POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Thesis

System Design and Evaluation of Multi-Modal Magnetic Tactile Sensor for Robotic Grasping

Supervisors
Prof. Marcello Chiaberge
Oliver Neumann

Candidate
Kaliroi Mignone

December 2024

Summary

In robotic manipulation and grasping tasks, visual sensing is commonly used to gather information about external object properties, including position, shape, and orientation. However, in unstructured environments where occlusions and low visibility may occur, visual information alone is insufficient for providing the necessary tactile feedback. In contrast, tactile sensors offer critical information regarding an object's pose, texture, and force, thereby enhancing a robotic system's ability to grasp and manipulate objects with greater precision. These sensors enable robots to perform tasks such as delicate grasping, shape recognition, and texture detection.

This thesis work, developed at the Institute of Robotics and Mechatronics (RMC) of the German Aerospace Center (DLR), addresses the pose estimation problem of both the fingertips of robotic grippers, as well as the objects being grasped. It emphasizes the role of tactile sensors in object manipulation and grasping operations, particularly in scenarios where visual data alone may be inadequate. Among the tactile sensing technologies developed in recent years—such as capacitive, piezoresistive, and optical sensors—Hall-effect-based tactile sensors are notable for their multi-directional sensing capabilities, relatively low cost, and ease of fabrication.

To achieve effective pose estimation for robotic grasping applications, this work explores the firmware design and evaluation of a magnetic-based multi-modal tactile sensor specifically developed by the Institute of Robotics and Mechatronics at DLR. The proposed tactile sensor consists of four Hall-effect sensors and a 9-degree-of-freedom (DoF) inertial measurement unit (IMU). Together, these sensors facilitate responsive measurement of touch-related force vectors while supporting orientation tracking through the fusion of IMU sensor data. It is crucial to overcome magnetic interference and drift effects that can distort readings, which are important considerations in sensor design and calibration.

For reliable real-time operation and communication within robotic systems, this work utilizes the Zephyr Real-Time Operating System (RTOS). Since Zephyr's framework lacked built-in sensor drivers for the Hall-effect and IMU sensors, custom drivers were developed in C based on detailed sensor datasheet specifications. This development facilitated sensor management and signal processing within the Zephyr RTOS.

The thesis proceeds by analyzing the Hall sensor data independently of the IMU data, conducting separate signal analyses. Calibration and sensor fusion algorithms for the IMU sensors (accelerometer, magnetometer, and gyroscope) allow for the derivation of quaternions or Euler angles, providing precise orientation of objects in contact with the tactile sensor. Analyses and evaluations of both the tactile and IMU data were performed, yielding a comprehensive overview of the overall functionalities of the tactile sensor.

Through this work, a novel tactile sensing approach is proposed that integrates multi-modal force and orientation feedback. This approach enhances the feasibility of pose estimation for robotic grasping, with potential applications in unstructured environments where traditional vision-based systems may not be suitable.

Acknowledgements

This work is supported by the European Union's Horizon Europe research and innovation framework under grant agreement No. 101070136, project IntelliMan.

Contents

List of Tables

List of Figures

Acronyms

ADC Analog to Digital Converter. 32, 33

AHRS Attitude and Heading Reference System. 53

API Application Programming Interface. 23, 24, 27, 29, 30, 43

CAN Controller Area Network. 17, 26, 71, 74

CSV Comma Separated Value. 55

DLR Deutsches Zentrum für Luft- und Raumfahrt (German Aerospace Center). 12, 15, 47, 48, 51

dof Degrees Of Freedom. 53, 59

EKF Extended Kalman Filter. 14

I^{2}C Inter-Integrated Circuit. 8, 12, 15, 17, 23, 27, 30, 35, 37, 38, 40, 42–45, 53, 69, 70, 74

IC Integrated Circuit. 15

IDE Integrated Development Environment. 17

IMU Inertial Measurement Unit. 8, 12, 15, 26, 27, 41, 47–49, 51, 53, 59, 63, 66, 67

IoT Internet of Things. 17

JTAG Joint Test Action Group. 72

LSB Least Significant Bit. 34

LSTM Long Short-Term Memory. 14

PCB Printed Circuit Board. 15, 16, 48

RAM Random Access Memory. 17

RTOS Real-Time Operating System. 12, 19, 20, 26, 27, 29, 37–39, 43, 44, 59, 66, 67

SB Start Burst. 36

SM Start Single Measurement. 36

SPI Serial Peripheral Interface. 17, 23, 74

UART Universal Asynchronous Receiver-Transmitter. 39

URL Uniform Resource Locator. 20, 21

USB Universal Serial Bus. 40, 71

Chapter 1

Introduction

1.1 Motivations

Robotic manipulation in unstructured environments often faces challenges due to the limitations of vision-based systems, particularly under conditions of occlusion or poor visibility. To address these limitations, tactile sensing has emerged as a critical complement, providing information about an object's pose, texture, and applied forces that enhance precision in grasping and manipulation tasks.

Motivated by the need to improve pose estimation in robotic grasping, this thesis, conducted at the Institute of Robotics and Mechatronics (RMC) of the German Aerospace Center (DLR), focuses on designing and evaluating a magnetic-based multi-modal tactile sensor. The sensor integrates Hall-effect sensors with a 9-degree-of-freedom inertial measurement unit (IMU) to deliver real-time force and orientation feedback, addressing key challenges such as magnetic interference and sensor drift through calibration and fusion algorithms.

Unlike previous DLR approaches to pose estimation, which relied on prerequisites such as inaccurate joint position and torque measurements, rigid assumptions about object geometry, and initial pose information often obtained through vision, this work aims to reduce these dependencies. The IMU-based tactile sensor can independently determine the initial pose of a grasped object, eliminating the need for vision-based initialization. Additionally, by placing an IMU directly on the robot's links, the system improves the precision of joint angle estimation, offering a more robust alternative to existing methods.

Custom drivers for these sensors were developed in C within the Zephyr Real-Time Operating System (RTOS), enabling reliable real-time communication and processing. This novel approach enhances the feasibility of tactile-based pose estimation in scenarios where visual feedback alone is insufficient, paving the way for more adaptive and effective robotic systems in complex environments.

1.2 Chapters organization

This section provides a brief description of the content of each chapter composing the thesis, thus providing an overview on the thesis work content.

Chapter 3 is about the DLR multi-modal tactile sensor. It includes sections about the general description of the sensor and the boards and Inter-Integrated Circuit (I^2C) communication protocol.

Chapter 4 is about the firmware development. In particular, the Zephyr RTOS basics and the debugging tools used are described. The following section is about the driver development first of the MLX90395 Hall sensor, and then of the ICM20948 Inertial Measurement Unit (IMU).

Chapter 5 is about the experimental evaluation of the sensors. First, the Hall sensor characterization is described; then, the IMU sensor calibration and comparison with benchmark sensors and libraries, is analyzed.

Chapter 6 describes the need of a sensor fusion procedure to effectively use IMU data. Therefore, the chosen sensor fusion algorithm and its implementation is described.

Chapter 7 shows the conclusions and the possible future work.

Chapter 8 contains the appendix showing the debugging tools used in the firmware development.

Chapter 2

State of the Art

This chapter reviews the current advancements and methodologies in tactile sensing, focusing on the technologies and approaches relevant to robotic manipulation and grasping. The covered topics are tactile sensing, multi-modal tactile sensing, and magnetic-based sensors in robotics, and the final paragraph will be about methodologies used for pose estimation by using tactile sensing.

2.1 Tactile Sensing in Robotics

Tactile sensing plays a central role in robotic applications, enabling robots to interact with their environment with dexterity and precision. Modern tactile sensors are designed to capture a wide range of physical interactions, such as force, texture, and curvature, providing critical feedback for grasping and manipulation tasks. Various sensor technologies, including piezoresistive, capacitive, optical, and multimodal sensors, have been developed, each offering distinct advantages in sensitivity, robustness, and cost-effectiveness. However, challenges such as non-linear responses, environmental susceptibility, and integration complexities remain a focus of ongoing research ([1], [2], [3], [4], [5]). The applications are robust grasping, slip detection, and object recognition ([1], [2], [3], [5], [6]). In soft robotics, tactile sensors enhance the capabilities of soft grippers, enabling slip detection, bend sensing, and electronics-free tactile feedback [4]. In addition, tactile sensing is being leveraged in niche sectors like agri-food robotics for tasks such as fruit firmness evaluation and texture-based object classification, demonstrating its versatility and transformative potential [7]. These advancements position tactile sensing as a cornerstone of modern robotic systems, driving innovation in pose estimation for grasping and beyond.

2.2 Multi-modal Tactile sensing

Multi-modal tactile sensors have revolutionized robotic grasping by integrating diverse sensing modalities to enhance object interaction and manipulation. These systems combine sensors capable of measuring forces, detecting motion, and sensing proximity, enabling precise object recognition, manipulation, and grasp stability prediction ([8], [9]).

Advanced models align tactile data with other modalities, such as vision and language, to achieve zero-shot learning and broaden application possibilities [10]. Additionally, fusion techniques that integrate motion and environmental sensing improve spatial resolution, force estimation, and interaction analysis [11]. While challenges like computational complexity remain, these sensors enable real-time insights, adaptability, and robust control in dynamic environments, making them indispensable for modern robotic systems.

2.3 Magnetic Tactile Sensing

Magnetic-based tactile sensors are a versatile and robust solution for robotic applications, offering high sensitivity, compactness, and environmental resilience. These sensors combine a magnetic field source, magnetic field sensors (e.g., Hall effect sensors), and a deformable medium to measure forces, pressures, and, in some cases, contact locations and shapes ([12], [13]). Their ability to deliver multi-directional force measurements and their immunity to environmental factors such as temperature and humidity make them reliable for dynamic applications [12]. In robotics, magnetic tactile sensors are widely used in grasping and manipulation tasks due to their flexibility and ease of integration into robotic grippers. Advanced designs utilizing sensor arrays and deformable media enable precise contact location detection, enhancing pose estimation and adaptive grasping ([13], [14]). While challenges such as susceptibility to stray magnetic fields and limited spatial resolution remain, innovations like magnetic shielding and optimized sensor designs continue to improve their performance ([13], [14]). Magnetic tactile sensors also excel in multi-modal systems, complementing other sensing modalities with their high spatial resolution and robust force measurement capabilities. This makes them highly effective for tasks requiring precise pose estimation and grasping in robotic applications.

2.4 Pose Estimation using Tactile Sensing

Tactile sensors have become integral to improving pose estimation in robotic grasping by leveraging diverse sensing modalities and computational techniques. Early methods refined pose estimates using contact location and force data, employing optimization algorithms like evolutionary strategies to minimize discrepancies between measured and estimated contact information [15]. Temporal tactile data has also been utilized with neural networks, such as those with Long Short-Term Memory (LSTM) layers, to enhance orientation estimation by integrating inertial and magnetic field data [16]. Particle filter-based approaches combine tactile and kinematic data for real-time pose tracking, while Extended Kalman Filter (EKF) have been used to estimate pose and contact forces with minimal or no specialized tactile sensors ([17], [18]). Bayesian methods efficiently estimate six-degree-of-freedom poses under high uncertainty [19]. Integrating multi-modal tactile data, including force/torque, inertial, and magnetic field sensing, has proven effective in addressing challenges like contact localization ambiguity and dynamic object interactions. By combining these modalities, modern systems achieve robust and accurate pose estimation for complex and dynamic grasping tasks.

Chapter 3

DLR Multi-Modal Magnetic Tactile Sensor

This chapter is devoted to the description of the sensor's general characteristics. First, the layout of the sensor and the main characteristics of the single sensors composing the system are presented. Then, the physical working principle is briefly assessed. Lastly, the boards used for the thesis work are briefly described.

3.1 General Description

The DLR tactile sensor is a multi-modal tactile sensor that exploits the modalities relative to the Hall sensors (MLX90395), and the one relative to the IMU sensor (ICM20948). The Hall sensors are needed to measure the applied force in different directions, while the IMU is needed to obtain the orientation of the tactile sensor. These modalities together enable the sensor to provide information on the pose of the fingertips on which the sensor is placed and of the grasped objects.

- MLX90395 Hall sensor from Melexis is a miniature monolithic sensor Integrated Circuit (IC) sensitive to the three orthogonal components of the magnetic flux density applied to the IC (i.e. B_x, B_y, and B_z).
- ICM20948 IMU from TDK Invensense is a low-power 9-axes MotionTracking device. The integrated sensors needed for this thesis' purposes are a 3-axis gyroscope, 3-axis accelerometer, 3-axis compass (AK09916).

The following chapters will present more details about the characteristics of those sensors.

The tactile sensor is made of a small Printed Circuit Board (PCB). On top of that the four MLX90395 Hall sensors which form the taxels of the tactile sensor are soldered. On the bottom side of the same PCB, the ICM20948 IMU is soldered, together with the spots for the cables needed for the I^{2}C communication. To obtain the final sensor, four permanent magnets with 1 millimeter of side length, are placed at a height of approximately one millimeter, on top of each Hall sensor. These magnets are then fused into a piece of silicon through a molding process. The chosen silicon has a shore hardness of 30. This parameter will have a consequence on the sensor's sensitivity.

Figure 3.1 shows a 3D rendering of the tactile sensor system.

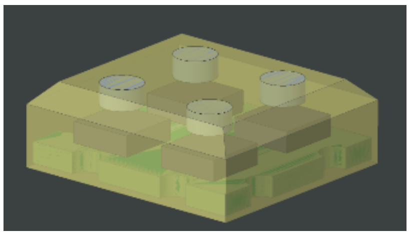
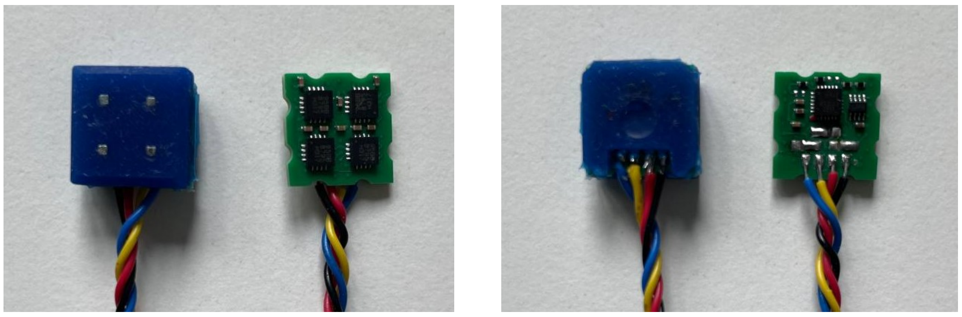


Figure 3.1: DLR Tactile Sensor Rendering

The green surface on the bottom represents the PCB, the black parallelepipeds are the four Hall sensors, the gray cylinders are the cubic permanent magnets, and the yellow layer is the silicon molded around the components of the sensor system.

A picture of the real sensor is shown in 3.2. On the left, the sensor is visible from the front side (3.2a), while on the right the view is from the back (3.2b). Both the pictures show the sensor with the silicon and the magnets on top of the Hall sensors, on the left, and the sensor without the silicon and magnets, on the right.



(a) Tactile Sensor - Front (b) Tactile Sensor - Back

Figure 3.2: DLR Tactile Sensor

Working principle

As external forces are applied to the elastomer, this experiences a deformation or a strain. With the magnet embedded inside the elastomer, these strains create a change in magnetic field which can be measured by the Hall effect sensors.

The change in the magnetic flux density measurements is detected and, through the calibration process, it can be reported to the value of the force that is applied on the sensor when this is pressed.

3.2 Development Boards

The main board used during the entire work is the Teensy 4.0 development board. The Teensy 4.0 is a compact, high-performance microcontroller board designed for applications requiring significant processing power and low power consumption. Built around the NXP i.MX RT1062 ARM Cortex-M7 processor, it operates at clock speeds up to 600 MHz, making it one of the fastest microcontrollers available in its class. Despite its small size, the board includes an array of versatile input/output options, as well as multiple communication protocols such as SPI, I^{2}C, and CAN bus. Additionally, the Teensy 4.0 provides 1 megabyte of RAM and 2 megabytes of flash memory, supporting substantial codebases and data storage, which is particularly advantageous for intensive applications like signal processing, robotics, and IoT development. Its compatibility with the Arduino IDE and support for the Teensyduino software extension also facilitates rapid prototyping and simplifies the development of embedded systems.

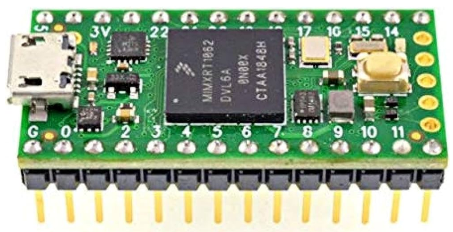


Figure 3.3: Teensy 4.0 Development Board

A second board was extensively used. This is the MIMRX1060-EVK Target Board, which is addressed in Appendix B (B), devoted to the debugging tools. In fact, this second board was mainly used for debugging purposes.

The communication protocol used by all the sensors is the I^{2}C. A detailed description of the protocol is provided in Appendix A (A).

Chapter 4

Firmware Development

This chapter details the methodology and procedure for firmware development. The first section introduces the framework used, beginning with an overview of the Zephyr RTOS, which is the chosen RTOS for dealing with the whole tactile sensor, its structure, and working principles, followed by a concise explanation of the methodology employed for real-time firmware development. The second section focuses on sensor driver development, providing a detailed description of the driver implementation for the MLX90395 Hall sensor, followed by the development process for the ICM20948 IMU driver.

4.1 Framework and Methodology

In this section the main tools for writing the drivers for the MLX90395 and ICM20948 sensors, are described. First, the main characteristics of the Zephyr RTOS will be addressed. Then, a short overview on how is the RTOS used for meeting real-time needs in this thesis work, is shown.

4.1.1 Zephyr RTOS

A Real-Time Operating System (RTOS) is a specialized operating system designed to manage hardware resources and run applications with precise timing and high reliability. The difference with general-purpose operating systems is that an RTOS prioritizes tasks based on their urgency and ensures that critical processes are executed within strict time constraints. Therefore, RTOS are ideal for applications where timing is crucial, such as embedded systems, robotics, automotive systems, and medical devices. Key features include multitasking, low latency, and deterministic behavior. The Zephyr Project[1] is a scalable real-time operating system (RTOS) supporting multiple hardware architectures, optimized for resource-constrained devices, and built with security in mind.

[1]Zephyr Project Website, `https://www.zephyrproject.org/`

The information presented in the following paragraphs about Zephyr RTOS was taken from the Zephyr official documentation [20]. For the installation and setup, the Getting Started Guide in [21] was followed.

Zephyr Github and Tools

The Zephyr Project GitHub [22] manages all the Zephyr-related repositories. The main Zephyr repository (zephyr) contains Zephyr's source code, configuration files, and build system. The build system is based on *CMake*, it is application-centric, and requires Zephyr-based applications to initiate building the Zephyr source code. The application build controls the configuration and build process of both the application and Zephyr itself, compiling them into a single binary.

West: Zephyr's meta-tool The Zephyr project includes a command line tool named *west*, which is inspired by Git submodules[2]. West's built-in commands simplify multi-repository management and allow developers to work with Git repositories under a common workspace directory. It's especially useful for projects where Zephyr is used as part of a larger codebase spread across multiple Git repositories. The *west* tool is developed in its own repository, which can be found in [22]. The documentation for *west* can be found in [20], in the *Developing with Zephyr* section.

West Manifests In Zephyr, *west manifests* play a crucial role in managing the complex dependencies and repository structure often involved in embedded systems projects. West manifests are YAML files, usually named *west.yaml*. Manifests have a top-level *manifest* section with some subsections, as shown and described in the following.

```yaml
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos

  projects:
    - name: zephyr
      remote: private-repo
      revision: dev-sensors
      import:
        name-allowlist:
          - cmsis       # required by the ARM port
          - hal_nxp     # required by the NXP port
          - hal_st      # required by the ST port
```

- *remotes*: contains a sequence of name and url-base which specifies the base URLs where projects can be fetched from;

[2]Git Tools - Submodules, `https://git-scm.com/book/en/v2/Git-Tools-Submodules`

- *projects*: contains a sequence describing the project repositories in the west workspace. Git remote URLs to use when cloning and fetching the projects need to be specified. Projects also have optional keys.
  - *revision*: refers to the Git repository branch to track;
  - *import*: imports projects from manifest files in the given repository into the current manifest. If used with the *name-allowlist* key as in the example, modules strictly needed by the application can be selected to be cloned;
  - *path*: path specifying where to clone the repository locally, relative to the top directory in the west workspace.

Other subsections can be also set up if needed. More information can be found in the documentation [20].

Kconfig: Configuration System The Zephyr kernel and subsystems can be configured at build time to adapt them for specific application and platform needs. Configuration is handled through *Kconfig*, which is the same configuration system used by the Linux kernel. Configuration options (symbols) are defined in *Kconfig* files. The documentation for *Kconfig* can be found in [20], in the *Build and Configuration Systems* section.

Zephyr Applications

Directory and Configuration Files The Zephyr application directory contains all application-specific files, such as application-specific configuration files and source code. The following scheme represents the directory of the application named firmware_app.

```
firmware_app
├── CMakeLists.txt
├── boards
│   ├── mimxrt1060_evk.overlay
│   └── teensy40.overlay
├── prj.conf
├── VERSION
└── src
    └── main.c
```

The files in the various folders will be briefly described in the following. More detailed information can be found in the *Application Development* section of the Zephyr documentation ([20]).

- *CMakeLists.txt*: This file tells the build system where to find the other application files, and links the application directory with Zephyr's CMake build system.
- *board.overlay*: The devicetree overlays specify which pieces of hardware will be used for the specific application. They can be multiple for multiple boards used for the same application.

- *prj.conf*: This is a *Kconfig* fragment that specifies application-specific values for one or more *Kconfig* options. These application settings are merged with other settings to produce the final configuration.

- *src*: This folder contains the source code files for the specific application, typically written in C, C++, or assembly language.

In the next chapters, examples of the content of the single application configuration files will be shown.

Application Types The Zephyr application types are three, but for the purposes of this thesis work two of them were mainly used.

- *Zephyr workspace application*
  This is located within the workspace but outside the zephyr repository. An example is shown in the following, where *zephyrproject* is the Zephyr workspace and *app* is the Zephyr workspace application.

```
zephyrproject/
├── .west/
├── zephyr/
├── build/
├── modules/
└── zephyr-app/
    └── app/
```

- *Zephyr freestanding application*
  This is located outside of a Zephyr workspace. An example is shown in the following, where *zephyrproject* is the Zephyr workspace and *app* is the Zephyr freestanding application.

```
<home>/
├── zephyrproject/
│   ├── .west/
│   ├── zephyr/
│   ├── build/
│   └── modules/
└── app/
    ├── CMakeLists.txt
    ├── Kconfig
    ├── prj.conf
    ├── debug.conf
    └── src
```

Application Initialization In Zephyr, applications can either created by hand, or it is possible to refer to already existing workspace applications. To initialize an already existing application, the following commands need to be followed in sequence in the folder where the workspace *ex-workspace* wants to be placed.

```
$ west init -m https://git.example.com/example-repo ex-workspace
$ cd ex-workspace
$ west update
```

The *west init* command creates the *ex-workspace* west workspace, and clones the manifest repo, while the *west update* command initially clones, and later updates, the projects listed in the manifest in the workspace.

Application Build and Flash Whenever an application is initialized, to deploy it on the microcontroller and test any device behavior, the code must be built and flashed onto the board. The west commands to do this are the following.

```
west build -b <BOARD> path/to/source/directory
west flash
```

Zephyr Devices

Zephyr devices are represented by a `struct device`, defined in *zephyr/device.h*. This structure holds a series of references to resources defined by drivers or defined internally by the device definition macros. The most important fields for the device structure are shown below.

```
struct device {
    const char *name;
    const void *config;
    const void *api;
    void *data;
};
```

Device Driver Model

The sensor driver is needed to interact with the hardware, and this happens through a communication protocol as I^{2}C or SPI. The driver handles initialization, configuration, and data acquisition. It also uses the Sensor Interface SPI to make data available to applications. In this thesis work, instance-based API, recommended for use within device drivers, was used. The hardware configuration is specified in the device tree. All of these aspects will be presented in more detail in the following.
The sensor driver folder is in a different directory depending on whether the application is a workspace or a freestanding one. For a Zephyr workspace application, the drivers are directly added to the *zephyr/drivers/sensor/vendor/sensor-name* folder, while for a Zephyr freestanding application, the directory is *zephyr-app/drivers/sensor/sensor-name*.

Sensor Interface API The Sensor Interface API is a standardized interface for accessing and interacting with different kind of sensors. It provides a set of functions, macros, and data structures, providing a common interface for applications. The sensor API referred to here is `sensor_driver_api`. The key functions to interact with the sensors are the *initialization* function, where the set up of the communication protocol and sensor settings happens, the *fetch* function to get a data sample from the sensor, and the *get* function, which takes the actual data from a specific channel of the sensor.
In fact, sensors may in general have different channels. A single channel can represent same physical properties (e.g. velocity if the sensor is an accelerometer) around different axes, or a different physical property measured by the same sensor. The sensor channels are defined in the *sensor.h* file located at *zephyr/include/zephyr/drivers*.

Instance-based API Instance-based API is a more flexible way of handling sensor drivers in Zephyr. It is especially used when multiple sensors of the same kind need to be implemented, each of which with different configurations and behaviors. To use this API, the `DT_DRV_COMPAT` macro needs to be defined. Then, after the API functions, an instantiation macro for each instance needs to be defined.

Device Tree A *devicetree*[3] is a hierarchical data structure used to define and configure hardware devices. It provides a hardware-agnostic configuration system where the sensor properties can be specified separately from the application code. The sensor driver can then use the properties defined in the device tree at runtime. Multiple instances of the same device can be defined in the device tree, each with its own properties.

Being the device tree a tree data structure, it has a hierarchy of nodes. Each node in the device tree is named according to the following convention: `node-name@address`. The `address` part is however only added if the node has the *reg* property defined. An example of a device tree node is shown below.

```
/ {
        device {
             compatible = "vendor,device";
             num-foos = <3>;
        };
};
```

- `/`: the root node does not have a name and it is identified by a forward slash;

- `device`: node with two properties.
  The *compatible* property defines the name of the hardware device the node represents. The recommended format is "vendor,device". This property is used to find the bindings for the node.

[3]Devicetree Website, `https://www.devicetree.org/`

When developing a sensor driver, the device tree files to be configured are the device tree sources (.dts files) and the device tree overlays (.overlay files). These two, linked with the device tree bindings which will be explained in the next paragraph, compose the complete device tree header file (devicetree.h). The following scheme 4.1 shows this concept.

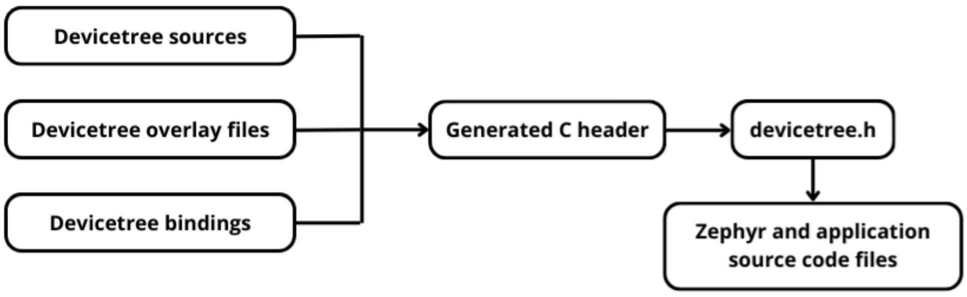


Figure 4.1: Devicetree Build Flow

Devicetree Bindings Device tree bindings are .yaml configuration files that define how the driver interacts with the device tree. During the configuration phase, the build system tries to match each node in the device tree to a binding file. This matching happens by checking that the compatible property has the same name for both the device tree node and the binding. The directory where the bindings are usually placed is *zephyr/dts/bindings*, and the files related to single sensors are named *vendor,device.yaml*. An example of binding referred to the device tree node in the previous paragraph is shown in the following.

```
compatible: "vendor,device"

properties:
  num-foos:
    type: int
    required: true
```

Information specific to the drivers for the MLX90395 and the ICM20948 sensors will be given in 4.2.

Zephyr Settings for Debugging

West Command for Debugging To compile a zephyr project for the MIMXRT1060_EVK target board described in B, the following command needs to be used.

```
west build -b mimxrt1060_evk -- -DCMAKE_BUILD_TYPE=Debug
```

- `--` is used to pass additional options to the build system;
- `-DCMAKE_BUILD_TYPE=Debug` option sets the build type to Debug, enabling debugging features such as specific debugging symbols and optimizations.

debug.conf File For debugging purposes, a specific *Kconfig* fragment can be set up. The file name is *debug.conf* and, together with the settings provided by the *prj.conf* file, produces the final configuration. The settings specified for the debugging of the application related to the tactile sensor firmware, are shown in the following.

```
# General
CONFIG_DEBUG=y                  # build kernel with debugging enabled
CONFIG_NO_OPTIMIZATIONS=y       # need for bigger stack size

# Compiler
CONFIG_DEBUG_OPTIMIZATIONS=y

# Segger
CONFIG_THREAD_NAME=y
CONFIG_THREAD_ANALYZER=y
CONFIG_SCHED_CPU_MASK=y
CONFIG_SEGGER_SYSTEMVIEW=y      # enable Segger SystemView
CONFIG_USE_SEGGER_RTT=y         # enable Segger J-Link RTT libraries
CONFIG_TRACING=y                # enable system tracing: needs SystemView
```

4.1.2 Methodology for Real-Time Firmware Development

The methodology for real-time firmware development in this work focuses on leveraging the Zephyr RTOS to design a modular and responsive system capable of meeting strict timing and performance requirements. The firmware incorporates two key threads: one dedicated to communication with a CANopen network and the other to sensor fusion. The CANopen thread handles the protocol-specific demands of a robotic application, ensuring real-time data exchange and synchronization over the CAN bus with precise timing to avoid delays and meet network deadlines. To meet these requirements, the CANopen thread is assigned a higher priority, ensuring it receives processor time before less critical tasks, such as sensor fusion. The sensor fusion thread processes data from the tactile sensor, integrating information from the Hall sensor and the IMU to derive meaningful insights. This separation of concerns allows each thread to operate at its appropriate frequency, prioritized according to its real-time constraints, and synchronized using Zephyr's scheduling and inter-thread communication primitives. This design ensures modularity, scalability, and fault isolation while meeting the performance demands of the robotic application.

4.2 Sensor Drivers Development

This section is devoted to the sensor drivers development. First, the MLX90395 sensor driver writing in Zephyr RTOS environment will be exploited in detail, focusing both on the driver and application configuration files and settings, and on the content of the driver source code. To this aim, the sensor characteristic will be described and discussed. Afterwards, the same content for the ICM20948 IMU sensor will be described, less in detail since most of the functions working principle is similar to the first sensor.

4.2.1 MLX90395 Hall Sensor Driver

The characteristics of Zephyr Device Drivers were already introduced in 4.1.1. This section, however, is specific to the MLX90395 sensor. The first part of the discussion will be about the configuration and source code files specific to the sensor driver, while the second part describes the configuration files for an application that uses the sensor driver. This latter part is kept general, to be adapted to any application using the sensor. The information for writing the driver was taken from the sensor datasheet, [23]. In this work, the high-field version of the Hall sensor is used.

The hardware used for the analysis and tests on the MLX90395 Hall sensor comprises the tactile sensor itself, shown in 3.2, and of the Teensy 4.0 board shown in 3.3.

Driver CMakeLists.txt

The *CMakeLists.txt* file contains the information to compile the MLX90395 sensor driver into a Zephyr library, which makes the driver source files available to all the other files of the project that want to use the functions and API defined in the *mlx90395.c* file. Therefore, after the compilation procedure, whatever application can refer to the sensor driver. The translation of this concept is shown in the following snippet, which is the content of the *CMakeLists.txt* file for the Hall sensor driver.

```
zephyr_library()
zephyr_library_sources(mlx90395.c)
```

Driver Kconfig

The *Kconfig* inside a sensor driver folder is used to define configuration options for that driver. In particular, it ensures that the driver is only available when the sensor is defined in the correct way in the device tree (though the *depends on* statement), and if this is verified, it automatically enables the I^{2}C driver (through the *select ... if* statement). The *menuconfig* option in the *Kconfig* enables the sensor, whenever `CONFIG_MLX90395` is set to yes (*y*) in the *prj.conf* file. The *Kconfig* file for the MLX90395 sensor is shown below.

```
menuconfig MLX90395
        bool "MLX90395 Three Axis Magnetometer"
        default y
        depends on DT_HAS_MELEXIS_MLX90395_ENABLED
```

```
select I2C if $(dt_compat_on_bus,$(DT_COMPAT_MELEXIS_MLX90395),i2c)
help
  Enable support for the MLX90395 Three Axis Magnetometer.
```

Devicetree Binding

The devicetree binding for the MLX90395 Hall sensor is the *melexis,mlx90395.yaml* file. Reference to this binding is made in 4.2.1, where the device tree specifications for the application using the sensor are defined. The compatible property is the same as the one in the .overlay files. The content of the devicetree binding is shown below. The *include* property makes the bindings for the other sensor properties or configurations available.

```
description: |
  Melexis MLX90395  3-axis magnetometer sensor accessed through I2C bus

compatible: "melexis,mlx90395"
include: [sensor-device.yaml, i2c-device.yaml]
```

mlx90395.h

The header file (.h) of the MLX90395 sensor driver contains definitions and declarations necessary for the driver to interact with other parts of the system, including the application and kernel. The *mlx90395.h* file contains the information described in the following. After listing each piece of information, an example is also provided.

- *Include Guards*: prevent multiple inclusions of the header file during compilation;

  ```
  #ifndef __SENSOR_MLX90395_H_
  #define __SENSOR_MLX90395_H_
  ```

- *Includes*: import necessary standard libraries and Zephyr modules;

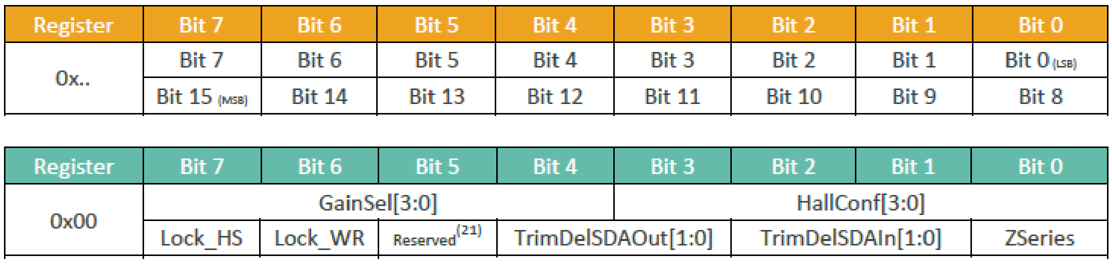
  ```
  #include <zephyr/drivers/i2c.h>
  ```

- *Enums and Macros Definitions*: define register addresses, bit masks, or other fixed values specific to the sensor;

  ```
  enum {GAIN_SEL_REG = 0x0,
      GAIN_SEL_MASK = 0x00f0,
      GAIN_SEL_SHIFT = 4};
  #define MLX90395_I2C_ADDR      0x30
  ```

  The example considered here is referred to the Gain properties. The information for the registers, masks and shift values are taken from the sensor datasheet [23], in particular from the Register Map table in Section 18.3. A snippet of the Register

Map is visible in 4.2. The GainSel value lies from Bit 4 to Bit 7 of register 0x00 (`GAIN_SEL_REG`). Therefore, `GAIN_SEL_MASK` is 0x00f0 and `GAIN_SEL_SHIFT` is 4.



Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x..	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0 (LSB)
	Bit 15 (MSB)	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8

Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x00	GainSel[3:0]				HallConf[3:0]			
	Lock_HS	Lock_WR	Reserved(21)	TrimDelSDAOut[1:0]		TrimDelSDAIn[1:0]		ZSeries

Figure 4.2: Register Map Example

- *Driver-Specific Data Structures*: structs to store runtime and configuration data.

```
struct mlx90395_config {
    struct i2c_dt_spec i2c;
    uint16_t bandwidth;
    uint16_t frequency;
    uint16_t prd_set;
    uint8_t zyxt;
    uint8_t bdr;
    enum mlx90395_mode mode;
};

struct mlx90395_data {
    uint16_t magn_x;
    uint16_t magn_y;
    uint16_t magn_z;
    struct k_sem sem;
};
```

mlx90395.c

The *mlx90395.c* file implements the logic that allows the Zephyr RTOS to communicate with the sensor using the driver's API. The .c file starts by including the necessary header files, which are the ones related to Zephyr RTOS, the communication protocol, and the *mlx90395.h* itself. Then, the helper functions to read from and write to the sensor's registers and a set of other functions to interact with the sensor and to modify its characteristics, are defined. Moreover, the sensor driver API structure (already described in 4.1.1) and the macros to associate the driver with the hardware in the device tree, are added. The structure for the sensor driver source code is shown below, with links referring to each function explained in the following paragraphs.

mlx90395.c Structure

Define `DT_DRV_COMPAT melexis_mlx90395`
Include necessary header files
Read Register
Write Register
Set Offset
Set Gain
Set Sensitivity
Set Resolution
Set Burst Rate
Digital Filtering
Convert x, y, z
Fetch Data
Channel Get
Exit
Reset
Start Burst
Start Single Measurement
Set Mode
Init
Define `MLX90395_DEFINE(inst)`

All the functions in the mlx90395.c file will be described one by one in the following. The information for the functions writing were extracted from the MLX90395 datasheet, [23]. Many functions have as argument `const struct device *dev`, which is a pointer to the device structure defined in the *device.h* header file, already introduced in 4.1.1. Some functions also use API related to I^2C protocol, which are defined in the *i2c.h* header file.

Read and Write Register For memory read and write commands, register access is provided directly through the I^2C protocol. To read the pictures in the following, 4.3 shows the conventions for I^2C.

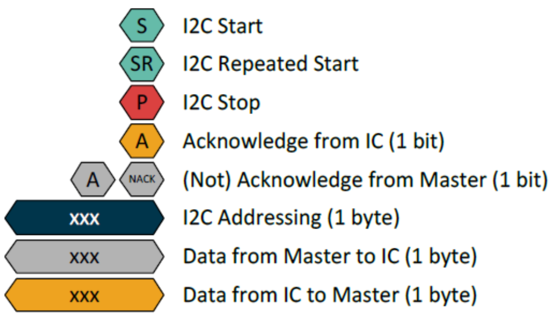

S
I2C Start
SR
I2C Repeated Start
P
I2C Stop
A
Acknowledge from IC (1 bit)
A
NACK
(Not) Acknowledge from Master (1 bit)
xxx
I2C Addressing (1 byte)
xxx
Data from Master to IC (1 byte)
xxx
Data from IC to Master (1 byte)

Figure 4.3: I^2C Convention

Important in read and write register commands is that the register address to be read and written must be shifted left by one bit.

Read Register The command implementation for the Read Register function is shown below in 4.4, followed by its pseudo-code.

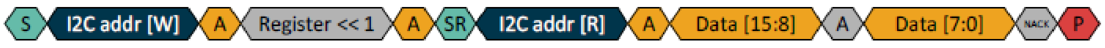


Figure 4.4: Read Register Command

```
Algorithm Read Register Function
Require: *device, register
  procedure READREGISTER
      Extract config struct
      Initialize a two-byte command cmd
      Shift left the register by 1 and assign it to cmd[0]
      return Output of i2c_write_read_dt function
  end procedure
```

Write Register The command implementation for the Write Register function is shown in 4.5, followed by its pseudo-code.

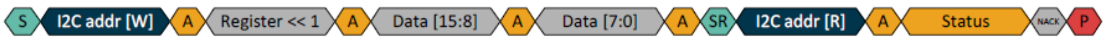
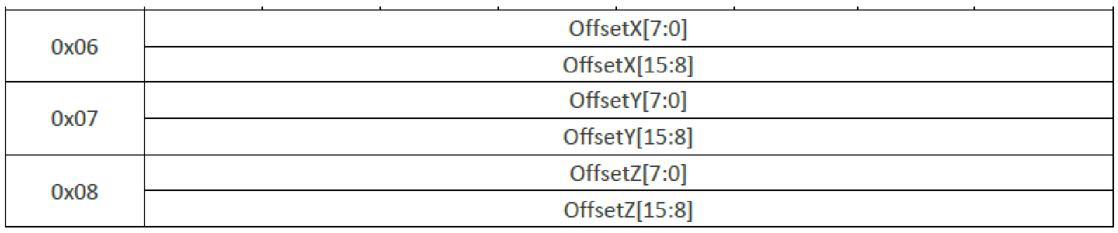


Figure 4.5: Write Register Command

```
Algorithm Write Register Function
Require: *device, register, data
  procedure WRITEREGISTER
      Extract config struct
      Initialize a three-byte command cmd
      Shift left the register by 1 and assign it to cmd[0]
      Assign 16-bit data to cmd[1] and cmd[2]
      return Output of i2c_write_read_dt function
  end procedure
```

Offset This function is needed to change the offset on the output of the MLX90395. This is done by using *OffsetX*, *OffsetY*, and *OffsetZ* parameters, which are each unsigned 16-bit variables. Figure 4.6 shows the register map for the Offset variables.



0x06	OffsetX[7:0]
	OffsetX[15:8]
0x07	OffsetY[7:0]
	OffsetY[15:8]
0x08	OffsetZ[7:0]
	OffsetZ[15:8]

Figure 4.6: Offset Register Map

The default offset value is 0x8000, which in binary is 1000 0000 0000 0000, and this corresponds to no offset adjustment. Moreover, the offset is adjusted on the 19-bit Analog to Digital Converter (ADC) value, so before the sensitivity adjustments and the resolution selection. Equation 4.1 explains the relationship between the raw measurement MAG, and the chosen offset value.

$$MAG = MAG - 4 * OFFSET_{X|Y|Z} \tag{4.1}$$

The pseudo-code for the Set Offset function is shown in the algorithm below.

Algorithm Set Offset Function

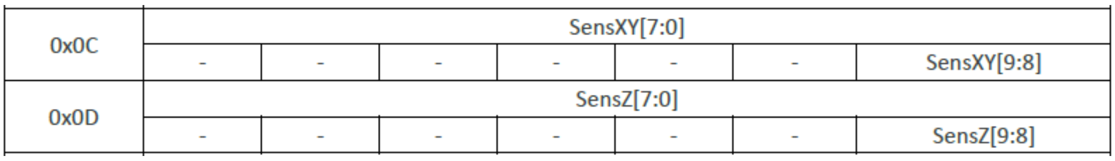
```
Require: *device, command for x, y, z
  procedure SetOffset
      Write command for x in X_OFFSET_REG
      Write command for y in Y_OFFSET_REG
      Write command for z in Z_OFFSET_REG
  end procedure
```

Gain This functions is needed to change the sensitivity of the MLX90395 sensor. This one can be adjusted with several parameters, one of which is *GainSel*. The *GainSel* parameter changes the gain stages before the input of the ADC. For small sensitivity adjustments, other parameters (SensXY, SensZ, ResX, ResY, ResZ, explained in the following paragraphs) are recommended. *GainSel* is recommended to be kept constant, and the default value for the high-field version of the sensor is 8, being the maximum possible value 15. The Register Map snippet for the *GainSel* parameter was already introduced in 4.2.1.

Algorithm Set Gain Function

```
Require: *device, command for x, y, z
  procedure SetGain
      Initialize two uint16_t variables: old_val, new_val
      Read GAIN_SEL_REG and store the value in old_val
      Update old_val assigning the new value to new_val
      Write new_val to GAIN_SEL_REG
  end procedure
```

Sensitivity Unlike GainSel, $SensXY$ and $SensZ$ parameters adjust the sensitivity after the ADC, so fully on the digital side. These parameters are always positive. Figure 4.7 shows the register map for the $SensXY$ and $SensZ$ parameters. Since both $SensXY$ and $SensZ$ occupy 8 bits, their maximum value is 255.



0x0C	SensXY[7:0]						
	-	-	-	-	-	-	SensXY[9:8]
0x0D	SensZ[7:0]						
	-	-	-	-	-	-	SensZ[9:8]

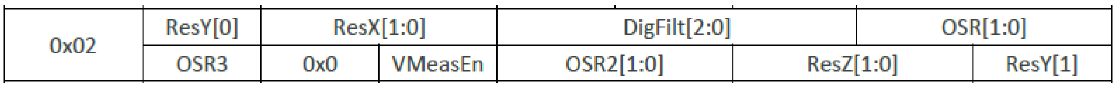
Figure 4.7: Sensitivity Register Map

Equation 4.2 explains the relationship between the raw measurement MAG, and the chosen $SensXY$ and $SensZ$ parameter value.

$$MAG = MAG \times \left[1 + \frac{SensXY \text{ or } SensZ}{2^9}\right] \tag{4.2}$$

The pseudo-code for the Set Sensitivity function is similar to the Set Gain one. However, the content of two different registers, `SENS_XY` and `SENS_Z`, are read and changed.

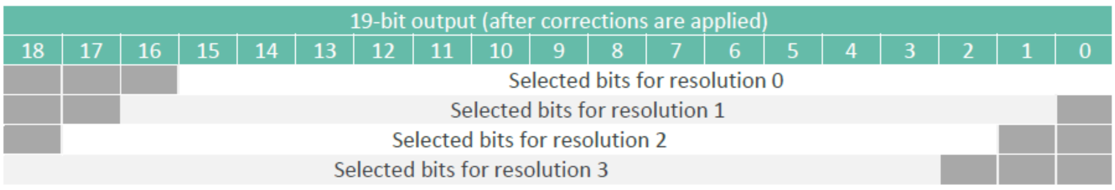
Resolution This function is needed to set the $ResX$, $ResY$, $ResZ$ parameters, with which 16 of the 19-bit ADC value can be selected. Figure 4.2 shows the register map for $ResX$, $ResY$, and $ResZ$ parameters. Since they occupy only two bits each, their value can go from 0 to 3.



0x02	ResY[0]	ResX[1:0]		DigFilt[2:0]			OSR[1:0]	
	OSR3	0x0	VMeasEn	OSR2[1:0]		ResZ[1:0]		ResY[1]

Figure 4.8: Resolution Register Map

The implementation for these parameters is shown in 4.9. For resolution 2 and 3 the saturation of the analog chain becomes visible and the used span is 17.2 bits of the available 19.



19-bit output (after corrections are applied)

18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
▨	▨	▨	Selected bits for resolution 0															
▨	▨	Selected bits for resolution 1																▨
▨	Selected bits for resolution 2																▨	▨
Selected bits for resolution 3																▨	▨	▨

Figure 4.9: Resolution Implementation

The pseudo-code for the Set Resolution function is similar to the Set Gain one. However, the `res_x`, `res_y`, and `res_z` parameters are first chained into a single variable, `res_xyz`, and then the content of the `RES_XYZ_REG` register is read and changed.

Digital Filtering The purpose of this function is to set up one of the existing measurement filters for the MLX90395 sensor. It is present on the magnetic measurements. This digital filter averages over $2^{DigFilt}$ measurements, where *DigFilt* is the parameter to set. The Register Map information for the Digital Filtering is visible in 4.8. Since the *DigFilt* parameter occupies 3 bits, its maximum possible value is 7. The pseudo-code for the Digital Filtering function is the same as the Set Gain one in 4.2.1, but the register to be read and written on is `DIG_FLT_REG`, and the mask and shift values are respectively `DIG_FLT_MASK` and `DIG_FLT_SHIFT`.

Convert The purpose of the converting functions is converting the sensor's raw digital output into meaningful physical values, which in the case of the Hall sensors is the magnetic field strength, measured in Tesla or Gauss (10^{-4} T). The raw output has a width of 16 bits and it's a signed integer, therefore the values are in the range -32768 to 32767.

The conversion consists of a multiplication (or division) by a scaling factor representing the sensor's sensitivity, whose value is specified in the sensor datasheet. The sensitivity refers to the relationship between the digital output (measured in LSB) and the magnetic field strength (measured in Tesla or Gauss). It quantifies how many digital units the sensor's output changes per unit of magnetic field or, inversely, how much magnetic field change is represented by a single step (LSB) in the sensor's digital output. The sensitivity values for the three axes x, y, and z, can be found in the sensor datasheet and are shown in 4.1. The quantities are described in both $\mu T/LSB_{16}$ and LSB_{16}/mT units of measurements.

Parameter	Symbol	Typical	Unit
Sensitivity X- or Y- axis	S_{XX50}	7.14	$\mu T/LSB_{16}$
	S_{YY50}	140	LSB_{16}/mT
Sensitivity Z- axis	S_{ZZ50}	7.14	$\mu T/LSB_{16}$
		140	LSB_{16}/mT

Table 4.1: Sensitivity values from Datasheet

Given this information, the formula for the conversion, which is the same for all x, y, and z coordinates, is shown below.

$$MagneticField\ [mT] = \frac{RawData\ [LSB]}{140\ [LSB/mT]} \tag{4.3}$$

Fetch This function fetches new data sample. The pseudo-code for the Fetch Sample function is shown in the algorithm below. A semaphore is used to ensure thread-safe access to the shared resource which is the device data structure. To take the semaphore, the `k_sem_take(&data->sem, K_FOREVER)` function was used, and to release it, the function is `k_sem_give(&data->sem)`.

Algorithm Fetch Sample Function

```
Require: *device, sensor channel
  procedure FetchSample
      Extract config struct
      Extract data struct
      Initialize a nine-byte buffer cmd
      Assign the content of MLX90395_CMD_READ_MEASUREMENT to cmd[0]
      Acquire the semaphore
      Throw an error if the $I^2C$ read fails
      Release the semaphore
      Extract the magnetometer's x, y, and z axis measurements from the buffer
  end procedure
```

Channel Get As already said in 4.1.1, the *Channel Get* function takes the actual data from a specific channel of the sensor, after the data has been fetched. The pseudo-code for the Fetch Sample function is shown in the algorithm below.

Algorithm Channel Get Function

```
Require: *device, sensor channel
  procedure ChannelGet
      Extract data struct
      Acquire the semaphore
      Switch based on the sensor channel
      Convert data with the converting function
      Release the semaphore
  end procedure
```

Exit The command implementation shown in 4.10 has the same structure of the one for the Exit Function, but shows an example for a start of burst mode with X and Y being measured. In the case of the Exit function, the command is 1000 0000, which corresponds to 128.

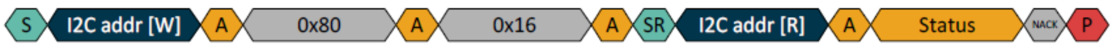


Figure 4.10: Exit Command

The pseudo-code for the Exit function is shown in the algorithm below. The parameter referred to as `MLX90395_CMD_EXIT` is the Exit command, 128.

Reset The command implementation for the Reset Function is shown in 4.11. The command value is now 0xf0, which in binary is 1111 0000, and in decimal 240.

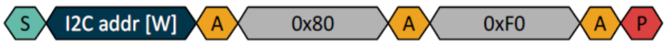


Figure 4.11: Reset Command

Sensor Modes The MLX90395 Hall sensor has three different operating modes: single measurement, burst, and wake-up on change. In this sensor driver implementation only the first two were considered, and the functions to set those modes are described in the next two paragraphs.

- *Single Measurement*: in this mode, a single measurement is requested by the master through either the Start Single Measurement (SM) command, or via a rising edge on the trigger pin;
- *Burst*: in this mode, the sensor will continuously make measurements. The mode is started by the Start Burst (SB) and ended by the Exit commands. The rate of measurements is programmable throught the Set Burst Rate function.

The structure of the command implementation for the Start Burst and Start Single Measurement functions is the one shown in 4.10. The commands for the functions referred to sensor modes are shown in 4.2. The zyxt is used to select which of the x, y, z axes or temperature (t) components need to be measured, by setting the corresponding bit to 1.

Name	Symbol	#	CMD Byte #1	CMD Byte #2	CMD Byte #3	CMD Byte #4
Start Burst Mode	SB	1	0b0001 zyxt	N/A	N/A	N/A
Start WOC Mode	SWOC	2	0b0010 zyxt	N/A	N/A	N/A
Start Single Measurement Mode	SM	3	0b0011 zyxt	N/A	N/A	N/A

Table 4.2: Sensor Modes Commands

Start Burst This function enables the activation of the Burst Mode, as explained in the general Sensor Modes paragraph in 4.2.1. The pseudo-code is shown in the following algorithm. The complete command structure for the Burst Mode is shown in the first row of Table 4.2, which in code translates to `MLX90395_CMD_START_BURST | zxyt_flags`.

Algorithm Start Burst Function

```
Require: *device, zyxt_flags
  procedure StartBurst
      Extract config struct
      Initialize a two-byte command cmd
      Assign 0x80 to cmd[0]
      Assign MLX90395_CMD_START_BURST | zxyt_flags to cmd[1]
      return Output of i2c_write_read_dt function
  end procedure
```

Start Single Measurement This function enables the activation of the Single Measurement Mode, as explained in the general Sensor Modes paragraph in 4.2.1. The pseudo-code is shown in the following algorithm. The complete command structure for the Single Measurement Mode is shown in the last row of figure 4.2, which in code translates to `MLX90395_CMD_START_MEASUREMENT | zxyt_flags`.

Algorithm Start Single Measurement Function

```
Require: *device, zyxt_flags
  procedure StartSingleMeasurement
      Extract config struct
      Initialize a two-byte command cmd
      Assign 0x80 to cmd[0]
      Assign MLX90395_CMD_START_MEASUREMENT | zxyt_flags to cmd[1]
      return Output of i2c_write_read_dt function
  end procedure
```

Set Mode This function is needed to set the operating mode of the sensor. The pseudo-code is shown in the algorithm below.

Algorithm Set Mode Function

```
Require: *device, register, zyxt_flags, mode, BaudRate
  procedure SetMode
      Switch(mode)
      Case MLX90395_MODE_IDLE: Exit
      Case MLX90395_MODE_BURST: Start Burst
      Case MLX90395_MODE_SINGLE_MEASUREMENT: Start Single Measurement
      EndSwitch
  end procedure
```

Burst Rate The Set Burst Rate function is needed to set the rate of measurements for the sensor in burst rate mode. The pseudo-code for the Set Burst Rate function is the same as the one for the Set Gain function in 4.2.1. However, the register to be read and written on is `BURST_SEL_REG`, and the mask and shift values are respectively `BURST_SEL_MASK` and `BURST_SEL_SHIFT`.

Init The Init function initializes the MLX90395 sensor. In particular, it verifies that the sensor is ready for the communication through I^2C protocol, it configures the right settings for the sensor, and prepares it to be used in whatever application inside the Zephyr RTOS environment. The pseudo-code for the initialization function is shown in the algorithm below.

Algorithm Initialization Function

```
Require: *device
  procedure INIT
      Extract config struct
      Extract data struct
      Verify if the $I^2C$ bus is ready; throw an error otherwise
      Exit the sensor
      Put the thread to sleep for 50 ms
      Reset the sensor
      Put the thread to sleep for 50 ms
      Set Offset
      Set Gain
      Set Sensitivity
      Set Resolution
      Digital Filtering
      Set Burst Rate
  end procedure
```

Application Configuration Files

To be able to use the sensor data for any application or project purposes, the application configuration files also need to be set up. In particular, the files introduced in 4.1.1, when describing the generic Zephyr application directory and configuration files, will be described in detail. Therefore, the CMakeLists.txt, Kconfig, <board>.overlay, prj.conf files, will be discussed in the following paragraphs.

CMakeLists.txt The *CMakeLists.txt* file contains the settings for the configuration of the application build system. The file content specifies the minimum version required for CMake to build the project (*cmake_minimum_required(...)*), the project name and the programming language used (*project(...)*), and the source files needed for the application (*target_sources(...)*). The content is shown below.

```
cmake_minimum_required(VERSION 3.13.1)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(mlx9039x LANGUAGES C)
target_sources(app PRIVATE src/main.c)
```

Kconfig The *Kconfig* file for the application only contains the line below. It refers to a file provided by the Zephyr RTOS which contains the essential configuration settings that can apply to most Zephyr projects. The *Kconfig.zephyr* file represents the standard settings for a Zephyr application.

```
source "Kconfig.zephyr"
```

.overlay Files Each .overlay file refers to a single board to be used in the application. The files for both the boards used for the MLX90395 application are described in the following.

- *teensy40.overlay*

```
/ {
    chosen {
        zephyr,console = &cdc_acm_uart0;
    };
};

&zephyr_udc0 {
    cdc_acm_uart0: cdc_acm_uart0 {
            compatible = "zephyr,cdc-acm-uart";
    };
};

&lpi2c1 {
    status = "okay";
    clock-frequency = <400000>;

    mlx90395c: mlx90395@c {
        compatible = "melexis,mlx90395";
        reg = <0x0c>;
    };
};
```

The `chosen` node under the root node does not refer to any hardware device, unlike general devicetree nodes. In fact, its properties are used to configure system-wide values, in this case the `zephyr,console` property sets Universal Asynchronous Receiver-Transmitter (UART) device used by the console driver. More details about the `cdc_acm_uart0` node can be found in the documentation[4].

- *mimxrt1060_evk.overlay*

```
&lpi2c1 {
    status = "okay";
    clock-frequency = <100000>;

    mlx90395: mlx90395@0c {
        compatible = "melexis,mlx90395";
        reg = <0x0c>;
    };
};
```

[4]Zephyr RTOS Documentation - USB Device Support, `https://docs.zephyrproject.org/latest/connectivity/usb/device/usb_device.html`

The `&lpi2c1 {...}` region is the same for both the considered boards. This refers to an I^2C bus instance in the Zephyr device tree. The *status* property indicates the operational status of the device, and "okay" indicates that the node is enabled. The *clock-frequency* property specifies the clock frequency for the I^2C bus, which in this case is 400 kHz, corresponding to the fast mode. Lastly, the *reg* property specifies the I^2C register of the device.

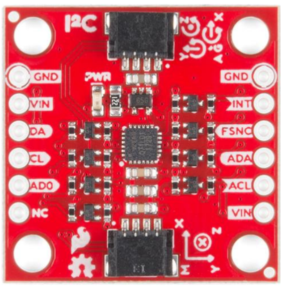
prj.conf The settings for the *prj.conf* configuration file are shown below. Other than the general assignments, some of the settings are specifically related to the board used, in this case the Teensy 4.0. These refer to the configuration of the USB device and serial output. Moreover, settings related to the communication protocol (`CONFIG_I2C=y`) and the specific sensor used (`CONFIG_MLX90395=y`) are specified.

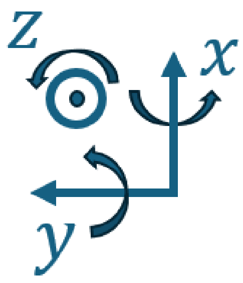
```
# GENERAL
CONFIG_LOG=y
CONFIG_GPIO=y
CONFIG_STDOUT_CONSOLE=y

# START - Only for Teensy 4.0
# USB (incl. console output)
CONFIG_USB_DEVICE_STACK=y
CONFIG_USB_DEVICE_VID=0x16C0
CONFIG_USB_DEVICE_PID=0x0483
CONFIG_USB_DEVICE_INITIALIZE_AT_BOOT=n
CONFIG_USB_DEVICE_PRODUCT="RTOS Teensy"
CONFIG_USB_DRIVER_LOG_LEVEL_ERR=y

# SERIAL OUTPUT
CONFIG_SERIAL=y
CONFIG_CONSOLE=y
CONFIG_UART_CONSOLE=y
CONFIG_UART_LINE_CTRL=y
CONFIG_CBPRINTF_FP_SUPPORT=y
## END - Only for Teensy 4.0

# I2C
CONFIG_I2C=y
CONFIG_SENSOR=y
CONFIG_MLX90395=y

# PRINTF SUPPORT
CONFIG_CBPRINTF_FP_SUPPORT=y
```

4.2.2 ICM20948 IMU Sensor Driver

The content of this section is the same as the one for the MLX90395 driver, detailed in 4.2.1. Therefore, the structure of the section is the same. The first part will be about the configuration and source code files specific to the sensor driver, while the second part describes the configuration files for an application that uses the sensor driver. The information for writing the driver was taken from the sensor datasheet, [24].

The hardware used for the analysis and tests on the ICM20948 IMU sensor is composed of the SparkFun Qwiic ICM20948 Breakout Board, shown in 4.12a, and of the Teensy 4.0 board shown in 3.3.


(a) Breakout Board



(b) Reference System

Figure 4.12: ICM20948 Sparkfun Breakout Board

Driver CMakeLists.txt The *CMakeLists.txt* file of the sensor driver has already been described in 4.2.1. The content, shown below, is the same as the one for the MLX90395 sensor, but the source code file referred to is icm20948.c.

```
zephyr_library()
zephyr_library_sources(icm20948.c)
```

Driver Kconfig The Kconfig file description was also already provided in 4.2.1.

```
menuconfig ICM20948
    bool "Custom ICM20948 sensor driver"
    default y
    depends on DT_HAS_INVENSENSE_ICM20948_ENABLED
    select I2C if $(dt_compat_on_bus,$(DT_COMPAT_INVENSENSE_ICM20948),i2c)
    help
      Enable ICM20948 sensor
```

Devicetree Binding The devicetree binding for the ICM20948 IMU sensor is the *tdk,icm20948.yaml* file. Reference to this binding is made in 4.2.1, where the device tree specifications for the application using the sensor are defined. The compatible property is the same as the one in the .overlay files desribed in 4.2.2. The content of the devicetree binding is shown below. The *include* property makes the bindings for the other sensor properties or configurations available.

```
description: |
TDK ICM20948 9-axis IMU (Inertial Measurement Unit) sensor
accessed through I2C bus

compatible: "tdk,icm20948"
include: ["i2c-device.yaml"]
```

Header Files

The header files (.h) of the ICM20948 sensor driver contains definitions and declarations necessary for the driver to interact with other parts of the system, including the application and kernel. The header files for this driver are two, the *icm20948_reg.h* and the *icm20948.h*, whose content will be described in the following.

icm20948_reg.h The ICM20948 sensor has several user banks (0, 1, 2, and 3), each with specific registers that can be configured for different sensor operations. This header file defines the sensors default addresses, the content of the registers addresses contained in Bank 0, Bank 1, Bank 2, and Bank 3, and the registers for the AK09916 magnetometer.

icm20948.h The second header file contains, in order, the include guards, the inclusion of the I^2C communication protocol header file, and the definition of driver-specific data structures related to runtime data and configuration.

```
#ifndef DRIVERS_SENSOR_ICM20948_H_
#define DRIVERS_SENSOR_ICM20948_H_

#include <zephyr/drivers/i2c.h>

struct icm20948_data {
    int16_t accel_x;
    int16_t accel_y;
    int16_t accel_z;

    int16_t gyro_x;
    int16_t gyro_y;
    int16_t gyro_z;

    int16_t magn_x;
    int16_t magn_y;
    int16_t magn_z;

    int16_t temp;
};
```

```
struct icm20948_config {
    struct i2c_dt_spec i2c;
};

#endif
```

icm20948.c

The *icm20948.c* file implements the logic that allows the Zephyr RTOS to communicate with the sensor using the driver's API. The .c file starts by including the necessary header files, which are the ones related to Zephyr RTOS, the communication protocol, and the ICM20948 header files themselves. Then, a set of functions to interact with the sensor and to modify its characteristics, are defined. Moreover, the sensor driver API structure (already described in 4.1.1) and the macros to associate the driver with the hardware in the device tree, are added. The structure for the sensor driver source code is shown below, with links referring to each function explained in the following paragraphs.

icm20948.c Structure

```
Define DT_DRV_COMPAT invensense_icm20948
Include necessary header files
Set Correct Bank
ICM20948 Read Register
ICM20948 Write Register
AK09916 Write Register
AK09916 Read Register
Convert Accelerometer
Convert Gyroscope
Convert Magnetometer
Fetch Data
Channel Get
Init
Define ICM20948_DEFINE(inst)
```

Some functions will be described in the following. The information for the functions writing were extracted from the ICM20948 datasheet, [24]. Many functions have as argument `const struct device *dev`, which is a pointer to the device structure defined in the *device.h* header file, already introduced in 4.1.1. Some functions also use API related to I^2C protocol, which are defined in the *i2c.h* header file.

Set Correct Bank The Set Correct Bank Function selects the wanted user bank. The pseudo-code is shown in the algorithm below.

Algorithm Set Correct Bank Function

```
Require: *device, bank
  procedure SetCorrectBank
      Extract config struct
      Save the previously set bank in current_bank
      Return if the wanted user bank is the same as previously set
      return Output of i2c_reg_write_byte_dt function
      set current_bank to bank
  end procedure
```

Init The Init function initializes the ICM20948 sensor. In particular, it verifies that the sensor is ready for the communication through I^2C protocol, it configures the wanted settings for each user bank of the sensor, and prepares it to be used in whatever application inside the Zephyr RTOS environment. The pseudo-code for the initialization function is shown in the algorithm below.

Algorithm Init Function

```
Require: *device
  procedure Init
      Extract config struct
      Verify if the I²C bus is ready; throw an error otherwise
      Reset unit: write 0x81 in ICM20948_REG_PWR_MGMT_1
      Wait for Reset to complete
      Exit sleep mode: write 0x01 in ICM20948_REG_PWR_MGMT_1
      Check which device is being accessed: if the address is not
  ICM20948_DEFAULT_ADDRESS, throw an error.
      Set Gyroscope settings
      Set accelerometer settings
      Set user bank 3 settings
      return 0
  end procedure
```

Application Configuration Files

The structure of the application configuration files was already introduced in 4.1.1, and the detailed structure for a generic application using the MLX90395 sensor was already described in 4.2.1.

The structure of many of the configuration files is the same also for the ICM20948 sensor. In particular, the *Kconfig* file is the standard one, the *CMakeLists.txt* file is the same except for the project name, which will change depending on the application, and the *prj.conf* file is the same, except for the `CONFIG_MLX90395=y`, which is substituted by `CONFIG_ICM20948=y`.

The only file that changes slightly is the *<board>.overlay* for both the Teensy 4.0 and the mimxrt1060_evk boards. In fact, the `&lpi2c1 {...}` region does not contain the `mlx90395` node, but the `icm20948` one, therefore the property specifying the register is different since the I^{2}C address for the ICM20948 sensor is 0x69. The content of the new region is shown below.

```
&lpi2c1 {
    status = "okay";
    clock-frequency = <400000>;

    icm20948: icm20948@69 {
            compatible = "invensense,icm20948";
            reg = <0x69>;
    };
};
```

Chapter 5

Experimental Evaluation

This chapter is devoted to the description of the experimental evaluation done on both MLX90395 and ICM20948 sensors. In particular, the first section is about the tests and results for the Hall sensor characterization. The second section will instead describe the ICM20948 IMU calibration and comparison with benchmark sensors and libraries.

5.1 MLX90395 Sensor Characterization

To characterize the behavior of the sensor, some tests have been performed. The first test is about the behavior of the single Hall sensors. Then, the effect of the permanent magnets on the IMU magnetometer is considered. The third test was done to measure the sensor sensitivity, lastly a comparison with the Xela tactile sensor in terms of resolution was made. The next sections will describe the tests and their results in detail.

5.1.1 Single Hall Sensors Behavior

The first test was done to show the behavior of the single taxels of the DLR tactile sensor, each one composed by a MLX90395 Hall sensor. The single taxels were pressed one by one for a duration of around two seconds, data was collected for 20 seconds, and a plot was produced. The result of the experiment is shown in Figure 5.1.

The x axis in the plot show the time, measured in seconds, while the y axis measures the magnetic field magnitude, measured in milliTesla. The first and second subplot show the behavior of the magnetic data along the x and y axes of the Hall sensor. However, since the direction of biggest solicitation for the Hall sensors is the z, the third subplot is the most important. It shows the value in milli-Tesla of the magnetic flux density along the z coordinate, which is always the most involved in these tactile sensing modalities.

The most immediate verification is about the magnitude of the magnetic field. The raw magnetic data is a 16-bit signed integer, therefore its range goes from -32768 to 32767, as said in 4.2.1. According to the converting function, when one of the permanent magnets

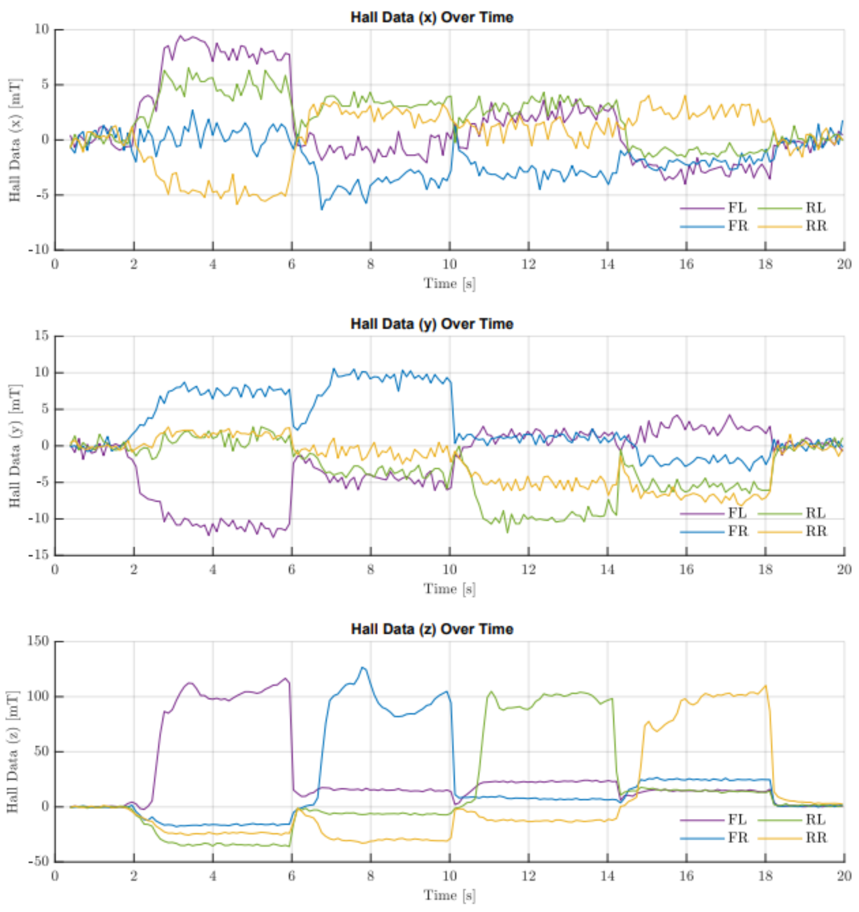


Figure 5.1: Single Hall Sensors Behavior

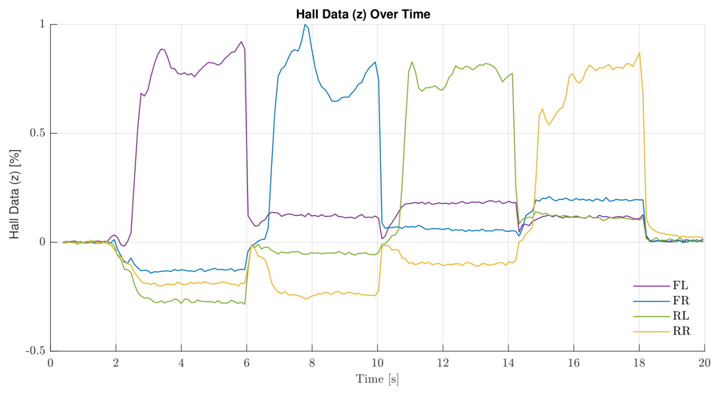
is pressed on one Hall sensor with a force leading to the maximum value in the range, the corresponding value for the real magnetic field magnitude is derived in 5.1 and corresponds to 234,05 mT.

$$MaxMagneticField\ [mT] = 32767\ [LSB]\ /\ 140\ [LSB/mT] = 234{,}05\ mT \tag{5.1}$$

The average maximum magnetic field value in the third subplot is around 100 mT, and the maximum possible magnetic field magnitude is 234,05 mT. Therefore, the values obtained after the converting function in the sensor driver are comparable to the expected ones.

5.1.2 Effect of the permanent magnets on the IMU

The design of the DLR tactile sensor makes the IMU very close to the side of the PCB with the Hall sensors and permanent magnets. Since the IMU also has an embedded magnetometer, needed to obtain the best results for the sensor fusion procedure, the permanent magnets should not influence the magnetic measurements. Therefore, a test was done to verify whether the permanent magnets embedded in the silicon on top of the Hall sensor taxels had an effect on the IMU magnetometer. The tactile sensor was pressed in a similar way with respect to the previous experiment, but this time data relative to the IMU magnetometer was recorded and plot. Figure 5.2 shows the results of the test.



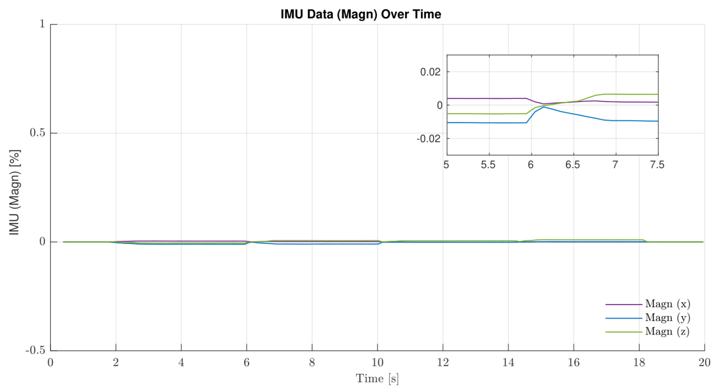


Figure 5.2: Effect of the permanent magnets on IMU magnetometer

The first subplot shows the magnetic Hall sensor data along the z coordinate, against the time. The second subplot shows instead the measurements of the IMU magnetometer along the three axes. The plot also shows a zoom on the area with biggest magnitude, to better read the corresponding y value. The two subplots are both normalized along the y axis with respect to the maximum magnetic flux density value for the Hall sensors. This is done to compare the plots with the same scaling. The comparison shows that, when pressing the Hall sensors along the z direction leading to the maximum value in magnitude (1), the maximum value for the data read by the IMU magnetometer is only 0.01. Therefore, the effect of the permanent magnets is negligible.

5.1.3 Sensor Sensitivity

The third test was done to measure the sensor sensitivity. In this case, the sensitivity is defined as the ratio between the force applied to the object (measured through the object weight), and the magnetic field magnitude measured by the sensor as a response to the applied force. The whole tactile sensor was pressed with objects of different weights. The weights, the corresponding force values, and the measured average field, are collected in Table 5.1. The table also shows the sensitivity coefficient, K, for each measurement.

Object Weight [g]	Object Weight [N]	Average Field[mT]	Coefficient K [N/mT]
17.000	0.167	0.025	6.637
26.000	0.255	0.066	3.849
43.000	0.421	0.212	1.989
79.000	0.774	0.285	2.718
106.000	1.039	0.503	2.066
164.000	1.607	0.759	2.117

Table 5.1: Sensor Sensitivity Table

The plot in Figure 5.3 translates the information of the table. Except for the smallest measurements, the sensitivity value is on average 2 [N/mT].

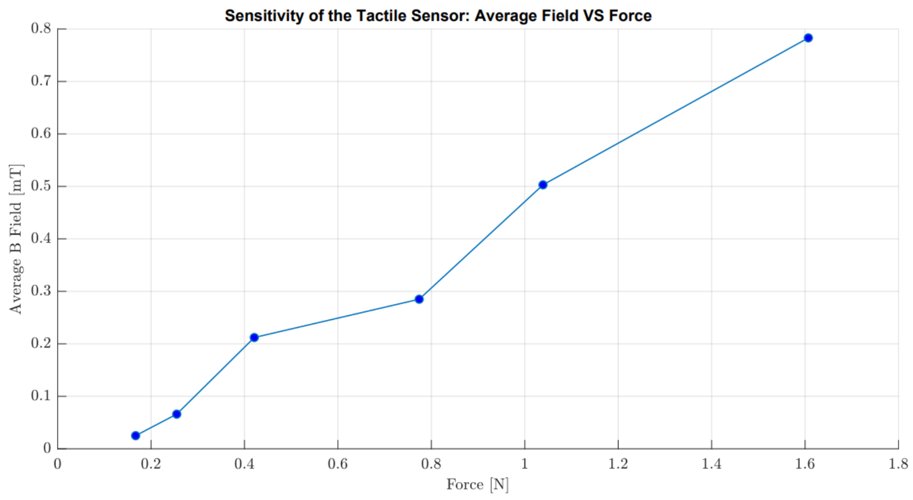


Figure 5.3: Sensor Sensitivity

5.1.4 Comparison with Xela Sensor Resolution

For the last test, the DLR tactile sensor was compared to a commercial tactile sensor from XELA Robotics, uSkin Patch model uSPa 44[1], whose datasheet is [25]. The XELA uSkin is a 4x4 taxels tri-axial tactile sensor module, and each individual sensing point measures three axis touch (x, y, z), just like DLR tactile sensor. However, uSkin does not have an integrated IMU sensor.

The uSPa44 has two different sensitivity settings, indicated by the letter "S" or "H". The standard model has sensitivity "H", so we will refer to this model for the following considerations. The details of the sensor's response, obtained by using one sensing point, so applying the force to one single taxel, can be seen in 5.4. This plot shows the uSkin raw output, measured in LSB, against the force, measured in Newtons.

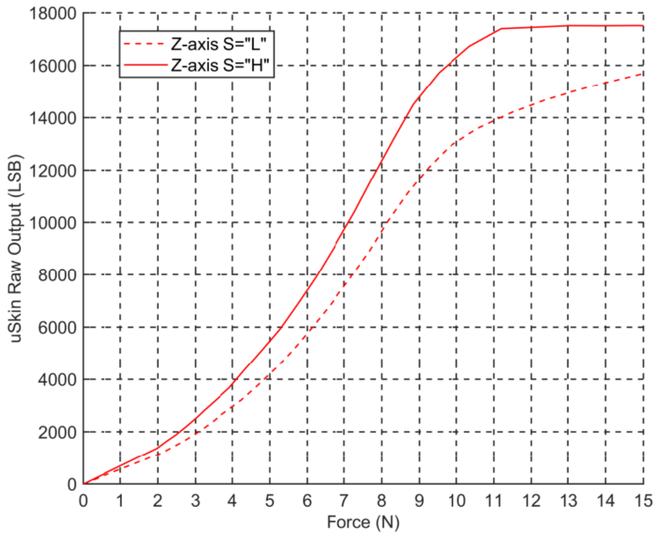
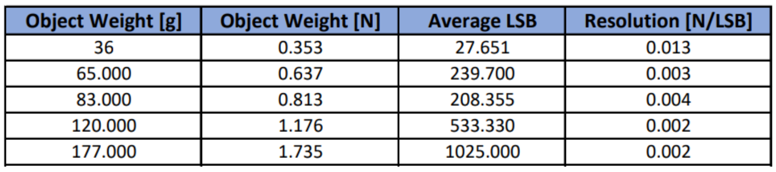


Figure 5.4: uSPa44 normal force (z-axis)

To compare the resolution of the XELA tactile sensor, a single taxel (front left) of the DLR tactile sensor was pressed with objects of different weights. The weights, the corresponding force values, and the measured average LSB, are collected in Table 5.2. The table also shows the resolution value, measured in [N/LSB], for each measurement. The small Table 5.3, instead, shows the maximum measurable range and the resolution values for the uSPa44 tactile sensor with default sensitivity. By simultaneously looking the two tables, the resolution values along the z axis are comparable and of the order of 0.002 [N/LSB].

[1]XELA uSPa 44 Tactile Sensor, `https://www.xelarobotics.com/sensor-collection/uspa-44`



Object Weight [g]	Object Weight [N]	Average LSB	Resolution [N/LSB]
36	0.353	27.651	0.013
65.000	0.637	239.700	0.003
83.000	0.813	208.355	0.004
120.000	1.176	533.330	0.002
177.000	1.735	1025.000	0.002

Table 5.2: Sensitivity of the sensor considering only the front left taxel

Sensitivity	Measurable Range				Resolution	
	Newton		LSB		N/LSB	
	x/y	z	x/y	z	x/y	z
H (default)	±1.3	11	±1400	17400	0.001	0.0018

Table 5.3: uSPa44 maximum measurable range and resolution

The similarity can be also shown by means of the plots. The first plot in Figure 5.5 shows the graphical representation of Table 5.2. The second plot is a snippet taken from Figure 5.4, more or less in the same range as the first one. From the graphical comparison of the two plots, the resolution slope is very similar.

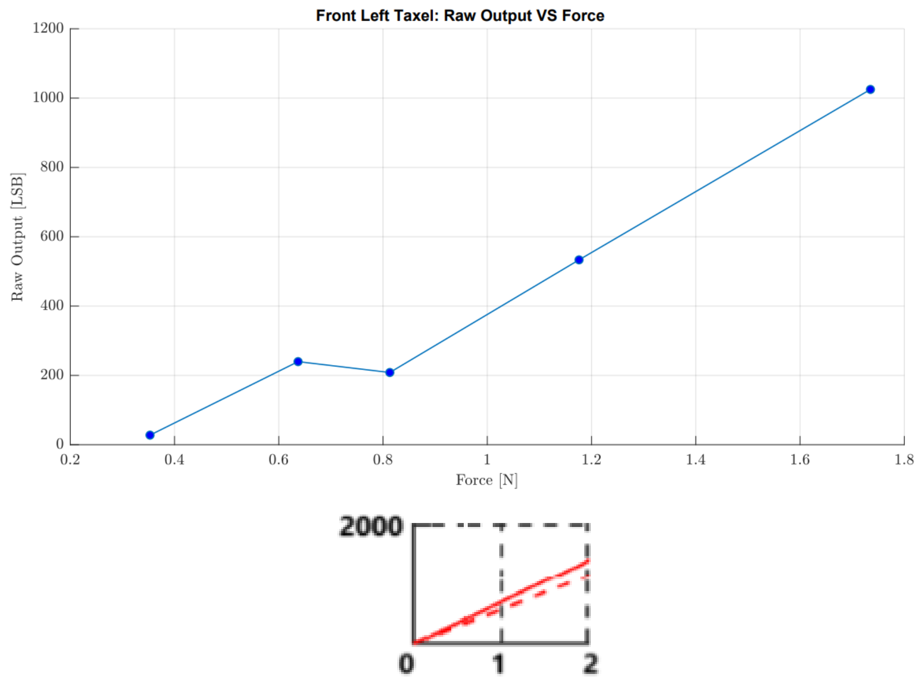


Figure 5.5: Sensitivity of the sensor considering only the front left taxel

5.2 IMU Sensor Characterization

This section is devoted to the characterization of the ICM20948 IMU sensor. The first thing that was done is the calibration of the sensor. This is needed to ensure the accuracy and reliability of the IMU data. The second step was comparing the measurements obtained by using the firmware written in Zephyr, and the ones obtained by using an ICM20948 already existing library. The last coparison was done between the ICM20948 sensor, whose data was read by using the firmware, and the ISM330DHCX 6 dof IMU sensor paired with the MMC5983MA magnetometer.

5.2.1 IMU Calibration

Raw data from IMU sensors often contain errors that can lead to significant inaccuracies in the measured data. Moreover, to use the IMU data in the best possible way, so trying to get the best part out of all the sensors composing the IMU, sensor fusion algorithms needed to be applied (see chapter 6, but if the calibration is not performed their outcome is not usable. Therefore, the calibration procedure is essential.

The sensor calibration procedure was performed by using the `ICM_20948-AHRS` Github repository from jremington, [26]. The repository contains a Mahony Attitude and Heading Reference System (AHRS) 3D Fusion Filter for Arduino and the ICM20948 sensor, and a calibrate3.py file to perform sensor calibration. This python file optionally replaces a C program called Magneto which performs calibration. The library is written and tested for the Sparkfun breakout board shown in 4.12, using I^{2}C connection on an Arduino Pro Mini. In this thesis, however, the target board used is the Teensy 4.0 shown in 3.3.

Sensors Measurement Model

The sensors measurement models describe how the data measured from the sensor relates to the true quantities. In fact, bias, scale and alignment errors can be typically found in sensor readings. Biases are constant offsets in the sensor readings and they are estimated by considering the mean of the measurements under static conditions. They can be therefore subtracted from the measured data to obtain the true one. Instead, errors due to scale factors and non-orthogonality of the sensor axes are corrected by means of a matrix transformation, calculated based on data from controlled rotation and 3D space sampling. Calibration aims at determining the bias vectors $(b_{acc}, b_{gyro}, b_{magn})$ and the correction matrices $(M_{acc}, M_{gyro}, M_{magn})$ for each sensor.

The measurement models, which take into account the presence of these errors, for each sensor composing the IMU are described in the following.

Accelerometer The accelerometer measures acceleration (a) along the three axes. The relationship between the real and measured acceleration data is shown below in 5.2.

$$a_{measured} = M_{acc}\,(a_{true} + b_{acc}) \tag{5.2}$$

Gyroscope The gyroscope measures angular velocities (ω) along the three axes. The relationship between the real and measured angular velocity data is shown below in 5.3.

$$\omega_{measured} = M_{gyro} \left(\omega_{true} + b_{gyro}\right) \tag{5.3}$$

Magnetometer The magnetometer measures the magnetic field vector magnitude (B). From a geometric point of view, magnetometer data samples should form a sphere. However, usually they form an ellipsoid[2]. This happens because of two contributions:

- Hard Iron Distorsion: caused by permanent magnetic objects near the sensor (like ferrous materials). It results in an offset in the magnetometer readings, which are modeled by the bias vector b_{magn}.
- Soft Iron Distortion: caused by nearby magnetic materials that distort the magnetic field, leading to an elliptical response. This contribution is corrected by using the matrix M_{mag}.

The ellipsoid problem occurs also for the accelerometer sensor, although it's less prominent. It does not happen in the gyroscope case. The relationship between the real and measured magnetic field data is shown in 5.4.

$$B_{measured} = M_{magn} \left(B_{true} + b_{magn}\right) \tag{5.4}$$

The equations to find the true measurements for accelerometer, gyroscope, and magnetometer sensors are respectively shown in the following.

$$a_{true} = M_{acc}^{-1} \left(a_{measured} - b_{acc}\right) \tag{5.5}$$

$$\omega_{true} = M_{gyro}^{-1} \left(\omega_{measured} - b_{gyro}\right) \tag{5.6}$$

$$B_{true} = M_{magn}^{-1} \left(B_{measured} - b_{magn}\right) \tag{5.7}$$

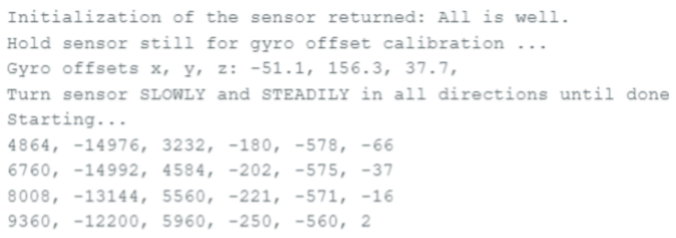
Calibration Procedure

The steps for using the library for calibration purposes is described in the following. The `ICM_20948_get_cal_data.ino` and the calibrate3.py files are used to complete the procedure and obtain the bias vectors and correction matrices.

First, the Arduino code file `ICM_20948_get_cal_data.ino` is deployed on the board. The instructions printed on the serial monitor must be followed. The sensor has to be kept still while the program collects gyroscope data and computes the gyroscope bias vector b_{gyro}. The same program then records 300 accelerometer and magnetometer data points, while

[2]Magnetometer Calibration, `https://teslabs.com/articles/magnetometer-calibration/`

the user slowly and steadily rotates the sensor in all directions. This data is copied from the serial monitor output, and it is used to create two different Comma Separated Value (CSV) files containing accelerometer and magnetometer data respectively. A snippet of the serial monitor content described is shown in Figure 5.6.



```
Initialization of the sensor returned: All is well.
Hold sensor still for gyro offset calibration ...
Gyro offsets x, y, z: -51.1, 156.3, 37.7,
Turn sensor SLOWLY and STEADILY in all directions until done
Starting...
4864, -14976, 3232, -180, -578, -66
6760, -14992, 4584, -202, -575, -37
8008, -13144, 5560, -221, -571, -16
9360, -12200, 5960, -250, -560, 2
```

Figure 5.6: Serial Monitor Output

The two CSV files are then loaded one at a time into the calibrate3.py program file, through the command `data = np.loadtxt("<sensor>_data.csv",delimiter=',')`. First, the accelerometer data file is loaded and the values for b_{acc} and the inverse of M_{acc} are obtained. Then, the same is done with the magnetometer data file, and the values for b_{magn} and the inverse of M_{magn} are obtained. The calibrate3.py code directly gives the inverted calibration matrices.

When the calibrate3.py file is run, the distorted ellipsoid coming from the plot of the uncalibrated data points, is shown. An example of this ellipsoid is visible in the following Figure 5.7.

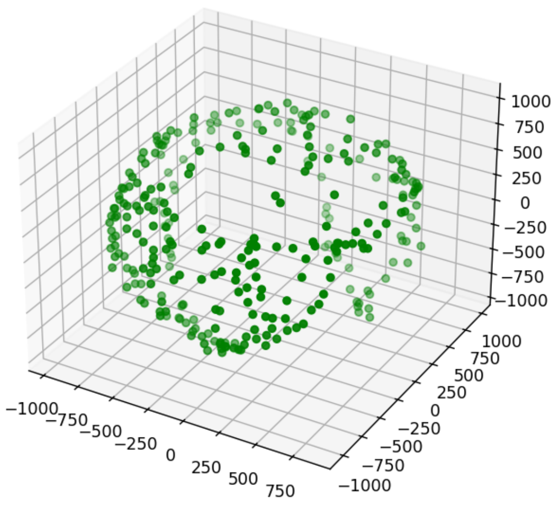


Figure 5.7: Data Points Ellipsoid

The parameters obtained after the calibration procedure are then directly used inside the equations 5.5, 5.6, and 5.7, to find the real calibrated measurements.

5.2.2 IMU Data Comparison Between Library and Firmware

After the calibration procedure, the data measured by the firmware was compared to the data obtained by using the same library used for calibration purposes, which uses the Sparkfun ICM_90248 Arduino library[3].

Taking as a reference system the one showed in 4.12b, the test made consisted in rotating the sensor in different ways. The sequence of rotations consisted in +90 and -90 degrees around x axis, -90 and +90 degrees around y axis, and 90 and -90 degrees around z axis. The expected behavior for each sensor is explained in the following:

- *Accelerometer*: the gravitational acceleration starts with a value of 9.8 m/s^2 along the z axis. When the sensor is rotated of 90 degrees around the x axis, the gravitational acceleration moves along the y axis, and when the sensor is rotated of 90 degrees around the y axis, the vector lies along the x axis;

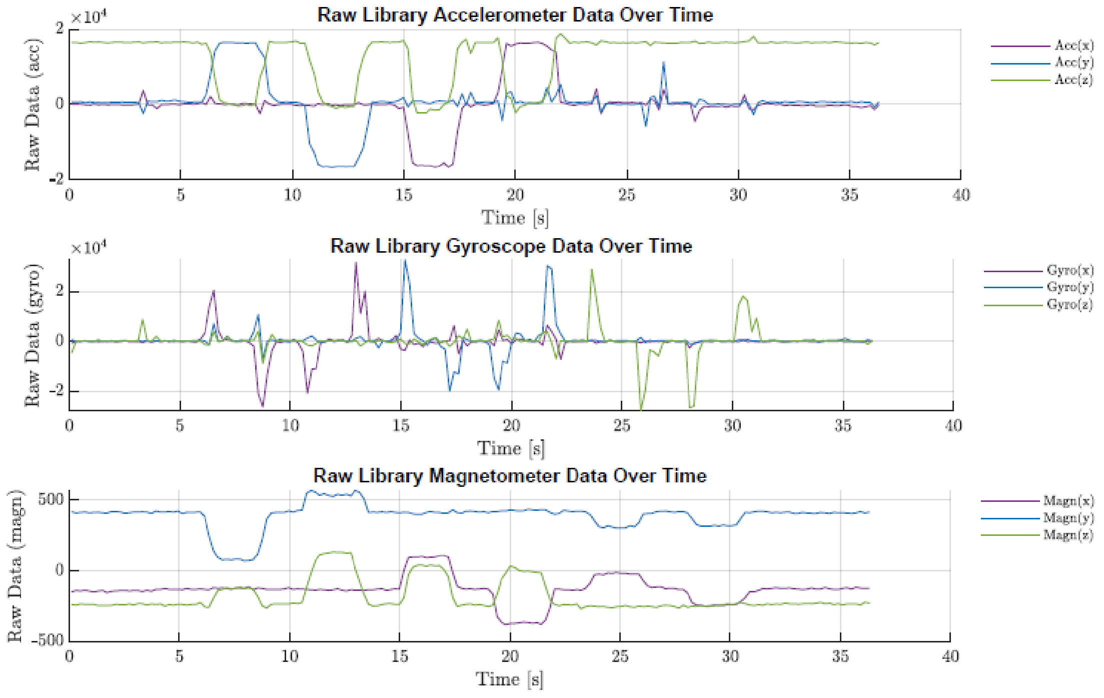
- *Gyroscope*: a spike should be visible in correspondence to each rotation;

- *Magnetometer*: when a rotation is made around the x axis, the magnetic measurement along that same axis should be constant. Same happens for y and z axes.

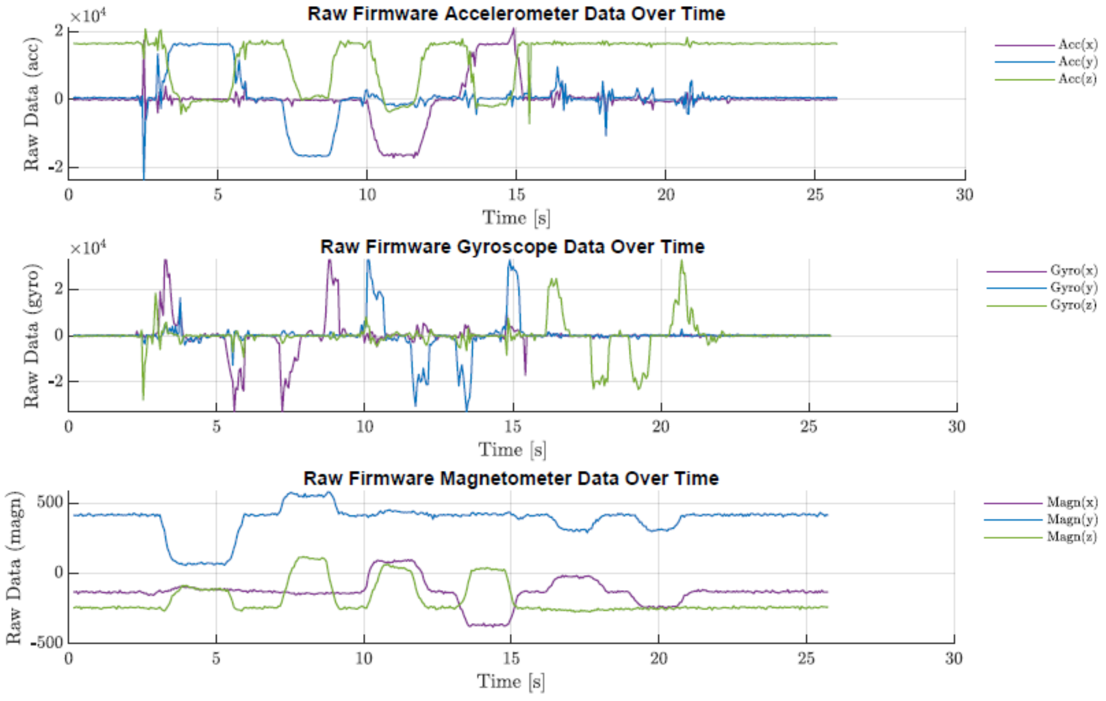
The raw data between the already existing and working library and the firmware, were compared. Figure 5.8a shows the raw data acquired by using the library, while Figure 5.8b shows the raw data acquired by using the firmware. It should be noted that the raw data are the same in terms of behavior and magnitude.

Then, a similar comparison was done between the calibrated data acquired with the library code and with the firmware one. Figure 5.9a shows the calibrated data acquired by using the library, while Figure 5.9b shows the calibrated data acquired by using the firmware. Also in this case the calibrated data are the same in terms of behavior and magnitude.

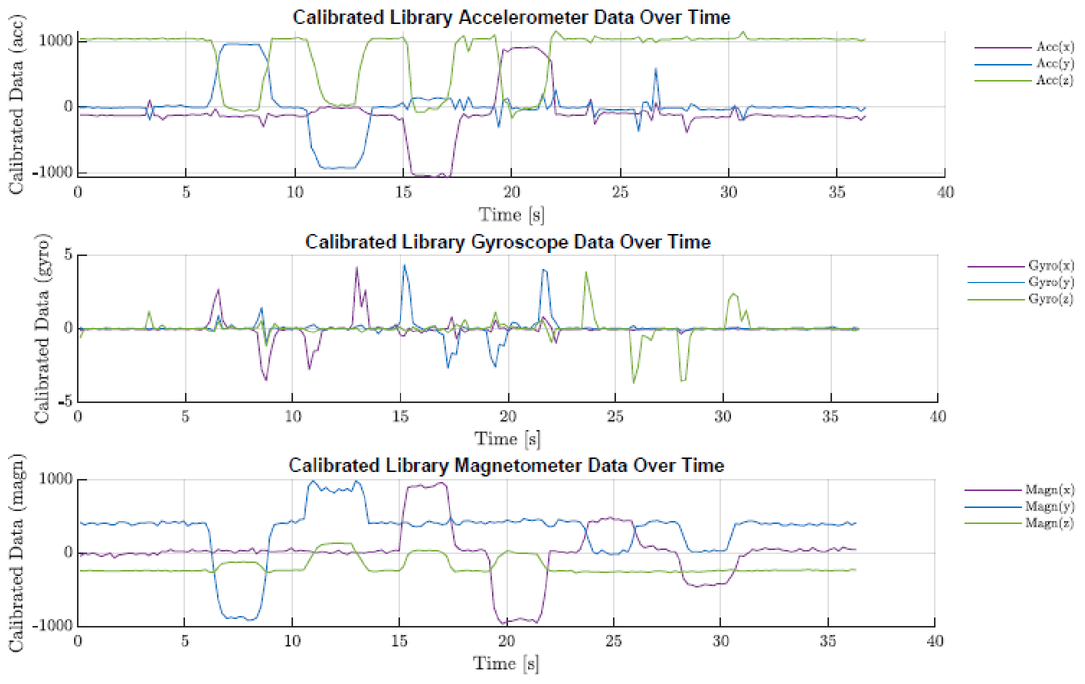
[3]https://www.arduinolibraries.info/libraries/spark-fun-9-do-f-imu-breakout-icm-20948-arduino-library



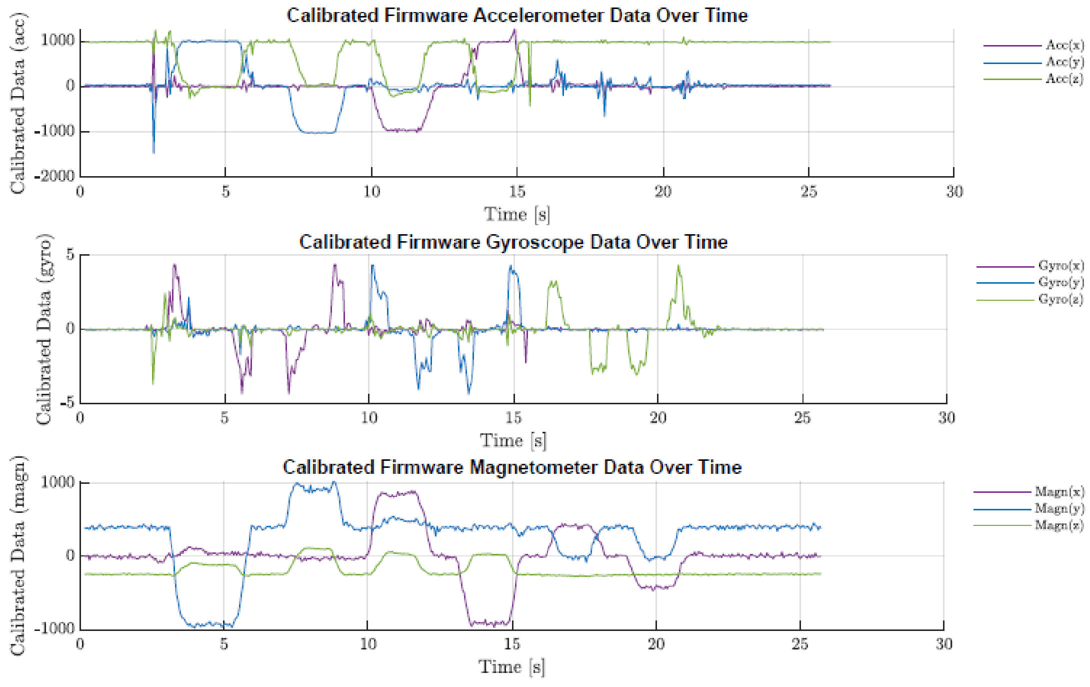
(a) Library Raw Data



(b) Firmware Raw Data



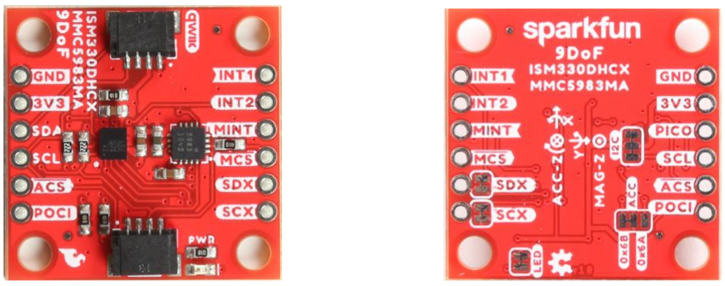
(a) Library Calibrated Data



(b) Firmware Calibrated Data

5.2.3 Comparison with benchmark sensors

The last tests done for the ICM20948 IMU sensor characterization consist in the comparison with two benchmark sensors whose drivers are already implemented in Zephyr RTOS. The two sensors are the ISM330DHCX 6 dof IMU and the MMC5983MA magnetometer. The hardware used for these tests consists of the ICM20948 Sparkfun breakout board in 4.12a, and the ISM330DHCX + MMC5983MA Sparkfun breakout board in 5.10, placed one on top of the other and fixed to each other with some screws and nuts to ensure the same movement of both the boards. Before being able to compare the data measured by both sensors, the reference systems, shown in 4.12b and 5.10b, were aligned.



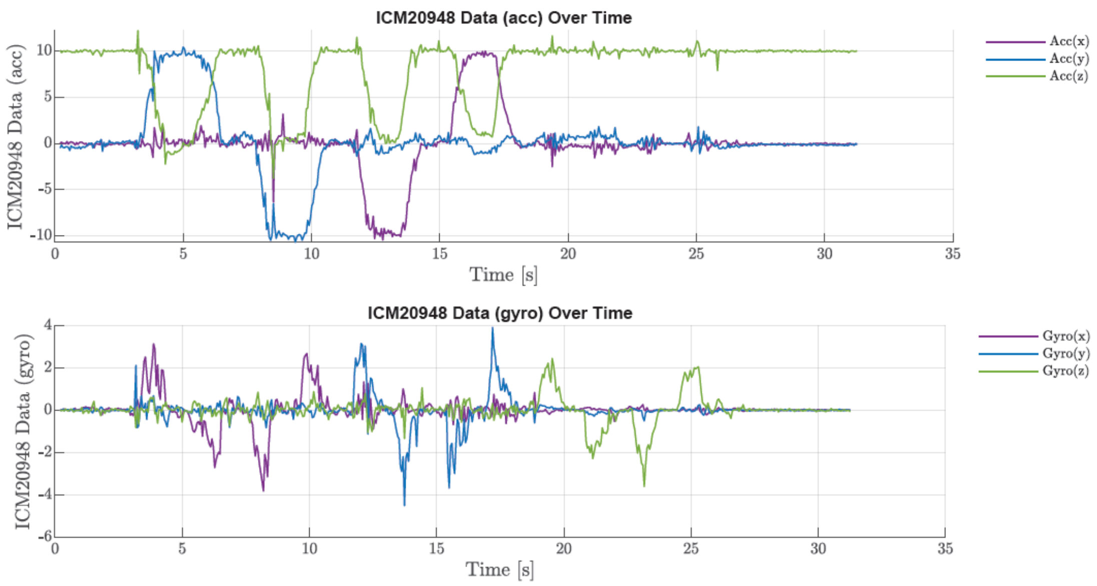
(a) Front (b) Back

Figure 5.10: SparkFun ISM330DHCX, MMC5983MA Breakout Board

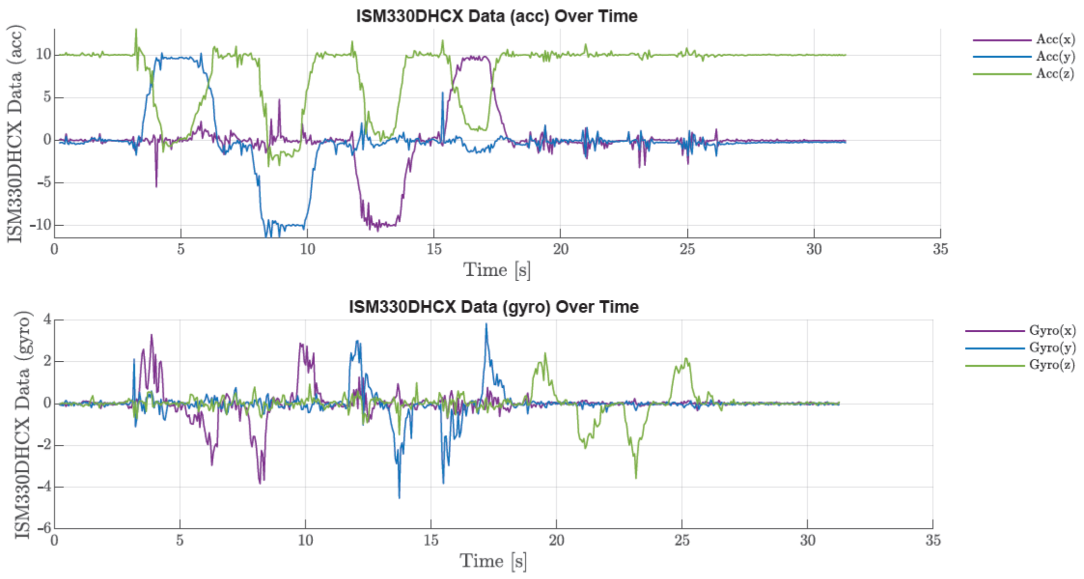
The same tests described in the previous paragraph 5.2.2 were reproduced for this setup.

Plots 5.11a and 5.11b show the comparison of the data measured by the accelerometer and gyroscope sensors inside the IMUs. They show the same data both in terms of behavior, which is the expected one, and of magnitude.

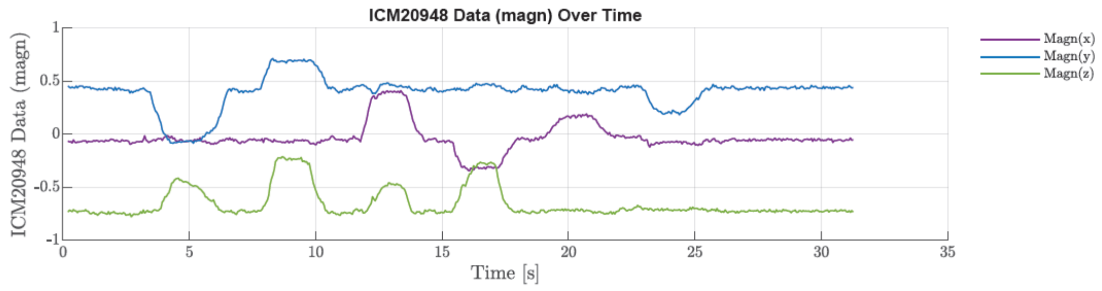
Plots in 5.12a and 5.12b, instead, show the comparison between the two magnetometers. In this case, data is also the same in terms of behavior and magnitude, but the offsets are different due to the missing calibration procedure for both the magnetometer sensors.



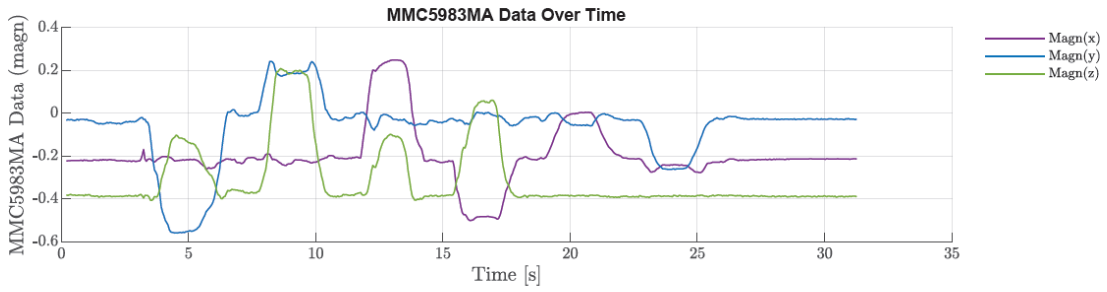
(a) ICM20948 Acc Gyro Data



(b) ISM330DHCX Data



(a) ICM20948 Magnetometer Data



(b) MMC5983MA Data

Chapter 6

Sensor Fusion

Sensor fusion is the process of combining data coming from different sensors to produce a more accurate and reliable representation of a system's state than the one that can be obtained by using single sensors alone. In IMUs context, no single accelerometer, gyroscope, or magnetometer can provide complete data about orientation in all conditions. Each sensor has strengths and weaknesses, and sensor fusion produces a more robust, accurate, and stable estimate by combining the sensor outputs to overcome individual limitations.

The three types of sensors already present in the ICM20948 IMU are commonly used for orientation estimation. Gyroscopes measure angular velocity. If the initial conditions are known, it may be integrated over time to obtain the sensor's orientation. Since the gyroscope measurement errors are integrated, errors will accumulate in the calculated orientation. Therefore, gyroscopes alone cannot provide an absolute measurement of orientation. Accelerometers and magnetometers will measure the earth's gravitational and magnetic fields respectively and so provide an absolute reference of orientation. However, they are likely to be subject to high levels of noise.

The task of sensor fusion algorithms, and therefore of an orientation filter, is to compute a single, accurate, and robust estimate of orientation through the optimal fusion of the measurements from these sensors maintaining a low computational cost for running on the onboard processor.

6.1 Mahony Filter

Among the existing sensor fusion algorithms, the ones with the best balance between computational efficiency, accuracy, and real-time capability are a few: Madgwick filter, Mahony filter, Complementary filter[1]. Based on the comparative analysis reported in [27], where the authors compare three sensor fusion algorithms for a setup that is similar

[1]IMU Data Fusing: Complemetary, Kalman, and Mahony Filter, `https://www.olliw.eu/2013/imu-data-fusing/`

to the one described in this thesis and suitable for real-time operation, the best filter in terms of computational efficiency, which is a very critical property in embedded systems applications, is the Mahony filter. Therefore, it was chosen to be implemented among the others.

The Mahony filter was first described by Mahony et al. in [28]. The filter's simplicity and efficiency have made it a popular choice for embedded systems, especially in applications like robotics and aerial navigation. The algorithm related to the Mahony filter is shown below. K_i and K_p are parameters to be chosen based on the filter wanted performance.

Algorithm Mahony Filter Algorithm

```
1: Set algorithm coefficients $K_i, K_p$ and initialize quaternion $q_1 = 1, q_2 = q_3 = q_4 = 0$
   while: sensor data is available
2: Read accelerometer measurements $a_x, a_y, a_z$ and gyroscope measurements $g_x, g_y, g_z$
3: Compute orientation error from accelerometer data, where $e_{i,t}$ represents the integral
   error of the measurements at time $t$:
       $e_{t+1} = \begin{bmatrix} a_{x,t} \\ a_{y,t} \\ a_{z,t} \end{bmatrix} \times \begin{bmatrix} 2(q_2 q_4 - q_1 q_3) \\ 2(q_1 q_2 + q_3 q_4) \\ (q_1^2 - q_2^2 - q_3^2 + q_4^2) \end{bmatrix}$
       $e_{i,t+1} = e_{i,t} + e_{t+1} \Delta t$
4: Update angular velocity computed from gyroscope with the $K_i$ and $K_p$ terms using
   feedback (fusion):
       $\omega_{t+1} = \omega_t + K_p e_{t+1} + K_i e_{i,t+1}$
5: Compute orientation increment from gyroscope measurements:
       $\dot{q}_{\omega,t+1} = \frac{1}{2} \hat{q}_t \otimes [0, \omega_{t+1}^T]$
6: Numerical integration:
       $q_{t+1} = \hat{q}_t + \Delta t \dot{q}_{\omega,t+1}$
   endwhile
```

Mahony Filter in zsclib

The Zephyr RTOS framework used has already implemented a library that, among many other scientific computing and data analysis features, implements various sensor fusion algorithms. This is the *Zephyr Scientific Library (zsclib)* and its Github repository is at [29].

All orientation fusion algorithms in *zscilib* use the same interface, therefore it's trivial to switch between different algorithms. The interface for the sensor fusion algorithms will be shown in the following taking as example the Mahony filter.

The Mahony filter can be instantiated and configured with the following piece of code, placed in the source code file.

```
/* Config settings for the Mahoney filter. */
static zsl_real_t _mahn_intfb[3] = { 0.0, 0.0, 0.0 };

static struct zsl_fus_mahn_cfg mahn_cfg = {
        .kp = 50,
        .ki = 0,
        .integral_limit = 10000.0,
        .intfb = {
                .sz = 3,
                .data = _mahn_intfb,
        },
};

static struct zsl_fus_drv mahony_drv = {
        .init_handler = zsl_fus_mahn_init,
        .feed_handler = zsl_fus_mahn_feed,
        .error_handler = zsl_fus_mahn_error,
        .config = &mahn_cfg,
};
```

The filter can then be initialized by using the `init_handler` function, and fed by using the `feed_handler` function. Both the functions are shown below.

```
struct zsl_fus_drv *drv = &mahony_drv;

/* Initialize the filter at 100 Hz*/
drv->init_handler(100, drv->config);

/* Feed the filter */
drv->feed_handler(&av_cal, &mv_cal, &gv_cal, &incl, &quat, drv->config);
```

Mahony Filter Implementation

After performing the calibration procedure on the ICM20948 IMU sensor, checking that the units of measurement of the gyroscope data were the right ones (accelerometer and magnetometer vectors are normalized), the implementation of the Mahony filter inside the application was done. Before testing the filter, the correctness of the input data was checked by using the external library [26] also used for the calibration procedure. The same test set was applied for two different data acquisitions, first with the library and then with the firmware (the code which also implements the Mahony filter in Zephyr RTOS). However, the filter did not give satisfactory results.

A different attempt was then made at using the Arduino code for the Mahony AHRS filter in [26], directly as a C function implementing the filter algorithm. Although the code used was the same as the one in the `MahonyQuaternionUpdate` function inside the `ICM_20948_Mahony.ino` file, and the data management was also the same, this attempt was also not satisfactory.

Chapter 7

Conclusions and Future Work

This chapter presents a conclusive summary of the thesis, highlighting its key contributions to addressing the pose estimation challenges in robotic grasping. The chapter outlines potential directions for future research, positioning this work as a foundational step toward further improving pose estimation methods and expanding the capabilities of robotic systems in grasping and manipulation tasks.

7.1 Conclusions

This thesis has addressed key challenges in robotic grasping and manipulation by developing and evaluating a novel multi-modal tactile sensing system, leveraging magnetic and inertial sensing for enhanced pose estimation. Building on the insights from state-of-the-art tactile sensing technologies, this work introduced an integrated system combining Hall-effect sensors and a 9-degree-of-freedom IMU to provide real-time feedback on forces and orientations.
The proposed sensor system was complemented by the custom firmware detailed in Chapter 4, which included the development of the drivers for the MLX90395 Hall sensor and the ICM20948 IMU within the Zephyr RTOS framework. The real-time needs of the robotic system were met by leveraging Zephyr RTOS features like thread priorities and synchronization.
Experimental evaluations highlighted the sensor's performance, addressing challenges such as magnetic interference and drift, while validating its robustness against existing benchmarks. An extensive study on possible effective sensor fusion algorithms for the IMU was carried out, and attempts for successful implementation of the chosen algorithm, Mahony filter, were made by using different methodologies.

7.2 Future Work

This section provides a list of recommendations for future work that must be taken into consideration to further enhance the capabilities and applications of the developed multimodal tactile sensing system, building on the foundational contributions of this thesis.

Implementing Sensor Fusion Algorithm

A key direction for future work is the further optimization and implementation of the chosen sensor fusion algorithm, the Mahony filter, to fully harness the potential of IMU data and effectively integrate it with tactile data from the Hall-effect sensors. By combining the tactile force measurements with quaternion information derived from the sensor fusion process, the system can achieve enhanced accuracy and robustness in pose estimation for robotic grasping applications. This approach aims to maximize the synergy between tactile and inertial data, providing a comprehensive and precise feedback mechanism that can significantly improve the performance of robotic manipulation tasks in complex environments.

Test the Firmware on the Real Robot

Another important direction for future work is testing the developed tactile sensor system on a real robotic platform. While the experimental evaluations conducted in this thesis demonstrated the sensor's performance in controlled scenarios, deploying it on a real robot would provide valuable insights into its practical reliability and effectiveness in dynamic environments. This step would allow for assessing the sensor's integration with robotic controllers, its responsiveness during real-time tasks, and its robustness in handling diverse objects and grasp conditions, ultimately bridging the gap between experimental validation and real-world applications.

Using multiple tactile sensors on a robotic hand

A promising direction for future work is the integration of multiple tactile sensors across the same robotic hand to enable comprehensive force and pose feedback for multi-fingered manipulation tasks. The modular design of the Zephyr RTOS, employed in this thesis, provides a scalable framework for managing multiple sensors efficiently. Its real-time capabilities and support for multi-threading allow seamless communication and coordination among sensors, ensuring synchronized data acquisition and processing. This scalability could facilitate the development of more advanced robotic systems capable of complex object handling and adaptive manipulation in dynamic environments.

Appendix A

I^2C Communication Protocol

This appendix describes the Inter-Integrated Circuit (I^2C) communication protocol. In fact, for the Teensy 4.0 board to communicate with the sensors and obtain data, the I^2C communication protocol[1] is used. For sensor applications, I^2C is particularly valuable due to its simplicity and efficiency in handling multiple devices with minimal wiring.

The Inter-Integrated Circuit (I^2C) protocol is a widely used, synchronous serial communication protocol, meaning that data bits are transferred one by one at regular intervals of time set by a reference clock line. It allows up to 1008 slave devices to communicate with a master device over just two wires: a data line (SDA) and a clock line (SCL). The scheme showing the communication between the master and the generic slave devices is shown in A.1.

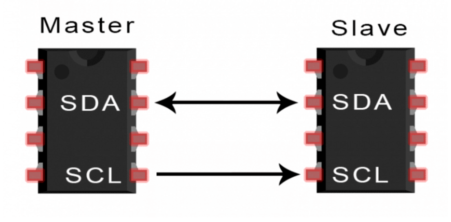


Figure A.1: Master-Slave Communication

Both the I^2C bus lines are operated as open drain drivers. This means that any device on the I^2C network can drive SDA and SCL low, but they cannot drive them high. So, a pull-up resistor is used for each bus line, to keep them high, at positive voltage, by default. This concept is shown in A.2. By using this system there will be no chances of shorting, which might happen when one device tries to pull the line high and some other device tries to pull the line low.

[1]Basics of I2C Communication | Hardware, Data Transfer, Configuration, `https://www.electronicshub.org/basics-i2c-communication/`

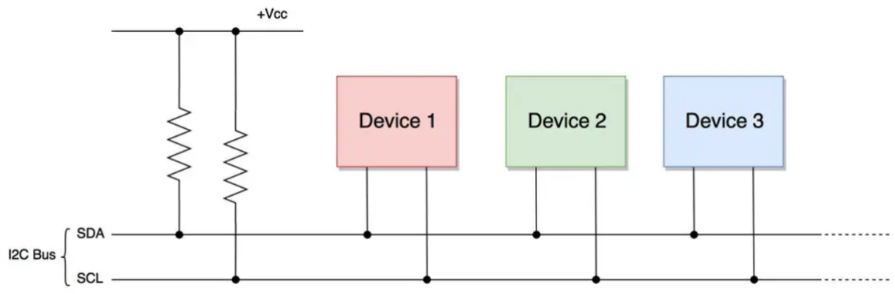


Figure A.2: Multiple Devices Communication

Designed for short-range, low-speed communication within electronic devices, I^{2}C operates by having the master device initiate communication, control the clock signal, and send or request data from designated slave devices through 7-bit or 10-bit addresses which are different for each slave device. The uniqueness of the 7-bit addresses and a more complete scheme for the Inter-Integrated Circuit (I^{2}C) communication protocol is visible in A.3.

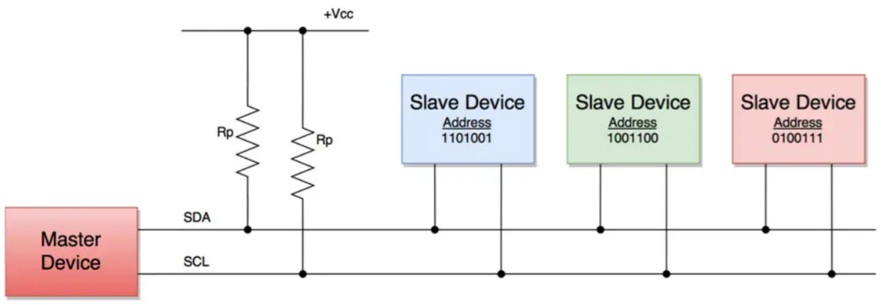


Figure A.3: Master-Slave Addresses

The data transfer uses a series of bit-based acknowledgments to ensure data integrity, and devices can be configured to communicate at different speeds, typically up to 400 kHz in standard applications or up to 3.4 MHz in high-speed mode.

Appendix B

Debugging Tools

This appendix shows the tools used in the thesis work for debugging purposes, and shows both the hardware components and the graphical tools.

Debugging is about ensuring that a system works as intended. This involves identifying and fixing errors, optimizing performance, and enhancing stability. It is a crucial aspect of embedded systems development. In fact, those systems are often deployed in resource-constrained environments, operate in real-time, and interact directly with hardware, making them particularly susceptible to complex and hard-to-find errors. Effective debugging is essential to ensure system performance, reliability, and safety.

The debugging system considered here is composed of:

- Target board, on which the code to analyze is deployed;
- Debug probe, which acts as an intermediary between the development environment on the host computer, and the target device;
- Graphical Debugger, which provides a user-friendly, visual interface for debugging the application;
- Oscilloscope, which uses triggers to capture data and verify if the firmware sends/receives correct instructions.

MIMRX1060-EVK Target Board

The target board used for debugging purposes is the MIMRX1060-EVK board, shown in B.1. This is an evaluation kit based on i.MX RT1060, an ultra-low-power, high-performance crossover processor by NXP Semiconductor. The MIMXRT1060-EVK board includes various peripherals and interfaces and multiple communication options like USB, Ethernet, and CAN. It's compatible with J-Link for more advanced debugging.

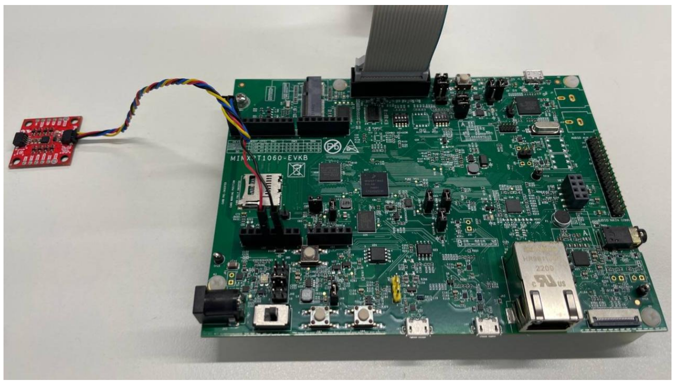


Figure B.1: MIMRX1060-EVK Target Board

J-Link PRO Debug Probe

The main tool used for debugging is the J-Link Pro debug probe from SEGGER. This high-performance debugging tool is widely used in embedded systems development due to its versatility, speed, and compatibility with a range of microcontrollers. It supports a broad range of development tools and software, including SEGGER's Ozone, which will be described in the next subsection. It's a multi-platform solution, making it adaptable to varied development environments.The debug probe is shown in B.2, being the black device on the left, connected to the board with a gray JTAG cable.

Complete Debug Hardware

The complete debug hardware is shown in B.2. The picture shows the JLink Pro Debugger probe connected to the chosen board, the MIMXRT1060-EVK. The specific sensor to be tested is connected to the board.

Ozone Graphical Debugger

The Ozone Debugger by SEGGER is a powerful, stand-alone graphical debugger designed for embedded systems, providing developers with comprehensive tools for debugging, performance analysis, and code optimization. It supports source-level debugging in C, C++, and Rust, as well as assembly instruction debugging. Ozone is also multi-platform. The tool integrates tightly with SEGGER's J-Link and J-Trace debug probes, enabling high-speed programming and the use of powerful built-in features.

Figure B.3 shows the default Ozone screen. The screen is divided in two horizontal strips. The first one comprises, from left to right, the source files list and breakpoints window, the source code viewer, where the code to analyze is shown and breakpoints can be set, the disassembly, local and global data windows, and the field showing the content of the registers. The second horizontal strip contains the console and the memory content.

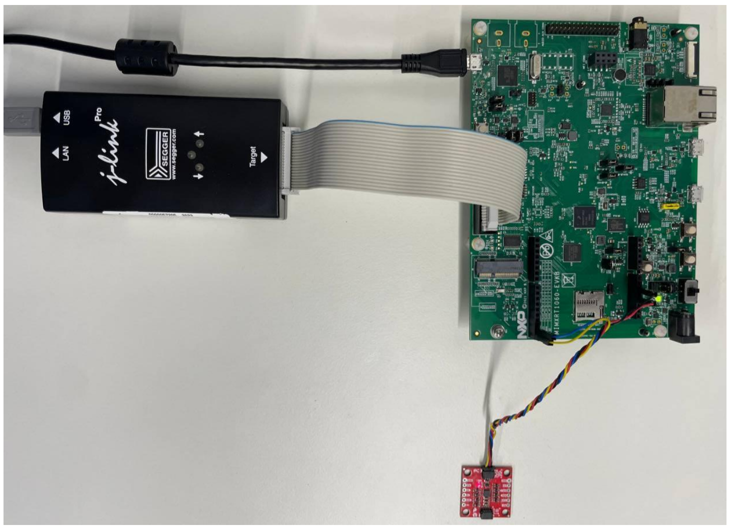
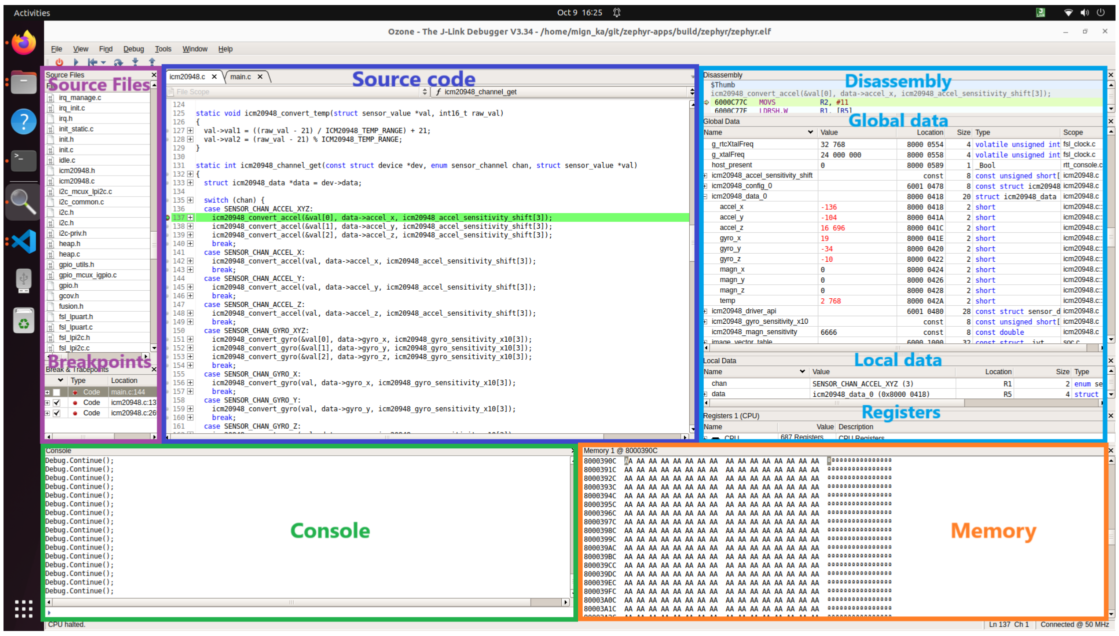


Figure B.2: Debug Hardware



Source Files

```
irq_manage.c
irq_init.c
irq.h
init_static.c
init.h
init.c
idle.c
icm20948.h
icm20948.c
i2c_mcux_lpi2c.c
i2c_common.c
i2c.h
i2c.h
i2c-priv.h
heap.h
heap.c
gpio_utils.h
gpio_mcux_igpio.c
gpio.h
gcov.h
fusion.h
fsl_lpuart.h
fsl_lpuart.c
fsl_lpi2c.h
fsl_lpi2c.c
```

Break & Tracepoints

	Type	Location
	Code	main.c:144
✓	Code	icm20948.c:13
✓	Code	icm20948.c:26

icm20948.c | main.c — File Scope — icm20948_channel_get

```c

static void icm20948_convert_temp(struct sensor_value *val, int16_t raw_val)
{
  val->val1 = ((raw_val - 21) / ICM20948_TEMP_RANGE) + 21;
  val->val2 = (raw_val - 21) % ICM20948_TEMP_RANGE;
}

static int icm20948_channel_get(const struct device *dev, enum sensor_channel chan, struct sensor_value *val)
{
  struct icm20948_data *data = dev->data;

  switch (chan) {
  case SENSOR_CHAN_ACCEL_XYZ:
    icm20948_convert_accel(&val[0], data->accel_x, icm20948_accel_sensitivity_shift[3]);
    icm20948_convert_accel(&val[1], data->accel_y, icm20948_accel_sensitivity_shift[3]);
    icm20948_convert_accel(&val[2], data->accel_z, icm20948_accel_sensitivity_shift[3]);
    break;
  case SENSOR_CHAN_ACCEL_X:
    icm20948_convert_accel(val, data->accel_x, icm20948_accel_sensitivity_shift[3]);
    break;
  case SENSOR_CHAN_ACCEL_Y:
    icm20948_convert_accel(val, data->accel_y, icm20948_accel_sensitivity_shift[3]);
    break;
  case SENSOR_CHAN_ACCEL_Z:
    icm20948_convert_accel(val, data->accel_z, icm20948_accel_sensitivity_shift[3]);
    break;
  case SENSOR_CHAN_GYRO_XYZ:
    icm20948_convert_gyro(&val[0], data->gyro_x, icm20948_gyro_sensitivity_x10[3]);
    icm20948_convert_gyro(&val[1], data->gyro_y, icm20948_gyro_sensitivity_x10[3]);
    icm20948_convert_gyro(&val[2], data->gyro_z, icm20948_gyro_sensitivity_x10[3]);
    break;
  case SENSOR_CHAN_GYRO_X:
    icm20948_convert_gyro(val, data->gyro_x, icm20948_gyro_sensitivity_x10[3]);
    break;
  case SENSOR_CHAN_GYRO_Y:
    icm20948_convert_gyro(val, data->gyro_y, icm20948_gyro_sensitivity_x10[3]);
    break;
  case SENSOR_CHAN_GYRO_Z:
```

Disassembly

```
$Thumb
icm20948_convert_accel(&val[0], data->accel_x, icm20948_accel_sensitivity_shift[3]);
6000C77C   MOVS        R2, #11
6000C77E   LDRSH.W     R1, [R5]
```

Global Data

Name	Value	Location	Size	Type	Scope
g_rtcXtalFreq	32 768	8000 0554	4	volatile unsigned int	fsl_clock.c
g_xtalFreq	24 000 000	8000 0558	4	volatile unsigned int	fsl_clock.c
host_present	0	8000 0589	1	_Bool	rtt_console.c
icm20948_accel_sensitivity_shift		const	8	const unsigned short[	icm20948.c
icm20948_config_0		6001 0478	8	const struct icm20948	icm20948.c
icm20948_data_0		8000 0418	20	struct icm20948_data	icm20948.c
accel_x	-136	8000 0418	2	short	icm20948.c:
accel_y	-104	8000 041A	2	short	icm20948.c:
accel_z	16 696	8000 041C	2	short	icm20948.c:
gyro_x	19	8000 041E	2	short	icm20948.c:
gyro_y	-34	8000 0420	2	short	icm20948.c:
gyro_z	-10	8000 0422	2	short	icm20948.c:
magn_x	0	8000 0424	2	short	icm20948.c:
magn_y	0	8000 0426	2	short	icm20948.c:
magn_z	0	8000 0428	2	short	icm20948.c:
temp	2 768	8000 042A	2	short	icm20948.c:
icm20948_driver_api		6001 0480	28	const struct sensor_d	icm20948.c
icm20948_gyro_sensitivity_x10		const	8	const unsigned short[	icm20948.c
icm20948_magn_sensitivity	6666	const	8	const double	icm20948.c
image_vector_table		6000 1000	32	const struct _ivt	soc.c

Local Data

Name	Value	Location	Size	Type
chan	SENSOR_CHAN_ACCEL_XYZ (3)	R1	2	enum se
data	icm20948_data_0 (0x8000 0418)	R5	4	struct

Registers 1 (CPU)

Name	Value	Description
CPU	687 Registers	CPU Registers

Console

```
Debug.Continue();
Debug.Continue();
Debug.Continue();
Debug.Continue();
Debug.Continue();
Debug.Continue();
Debug.Continue();
Debug.Continue();
Debug.Continue();
Debug.Continue();
Debug.Continue();
Debug.Continue();
Debug.Continue();
Debug.Continue();
Debug.Continue();
Debug.Continue();
```

Memory 1 @ 8000390C

```
8000390C  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
8000391C  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
8000392C  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
8000393C  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
8000394C  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
8000395C  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
8000396C  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
8000397C  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
8000398C  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
8000399C  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
800039AC  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
800039BC  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
800039CC  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
800039DC  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
800039EC  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
800039FC  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
80003A0C  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
80003A1C  AA AA AA AA AA AA AA AA  AA AA AA AA AA AA AA AA
```

Figure B.3: Ozone Debugger

SEGGER SystemView

When dealing with complex embedded systems comprising multiple threads and interrupts, the *SEGGER SystemView* tool is needed. This is a real-time recording and visualization tool for embedded systems. It reveals the true runtime behavior of an application, going far deeper than the system insights provided by debuggers.

Oscilloscope

Another very useful tool when debugging embedded systems is a digital oscilloscope. In this thesis work, the Rigol MSO5354 digital oscilloscope was used. The MSO5354 excels in analyzing communication protocols such as I^{2}C, SPI, or CAN, and can decode them in real-time.

To decode the I^{2}C protocol, used for the tactile sensor communication, the oscilloscope probes were connected to the clock and data buses. Then, triggers are used to capture data during a specific transaction, and this data is decoded to verify if the firmware sends/receives correct instructions. If the registers and values read are the expected ones, the application is working correctly.

Figure B.4: Rigol MSO5354 Digital Oscilloscope

Bibliography

[1] Q. Li, O. Kroemer, Z. Su, F. F. Veiga, M. Kaboli, and H. J. Ritter, “A review of tactile information: Perception and action through touch”, *IEEE Transactions on Robotics*, vol. 36, no. 6, pp. 1619–1634, 2020. DOI: `10.1109/TRO.2020.3003230`.

[2] Z. Kappassov, J.-A. Corrales, and V. Perdereau, “Tactile sensing in dexterous robot hands”, *Robotics and Autonomous Systems*, vol. 74, pp. 195–220, 2015.

[3] R. S. Dahiya, M. Valle, *et al.*, “Tactile sensing for robotic applications”, *Sensors, Focus on Tactile, Force and Stress Sensors*, pp. 298–304, 2008.

[4] C. Hegde, J. Su, J. M. R. Tan, K. He, X. Chen, and S. Magdassi, “Sensing in soft robotics”, *ACS nano*, vol. 17, no. 16, pp. 15 277–15 307, 2023.

[5] J. Jiang and S. Luo, “Robotic perception of object properties using tactile sensing”, in *Tactile Sensing, Skill Learning, and Robotic Dexterous Manipulation*, Elsevier, 2022, pp. 23–44.

[6] R. Bogue, “Recent developments in robotic tactile perception”, *Industrial Robot: An International Journal*, vol. 44, no. 5, pp. 565–570, 2017.

[7] W. Mandil, V. Rajendran, K. Nazari, and A. Ghalamzan-Esfahani, “Tactile-sensing technologies: Trends, challenges and outlook in agri-food manipulation”, *Sensors*, vol. 23, no. 17, p. 7362, 2023.

[8] Y. Jiang, L. Fan, X. Sun, *et al.*, “A multifunctional tactile sensory system for robotic intelligent identification and manipulation perception”, *Advanced Science*, vol. 11, no. 41, p. 2 402 705, 2024.

[9] P. Weiner, C. Neef, Y. Shibata, Y. Nakamura, and T. Asfour, “An embedded, multi-modal sensor system for scalable robotic and prosthetic hand fingers”, *Sensors*, vol. 20, no. 1, p. 101, 2019.

[10] F. Yang, C. Feng, Z. Chen, *et al.*, “Binding touch to everything: Learning unified multimodal tactile representations”, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 26 340–26 353.

[11] W. Fan, H. Li, W. Si, S. Luo, N. Lepora, and D. Zhang, “Vitactip: Design and verification of a novel biomimetic physical vision-tactile fusion sensor”, *arXiv preprint arXiv:2402.00199*, 2024.

[12] X. Li, R. Deng, W. Jiao, *et al.*, "A high-sensitivity magnetic tactile sensor with a structure-optimized hall sensor and a flexible magnetic film", *IEEE Sensors Journal*, vol. 24, no. 10, pp. 15 935–15 944, 2024. DOI: `10.1109/JSEN.2024.3385299`.

[13] S. Park, S.-R. Oh, and D. Hwang, "Magtac: Magnetic six-axis force/torque fingertip tactile sensor for robotic hand applications", in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 10 367–10 372. DOI: `10.1109/ICRA48891.2023.10161042`.

[14] A. Mohammadi, Y. Xu, Y. Tan, P. Choong, and D. Oetomo, "Magnetic-based soft tactile sensors with deformable continuous force transfer medium for resolving contact locations in robotic grasping and manipulation", *Sensors*, vol. 19, no. 22, p. 4925, 2019.

[15] J. Bimbo, P. Kormushev, K. Althoefer, and H. Liu, "Global estimation of an object's pose using tactile sensing", *Advanced Robotics*, vol. 29, no. 5, pp. 363–374, 2015.

[16] V. R. Galaiya, M. Asfour, T. E. Alves de Oliveira, X. Jiang, and V. Prado da Fonseca, "Exploring tactile temporal features for object pose estimation during robotic manipulation", *Sensors*, vol. 23, no. 9, p. 4535, 2023.

[17] M. Chalon, J. Reinecke, and M. Pfanne, "Online in-hand object localization", in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2013, pp. 2977–2984.

[18] M. Pfanne and M. Chalon, "Ekf-based in-hand object localization from joint position and torque measurements", in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2017, pp. 2464–2470.

[19] A. Petrovskaya, O. Khatib, S. Thrun, and A. Ng, "Bayesian estimation for autonomous object manipulation based on tactile sensors", in *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, 2006, pp. 707–714. DOI: `10.1109/ROBOT.2006.1641793`.

[20] Zephyr Project, *Zephyr Project Documentation.* [Online]. Available: `https://docs.zephyrproject.org/latest/index.html`.

[21] Zephyr Project, *Zephyr RTOS - Getting Started Guide.* [Online]. Available: `https://docs.zephyrproject.org/latest/develop/getting_started/index.html`.

[22] Zephyr Project, *Zephyr - GitHub*, Zephyr Project GitHub Profile. [Online]. Available: `https://github.com/zephyrproject-rtos`.

[23] Melexis, *MLX90395 Datasheet.*

[24] T. Invensense, *ICM-20948 IMU Datasheet.*

[25] X. Robotics, *uSkin Sensor Datasheet.*

[26] jremington, *ICM-20948-AHR*, GitHub repository. [Online]. Available: `https://github.com/jremington/ICM_20948-AHR`.

[27] K. Çoçoli and L. Badia, "A comparative analysis of sensor fusion algorithms for miniature IMU measurements", in *2023 International Seminar on Intelligent Technology and Its Applications (ISITIA)*, Surabaya, Indonesia: IEEE, Jul. 26, 2023, pp. 239–244, ISBN: 9798350313956. DOI: `10.1109/ISITIA59021.2023.10220994`. [Online]. Available: `https://ieeexplore.ieee.org/document/10220994/`.

[28] R. Mahony, T. Hamel, and J.-M. Pflimlin, "Nonlinear complementary filters on the special orthogonal group", *IEEE Transactions on Automatic Control*, vol. 53, no. 5, pp. 1203–1218, Jun. 2008, ISSN: 0018-9286. DOI: `10.1109/TAC.2008.923738`. [Online]. Available: `http://ieeexplore.ieee.org/document/4608934/`.

[29] Zephyr Project, *Zsclib - GitHub*, Zsclib GitHub Repository. [Online]. Available: `https://github.com/zephyrproject-rtos/zscilib`.