UFSC 2023 - Embedded Systems Programming in C++

Helmholtz Cage Driver

Ramon de Araujo Borba[1]

[1]UFSC - Universidade Federal de Santa Catarina

December 6, 2023

1 Introduction

The Helmholtz cage is composed of a set of coils, positioned concentrically along three orthogonal axis. By controlling the current flowing through the coisl, it is possible to genereate a uniform magnetig field in the center of the cage, in any desired direction.

Some of the applications of this equiment includes adjustment and calibration of magnetic sensors, testing of electronic devices, development and environmental simulations for nanosatellites, academic researches, scientific experiments and more.

In the Drakkar Atlantec Group calibration laboratory, a Helmholtz cage is used for adjustment and calibration of oceanography equipments, which includes magnetic sensors. It was observed by one of the owners of this company that the driver and software used in their laboratory for controlling their Helmholtz cage is becoming obsolete, and maintenance and repair of this driver is of great difficulty.

In this context, this work presents the design and implementation of a Helmholtz cage driver, capable of controlling Helmholtz coils in 3 axis, and adequate for use in the calibration laboratory of the Drakkar Atlantec Group.

The development of this project is part of the course EEL510265 - Embedded Systems Programming, the complete project requirements provided by professor Eduardo Augusto Bezerra can be found in his website[1] and in the project's GitHub repository's[2] Documentation Folder.

2 Application Requirements

For adequate use in the Drakkar Atlantec laboratory, the embedded controller must fulfill the following requirements:

- Provide currents of up to $1.5A$ in both directions to the coils of each axis.
- Have current monitoring capability.
- Allow control of the axes via serial communication with a host computer.

Besides the requirements for laboratory use, this controller will have the following features:

- Control and monitoring via wireless communication with a host smartphone.
- Store a log of operations and events.

The host computer and smartphone software have the following requirements:

- Allow user to control each axis individually.
- Read the event log from the embedded system.
- Allow user to visualize the log of events:
  - List all events between a given time interval.
  - Checking total active time of the embedded system between a given time interval.

3 Materials & Methods

The hardware for the embedded system will be comprised of a processing unit, voltage controlled current sources, helmholtz coils, current sensors and a real time clock.

The chosen processing unit is an ESP32 based development kit featuring the ESP32-WROOM-32[3] dual-core microcontroller module, with built-in Wi-Fi and Bluetooth capabilities, developed by Espressif [4]

The current sources were designed using operational apmlifiers and power transistors, and the ESP32 module controlls the current sources using its internal DAC or PWM signals. The current sources are designed to provide currents of up to $\pm 1.5A$.

The current sensors are based on INA219 [5] and ACS712 [6] ICs. The INA219 current sensor has a resolution of $0.8mA$ and the ACS712 current sensor has a sensibility of $66mV/A$.

The software for the entire system, embedded controller, host computer and host smartphone were developed using C++ with object oriented design.

For the embedded controller, the software was developed using Espressif's ESP-IDF [7] framework, which is composed of libraries and drivers for the microcontroller, made by the manufacturer.

4 Proposed design

The proposed system is composed of two parts, an embedded controller, for which both hardware and firmware were designed and implemented, and a host computer and smartphone software to interact with the embedded controller. A block diagram for the system is shown in Figure 1.

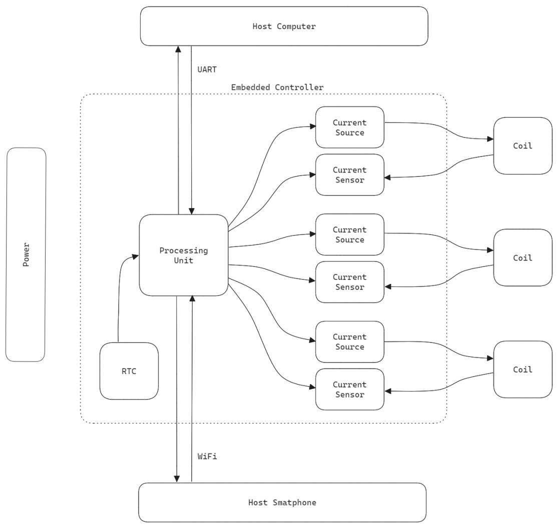


Figure 1: System block diagram

In the sections below, the different parts of the system are explained in more detail.

4.1 Embedded Hardware

The embedded controller is composed of a processing unit, three current sources, three current sensors and a real-time clock.

The processing unit chosen was an ESP32 based module [3], which has a dual-core 32-bit microcontroller with a clock frequency of up to $240MHz$, built-in Wi-Fi and Bluetooth capability, 4Mb of Flash memory and a wide variety of peripherals and sensors.

Due to limited availability, two types of current sensors were used, one INA219[5] based current sensor and two ASC712 [6] based current sensors. The INA219 current sensor can measure currents of up to $\pm 3.2A$ with a resolution of $0.8mA$, provides digital output through an I^2C interface. The ACS712 current sensor can measure currents of up to $\pm 30A$ with a sensibility of $66mV/A$ and porvides analog output, requiring the use of an ADC.

The real-time clock module is based on the DS3231 [8] chip, which has an integrated temperature compensated oscilator crystal for increased accuracy, integrated temperature sensor and programmable square wave output signal.

The voltage controlled current sources were designed using LM324 [9] operatoinal amplifiers and TIP41 and TIP42 [10] power transistors. The current sources are designed to provide currents of up to $\pm 1.5A$ and can be controlled by a voltage signal. The EPS32 has two integrated 8-bit DAC channels that are used to control two af the three current sources, the remaining one is controlled through a PWM output filtered by an passive RC filter. The schematic for the current source circuit is shown in Figure 3 of Appendix B.

4.2 Embedded Firmware

The firmware for the embedded system was implemented using the Espressif's ESP-IDF[7] official development framework and it's available tools. This framework runs a slightly modified version of FreeRTOS[11] natively, and allows the developer to use all of it's features. The FreeRTOS was modified by Espressif in order to allow it perform in ESP32's dual-core architecture, and manage both cores effectively, allowing for multithreaded development. Due to that, the embedded firmware was designed with the FreeRTOS workflow in mind, mainly through the use of tasks, to accomplish its goals. Details about the modifications can be found in the ESP-IDF Programming Guide[7].

Although the ESP-IDF framework's features and APIs being implemented in C, it is natively compatible with C++ development, due to the compilers being built upon GCC, it even provides example components and APIs implemented in C++ which are used in this application where possible.

The firmware must fulfillthe following tasks:

- Host computer communication via UART.
- Host smartphone communication via Wi-Fi/Bluetooth.

- Control each Helmholtz coil idependently through PWM or DAC.
- Read current sensors for each Helmholtz coil.
- Record a log of events.
- Transfer the event log to the hosts.

4.2.1 Abstraction Layers

The firmware will be organized in layers with different levels of abstraction. The highes layer will be responsible for the application logic and operation, with lower layers being responsible for external devices and basic functions of the microcontroller and its peripherals, astracting the manufacturer's C labraries using C++ classes.

The abstraction layers are divided as follows:

- Application layer.
- Components layer.
- Device Drivers layer.
- MCU Drivers layes.

The MCU Drivers layer is composed of classes responsible for interfacing with the ESP32's internal peripherals and functionalities. In this layer, most of the ESP-IDF's C APIs are encapsulated into C++ classes and, where possible, the ESP-IDF's own C++ APIs are employed.

The Device Drivers layer is composed of classes responsible for implementing drivers for the external devices used in the system, such as current sensors, current sources and real time clock, taking advantage of the abstractions of the internal drivers of the ESP32.

The Components layer is composed of classes responsible for implementing the interface between the application logic and the hardware devices. They provide the main functionalities needed by the application.

The Application layer, which is mainly composed of the Application class, is the highes level layes and is responsible for the logic of operation of the embedded controller and creation of the main application task.

4.2.2 Detailed Description

In order to fulfill the requirements for this application, and following the abstraction layers described above, several C++ classes were designed. This section porvides a basic description of each of the main classes responsible for the application logic. The class diagram for the embedded firmware is shown in Figure 2 of Appendix A.

The Application class represents the application as a hole and deals with the application logic, it is the entry point of the system firmware. This class is responsible for creating the main tasks of the system, and integrating the othe component classes to control the system's operation.

The Host, HostPC and HostMobile classes represent the hosts of the system, which will receive the log of events and send commands to de embedded controller. The Host class is an abstract class, providing methods for the Application class to interface with all of its hosts through polymorphism and virtual functions. The HostPC class inherits from the Host class and represents the host PC, which will be interfaced with via UART. The HostMobile class also inherits from the Host class and represents the host PC, which will be interfaced with via WiFi. Both hosts will be able to send commands for controlling the current sources individually, as well as request the transmisison of the log of events.

The Queue and Node template classes implemet a queue data structure. They are designed to be platform independant, flexible and reuseble to store any type of object as necessary. In the embedded controller, the main use for this templated classes is to implmement the log of events.

The Logger and LogData classes represent the system log of events. The LogData encapsulates all the information needed in a log entry, while the Logger class manages the storage of this information in a queue, to later be sent to the hosts.

The ClockCalendar class is responsible for storing the date and time of each entry in the event log, it inherits from both the Clock and the Calendar classes, which are responsible for dealing with time and dalendar dates respectively.

4.3 Host computer software

The host computer software was developed in C++ with an object oriented approach.

It was developed using G++13 compiler, with C++23 standard.

WSL was used to develop the software and for now it only supports Linux. It is being implemented as platform independant as possible, with portability in mind. This way, if it is to be used in a different platform, only a small amount od change will be needed.

4.3.1 Abstraction Layers

The host computer software is organized in a similar manner as the embedded firmware, with abstraction layers, except it has less layer since its not dealing directly with hardware.

The abstraction layers are divided as follows:

- Application layer.
- Libraries layer.

The application layer is where the logic of operation is implemented, being the highest level of abstraction in the software. It's mainly composed of a Application class.

The libraries layer is composed of several classes, organized as differen libraries, that implement specific functionalities used by the Application class to fulfill the requirements for the software.

4.3.2 Detailed Description

In order to fulfill the requirements for this application, and following the abstraction layers described above, several C++ classes were designed. This section porvides a basic description of each of the main classes responsible for the application logic.

The Application class is te entry point for the program and is responsible for starting the program's threads and sending commands to the embedded controller.

The SerialPort class is responsible for managing serial ports and is used to communicate with the embedded system. Here, an exception is thown to signal if the serial port was successfully opened upon construction of the SerialPort object.

Due to C++'s abstractions mechanisms, some of the classes implemented for the embedded controller were re-used for the host computer software, since they were implmented in a platform agnostic manner. The Logger, LogData, ClockCalendar, Queue and Node classes were taken from the embedded controller code base, with the only modifications made for adding functionality.

The ability to compare dates throught the ClockCalendar classes was implemented through operator overloading and friends. This functionality was used to compare and select the entries in the event log.

4.4 Tests

A test plan was implemented and is present at Appendix C.

Conclusions

The system was implemented at a proof of concept level, using all of the concepts studied in the EEL510265 course.

This project will serve as a stating point for a future implementation of an equipment fully capable of replacing the existing technology used in the Drakkar Atlantec Group's calibration laboratory.

References

[1] E. A. Bezerra. Project specifications. [Online]. Available: https://gse.ufsc.br/bezerra/wp-content/uploads/2022/02/Especificacao_2021_2-v2.pdf

[2] R. de Araujo Borba. Project's github repository. [Online]. Available: https://github.com/ramonborba/helmholtz-cage-driver

[3] Espressif. Esp32-wroom-32 datasheet. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf

[4] ——. About espressif. [Online]. Available: https://www.espressif.com/en/company/about-espressif

[5] T. Instruments. Ina219 datasheet. [Online]. Available: https://www.ti.com/lit/ds/symlink/ina219.pdf?ts=1699430728861&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252Fes-mx%252FINA219

[6] A. Microsystems. Adc712 datasheet. [Online]. Available: https://www.allegromicro.com/-/media/files/datasheets/acs712-datasheet.pdf

[7] Espressif. Esp-idf programming guide. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/index.html

[8] A. Devices. Ds3231 datasheet. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/ds3231.pdf

[9] T. Instruments. Lm324n datasheet. [Online]. Available: https://www.ti.com/lit/ds/symlink/lm324-n.pdf?ts=1699442755539&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FLM324-N%253Futm_source%253Dgoogle%2526utm_medium%253Dcpc%2526utm_campaign%253Dasc-null-null-GPN_EN-cpc-pf-google-eu%2526utm_content%253DLM324N%2526ds_k%253DLM324-N%2BDatasheet%

2526DCM%253Dyes%2526gclid%253DEAIaIQobChMIhtWRoqW0ggMVaYxoCR0_gADEEAAYASAAEgLnMPD_BwE%2526gclsrc%253Daw.ds

[10] O. Semiconductors. Tip41 and tip42 datasheet. [Online]. Available: https://www.onsemi.com/download/data-sheet/pdf/tip41a-d.pdf

[11] Freertos website. [Online]. Available: https://www.freertos.org/

Appendices

A Class Diagrams

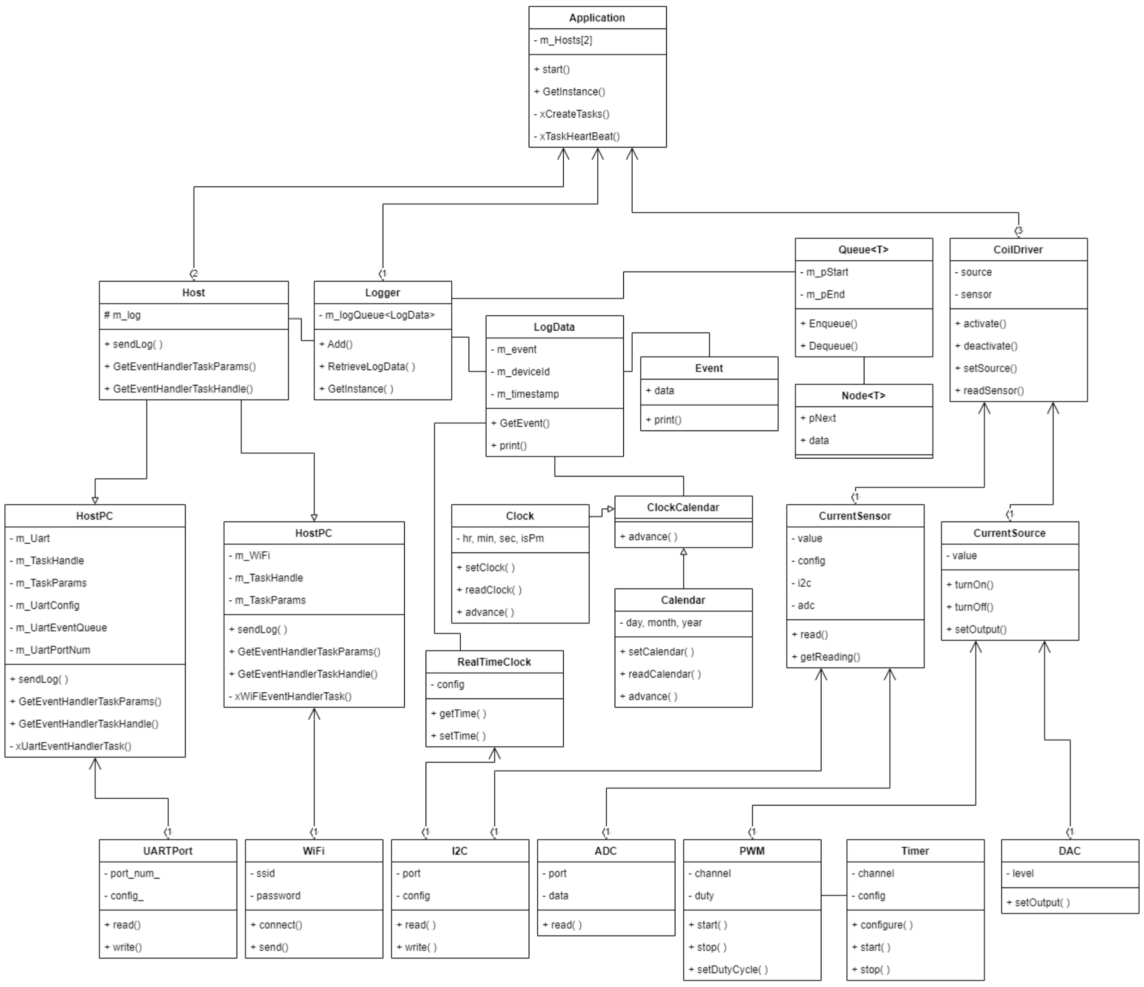
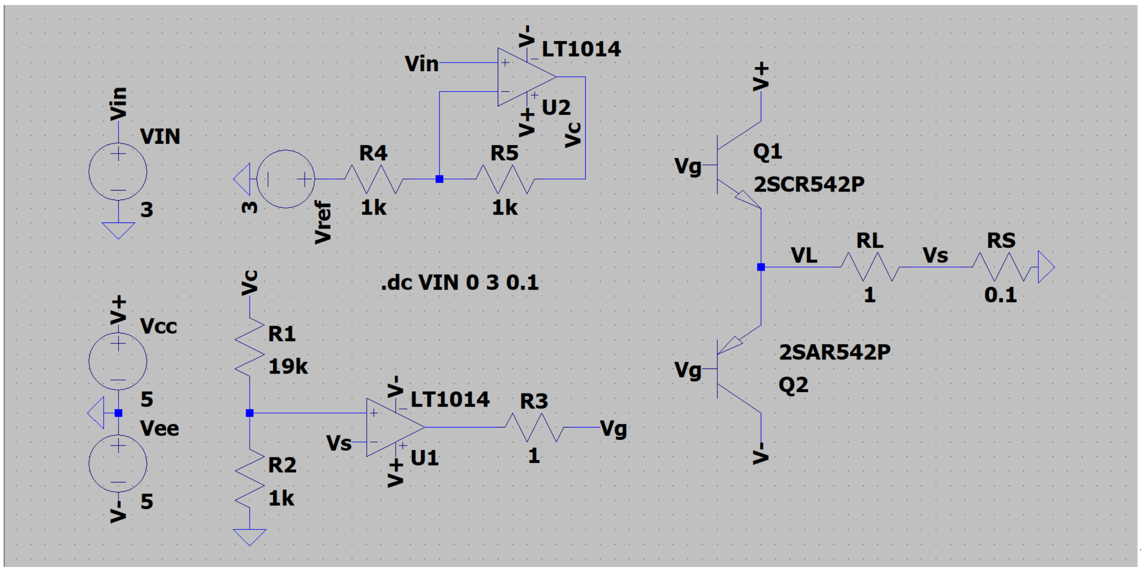


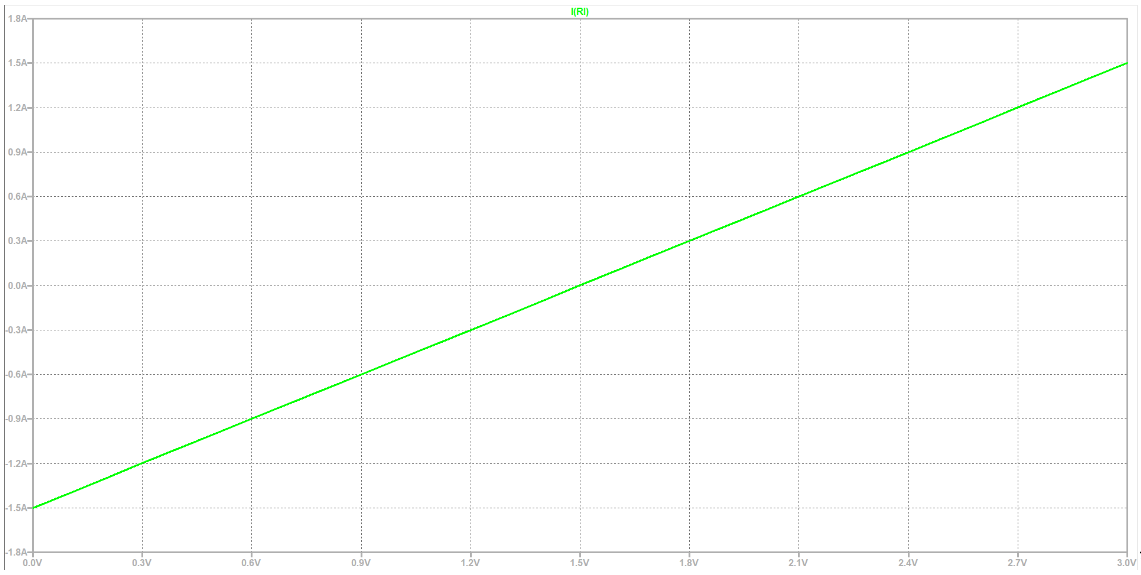
Figure 2: Embedded firmware class diagram

B Schemaitcs & Simulations



,

Figure 3: Voltage controlled current source schematic



,

Figure 4: Voltage controlled current source output current simulation

C Test Plan

C.1 Required Tools

To perform the tests described in the next section, the following intruments and software tools are necessary:

- Required software tools:
  - PC with a Linux OS (Ubuntu preferably)
  - ESP-IDF v5.1.2
  - g++ 13.1.0

  - A terminal emulator (screen or minicom for example)

- Required hardware and instruments:
  - Voltimeter
  - Amperimeter
  - USB/UART converter
  - ESP32 development board

C.2 HostPC Software Tests

The following tests verify that the host PC software is operating correctly.

C.2.1 Test 1

Verify that the host PC software can send commands through a serial port.

Setup

- Clone the project's GitHub repository [2]
- Compile the host PC software
- Connect the ESP32 to the PC with the appropriate USB cable
- Load the uart_events example from ESP-IDF onto the ESP32

Steps

- Open a terminal window and run the ESP-IDF serial monitor on the ESP32 serial port
- Execute the host PC software configured for the USB/UART converter serial port
- Select option 1 to set the X axis current
- Verify on the ESP32's logs that the correct command was sent
- Select option 2 to set the Y axis current
- Verify on the ESP32's logs that the correct command was sent
- Select option 3 to set the Z axis current
- Verify on the ESP32's logs that the correct command was sent
- Select option 4 to request transmission of the event log from the embedded controller
- Verify on the ESP32's logs that the correct command was sent

Expected Results

- The correct commands are being sent from the host PC.

C.3 Embedded Controller Tests

The following tests verify that the embedded controller is operating correctly.

C.3.1 Test 1

Verify that the embedded controller can read commands from serial port.

Setup

- Assert that the host PC software tests were executed and the expected results were obtained
- Clone the project's GitHub repository [2]
- Compile the host PC software
- Connect the embedded controller to the PC with the appropriate USB cable
- Connect the USB/UART converter to the embedded controller's command serial port and the host PC.
- Compile and load the firmware of the embedded controller using ESP-IDF's tools

Steps

- Open a terminal window and run the ESP-IDF serial monitor on the embedded controller's debug serial port
- Execute the host PC software configured for the USB/UART converter serial port
- Select option 1 to set the X axis current
- Verify on the embedded controller's logs that the correct command was received
- Select option 2 to set the Y axis current
- Verify on the embedded controller's logs that the correct command was received
- Select option 3 to set the Z axis current
- Verify on the embedded controller's logs that the correct command was received
- Select option 4 to request transmission of the event log from the embedded controller
- Verify on the embedded controller's logs that the correct command was received

Expected Results

- The correct commands are being received by the embedded controller.