 Thesis Objectives and Contributions

The main objective of this thesis was to develop a Zynq-7000-based control unit for the external part of the NerveRepack system. The control unit was intended to provide a common platform for controlling and testing the external system through the required communication and control interfaces. These interfaces also had to be accessible from Linux, allowing the connected external modules to be configured, monitored, and tested through software. The main contributions of this work were the implementation of the required hardware interfaces, the development of the corresponding Linux-side software environment, and the validation of the developed control unit. The hardware architecture was designed to provide three independent I2C buses, SPI, CAN, and GPIO according to the requirements defined in Section 2.2. A PetaLinux-based environment and user-space applications were developed to provide software access to these interfaces and support communication, control, and testing.

The control unit was initially implemented on an AX7Z020B-based platform and validated through Linux-side interface checks, functional interface tests, and tests with the wireless power transfer, power management, and data communication subsystems. The control unit was then migrated to the more compact AC7020C platform, where the necessary validation tests were repeated to confirm the required interface functionality.

1.3 Thesis Organization

This thesis consists of six chapters:

- **Chapter 1** introduces the context and motivation of the work and presents the objectives and main contributions of the thesis.
- **Chapter 2** provides the background of the work by describing the NerveRepack system architecture, the control unit requirements, and the main hardware, communication, and software concepts relevant to the development.
- **Chapter 3** describes how the control unit hardware was developed and adapted to the target platforms, including the Vivado design, pin assignment and input/output (I/O) constraints, hardware export, and migration to the AC7020C platform.
- **Chapter 4** explains how the embedded Linux environment was prepared and deployed on the control unit, and how the implemented interfaces were configured and verified from Linux.
- **Chapter 5** presents the validation of the control unit, starting with functional tests of the individual interfaces and continuing with tests involving the NerveRepack subsystems.
- **Chapter 6** summarizes the main outcomes of the work and discusses possible future developments.

Chapter 2

System Background and Requirements

2.1 NerveRepack System Architecture

The NerveRepack system, introduced in Section 1.1, can generally be divided into two main parts: the implantable section and the external section. The implantable section contains electrodes that connect to the nerves and electronic circuits necessary for signal acquisition, processing, and stimulation. The external section contains the electronic systems required for communication, power transfer, power management, and control [4].

The general structure of the NerveRepack system is shown in Figure 2.1, where both data and energy transfer between the implantable and external sections are wireless. The figure also shows the main functional blocks involved in the control of the mechatronic device. The structure shown as Wireless Communication represents the data communication between the implantable and external sections, while the structure shown as Inductive Power Supply corresponds to the wireless power transfer between the implantable and external sections. The power management subsystem, which is part of the external electronic platform, is not represented as a separate block in this general diagram. In Figure 2.1, the control unit developed in this thesis forms part of the Control System block shown within the Mechatronic Structures part of the architecture and is intended to serve as a common control platform for both the exoprosthesis and the exoskeleton. Its connections to Artificial Intelligence (AI) Algorithms, Intelligent Interfaces, and Movements Algorithms/Database show how the Control System interacts with other functions in the overall NerveRepack architecture [4]. This thesis focuses on the embedded hardware and software infrastructure that enables the Control System block to interact with the power management, wireless power transfer, and

data communication subsystems. The AI Algorithms, Intelligent Interfaces, and Movements Algorithms/Database blocks are outside the scope of this work.

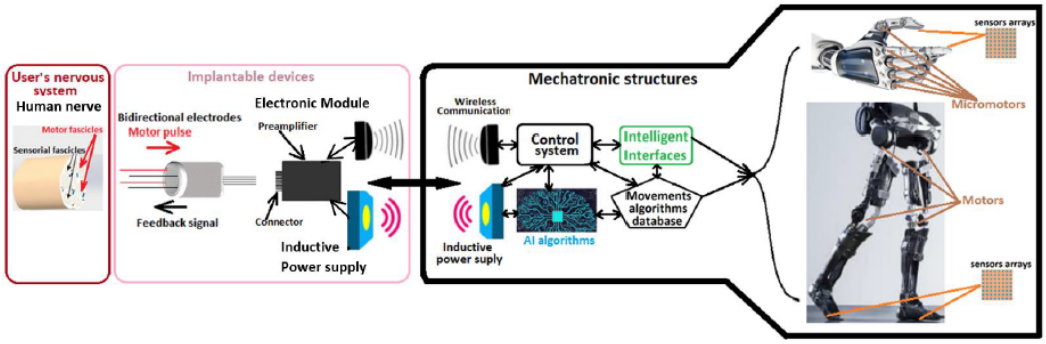


Figure 2.1: General architecture of the NerveRepack system [4].

The wireless power transfer system enables the transfer of energy from the transmitter on the external side to the receiver on the implant side. The power management system provides the power management and control functions required for the operation of the external electronic platform. The data communication system enables bidirectional data transfer between the electronics on the implant side and the external system [4].

The external control unit provides the necessary communication and control interfaces for the configuration, monitoring, and testing of these subsystems.

2.2 Control Unit Requirements

The control unit requirements are defined by the communication and control needs of the NerveRepack external system. Since the different external modules use different communication protocols and operating frequencies, the control unit must provide multiple independent interfaces.

The system requires three independent I2C interfaces, with their operating frequencies determined by the requirements of the corresponding subsystems. The 1 MHz (Fast-mode Plus) interface is used by the power management system for monitoring and control of its power-related peripherals. The higher bus rate was selected to support continuous acquisition of bus-voltage and shunt-voltage measurements from the three INA228 devices while also allowing control of the buck-boost converter and digital potentiometers on the same bus. The 400 kHz (Fast-mode) interface is used to access the monitoring and control devices on the wireless power transfer transmitter board. For the WPT subsystem, this rate was considered sufficient for the required device configuration and monitoring operations. The 100 kHz (Standard-mode) interface is also part of the power

management subsystem and is dedicated to System Management Bus (SMBus) communication with the smart battery module. The interface was configured at 100 kHz, corresponding to the maximum SMBus clock frequency specified for the PMM240 [7].

For the data communication system, the control unit must provide an 8 MHz SPI interface acting as the master. This operating speed was selected according to the maximum SPI connection speed specified for the ISP1507 device used in the subsystem [8]. A CAN interface is also required for communication with the electronics of the hand exoprosthesis. In the final prototype, a dedicated printed circuit board (PCB) will control the motors and acquire data from the sensors, and this board will communicate with the embedded control system through the CAN bus. General-purpose digital inputs and outputs are also required for control and status signals. A total of 32 GPIO lines are provided, while the remaining unused lines are reserved for future extensions of the system.

The I2C, SPI, and GPIO signals connected directly to the external electronic boards use 3.3 V logic levels. CAN communication differs because the physical bus requires a CAN transceiver to interface the controller signals with the differential CAN High (CANH) and CAN Low (CANL) lines. In addition, the interfaces must be accessible from software. Therefore, support for an embedded Linux system developed using PetaLinux is also a requirement of the control unit.

The Zynq-7000 architecture is suitable for these requirements because it combines peripherals in the Processing System and Programmable Logic in the same device. The Zynq-7020 was used for the control unit implementation. The following sections provide the background for these requirements by introducing the Zynq-7000 architecture in Section 2.3, the communication and control interfaces in Section 2.4, and embedded Linux and PetaLinux in Section 2.5.

2.3 Zynq-7000 SoC Architecture

The Zynq-7000 family from Advanced Micro Devices (AMD) is based on a SoC architecture that combines the PS and PL within the same device [5]. This architecture allows software to run on the processor while enabling additional hardware functions on the Field-Programmable Gate Array (FPGA) side.

The Processing System includes ARM processors and various peripherals, while the Programmable Logic consists of FPGA resources. By using these two sections together, software running on the processor can be combined with hardware implemented in the Programmable Logic.

The general architecture of the Zynq-7000 SoC is shown in Figure 2.2. The figure shows the Processing System and Programmable Logic sections, the connections between them, the processor cores within the Processing System, and the main

peripherals.

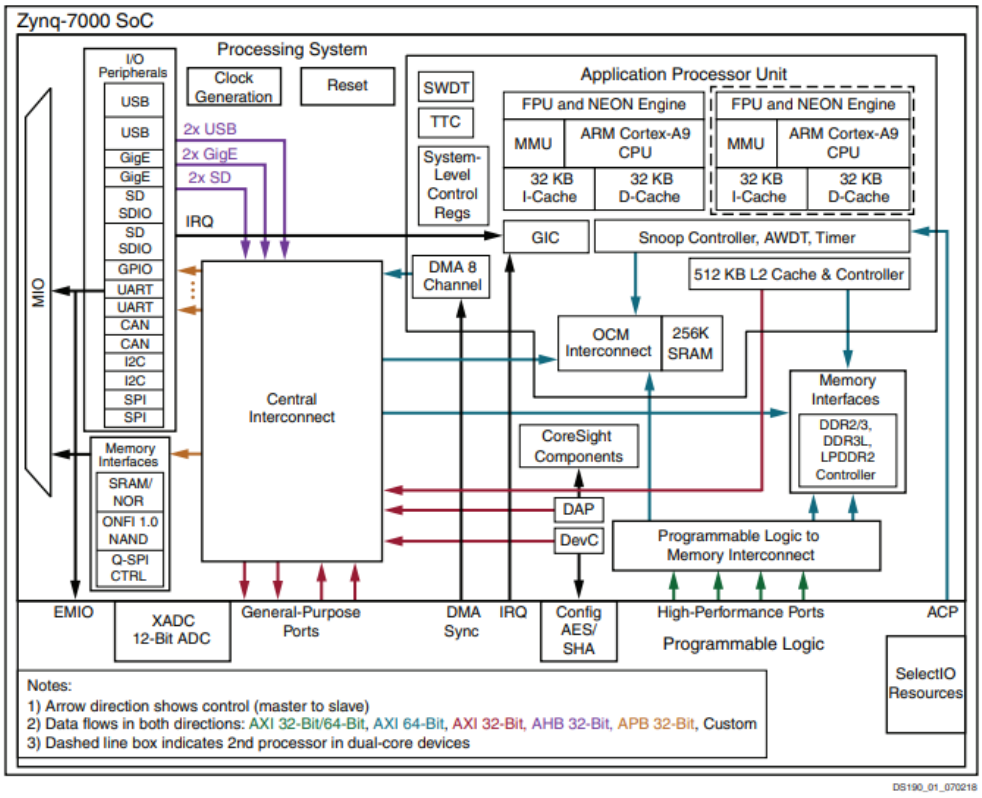


Figure 2.2: Architectural overview of the Zynq-7000 SoC [5].

This architecture is important for the control unit. Most of the necessary communication interfaces can be provided using the peripherals within the Processing System. Additional hardware requirements that cannot be met by the Processing System can be implemented in the Programmable Logic.

2.3.1 Processing System

The Processing System contains the main processing resources and integrated peripherals of the Zynq-7000 device. The Zynq-7020 used in this work includes a dual-core ARM Cortex-A9 processor. For the -2 speed grade device used in the control unit, the maximum processor frequency is 766 MHz [5]. The device part selection and speed grade are described in Section 3.2. The Processing System also includes on-chip memory, a Double Data Rate (DDR) memory controller, interrupt and system-control resources, together with several communication and general-purpose peripherals.

The integrated peripherals include two I2C controllers, two SPI controllers, two CAN controllers, and GPIO. The PS I2C controllers support bus frequencies up to 400 kHz. The SPI controllers support external clock frequencies up to 50 MHz when routed through multiplexed input/output (MIO) and up to 25 MHz when routed through extended multiplexed input/output (EMIO). The CAN controllers support bit rates up to 1 Mb/s [9].

Signals from the peripherals in the Processing System can be routed to external connections through MIO or EMIO. With MIO, the signals are connected directly to the pins dedicated to the PS. With EMIO, the signals are routed through the PL to the FPGA I/O pins. The PS GPIO controller provides up to 54 GPIO signals through MIO and 64 GPIO signals through EMIO [9]. This allows PS peripheral signals to be connected to suitable PL I/O pins instead of being limited to the dedicated MIO pins.

2.3.2 Programmable Logic

Programmable Logic is the FPGA section of the Zynq-7000. It contains configurable logic resources such as look-up tables (LUTs), flip-flops, block random-access memory (RAM), digital signal processing (DSP) resources, and programmable interconnections. For the Zynq-7020 (XC7Z020) device used in this work, the Programmable Logic contains 53,200 LUTs, 106,400 flip-flops, 140 36-Kb block RAMs, and 220 DSP slices [5]. Intellectual property (IP) cores can be added in this section, and custom hardware blocks can also be designed when necessary. This allows peripherals or hardware functions that are not available in the Processing System to be implemented in the Programmable Logic.

2.3.3 PS–PL Interconnection

Communication between the Processing System and peripherals implemented in the Programmable Logic is provided through the AXI interfaces available in the Zynq architecture. AXI supports memory-mapped communication, allowing the processor to access registers of peripherals located in the PL through its address space.

In a memory-mapped AXI connection, the processor can write configuration values to peripheral registers and read status or data registers when required. AXI4-Lite is commonly used for this type of register-based control because it provides a simplified memory-mapped interface without the additional features required for high-bandwidth data transfers [10].

When multiple AXI components are present, an AXI Interconnect routes transactions between the Processing System and the appropriate peripheral [10].

2.4 Communication and Control Interfaces

2.4.1 I2C and SMBus

I2C is a synchronous serial communication protocol that allows multiple electronic components to communicate on the same bus over short distances. Two lines are used for communication: Serial Data (SDA) and Serial Clock (SCL). The master device initiates the communication and generates the clock signal, while the selected slave device responds when its address is transmitted. Data is transferred through the SDA line.

Multiple devices can be connected to the same bus and are distinguished by their addresses. A communication process begins with a start condition, continues with address and data transfer, and ends with a stop condition. An acknowledgement bit is sent by the receiver to indicate that the transmitted data has been received [11]. The I2C specification defines different operating modes, including Standard-mode at 100 kHz, Fast-mode at 400 kHz, and Fast-mode Plus at 1 MHz [11].

SMBus is a communication standard closely related to I2C and is used particularly in system management and smart battery applications. While SMBus communication uses the basic SDA and SCL lines of I2C, additional rules such as timeouts and data integrity checks may also be included [12].

2.4.2 SPI

SPI is a communication interface used for synchronous serial data transfer between electronic devices. A typical SPI connection includes a serial clock (SCLK), a Master Out Slave In (MOSI) data line, a Master In Slave Out (MISO) data line, and a chip select (CS) signal [13].

The clock signal is generated by the master. The chip select signal is used to select the slave device for communication. The separate transmit and receive lines allow data transfer in both directions within the same connection.

Unlike I2C, the SPI protocol does not have a device addressing mechanism. Instead, connected devices are selected using chip select signals. The SPI operating mode is determined by the clock polarity (CPOL) and clock phase (CPHA) settings, and the communicating devices must use the same SPI mode [13].

2.4.3 CAN

CAN is a serial communication protocol developed to allow multiple electronic nodes to reliably exchange data over a common bus. It is widely used in automotive and industrial control systems due to its ability to provide reliable communication in environments with electrical noise.

In CAN communication, data is transmitted in frames, and each frame contains an identifier. When multiple nodes attempt to send data simultaneously, an arbitration mechanism determines which frame gains access to the bus [9].

Physical CAN connections typically use two differential signal lines called CANH and CANL. Differential communication reduces the impact of electrical noise on data transmission [14].

2.4.4 GPIO

GPIO lines are general-purpose digital signals that can be configured as inputs or outputs and controlled by software. Unlike communication protocols, they are not tied to a specific data transmission format. Therefore, they can be used for signals such as enable, reset, ready, or status.

When a GPIO line is configured as an output, it can drive a logic high or logic low level to an external circuit. When configured as an input, it can read the logic state of an external signal [9].

2.5 Embedded Linux and PetaLinux

Embedded Linux is a Linux operating system adapted for embedded computing platforms. It runs on the processor and manages system resources such as memory, processes, and hardware peripherals. Hardware devices are accessed through drivers in the Linux kernel, while applications normally run in user space and interact with these drivers through the interfaces provided by the operating system.

The hardware available to the operating system is described through the device tree, which provides information such as peripheral addresses, interrupts, and other hardware-specific configuration parameters required by the corresponding drivers [6].

PetaLinux is a development environment provided for creating embedded Linux systems for AMD devices such as Zynq SoCs. It provides tools for configuring the Linux kernel, device tree, root file system, and boot components. These elements can then be built and packaged to create the files required to boot Linux on the target hardware platform [6].

Chapter 3

Hardware Design and Implementation

3.1 Hardware Development Platforms

Two different hardware platforms were used during the development of the control unit. Initial design and interface tests were performed on a development platform where the AC7Z020 core board was mounted on the AX7Z020B carrier board. In a later stage, the system was migrated to the more compact AC7020C board, which uses the same Zynq-7020 SoC. The use of the same SoC on both platforms allowed a large part of the developed hardware architecture to be preserved. During the board migration, external connections were rearranged according to the connector layout of the new board.

3.1.1 Initial AX7Z020B-Based Platform

The initial development platform of the control unit consists of the AC7Z020 core board, shown in Figure 3.1, and the AX7Z020B carrier board. The AC7Z020 includes the XC7Z020-2CLG400I, a Zynq-7000 SoC. The core board measures approximately 35 mm $\times$ 42 mm and also features two 512 MB DDR3 memory devices, 256 Mbit Quad SPI (QSPI) Flash, and a 33.333 MHz clock source for the Processing System. Together, the two DDR3 devices provide a total memory capacity of 1 GB and are connected to the Processing System's DDR interface through a 32-bit data bus [15].

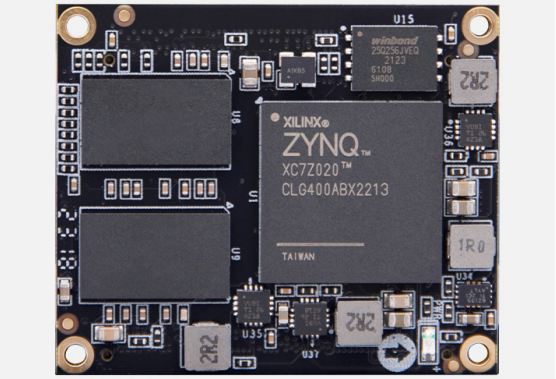


Figure 3.1: AC7Z020 core board used in the initial development platform [15].

The AC7Z020 core board provides the processor and programmable logic resources, while the AX7Z020B carrier board provides the external connections necessary for development and testing. The carrier board measures approximately 130 mm × 90 mm. The core board connects to the carrier board through board-to-board connectors, forming the complete initial development platform shown in Figure 3.2. The AX7Z020B features a microSD card slot, a Universal Serial Bus (USB)-to-Universal Asynchronous Receiver-Transmitter (UART) interface, two CAN interfaces with on-board transceivers for external CAN bus communication, and two 40-pin expansion connectors. The microSD card slot was used to boot the PetaLinux system, the USB-to-UART interface was used to monitor the boot process and access the Linux terminal, and the J20 and J21 expansion connectors were used to connect communication and GPIO signals to external devices [16].

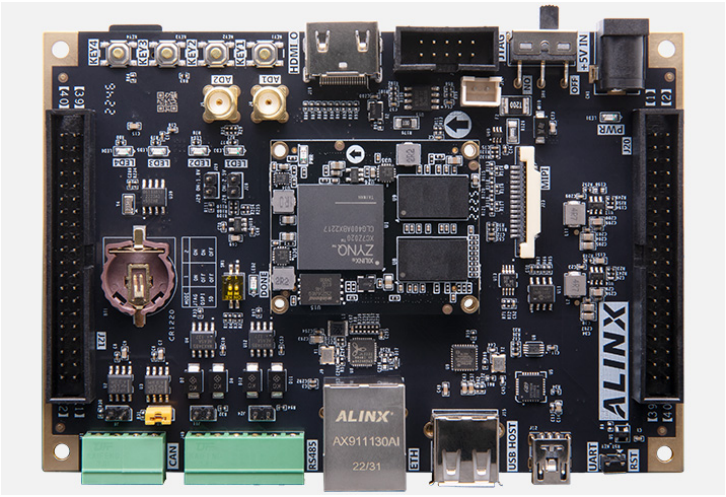


Figure 3.2: AX7Z020B development board with the AC7Z020 core board mounted [16].

This platform was used for hardware and software development of the control unit. The necessary I2C, SPI, CAN, and GPIO interfaces were first implemented on this platform and tested at different stages. Details of the functional interface and subsystem validation tests are presented in Chapter 5.

3.1.2 AC7020C Platform and Migration

After the initial development and validation work, the control unit was migrated to the AC7020C board shown in Figure 3.3. The AC7020C uses the same XC7Z020-2CLG400I Zynq-7000 SoC as the AC7Z020 core board in the initial platform. Compared with the initial platform, the processor, memory, and basic peripherals required during development are integrated on a single, more compact board. The physical dimensions of the board are approximately 75 mm × 64 mm [17].

The AC7020C features two 512 MB DDR3 memory devices, for a total memory capacity of 1 GB. The board also includes 256 Mbit QSPI Flash, a microSD card slot, a USB-to-UART interface, and three 40-pin expansion ports (J10, J11, and J12). A 33.333 MHz clock source is provided for the Processing System, while a separate 50 MHz clock source is available for use on the PL side [17].

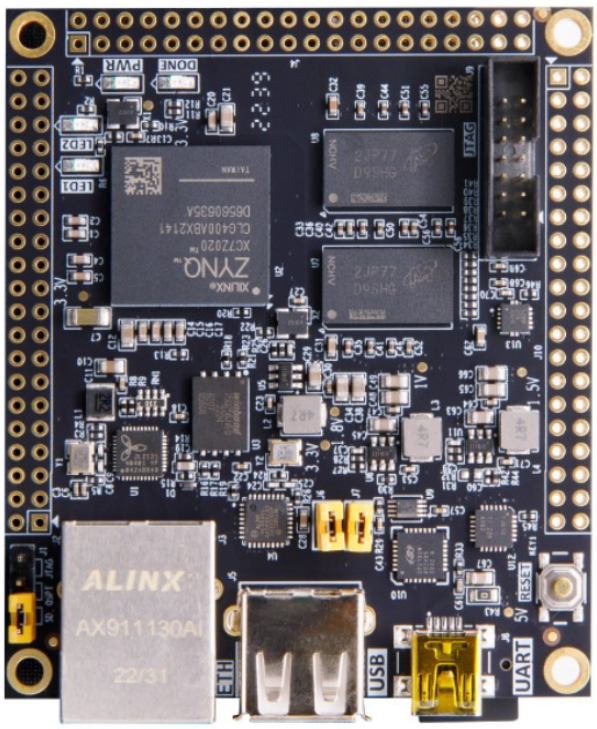


Figure 3.3: AC7020C board used for the final implementation of the control unit [17].

Since both platforms use the same Zynq-7020 SoC, most of the existing Vivado design, including the peripherals used in the Processing System and the main architecture on the PL side, could be retained. However, the boards have different external connectors, and the J10 and J11 connectors on the AC7020C were used instead of the J20 and J21 connectors on the initial platform. One of the main hardware changes required for the migration was therefore updating the physical pin assignments in the Xilinx Design Constraints (XDC) file. The changes made to the pin assignments are described in detail in Section 3.3.

3.2 Vivado Hardware Design

The hardware design of the control unit was created using Vivado 2023.1. When creating the Vivado projects, the xc7z020clg400-2 target part corresponding to the Zynq-7020 SoC used on both boards was selected. XC7Z020 indicates the Zynq-7000 SoC model used, CLG400 indicates the physical package option of the device, and -2 indicates the speed grade value. The package selection defines the physical pin arrangement of the device, while the speed grade indicates the timing and performance class of the device [5].

A new block design was then created using Vivado IP Integrator. The ZYNQ7 Processing System IP core from the Vivado IP Catalog was added to the block design. Using this block, the necessary DDR memory and clock settings were configured, and the peripherals used within the Processing System were enabled.

3.2.1 Zynq PS Configuration

The ZYNQ7 Processing System was configured according to the hardware specifications of the boards. Since both boards use the same Zynq-7020 SoC, the basic Processing System configuration required for the control unit was preserved in both Vivado designs. At this stage, Vivado Run Block Automation was used to create the DDR and FIXED_IO interfaces of the Processing System as external connections within the block design. The on-board 33.333 MHz reference clock was defined for the Processing System [15, 17].

The DDR configuration was set according to the characteristics of the memories on the boards. Although different DDR3 memory models were used on the two boards, the MT41J256M16 RE-125 memory part profile was used in both Vivado designs because the memories have compatible organization and operating characteristics. In this configuration, the DDR interface was used with a 32-bit data bus and a 533.33 MHz operating frequency [15, 17].

The fabric clock FCLK_CLK0, provided by the PS to the PL, was enabled and set to 100 MHz to provide the operating clock for the AXI components in the PL. It was used by the AXI Interconnect and AXI IIC components within the

block design. The separate 50 MHz PL clock source available on both hardware platforms was not used in this design [16, 17].

The necessary peripherals for the control unit, I2C0, I2C1, SPI0, CAN0, and GPIO, were enabled within the ZYNQ7 Processing System IP core. Signals from these peripherals were routed through MIO or EMIO, depending on the physical connections on the board. I2C0, I2C1, SPI0, and 32-bit GPIO were routed through EMIO so that their signals could reach the required external connectors through the PL I/O pins while the controllers remained in the PS.

CAN0 was configured on MIO10 and MIO11 pins. This choice was made because these pins were connected to the CAN transceiver on the carrier board of the initial development platform [16]. The CAN interface was validated using internal loopback, and no external CAN bus communication was performed through the transceiver. However, the CAN transceiver on the initial development platform can be used for external CAN bus communication if needed. Since the AC7020C board does not include an on-board CAN transceiver, an external transceiver would be required for an external CAN bus test.

The SD0 and UART1 peripherals were used for the on-board microSD card and USB-to-UART interface, respectively. SD0 was configured on MIO40–45, while UART1 used MIO48 and MIO49, which are physically connected to the USB-to-UART circuit on the board [16, 17].

For the peripherals used through MIO, the PS I/O bank settings were configured according to the electrical characteristics of the board. The PS MIO pins are divided into two I/O banks. MIO pins 0–15 belong to Bank 0, and MIO pins 16–53 belong to Bank 1. The I/O voltage values of these banks were set to match the physical I/O supply architecture of the hardware platforms used. Therefore, Bank 0 was configured at 3.3 V and Bank 1 at 1.8 V [16, 17]. The MIO pins used by CAN0 are located in Bank 0, while the MIO pins used by SD0 and UART1 are located in Bank 1. These bank settings are only valid for PS signals routed through MIO.

3.2.2 Block Design and PL Integration

The control unit requires three independent I2C interfaces, while the Processing System provides only two I2C controllers. In addition, the required 1 MHz I2C frequency exceeds the maximum 400 kHz bus frequency supported by the PS I2C controllers. Therefore, the 1 MHz I2C interface was implemented in the Programmable Logic using an AXI IIC IP core from the Vivado IP Catalog. The AXI IIC was configured for a 1 MHz SCL frequency, corresponding to Fast-mode Plus, as required in Section 2.2. This parameter defines the approximate SCL frequency generated by the AXI IIC when operating as an I2C master.

The necessary AXI, clock, and reset connections were created using Vivado

Connection Automation. The AXI IIC provides an AXI4-Lite slave interface for access to its control and status registers. It was connected to the Processing System through the M_AXI_GP0 general-purpose AXI master port and the AXI Interconnect, while the I2C interface was defined as an external port within the block design. In the Vivado Address Editor, the AXI IIC was mapped to the address range from 0x41600000 to 0x4160FFFF, corresponding to a 64 KB address space. The 100 MHz FCLK_CLK0 defined in Section 3.2.1 was used for the clock connections of the AXI IIC and AXI Interconnect. The necessary reset signals were generated by the Processor System Reset block.

The interrupt connection for the AXI IIC was also configured manually. Fabric Interrupts were enabled in the ZYNQ7 Processing System, and the IRQ_F2P port was enabled for interrupt transfer from PL to PS. The interrupt output of the AXI IIC was then connected to the IRQ_F2P input of the Processing System, allowing interrupt signals generated by the AXI IIC to be transferred to the PS.

Once the connections within the block design were complete, the Vivado Validate Design process was run. This process was used to check the consistency of the interfaces, connections, and basic configurations within the block design, and the design was successfully validated. A Hardware Description Language (HDL) wrapper was then created for the block design. The HDL wrapper defines the block design as the top-level HDL module of the project and exposes the external ports of the block design at the top level. This allows the block design to be used as the top-level hardware structure of the project during the synthesis and implementation phases.

The EMIO connections on the PS side and the external ports defined for the AXI IIC on the PL side were assigned to the physical FPGA pins and expansion connectors on the board in the XDC file. These physical pin assignments are described in Section 3.3. The completed Vivado block design structure of the control unit is shown in Figure 3.4. The figure shows the ZYNQ7 Processing System, AXI Interconnect, AXI IIC, and Processor System Reset blocks, as well as the AXI, clock, reset, and interrupt connections between them.

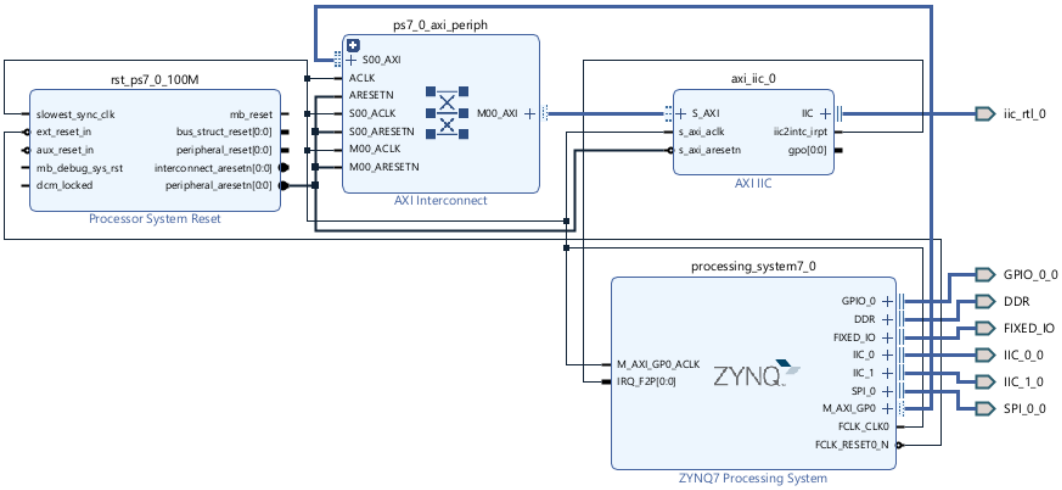


Figure 3.4: Vivado block design of the control unit.

3.3 Pin Assignments and I/O Constraints

External ports defined in the block design were assigned to the corresponding FPGA pins in the XDC file, where the I/O standards for these signals were also specified. Since the expansion connectors and associated FPGA pins of the AX7Z020B and AC7020C boards were different, separate physical pin assignments were used for the two platforms. CAN0, SD0, and UART1 are not included in these assignments because they are routed through MIO; their MIO configuration is described in Section 3.2.1.

3.3.1 Initial AX7Z020B Pin Assignment

On the initial development platform, the PS I2C and SPI signals routed through EMIO, together with the AXI IIC signals generated in the PL, were connected to the J20 expansion connector. The J21 expansion connector was used for the 32-bit GPIO interface. Signals belonging to the same communication interface were assigned to neighboring connector pins where possible, and the I2C and SPI signals were placed close to ground pins. The pin assignments used for the AX7Z020B are shown in Table 3.1.

Table 3.1: External interface pin assignments on the initial AX7Z020B platform.

Interface	Signals	Connector	Pins
PL I2C	SCL, SDA	J20	3–4
PS I2C0	SCL, SDA	J20	5–6
PS I2C1	SCL, SDA	J20	7–8
SPI0	SCLK, MOSI, MISO, CS0	J20	33–36
GPIO	GPIO[0:31]	J21	3–34

Since only one slave device was used on the SPI0 interface, CS0 was sufficient for communication. However, when SPI0 was routed externally through EMIO, CS1 and CS2 outputs were also generated as external signals; these signals were assigned to physical pins in the XDC file but were not used in the system. All I2C, SPI, and GPIO signals assigned in the XDC file were defined using the LVCMOS33 I/O standard.

3.3.2 AC7020C Pin Assignment

When the system was migrated to the AC7020C board, the basic communication structure within the block design was preserved, while the physical pin assignments were changed according to the new board's connector layout. The PS interfaces routed through EMIO and the AXI IIC interface implemented in the PL were assigned to the J10 and J11 expansion connectors on the AC7020C. Signals for the three I2C interfaces and SPI0 were grouped on J10, while the 32 GPIO signals were grouped on J11. The third expansion connector, J12, was not used because it was not required.

CAN0 remained configured on MIO10 and MIO11 in the AC7020C design. These MIO signals are available through the J10 expansion connector on pins 35 and 31, respectively [17]. An external transceiver is therefore required for external CAN communication.

Signals belonging to the same communication interface were assigned to neighboring connector pins where possible, and the I2C and SPI signals were placed close to ground pins. The pin assignments used for the AC7020C are shown in Table 3.2.

Table 3.2: External interface pin assignments on the AC7020C platform.

Interface	Signals	Connector	Pins
PL I2C	SCL, SDA	J10	3–4
PS I2C0	SCL, SDA	J10	5–6
PS I2C1	SCL, SDA	J10	7–8
SPI0	SCLK, MOSI, MISO, CS0	J10	9–12
GPIO	GPIO[0:31]	J11	3–34

Although only CS0 was used for SPI communication, CS1 and CS2 were also assigned to physical pins in the XDC file but were not used in the system. All I2C, SPI, and GPIO signals listed in Table 3.2 were configured using the LVCMOS33 I/O standard. In addition, since GPIO[31] was used as a READY/handshake input in later SPI communication tests, a pull-down was defined for this line in the XDC file. This kept the input at a low level when it was not actively driven by an external device.

3.4 Synthesis and Implementation

The hardware design was synthesized and implemented in Vivado once the block design, HDL wrapper, and physical pin assignments were completed. These processes were performed separately for both hardware platforms. In the synthesis phase, the IP cores used in the block design and their connections were converted into a hardware netlist for the target xc7z020clg400-2 device. After completing the synthesis, the implementation was run to place the logical resources used on the PL side onto the physical resources within the FPGA and create the required connections between them.

After completing the implementation, resource utilization reports were reviewed for both the initial design using the AC7Z020 core board on the AX7Z020B carrier board and the design created for the AC7020C. Post-implementation results are compared in Table 3.3.

Table 3.3: Post-implementation resource utilization for the two hardware platforms.

Resource	Available	AX7Z020B	AC7020C	Utilization (%)
LUT	53200	775	774	1.46 / 1.45
LUT RAM (LUTRAM)	17400	49	49	0.28 / 0.28
Flip-Flop (FF)	106400	820	820	0.77 / 0.77
I/O	125	44	44	35.20 / 35.20

Resource utilization results show that the utilization of LUTs, LUTRAM, and FFs on the PL side is low in both designs. This low resource utilization is expected because the PL is mainly used for the additional AXI IIC interface and the necessary AXI, clock, and reset connections. Similar resource utilization values obtained in both designs are consistent with the fact that the core hardware architecture was largely preserved during the board migration.

After implementation was completed in the final AC7020C design, the timing summary was also checked. For setup timing, the Worst Negative Slack (WNS) was 4.415 ns, and the Total Negative Slack (TNS), representing the sum of all negative setup slack values, was 0 ns. For hold timing, the Worst Hold Slack (WHS) was 0.095 ns, and the Total Hold Slack (THS), representing the sum of all negative hold slack values, was reported as 0 ns. Positive WNS and WHS values together with zero TNS and THS values indicate that there are no setup or hold timing violations in the design.

3.5 Bitstream Generation and Hardware Export

After successfully completing the synthesis and implementation phases, an FPGA bitstream file was created. The bitstream contains the configuration data required to implement the hardware design on the FPGA. This data determines how the programmable logic resources in the PL, the connections between these resources, and the I/O resources will be configured. When the bitstream is loaded onto the board, the FPGA resources in the PL are programmed according to this configuration data.

The hardware design was then exported in Xilinx Support Archive (XSA) format for use in the PetaLinux development environment. The XSA file contains the definition of the hardware platform and the generated FPGA bitstream and is used as a hardware input for the PetaLinux project.

When the system was migrated to the AC7020C board, the synthesis and implementation processes were repeated using the updated Vivado design and the pin assignments created for the new board. The same hardware export process was repeated for the AC7020C design, and an updated XSA file was exported along with the bitstream created for the new board. The PetaLinux project was then updated using this hardware platform.

Chapter 4

Embedded Linux Implementation and System Bring-Up

4.1 PetaLinux Development Environment and Project Setup

The PetaLinux development environment was prepared using a virtual machine created with Oracle VirtualBox. Oracle VirtualBox is virtualization software that allows virtual machines to be created and different operating systems to be run on a single computer. PetaLinux 2023.1 was used to ensure compatibility with Vivado 2023.1, which was used in the project. Ubuntu 20.04.6 Long-Term Support (LTS) was installed on the virtual machine to provide a Linux environment supported by PetaLinux 2023.1 [6].

PetaLinux 2023.1 was then installed in the Ubuntu environment. This installation was performed only once during the initial setup of the development environment. After the development environment was prepared, a PetaLinux project was created for the control unit.

The XSA file exported from Vivado was imported into the PetaLinux project. This step configured the PetaLinux project according to the hardware design defined in Vivado. When the hardware design was modified and a new XSA file was generated, the PetaLinux project was reconfigured using the updated hardware definition.

The PetaLinux project was then used to configure the Linux kernel, root filesystem, and device tree, as well as to build the system and generate the boot files. The Linux kernel, root filesystem, and device tree configurations are described in

Section 4.2, while the build and boot file generation processes are described in Section 4.3.

4.2 Board Support Package Customization

The BSP contains the software and configuration components necessary for an embedded Linux system to operate on the target hardware platform. In the PetaLinux project, the BSP was generated based on the hardware definition exported from Vivado and customized to meet the requirements of the control unit. As part of this customization, Linux kernel and root filesystem settings were adjusted, and necessary custom definitions were added to the device tree.

4.2.1 Linux Kernel and Root Filesystem Configuration

In the PetaLinux project, the Linux kernel was configured to provide the necessary support for the I2C, SPI, CAN, and GPIO interfaces used by the control unit. The root filesystem was configured to include the user-space tools required for subsequent verification and testing activities.

For the I2C interface, the Cadence I2C driver was used for the I2C controllers within the Processing System, while the Xilinx I2C driver was used for the AXI IIC controller located in the Programmable Logic. I2C character device support was also enabled to allow access to the I2C buses from Linux user space. This allowed the I2C controllers to be accessed through `/dev/i2c-*` device nodes.

For the SPI interface, the Cadence SPI driver, which supports the SPI controller within the Processing System, was used. Additionally, SPIDEV support was enabled, allowing user-space access to the SPI interface through the `/dev/spidev*` device node.

For the CAN interface, the Linux CAN infrastructure and the Xilinx CAN driver, which supports the CAN controller within the Zynq Processing System, were used. This configuration enabled user-space access to the CAN interface through the SocketCAN infrastructure, which Linux uses for CAN communication.

For GPIO, no additional kernel configuration was required. The Zynq PS GPIO driver and the Linux GPIO infrastructure were already enabled in the current kernel configuration, allowing Linux to use GPIO signals routed through EMIO.

For the root filesystem, the `i2c-tools` and `can-utils` packages were included to provide the necessary user-space tools. While `i2c-tools` enabled the control of I2C buses and connected devices from the Linux command line, `can-utils` provided the user-space tools used for testing CAN communication. Since the necessary user-space access for SPI was provided through SPIDEV, no separate root filesystem package was required for this purpose. Similarly, GPIO access did not require a separate user-space package.

4.2.2 Device Tree Configuration

When the XSA file exported from Vivado was imported into the PetaLinux project, the base device tree was automatically generated by PetaLinux. Additional user-specific settings and modifications were defined in the `system-user.dtsi` file. This file includes the `system-conf.dtsi` file generated by PetaLinux and ensures that user-defined settings are applied to the base device tree.

Alias definitions were added for the I2C and SPI controllers to explicitly define the bus numbering on the Linux side. The two I2C controllers within the PS were defined as `i2c0` and `i2c1`, respectively, while the AXI IIC controller located in the PL was defined as `i2c2`. Similarly, the PS SPI controller was mapped as `spi0`. These alias definitions were not required for the controllers to function but were used to make the logical bus numbering on the Linux side explicit.

In accordance with the control unit requirements defined in Section 2.2, the three I2C interfaces were configured with different operating frequencies. The AXI IIC controller located in the Programmable Logic had been configured with a 1 MHz SCL frequency parameter in the Vivado hardware design. A bus frequency of 1 MHz was also specified on the Linux side by defining `clock-frequency = <1000000>` in the `system-user.dtsi` file. The `clock-frequency` property specifies the desired I2C bus clock frequency used by the Linux driver to configure the controller timing. The bus frequencies of the I2C0 and I2C1 controllers within the Processing System were defined as 400 kHz and 100 kHz, respectively, in the device tree. For all three I2C nodes, the `status` property was explicitly set to `okay`.

For the SPI interface, the PS SPI0 controller was enabled and configured to use a single chip-select line. A `spidev` device was defined on chip-select 0, enabling the SPI interface to be accessed from Linux user space. In accordance with the 8 MHz SPI requirement specified in Section 2.2, the `spi-max-frequency` value was set to 8 MHz. While this value defines the maximum permissible clock frequency for the SPI device, the clock frequency actually used for transfers is determined by the user-space application without exceeding this limit.

No additional user-specific definitions were made in `system-user.dtsi` for CAN and GPIO. Since the base device tree definitions were sufficient for these interfaces, no additional parameters or overrides were required.

4.3 System Build and Deployment

The system was built after completing the PetaLinux project configuration to generate the necessary Linux and boot components. The resulting files were then transferred to an SD card and used to boot the control unit on the hardware platform.

4.3.1 PetaLinux Build and Boot Image Generation

The system was built using the `petalinux-build` command. This process generated the software components required for the target system based on the Linux kernel, root filesystem, and device tree configurations defined in the project. The build process produced `image.ub`, which contains the Linux kernel, device tree, and root filesystem, as well as U-Boot and the `boot.scr` file, which directs U-Boot to load the necessary images during the boot process. These files were located in the `images/linux` directory of the PetaLinux project [6].

After the build process, the boot image (`BOOT.BIN`) was created. This step was performed using the `petalinux-package --boot` command. The `BOOT.BIN` file included the First Stage Boot Loader (FSBL), which performs the initial hardware initialization, the FPGA bitstream generated from the Vivado hardware design, and the U-Boot bootloader, which manages the loading of the Linux image and the subsequent boot stage [6].

4.3.2 SD Card Deployment and System Boot

The generated `BOOT.BIN`, `image.ub`, and `boot.scr` files were transferred to the microSD card. These files were copied to the card's boot partition, which was formatted as File Allocation Table 32 (FAT32). A FAT32 partition was used because the Zynq-7000 SoC's BootROM loads the `BOOT.BIN` file from the SD card's FAT-formatted boot partition [9]. The `image.ub` and `boot.scr` files were also copied to the same partition for use in subsequent boot stages. A separate root filesystem partition was not used, as the root filesystem was included in `image.ub` and loaded from it during the boot process.

The control unit was then configured to boot from the SD card, and the system was started. The boot process was monitored through the UART connection using the TeraTerm serial terminal program configured at 115200 baud. This made it possible to monitor the messages generated during the initialization of the FSBL, U-Boot, and Linux. Successful booting of the system from the microSD card was confirmed when the PetaLinux login screen was reached.

Whenever changes were made to the hardware design or PetaLinux configuration, the updated boot files were copied to the microSD card's FAT32 boot partition, and the same boot procedure was repeated.

4.4 Verification of Interface Availability and Configuration

Once the system had booted successfully, the I2C, SPI, CAN, and GPIO interfaces used in the control unit were checked on Linux. This verification was performed

before the functional tests with external devices.

The `i2cdetect -l` command was used in Linux to check the I2C interfaces. The output listed the two Cadence I2C controllers within the Processing System as `i2c-0` and `i2c-1`, and the AXI IIC controller within the Programmable Logic as `i2c-2`. This confirmed that the three I2C interfaces in the design were recognized by Linux as distinct I2C adapters. The `clock-frequency` values in the device tree were also checked on the running system. These values appeared in 32-bit hexadecimal format as `00 06 1a 80`, `00 01 86 a0`, and `00 0f 42 40`, respectively. Their decimal equivalents are 400,000 Hz, 100,000 Hz, and 1,000,000 Hz, corresponding to operating frequencies of 400 kHz for PS I2C0, 100 kHz for PS I2C1, and 1 MHz for AXI IIC. The results obtained are consistent with the requirements defined in Section 2.2 and the device tree configuration established in Section 4.2.2. These results are shown in Figure 4.1.

```
my_project:/home/petalinux# i2cdetect -l
i2c-0   i2c             Cadence I2C at e0004000                 I2C adapter
i2c-1   i2c             Cadence I2C at e0005000                 I2C adapter
i2c-2   i2c             xiic-i2c 41600000.i2c                   I2C adapter
my_project:/home/petalinux# hexdump -C /sys/firmware/devicetree/base/axi/i2c@e0004000/clock-frequency
00000000  00 06 1a 80                                       |....|
00000004
my_project:/home/petalinux# hexdump -C /sys/firmware/devicetree/base/axi/i2c@e0005000/clock-frequency
00000000  00 01 86 a0                                       |....|
00000004
my_project:/home/petalinux# hexdump -C /sys/firmware/devicetree/base/amba_pl/i2c@41600000/clock-frequency
00000000  00 0f 42 40                                       |..B@|
00000004
```

Figure 4.1: Linux-side verification of the I2C interfaces and their configured clock frequencies.

The availability of the SPI interface on the Linux side was confirmed by the creation of the `/dev/spidev0.0` device node. The configuration was also verified by reading the SPI parameters from the device tree on the running system. The hexadecimal value `00 7a 12 00` read for `spi-max-frequency` corresponds to 8,000,000 Hz, which is equivalent to 8 MHz. Furthermore, a `num-cs` value of 1 indicates that a single chip-select line is used, while a `reg` value of 0 indicates that the SPI device is defined on chip-select 0. Thus, it was confirmed that the SPI configuration defined in Section 4.2.2 was correctly transferred to the Linux system. The corresponding SPI verification results are shown in Figure 4.2.

```
my_project:/home/petalinux# ls -l /dev/spidev*
crw-------    1 root     root      153,   0 Jan  1  1970 /dev/spidev0.0
my_project:/home/petalinux# find /sys/firmware/devicetree/base -name spi-max-frequency -print
/sys/firmware/devicetree/base/axi/spi@e0006000/spidev@0/spi-max-frequency
my_project:/home/petalinux# hexdump -C /sys/firmware/devicetree/base/axi/spi@e0006000/spidev@0/spi-max-frequency
00000000  00 7a 12 00                                       |.z..|
00000004
my_project:/home/petalinux# hexdump -C /sys/firmware/devicetree/base/axi/spi@e0006000/num-cs
00000000  00 00 00 01                                       |....|
00000004
my_project:/home/petalinux# hexdump -C /sys/firmware/devicetree/base/axi/spi@e0006000/spidev@0/reg
00000000  00 00 00 00                                       |....|
00000004
```

Figure 4.2: Linux-side verification of the SPI interface and its configuration.

To verify that the CAN interface was correctly recognized by Linux, the CAN node in the device tree and the Linux network interface were checked. The CAN node in the device tree was found to have its `status` property set to `okay`. By examining the `dmesg` output, which displays Linux kernel messages, it was confirmed that the CAN infrastructure had been loaded by the system. Finally, the command `ip -details link show can0` confirmed that the `can0` network interface had been created in the system. During this check, the interface appeared as `DOWN` and the CAN status as `STOPPED` because CAN communication had not yet been initiated. At this stage, only the recognition of the interface by Linux and the creation of the `can0` network interface were verified. The CAN interface verification is shown in Figure 4.3.

```
my_project:/home/petalinux# cat /proc/device-tree/axi/can@e0008000/compatible
xlnx,zynq-can-1.0my_project:/home/petalinux# cat /proc/device-tree/axi/can@e0008000/status
okaymy_project:/home/petalinux# dmesg | grep -i "can:"
can: controller area network core
can: raw protocol
can: broadcast manager protocol
can: netlink gateway - max_hops=1
my_project:/home/petalinux# ip -details link show can0
2: can0: <NOARP,ECHO> mtu 16 qdisc noop state DOWN mode DEFAULT group default qlen 10
    link/can  promiscuity 0 minmtu 0 maxmtu 0
    can state STOPPED (berr-counter tx 0 rx 0) restart-ms 0
```

Figure 4.3: Linux-side verification of the CAN interface.

The relevant GPIO controller information was checked to verify that the GPIO interface was recognized by Linux. The output showed that the controller was identified as `zynq_gpio` and provided a total of 118 GPIO lines. This result indicates that the GPIO controller within the Zynq Processing System was correctly detected by Linux. In the control unit, however, only the 32 GPIO lines routed externally through EMIO were used. The GPIO interface verification is shown in Figure 4.4.

```
my_project:/home/petalinux# grep . /sys/class/gpio/gpiochip906/label /sys/class/gpio/gpiochip906/base /sys/class/gpio/gpiochip906/ngpio
/sys/class/gpio/gpiochip906/label:zynq_gpio
/sys/class/gpio/gpiochip906/base:906
/sys/class/gpio/gpiochip906/ngpio:118
my_project:/home/petalinux# tr -d '¥0' < /sys/class/gpio/gpiochip906/device/of_node/compatible; echo
xlnx,zynq-gpio-1.0
```

Figure 4.4: Linux-side verification of the PS GPIO controller.

As a result of these checks, it was confirmed that the three I2C interfaces, as well as the SPI, CAN, and GPIO interfaces, were recognized by Linux and that the necessary configurations were correctly applied to the system. Functional communication with external devices was not performed at this stage. Chapter 5 covers functional validation of the interfaces, including tests with external devices and loopback configurations, as well as subsystem validation.

Chapter 5

Validation and System Integration

5.1 Validation Methodology

Validation tests were conducted using either Linux commands or test programs developed in C, depending on the requirements of each test. For tests performed using Linux commands, the relevant commands were executed directly in the PetaLinux terminal after the boot process described in Section 4.3.2 had completed.

For tests performed using C programs, the programs were cross-compiled in an Ubuntu 20.04.6 LTS environment running on VirtualBox on a Windows host computer. The programs were compiled on the host computer, while the resulting executable files were generated for the ARM processor within the Zynq Processing System. Compilation was performed using the `arm-linux-gnueabihf-gcc` compiler.

The generated executable files were copied to the FAT32 boot partition of the SD card containing the PetaLinux boot files. The SD card was inserted into the target board, and the system was booted using the method described in Section 4.3.2 to access the PetaLinux environment. Subsequently, the FAT32 partition of the SD card was mounted at the `/mnt/sd` directory within the Linux file system, and the executable files were accessed from this directory. The relevant program was then executed directly from the mounted SD card partition.

5.2 Functional Interface Validation

Functional Interface Validation focused on testing the communication and control interfaces independently before subsystem validation. External devices were used

for the I2C and SPI tests, while loopback configurations were used for CAN and GPIO. The I2C interfaces were tested using Linux I2C commands directly from the PetaLinux terminal, while the SPI, CAN, and GPIO interfaces were tested using user-space programs developed in C. Linux I2C tools were sufficient for basic device detection and register read/write operations, whereas the SPI, CAN, and GPIO tests required the generation of specific data sequences, comparison of transmitted and received data, and automatic evaluation of the test results.

Interface tests were initially conducted on a platform consisting of the AC7Z020 core board mounted on the AX7Z020B development board. Following the migration of the system to the AC7020C board, the operation of the relevant interfaces was also verified on the new platform.

5.2.1 I2C Interface

The I2C interfaces were functionally validated using an external MP8859 buck-boost DC-DC converter evaluation board. The objective of this test was to verify that the three I2C interfaces could communicate with a real external device at the register level and that the device's operating parameters could be modified using Linux. The MP8859 supports the configuration of various operating parameters, including the output voltage, through I2C [18].

The connected device was first checked to verify that it was correctly detected on the I2C bus. The `i2cdetect` command in Linux was used for this purpose, and the MP8859 was detected at address 0x62. This address matches one of the I2C slave addresses specified for the device in the MP8859 datasheet. The device's identification registers were then read using the `i2cget` command. Values of 0x09, 0x58, and 0x01 were obtained from addresses 0x27, 0x28, and 0x29, respectively. These values matched the expected values specified in the datasheet for the manufacturer ID, device ID, and IC revision [18]. This also confirmed that the device detected at address 0x62 was the MP8859.

I2C write operations were then tested using the converter's output voltage configuration registers. Initially, the value 0x04 was read from the VOUT_L (0x00) register and the value 0x3E from the VOUT_H (0x01) register using the `i2cget` command. According to the output voltage encoding defined in the datasheet, these values correspond to the device's default 5 V output voltage setting. The values 0x00 and 0x4B were then written to the VOUT_L and VOUT_H registers, respectively, using the `i2cset` command, changing the output voltage setting to 6 V. To apply the new value, the GO_BIT in the VOUT_GO (0x02) register was set to 1. The datasheet specifies that the new output voltage setting is applied only after this bit is set [18]. Finally, the VOUT_L and VOUT_H registers were read back using `i2cget` to verify that the written values, 0x00 and 0x4B, were correctly stored in the device.

In addition to register operations, the converter's physical output voltage was checked using a multimeter. It was observed that the output voltage, initially around 5 V, increased to approximately 6 V after the register settings were modified. This confirmed that the I2C write operation produced the expected change not only at the register level but also at the device's actual output. Figure 5.1 shows the terminal output for the device detection, register read, and register write operations performed on the `i2c-2` interface.

```
my_project:/home/petalinux# i2cdetect -y -r 2
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                         -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- 62 -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
my_project:/home/petalinux# i2cget -y 2 0x62 0x27
0x09
my_project:/home/petalinux# i2cget -y 2 0x62 0x29
0x01
my_project:/home/petalinux# i2cget -y 2 0x62 0x28
0x58
my_project:/home/petalinux# i2cget -y 2 0x62 0x00
0x04
my_project:/home/petalinux# i2cget -y 2 0x62 0x01
0x3e
my_project:/home/petalinux# i2cset -y 2 0x62 0x00 0x00
my_project:/home/petalinux# i2cset -y 2 0x62 0x01 0x4B
my_project:/home/petalinux# i2cset -y 2 0x62 0x02 0x01
my_project:/home/petalinux# i2cget -y 2 0x62 0x00
0x00
my_project:/home/petalinux# i2cget -y 2 0x62 0x01
0x4b
```

Figure 5.1: MP8859 device detection and register read/write operations on the `i2c-2` interface.

The device detection and register read/write operations demonstrated for the `i2c-2` interface in Figure 5.1 were also repeated on the `i2c-0` and `i2c-1` interfaces, and successful communication with the MP8859 was established on both. These tests confirmed that all three I2C interfaces could communicate with a real external device, read device registers, and write data to configuration registers. In particular, changing the MP8859 output voltage setting from 5 V to 6 V demonstrated that the I2C interfaces could be used not only to read but also to modify the operating parameters of an external device.

5.2.2 SPI Interface

The SPI interface was functionally validated using a test setup where the control unit acted as the master and an external nRF52840 development kit acted as the slave. SPI Mode 0 and an 8-bit word length were used in the tests. The 8 MHz SPI operating requirement defined for the data communication system in Section 2.2 was used as the target operating speed. Two different tests were conducted to verify that the interface could perform full-duplex data transmission and reliably sustain sequential packet transmission.

Full-duplex operation was first tested at 8 MHz. The C user-space application on the Zynq side used transmit and receive buffers within the same SPI transaction, allowing data to be sent to and received from the slave simultaneously. The terminal output on the Zynq side displayed the data "Zynq!!" in the transmit (TX) field and "Nordic" in the receive (RX) field, which was transmitted by the nRF52840 slave. The values shown in the TX and RX fields were hexadecimal byte values corresponding to the character strings "Zynq!!" and "Nordic," respectively. The terminal output on the nRF52840 side also confirmed the successful receipt of the "Zynq!!" data. This verified that the MOSI and MISO data lines functioned correctly for bidirectional data transfer. The Zynq master terminal output is shown in Figure 5.2, while the corresponding nRF52840 slave terminal output is shown in Figure 5.3.

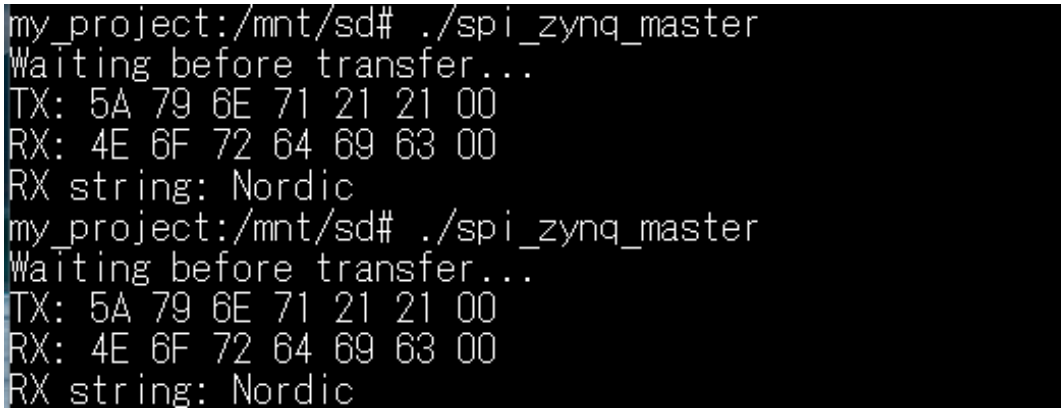


Figure 5.2: Zynq master terminal output during the full-duplex SPI test.



Figure 5.3: nRF52840 slave terminal output during the full-duplex SPI test.

The SPI interface was then tested using sequential packet transmission. The SPI master on the Zynq side was configured through `/dev/spidev0.0` to operate in Mode 0 with an 8-bit word length at 8 MHz. A total of 1,050 packets, each 244 bytes long, were transmitted with a 50 μs delay between consecutive packets, resulting in a total data transfer of 256,200 bytes.

The first two bytes of each packet contained a packet index starting from zero. The first transmitted packet therefore carried an index value of 0x0000, and this value was incremented for each subsequent packet. For example, in the nRF52840 terminal output shown in Figure 5.5, the packet displayed as the 50th received packet has an index value of 49, so its first two bytes appear as 0x00 0x31. The remaining 242 bytes were filled with a predefined, incrementing data pattern. This structure facilitated the monitoring of the packet sequence and content at the slave side. The slave application on the nRF52840 received the packets through SPI and printed the content of every 50th packet to the terminal. This allowed the received packets and their data pattern to be checked while the transfer was in progress. The Zynq master output for the sequential packet transfer test is shown in Figure 5.4, while the packet contents received by the nRF52840 slave are shown in Figure 5.5.

```
Waiting 2 seconds for Nordic to become ready...
Transfer starting...

Progress: 100/1050 (9.5%)
Progress: 200/1050 (19.0%)
Progress: 300/1050 (28.6%)
Progress: 400/1050 (38.1%)
Progress: 500/1050 (47.6%)
Progress: 600/1050 (57.1%)
Progress: 700/1050 (66.7%)
Progress: 800/1050 (76.2%)
Progress: 900/1050 (85.7%)
Progress: 1000/1050 (95.2%)

=== Transfer Complete ===
Success: 1050 / 1050 (100.00%)
Fail:    0 / 1050
Elapsed: 0.598 seconds
Throughput: 418.13 KB/sec
Avg per packet: 569.9 us
```

Figure 5.4: Zynq master terminal output during the sequential packet transfer test.

```
<info> app: Received packet number 50
<info> app: Content of packet 50:
<info> app:  00 31 00 01 02 03 04 05|.1......
<info> app:  06 07 08 09 0A 0B 0C 0D|........
<info> app:  0E 0F 10 11 12 13 14 15|........
<info> app:  16 17 18 19 1A 1B 1C 1D|........
<info> app:  1E 1F 20 21 22 23 24 25|.. !"#$%
<info> app:  26 27 28 29 2A 2B 2C 2D|&'()*+,-
<info> app:  2E 2F 30 31 32 33 34 35|./012345
<info> app:  36 37 38 39 3A 3B 3C 3D|6789:;<=
<info> app:  3E 3F 40 41 42 43 44 45|>?@ABCDE
<info> app:  46 47 48 49 4A 4B 4C 4D|FGHIJKLM
<info> app: Received packet number 100
<info> app: Content of packet 100:
<info> app:  00 63 00 01 02 03 04 05|.c......
<info> app:  06 07 08 09 0A 0B 0C 0D|........
<info> app:  0E 0F 10 11 12 13 14 15|........
<info> app:  16 17 18 19 1A 1B 1C 1D|........
<info> app:  1E 1F 20 21 22 23 24 25|.. !"#$%
<info> app:  26 27 28 29 2A 2B 2C 2D|&'()*+,-
<info> app:  2E 2F 30 31 32 33 34 35|./012345
<info> app:  36 37 38 39 3A 3B 3C 3D|6789:;<=
<info> app:  3E 3F 40 41 42 43 44 45|>?@ABCDE
<info> app:  46 47 48 49 4A 4B 4C 4D|FGHIJKLM
<info> app: Received packet number 150
<info> app: Content of packet 150:
<info> app:  00 95 00 01 02 03 04 05|........
<info> app:  06 07 08 09 0A 0B 0C 0D|........
<info> app:  0E 0F 10 11 12 13 14 15|........
<info> app:  16 17 18 19 1A 1B 1C 1D|........
<info> app:  1E 1F 20 21 22 23 24 25|.. !"#$%
<info> app:  26 27 28 29 2A 2B 2C 2D|&'()*+,-
<info> app:  2E 2F 30 31 32 33 34 35|./012345
<info> app:  36 37 38 39 3A 3B 3C 3D|6789:;<=
<info> app:  3E 3F 40 41 42 43 44 45|>?@ABCDE
<info> app:  46 47 48 49 4A 4B 4C 4D|FGHIJKLM
```

Figure 5.5: nRF52840 slave terminal output during the sequential packet transfer test.

All 1,050 SPI transfers initiated by the Zynq were completed successfully, with no transfer errors reported. The test was completed in approximately 0.598 seconds. The effective throughput, calculated using the total data transferred and the duration of the transfer cycle, was approximately 418.13 KiB/s. This value does not represent the SPI clock frequency itself but includes the entire transfer cycle, including inter-packet delays and software processing. The nRF52840 side output also confirmed that the packets were received and that the packet contents displayed at regular intervals matched the expected data pattern. These results confirmed that the SPI interface supported both full-duplex data transfer and sequential packet communication under the tested configuration.

5.2.3 CAN Interface

The CAN interface was functionally validated using the internal loopback mode of the CAN0 controller in the Zynq Processing System. The objective of this test was to verify that CAN frames could be transmitted by the software, processed by the controller, and correctly received back. In internal loopback mode, transmitted frames are routed internally to the controller's own receive path, so no external CAN node or physical CAN connection is required.

The `can0` interface was configured for operation at 1 Mbit/s with internal loopback enabled. This bitrate was selected for functional testing and was not a system-level design requirement. The interface was first brought down, configured with the required CAN parameters, and then brought up again.

A user-space application developed in C was used for the functional test. The application opened a raw CAN socket for `can0` through the Linux SocketCAN interface and transmitted a total of ten CAN frames. Each frame used a standard 11-bit CAN identifier and contained an 8-byte payload. During the test, frame identifiers were incremented from 0x001 to 0x00A, and a predefined data pattern was generated for each frame. Each transmitted frame was received back through the internal loopback path, and the received identifier, data length code, and payload were compared with the transmitted values. A timeout was also used on the receiving side to prevent the program from waiting indefinitely if a frame was not received.

All ten frames were successfully transmitted and received. The identifier, data length code, and payload values of the transmitted and received frames matched, and no mismatches were observed. The test therefore completed with a PASS result. These results confirmed the correct operation of the CAN controller and the frame transmission and reception performed through Linux SocketCAN. Because the test used internal loopback mode, it did not validate the physical CAN bus or an external CAN transceiver. The scope of this test was therefore limited to the functional verification of the CAN controller and its software interface, with the

CAN configuration and internal loopback test output shown in Figure 5.6.

```
my_project:/mnt/sd# ip link set can0 down
my_project:/mnt/sd# ip link set can0 type can bitrate 1000000 loopback on
xilinx_can e0008000.can can0: bitrate error 0.0%
my_project:/mnt/sd# ip link set can0 up
IPv6: ADDRCONF(NETDEV_CHANGE): can0: link becomes ready
my_project:/mnt/sd# chmod +x can_test
my_project:/mnt/sd# ./can_test
CAN internal loopback test on can0
Frames to send: 10, DLC: 8

[001] TX: ID=0x001 DLC=8 [ 00 01 02 03 04 05 06 07 ]
      RX: ID=0x001 DLC=8 [ 00 01 02 03 04 05 06 07 ]  MATCH
[002] TX: ID=0x002 DLC=8 [ 01 02 03 04 05 06 07 08 ]
      RX: ID=0x002 DLC=8 [ 01 02 03 04 05 06 07 08 ]  MATCH
[003] TX: ID=0x003 DLC=8 [ 02 03 04 05 06 07 08 09 ]
      RX: ID=0x003 DLC=8 [ 02 03 04 05 06 07 08 09 ]  MATCH
[004] TX: ID=0x004 DLC=8 [ 03 04 05 06 07 08 09 0A ]
      RX: ID=0x004 DLC=8 [ 03 04 05 06 07 08 09 0A ]  MATCH
[005] TX: ID=0x005 DLC=8 [ 04 05 06 07 08 09 0A 0B ]
      RX: ID=0x005 DLC=8 [ 04 05 06 07 08 09 0A 0B ]  MATCH
[006] TX: ID=0x006 DLC=8 [ 05 06 07 08 09 0A 0B 0C ]
      RX: ID=0x006 DLC=8 [ 05 06 07 08 09 0A 0B 0C ]  MATCH
[007] TX: ID=0x007 DLC=8 [ 06 07 08 09 0A 0B 0C 0D ]
      RX: ID=0x007 DLC=8 [ 06 07 08 09 0A 0B 0C 0D ]  MATCH
[008] TX: ID=0x008 DLC=8 [ 07 08 09 0A 0B 0C 0D 0E ]
      RX: ID=0x008 DLC=8 [ 07 08 09 0A 0B 0C 0D 0E ]  MATCH
[009] TX: ID=0x009 DLC=8 [ 08 09 0A 0B 0C 0D 0E 0F ]
      RX: ID=0x009 DLC=8 [ 08 09 0A 0B 0C 0D 0E 0F ]  MATCH
[010] TX: ID=0x00A DLC=8 [ 09 0A 0B 0C 0D 0E 0F 10 ]
      RX: ID=0x00A DLC=8 [ 09 0A 0B 0C 0D 0E 0F 10 ]  MATCH

--- Summary ---
Frames sent:        10 / 10
Frames received:    10 / 10
Mismatches:         0
Result: PASS
```

Figure 5.6: CAN interface configuration and internal loopback test results.

5.2.4 GPIO Interface

The GPIO interface was functionally validated using loopback tests between externally connected GPIO lines. The 32 GPIO lines used in the control unit were divided into two 16-bit groups for testing. The GPIO[15:0] lines were tested first, followed by the GPIO[31:16] lines.

Within the GPIO[15:0] group, the first eight lines were connected to the remaining eight lines using jumper wires. Specifically, GPIO[0]–GPIO[7] were connected to GPIO[8]–GPIO[15], respectively. A C user-space application developed for the test accessed the GPIO lines through the Linux GPIO interface to control their

direction and logic levels. In the initial phase, GPIO[7:0] were configured as outputs and GPIO[15:8] as inputs. The test patterns 0x00, 0xFF, 0xAA, 0x55, 0x0F, and 0xF0 were applied to the output group, and the values read from the input group were compared with the transmitted values. These patterns were selected to verify different logic level combinations. In particular, 0xAA and 0x55 tested alternating 0 and 1 levels on adjacent lines, while 0x0F and 0xF0 tested the lower and upper halves of the 8-bit group at different logic levels. The jumper connections and the fixed-pattern test results for this direction are shown in Figure 5.7.

Following the fixed pattern tests, a walking bit test was performed. In this test, a single bit was activated sequentially from 0x01 to 0x80. This allowed each GPIO line within the group to be driven individually and the corresponding line on the other side to be checked.

The input and output directions were then reversed, and the same tests were repeated with GPIO[15:8] configured as outputs and GPIO[7:0] as inputs. Testing both directions verified that the GPIO lines operated correctly as both inputs and outputs, rather than only confirming the physical connections. To prevent two connected lines from being driven as outputs at the same time during the direction change, both groups were first configured as inputs, after which only the required group was configured as an output. The corresponding fixed-pattern and walking bit test results after reversing the directions are shown in Figure 5.8.

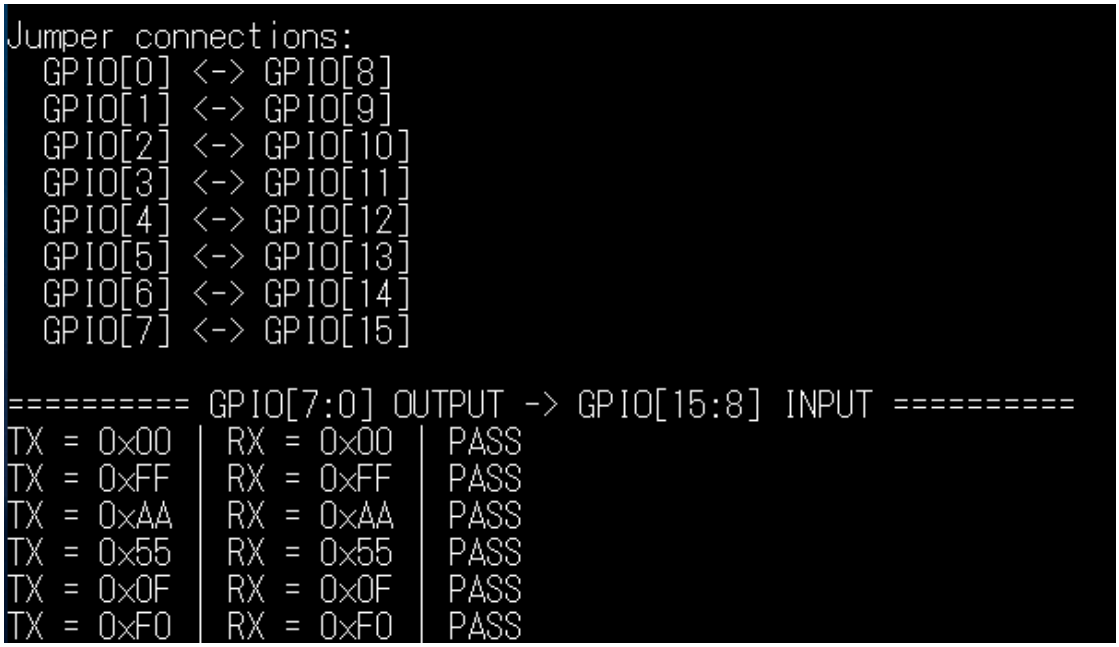


Figure 5.7: Jumper connections and fixed-pattern loopback test results with GPIO[7:0] configured as outputs and GPIO[15:8] as inputs.

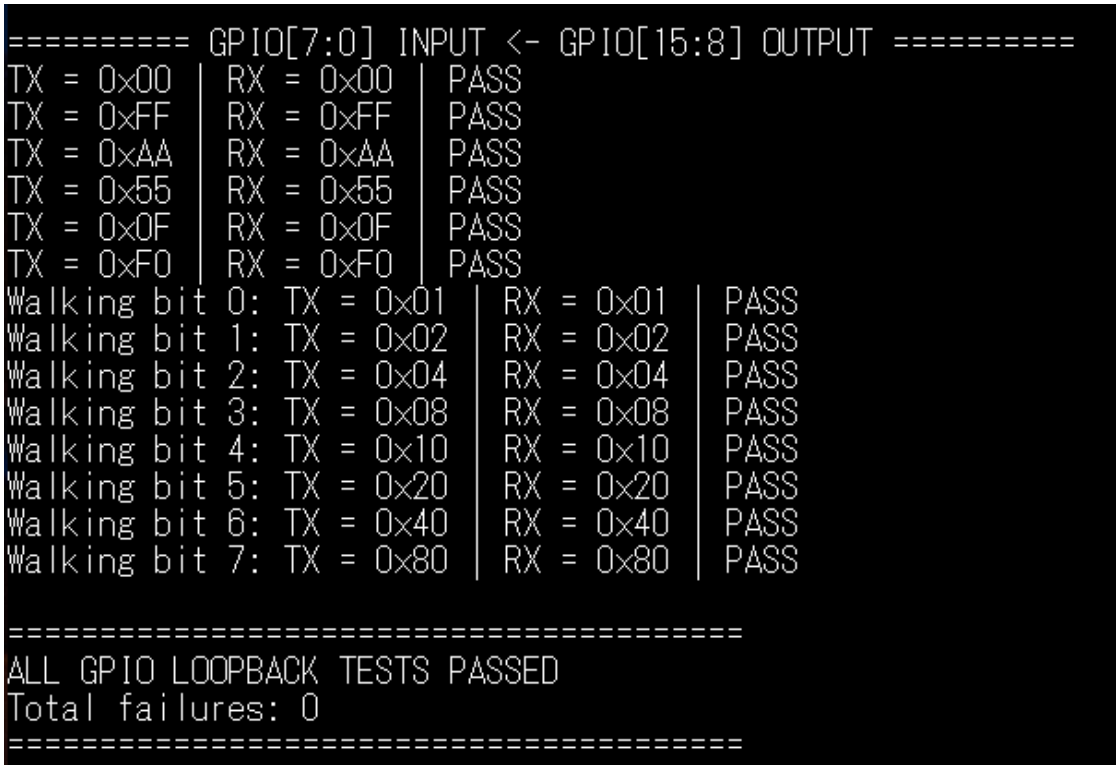


Figure 5.8: Fixed-pattern and walking bit loopback test results with GPIO[15:8] configured as outputs and GPIO[7:0] as inputs.

All transmitted and read values matched for both the fixed test patterns and the walking bit test. All checks produced a PASS result in both directions, and the reported error count was zero. The same test procedure was repeated for the GPIO[31:16] group. In this case, GPIO[16]–GPIO[23] were connected to GPIO[24]–GPIO[31], respectively, and the same patterns and walking bit test were applied in both directions. All transmitted and received values also matched for this group, and the test was completed without errors. These results confirmed that all 32 GPIO lines functioned correctly as both inputs and outputs in the tested loopback configuration and could be controlled through Linux.

5.3 Subsystem Validation

Subsystem Validation focused on testing the WPT, PMS, and Data Communication subsystems individually with the control unit. C user-space applications were employed to perform the control, monitoring, and communication functions of these subsystems; where necessary, control signals were also managed through the relevant Linux interfaces.

The WPT and PMS subsystem validation tests were performed on the initial AX7Z020B platform. The Data Communication subsystem was first tested on the AX7Z020B platform and was tested again after the control system was migrated to the AC7020C board. The hardware used for the subsystem validation tests is shown in Figure 5.9.

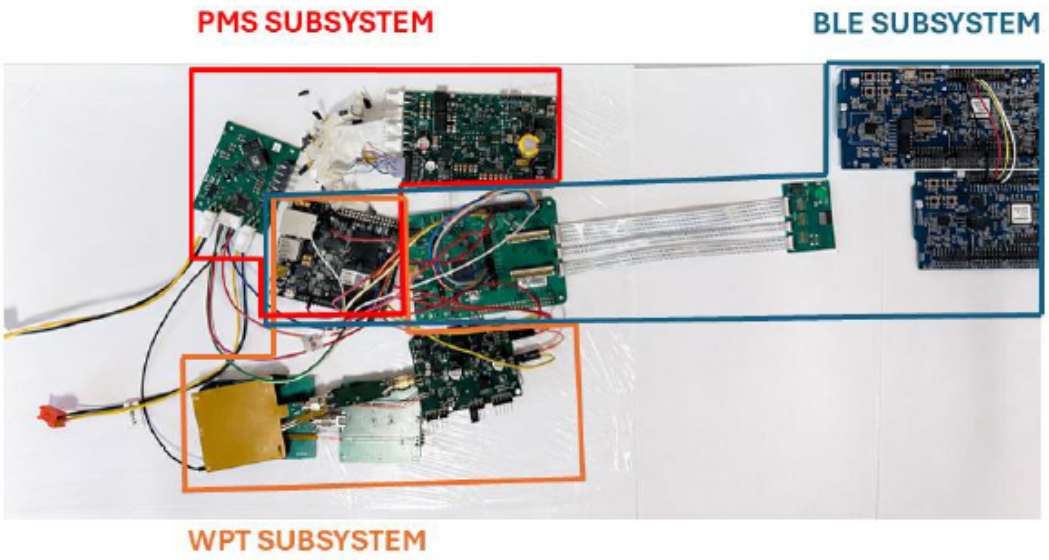


Figure 5.9: Overview of the control unit and subsystem hardware used in the WPT, PMS, and Data Communication validation tests.

5.3.1 Wireless Power Transfer Subsystem

The WPT subsystem was validated with the control unit connected to the WPT TX board. Based on the WPT function described in Section 2.1 and the control unit requirements defined in Section 2.2, the tests verified the control unit's ability to communicate with the onboard devices, adjust the supply voltage on the TX side, and read electrical and thermal measurement values during operation.

Three main components on the WPT TX board are used for control and monitoring by the control unit: the MP8859 DC-DC converter, which adjusts the TX supply voltage; the INA228 power monitor, which measures voltage, current, and power; and the ADS7128 analog-to-digital converter (ADC), which is used to monitor the TX coil temperature [18, 19, 20]. A negative temperature coefficient (NTC) thermistor located near the TX coil senses the coil temperature, while the ADS7128 reads the corresponding analog signal. These components have I2C addresses of 0x62, 0x40, and 0x14, respectively, and communicate with the control unit through the 400 kHz I2C interface defined for the WPT subsystem in

Section 2.2.

The WPT TX board was enabled before the test using Linux commands through a GPIO output of the control unit. While the TX board drove the TX coil, the RX coil was positioned coaxially at a distance of approximately 1 cm, and a 100 Ω load was connected to the RX side to evaluate the received power. The voltage across the RX load was measured using a multimeter. The test setup is shown in Figure 5.10. During the test, both the required 5 V main supply and the 3.3 V supply for I2C communication and monitoring components were provided by the Zynq platform.

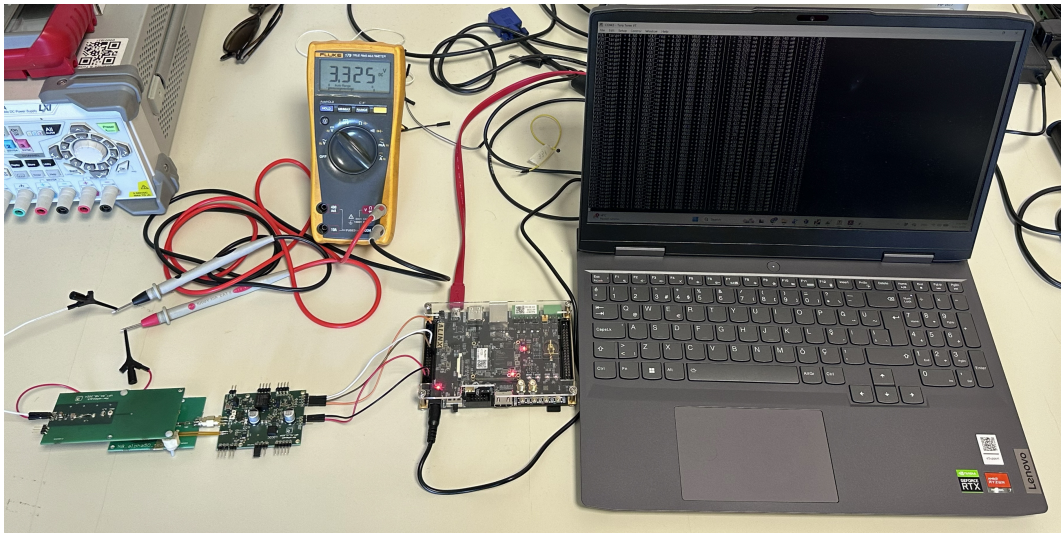


Figure 5.10: Experimental setup for WPT subsystem validation, including the control platform, WPT hardware, and RX voltage measurement.

A C user-space application running on Linux was used for the subsystem test. The application accessed the I2C devices on the WPT TX board through `/dev/i2c-0`. The program configured the MP8859 to set the converter output voltage on the TX side. Bus voltage, current, and power values were read from the INA228, while the TX coil temperature was calculated using the analog value obtained from the NTC sensor through the ADS7128. These values were displayed periodically on the terminal during the test. This allowed the required WPT operations to be performed within the same application.

During the validation test, an output voltage target of 4.50 V was set for the MP8859. The monitoring results obtained during the test are shown in Figure 5.11. The terminal output showed that the VOUT_target and VOUT_reg values both corresponded to 4.50 V. Readings from the INA228 indicated a bus voltage of approximately 4.523 V, a current of 79.4 mA, and a power of 359 mW. At the same time, the TX coil temperature, calculated from the NTC measurement acquired

through the ADS7128, was monitored at approximately 27.2 °C and remained at this level throughout the test.

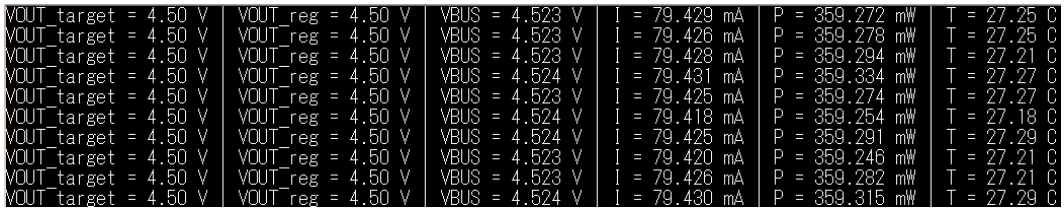


Figure 5.11: Real-time monitoring results during WPT subsystem validation.

In addition to the control and monitoring operations on the TX side, the RX load voltage was measured at approximately 3.33 V using a multimeter. Based on the 100 Ω load used on the RX side, the received power was calculated to be approximately 111 mW. Compared with the approximately 359 mW measured on the TX side, this corresponds to a wireless power transfer efficiency of approximately 31% at the tested operating point. These results confirmed that the control unit successfully performed the required control and monitoring functions for the WPT subsystem.

5.3.2 Power Management Subsystem

PMS validation focused on testing the control unit together with the components responsible for power management. Based on the role of the PMS described in Section 2.1, power monitoring and regulation operations, as well as battery management functions, were verified. The 1 MHz I2C interface defined in Section 2.2 was used for the monitoring and regulation components, while the 100 kHz SMBus was used for communication with the PMM240 and the smart battery.

Six peripherals are connected to the 1 MHz I2C bus on the PMS. Three INA228 devices monitor voltage and current values on the supercapacitor (0x41), motor/load (0x44), and battery (0x45) lines, respectively. The supercapacitor contributes to reducing peak current demand by supporting the battery, particularly during high power demands from the motors. The MP8859 (0x62) controls the output voltage of the DC-DC converter used for the motor supply, while a digital potentiometer at address 0x2E adjusts the boost converter output voltage and another at address 0x2C adjusts the supercapacitor charging current.

The interface between the PMS and the control unit also includes the control signals EN_CHARGE, EN_Switch, and EN_BOOST. These signals are used to enable the supercapacitor charging circuit, the power path switch, and the boost converter output, respectively. The battery and supercapacitor power paths are connected to the load through this structure.

The experimental setup used for PMS validation is shown in Figure 5.12. A C user-space application running on Linux was used for the tests. The application communicated with the three INA228 devices, the two digital potentiometers, and the MP8859 through `/dev/i2c-2`. Bus voltage and shunt voltage values were read from the INA228 devices, and the shunt voltage measurements were used to calculate the current on the supercapacitor, load, and battery lines. In the same application, the DC-DC converter output voltage, boost converter output voltage, and supercapacitor charging current could be adjusted using the `buck`, `boost`, and `chg` terminal commands, respectively.

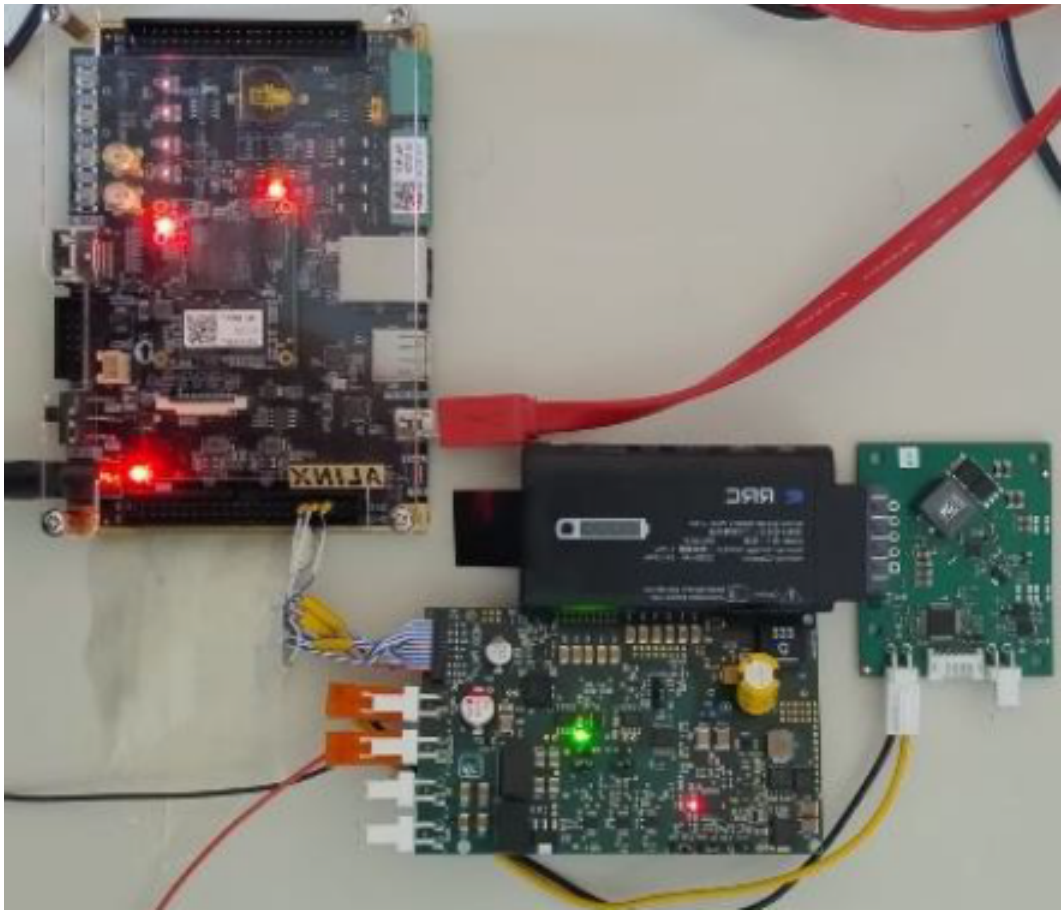


Figure 5.12: Experimental setup for PMS validation.

Measurement values were displayed periodically on the terminal during operation. In Figure 5.13, the supercapacitor, load, and battery lines are labeled SC, LOAD, and BATT, respectively. The results demonstrated that the sensing and control peripherals on the PMS could be accessed through the same I2C master and used for both monitoring and control.

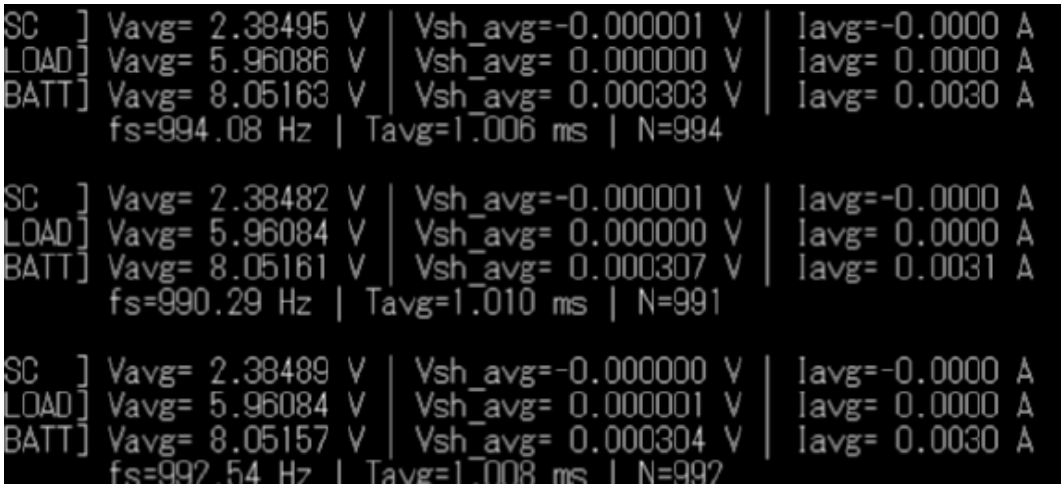


Figure 5.13: Real-time battery, load, and supercapacitor monitoring during PMS validation.

For battery management validation, a PMM240 smart battery interface (0x10) and an RRC2037 smart battery (0x0B) were used within the PMS. These two devices communicate with the control unit through a 100 kHz SMBus, which is separate from the 1 MHz I2C bus used for power monitoring and regulation. The PMM240 allows the charging parameters to be configured, while the battery current is read from the smart battery [7]. Two GPIO input lines are also used to provide battery/charging status and external power status information to the control unit.

A separate C user-space application was used for these tests. The application communicated with the PMM240 and the smart battery through `/dev/i2c-1` to configure the input current and charging current parameters and read the battery current. The same application periodically read the GPIO1 and GPIO2 inputs and displayed the battery/charging status, external power status, and battery current on the terminal.

The PMM240 validation setup is shown in Figure 5.14. The charging status was monitored through the 0.5 Hz blinking behavior of the GPIO1 status signal, and the same condition was confirmed by the light-emitting diode (LED) indicators on the battery. GPIO2 indicates the external power status; the signal was observed at logic high when the adapter was connected and changed to logic low when the adapter was disconnected [7].

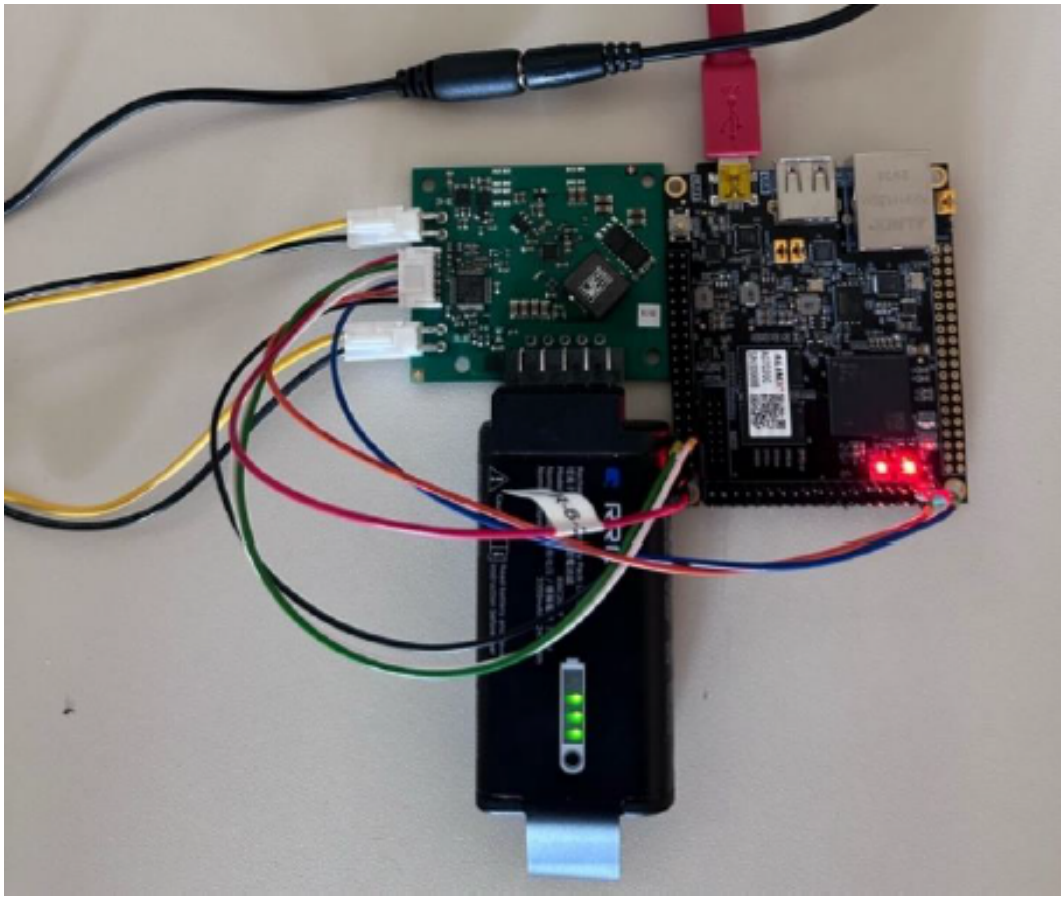


Figure 5.14: Experimental setup for smart battery validation.

The terminal output showing the real-time monitoring of battery charging and external power status is presented in Figure 5.15.

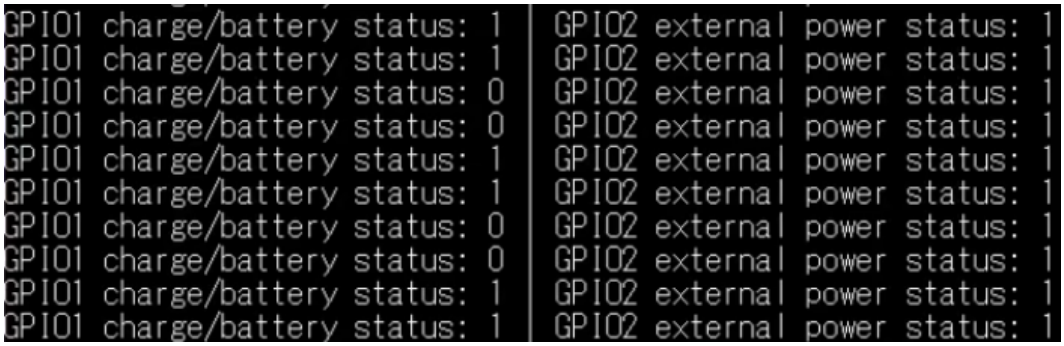


Figure 5.15: Real-time battery charging and external power status monitoring.

Charging current control was also verified as part of the same validation. A charging current value was set through the PMM240 and compared with the current read from the smart battery. The match between the set and read values confirmed that the charging current could be correctly configured by the control unit. The corresponding terminal output is shown in Figure 5.16.

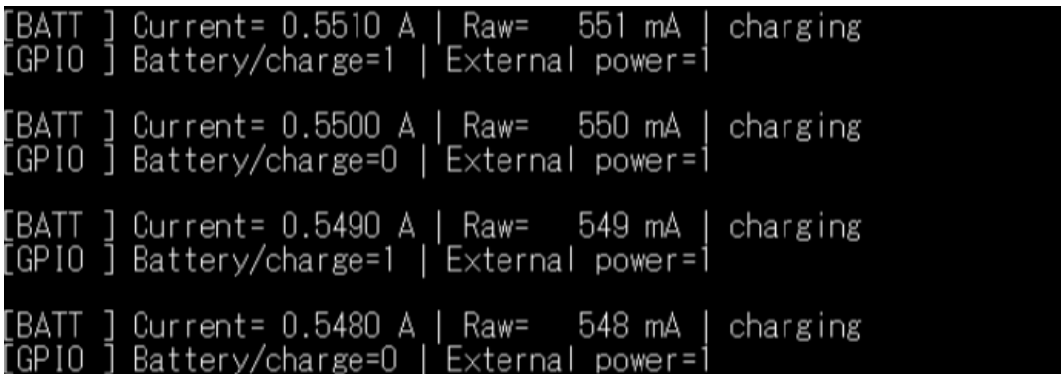


Figure 5.16: Charging current setting and monitoring during PMM240 validation.

5.3.3 Data Communication Subsystem

The operation of the control unit was evaluated as part of the bidirectional communication path between the implant and the external system. Based on the data communication function described in Section 2.1, the tests verified the transmission of data generated on the implant side to the external system and the transfer of response data in the opposite direction. The control unit used the 8 MHz SPI master interface defined in Section 2.2.

In the test setup, the nRF52840 development kit operated as the SPI master on the implant side and was used to generate and receive test data, while the nRF52832 operated as the SPI slave and Bluetooth Low Energy (BLE) Server. On the external side, the ISP1507 acted as the BLE Client and SPI slave and was connected to the control unit through SPI. Wireless communication between the nRF52832 and the ISP1507 was performed through BLE. Uplink data was transferred from the implant side to the external system, while downlink data followed the same communication path in the opposite direction. A 244-byte application payload was used in both directions.

A GPIO handshake signal between the ISP1507 and the control unit was used to coordinate downlink transmission. While uplink transmission continued, the ISP1507 activated the handshake signal when downlink data was required, thereby sending a request to the control unit. The control unit then transferred the response packets to the ISP1507 through SPI. The experimental setup is shown in Figure 5.17.

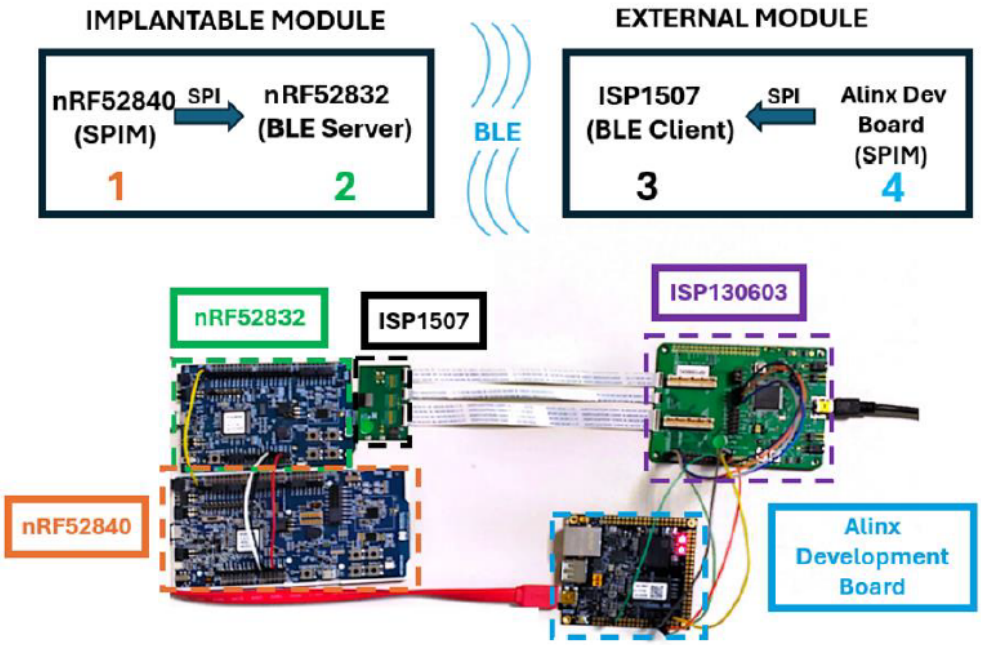


Figure 5.17: Experimental setup for bidirectional communication validation of the Data Communication subsystem.

Communication operations on the control unit side were performed using a C user-space application running on Linux. The application configured the SPI interface through `/dev/spidev0.0` to operate in Mode 0 with an 8-bit word length at 8 MHz and performed the required SPI transactions. The GPIO handshake was also handled by the same application. The handshake line was monitored as a GPIO input, and when the ISP1507 activated the signal to request a downlink transfer, the application initiated the corresponding SPI transaction. Test data were generated as an incrementing byte sequence and used for the transfers.

An Uplink Burst Trigger (UBT) mechanism was used to define how frequently downlink responses were requested. UBT represents the number of uplink packets received before a downlink response burst is triggered. The UBT value was set to 100 for the test. After every 100 uplink packets, the ISP1507 activated the GPIO handshake signal to request a downlink response from the control unit. A total of 1,000 uplink packets were transmitted, and each downlink response consisted of a four-packet burst. This resulted in 10 downlink bursts and 40 downlink packets, for a total of 1,040 packets during the test.

Uplink and downlink activity were monitored using an oscilloscope during the complete bidirectional transmission. In Figure 5.18, the yellow trace represents uplink activity, while the purple trace represents downlink activity. The total transmission duration was measured from the oscilloscope capture, and the results

yielded an uplink throughput of approximately 996 kbps, a downlink throughput of approximately 40 kbps, and a total throughput of approximately 1.04 Mbps. These results demonstrated that downlink responses could be transmitted while uplink communication was still active.

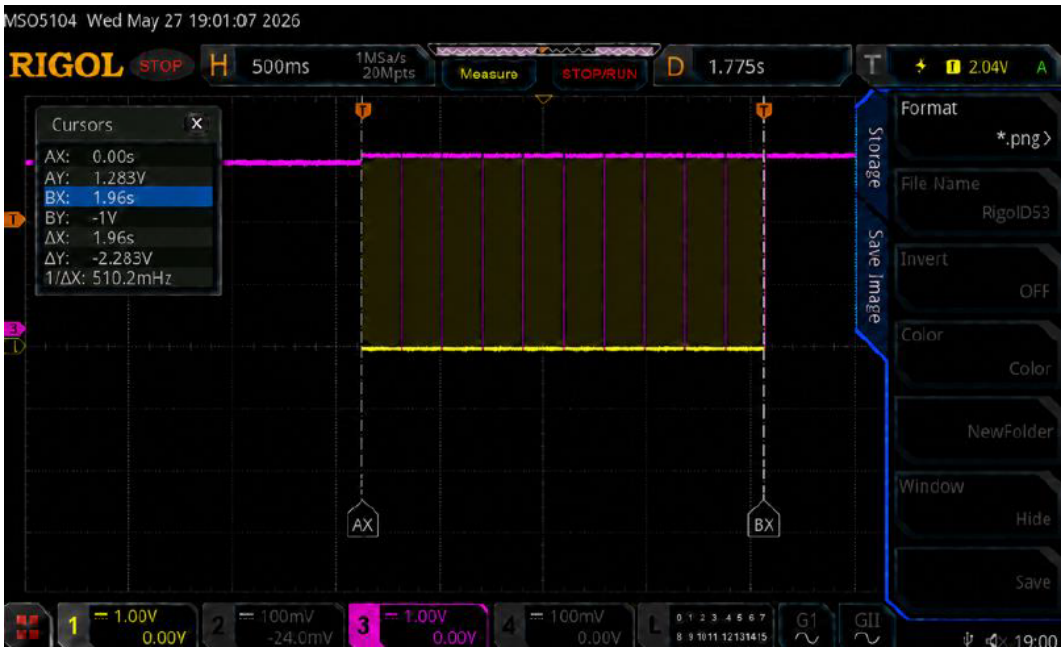


Figure 5.18: Oscilloscope capture of bidirectional data transmission during validation of the Data Communication subsystem.

In addition to transmission speed, packet integrity was also verified. The test data consisted of incrementing byte sequences, and the firmware checked the received payload content and packet order. This enabled the detection of missing, corrupted, or unexpected packet content. BLE traffic was also recorded using an nRF5340 sniffer, which was used to capture wireless packets without affecting the communication. The captured packets were then examined in Wireshark, a protocol analysis tool used to inspect packet contents and timing information. Payload bytes, packet direction, and timestamps were checked during this analysis.

Figures 5.19 and 5.20 show Wireshark captures of two consecutive packets. The first payload ends with 0x7F and the following payload begins with 0x80, showing that the byte sequence continued correctly across the packet boundary. This provided an independent verification of packet integrity in addition to the checks performed by the firmware.

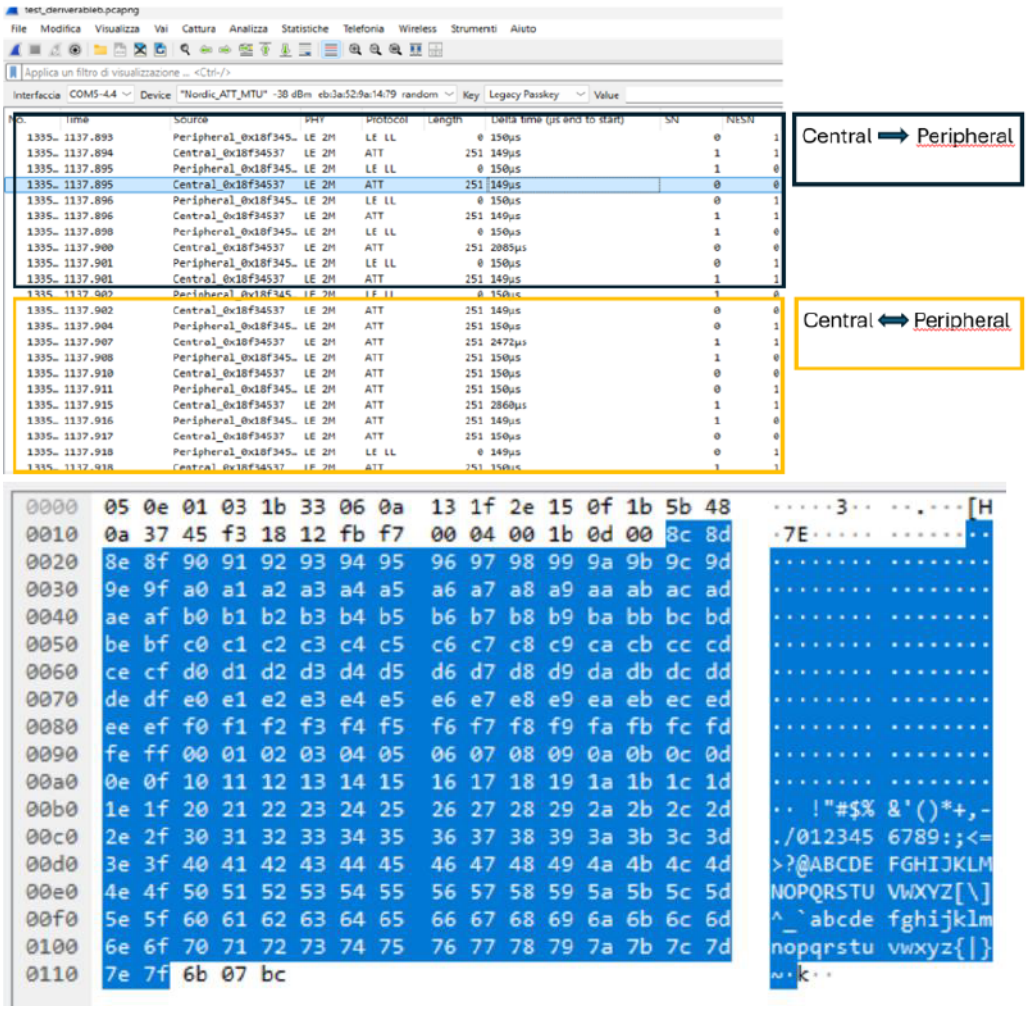


Figure 5.19: Wireshark capture showing the end of the first payload at 0x7F.

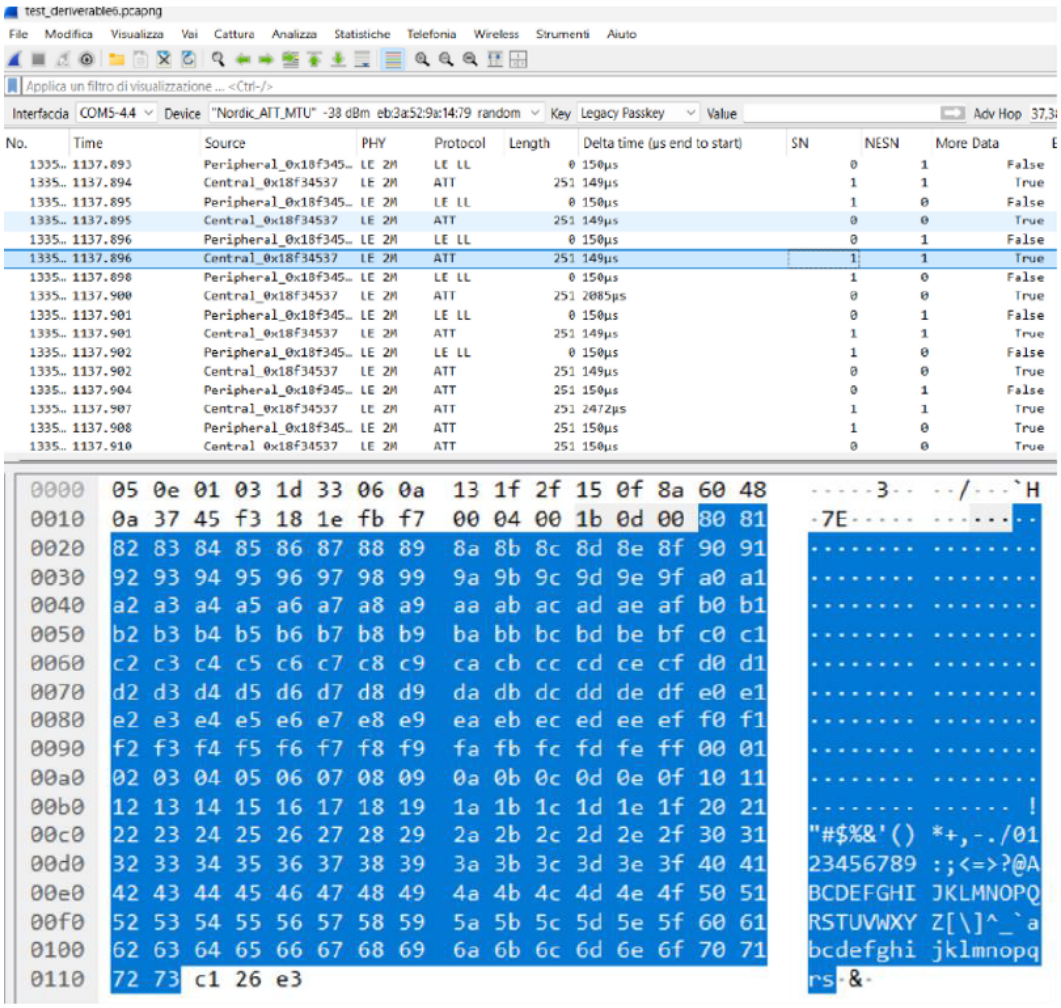


Figure 5.20: Wireshark capture showing the continuation of the byte sequence from 0x80 in the following payload.

These tests confirmed that the control unit operated as an SPI master within the Data Communication subsystem, handled downlink requests through the GPIO handshake signal, and supported bidirectional uplink and downlink communication across the complete communication path.

Chapter 6

Conclusion

The work presented in this thesis focused on the development and validation of a control unit utilizing the Zynq-7000 SoC for the control, monitoring, and testing of the external electronic subsystems of the NerveRepack system. The developed control unit integrates the required communication and control interfaces onto a common embedded platform.

The hardware architecture of the control unit was implemented using both the PS and PL sections of the Zynq-7000. While the necessary I2C, SPI, CAN, and GPIO interfaces were implemented within this structure, an additional I2C interface that could not be provided by the PS was created in the PL using the AXI IIC IP core. Alongside the hardware, a PetaLinux-based software environment was prepared to enable the use of these interfaces from Linux. Control, communication, and testing operations were performed on the same platform using Linux tools and user-space applications developed in C.

The developed system was validated in several stages. Initially, it was verified that the interfaces were recognized by Linux and that the required configurations were correctly applied. Subsequently, the functionality of the I2C and SPI interfaces was validated using external devices, while the CAN and GPIO interfaces were verified through loopback tests. Following the interface tests, the control unit was tested with the WPT, PMS, and Data Communication subsystems. These tests demonstrated that the control unit could perform its main functions, including power-related control and monitoring, smart battery communication, and bidirectional data transfer.

In a later stage of the development process, the control system was migrated to the more compact AC7020C platform, which uses the same Zynq-7020 SoC. Hardware connections and necessary system files were adapted to the new platform, and the operation of the control unit's relevant interfaces on the new board was verified. Overall, the developed system provides a common Linux-based control infrastructure for the configuration, monitoring, communication, and experimental

testing of NerveRepack's external subsystems.

6.1 Future Work

Several directions can be considered for the future development of the control platform:

- **Dedicated hardware functions:** The current design could be extended by adding dedicated functions in the PL. For example, neural data handled by the Data Communication subsystem could be processed after reception or before transmission. Similarly, some control functions for the WPT and PMS subsystems could be implemented in the PL using finite state machines (FSMs) with parameters configured from software. This would allow selected control and data processing tasks to be performed directly in the FPGA.
- **Custom carrier PCB:** A custom carrier PCB could be developed for the AC7Z020 core board according to the requirements of the NerveRepack control unit. The board could include only the communication and control interfaces, connections, and supporting components required by the system, allowing a more compact implementation. In this way, the current AC7020C development platform could be replaced by a smaller hardware platform designed specifically for the NerveRepack application.
- **Unified PetaLinux control software:** Separate C user-space applications were used for the control and validation of the different subsystems. In future work, the WPT, PMS, and Data Communication functions could be combined into a single PetaLinux application or a common software structure. Control, monitoring, and communication tasks could then be executed concurrently, allowing long-duration tests to evaluate system stability and possible communication or interface issues.
- **Final system integration and validation:** Once the mechatronic structures of the NerveRepack system are complete, the developed control unit could be integrated into the final exoprosthesis or exoskeleton platform together with the WPT, PMS, and Data Communication subsystems. Since the project aims for the electronic platform to operate together with these mechatronic structures and the implantable system, the existing validation tests could be repeated on the final system configuration. This would allow the control, monitoring, power management, and data communication functions of the developed platform to be validated while all system components are operating together.

Bibliography

[1] P. Geethanjali. «Myoelectric Control of Prosthetic Hands: State-of-the-Art Review». In: *Medical Devices: Evidence and Research* 9 (2016), pp. 247–255. DOI: 10.2147/MDER.S91102 (cit. on p. 1).

[2] J. W. Sensinger and S. Dosen. «A Review of Sensory Feedback in Upper-Limb Prostheses From the Perspective of Human Motor Control». In: *Frontiers in Neuroscience* 14, 345 (2020). DOI: 10.3389/fnins.2020.00345 (cit. on p. 1).

[3] S. Raspopovic, A. Cimolato, A. Panarese, F. Vallone, J. del Valle, S. Micera, and X. Navarro. «Neural Signal Recording and Processing in Somatic Neuroprosthetic Applications: A Review». In: *Journal of Neuroscience Methods* 337, 108653 (2020). DOI: 10.1016/j.jneumeth.2020.108653 (cit. on p. 1).

[4] European Commission. *Intelligent Neural System for Bidirectional Connection with Exoprostheses and Exoskeletons – NerveRepack*. Accessed: Sep. 9, 2026. CORDIS. URL: https://cordis.europa.eu/project/id/101112347/reporting (cit. on pp. 1, 2, 4, 5).

[5] *Zynq-7000 SoC Data Sheet: Overview*. Advanced Micro Devices, Inc. 2018. URL: https://docs.amd.com/v/u/en-US/ds190-Zynq-7000-Overview (cit. on pp. 2, 6–8, 14).

[6] *PetaLinux Tools Documentation: Reference Guide*. Advanced Micro Devices, Inc. 2023. URL: https://docs.amd.com/r/2023.1-English/ug1144-petalinux-tools-reference-guide (cit. on pp. 2, 10, 21, 24).

[7] *RRC-PMM240 Power Management Module for Mobile Applications*. RRC power solutions GmbH. 2022. URL: https://www.rrc-ps.com/fileadmin/Dokumente/PMM240_Specification_E.PDF (cit. on pp. 6, 42).

[8] *ISP1507 Serialization User Guide*. Insight SiP. 2019. URL: https://www.insightsip.com/fichiers_insightsip/pdf/ble/ISP1507/isp_ble_AN190401.pdf (cit. on p. 6).

[9] *Zynq-7000 SoC Technical Reference Manual*. Advanced Micro Devices, Inc. 2026. URL: https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM (cit. on pp. 8, 10, 24).

[10] *Vivado Design Suite: AXI Reference Guide*. Advanced Micro Devices, Inc. 2017. URL: https://docs.amd.com/v/u/en-US/ug1037-vivado-axi-reference-guide (cit. on p. 8).

[11] *I2C-bus Specification and User Manual*. NXP Semiconductors. 2021. URL: https://www.nxp.com/docs/en/user-guide/UM10204.pdf (cit. on p. 9).

[12] *System Management Bus (SMBus) Specification*. System Management Interface Forum, Inc. 2024. URL: https://smbus.org/specs/SMBus_3_3_1_20241020.pdf (cit. on p. 9).

[13] *AVR151: Setup and Use of the SPI on tinyAVR and megaAVR Devices*. Microchip Technology Inc. 2016. URL: https://www.microchip.com/en-us/application-notes/an2585-1 (cit. on p. 9).

[14] *TJA1057 High-Speed CAN Transceiver*. NXP Semiconductors. 2026. URL: https://www.nxp.com/docs/en/data-sheet/TJA1057.pdf (cit. on p. 10).

[15] *AC7Z020 User Manual*. Alinx Electronic Technology (Shanghai) Co., Ltd. 2019. URL: https://www.alinx.com/public/upload/file/AC7Z020_User_Manual.pdf (cit. on pp. 11, 12, 14).

[16] *AX7Z020B Development Board User Manual*. Alinx Electronic Technology (Shanghai) Co., Ltd. 2022. URL: https://www.en.alinx.com/upload/common/filesystem/fp/AX7Z020B%20V1.0%20User%20Manual.pdf (cit. on pp. 12, 15).

[17] *AC7010C/AC7020C User Manual*. Alinx Electronic Technology (Shanghai) Co., Ltd. 2022. URL: https://www.alinx.com/public/upload/file/AC7010C_AC7020C_User_Manual.pdf (cit. on pp. 13–15, 18).

[18] *MP8859 2.8–22V VIN, 3A IOUT, 4-Switch, Integrated Buck-Boost Converter with I2C Interface*. Monolithic Power Systems, Inc. 2018. URL: https://www.monolithicpower.com/en/documentview/productdocument/index/version/2/document_type/Datasheet/lang/en/sku/MP8859/document_id/4033/ (cit. on pp. 29, 38).

[19] *INA228 85-V, 20-Bit, Ultra-Precise Power/Energy/Charge Monitor With I2C Interface*. Texas Instruments. 2022. URL: https://www.ti.com/lit/ds/symlink/ina228.pdf (cit. on p. 38).

[20] *ADS7128 Small, 8-Channel, 12-Bit ADC With I2C Interface, GPIOs, and CRC*. Texas Instruments. 2020. URL: https://www.ti.com/lit/ds/symlink/ads7128.pdf (cit. on p. 38).