



UNIVERSITY OF PADUA

DEPARTMENT OF INFORMATION ENGINEERING

MASTER'S DEGREE IN
CONTROL SYSTEMS ENGINEERING

**Power management for controlled power
cycles of an integrated computer,
environmental and electrical
monitoring, and LED signaling: a C
implementation for PIC24
microcontrollers**

Thesis Supervisor:
PROF. ALBERTO MORATO

Graduate Student:
GIULIA PIZZATO
2089397

Academic Year 2023/2024
4 December 2024

Acknowledgements

Ringrazio prima di tutto i miei genitori, che mi sono stati accanto fin dai primi passi. Grazie mamma, grazie papà, non avrei mai potuto arrivare qui da sola, mi avete reso chi sono oggi. Non so esprimere a parole quanto vi voglio bene, il minimo che posso fare è dedicarvi questa laurea. Grazie anche a mio fratello, con cui ho condiviso un'infanzia, e con cui ancora rido alle lacrime.

Grazie alla mia famiglia: agli zii, ai nonni, ed ai cugini, grazie per i Natali, per i compleanni e le uscite, stare assieme è come una coperta calda, in cui mi sento al sicuro. Ringrazio anche la famiglia De Vicari, perché siete praticamente famiglia anche voi.

Ringrazio i miei amici, che so già leggeranno questa parte curiosi di cosa sto scrivendo di loro. Grazie ragazzi, perché mi siete accanto. Grazie a chi è qui da una vita, grazie Marta, Maria Laura, Maria ed Eugenia. Siete la mia base, il mio angolo sicuro. Grazie anche a chi invece ha vissuto al mio fianco questi cinque anni di università. Grazie per tutte le risate, le partite a carte, le gite, e le spiegazioni subito prima degli esami. Grazie a (in ordine alfabetico, non litigate): Ada, Alessia, Ambra, Cesco, Enrico, Giorgio, Grim, Lore, Madda, Marco, e Mina.

Infine voglio ringraziare Mattia, grazie di starmi accanto. Da quando sei entrato nella mia vita la stai riempiendo con tanta di quella pura e semplice gioia, che hai preso metà del mio cuore. Grazie, ti amo.

Abstract

The objective of this thesis is to illustrate the development of a C program for a PIC24 microcontroller, installed on a multipurpose control and power supply board. Mounted on a device named ISObox, the aim of the program is to manage a controlled and clean power-up and shutdown cycle of an embedded computer system. ISObox is mounted on machinery, and therefore its power supply is directly provided by the machine's engine. When the machinery is ignited on or turned off, the microcontroller must ensure a clean shutdown of the embedded computer, to prevent sudden power loss that could cause filesystem damages or corruption. Similarly, at startup, the system must manage voltage fluctuations to avoid unintended power-ups or shutdowns. The microcontroller utilizes environmental sensors to monitor voltage, temperature, and humidity, taking appropriate actions to signal anomalies via bicolour LED or to shut down the embedded computer in critical situations. The communication between microcontroller and both sensors and embedded computer system is established through the I2C protocol, one as master and the other as slave. Through this communication continuous monitoring of the system's status is ensured.

Contents

1	Introduction	1
1.1	The company and their main product	1
1.2	ISO BOX	3
1.3	Evolution of ISO BOX	4
1.4	The project	5
1.4.1	Thesis structure	7
2	Hardware	9
2.1	Mechanics	9
2.2	PIC24F MCU	10
2.3	Electronics	12
2.3.1	Front Panel Board	12
2.3.2	PCIE CAN bus controller board	14
2.3.3	PAPBB	14
3	Peripherals Communication	17
3.1	I/O communication	17
3.1.1	Polling	18
3.1.2	Interrupts	19
3.2	I2C communication	24
3.2.1	I2C Protocol	24
3.2.2	Read and Write operations as Master	27
3.2.3	I2C on the MCU side	29
4	Objective of the project	33
4.1	Objectives	33
4.1.1	General Objectives	33
4.1.2	Communication with computer	34
4.1.3	Error management	35

4.1.4	Power button actions	35
4.2	Preconditions	36
4.3	Constraints	37
5	Software	39
5.1	Setup and Diagnostics	39
5.2	Logic flow	41
5.2.1	Isoboard Firmware	41
5.2.2	New ISO BOX firmware	45
5.3	Code Schemes	53
5.4	Cranking	61
5.4.1	What is cranking	61
5.4.2	Cranking problem	63
5.4.3	Code for cranking	64
6	Validation	71
6.1	Power-On Tests	72
6.1.1	ISO BOX mode	73
6.1.2	POWER BOX mode	74
6.2	Normal Run Tests	74
6.3	Power-off Test	75
6.4	Errors	75
6.5	Registers functionality	77
7	Future Developments and Conclusions	79
	Bibliografia	81

List of Figures

1.1	ITPhotonics logo [1]	1
1.2	Polispec [2]	2
1.3	Original ISO BOX Rendering [3]	3
1.4	Modernized ISO BOX rendering	5
2.1	ISO BOX dimension scheme	9
2.2	ISO BOX front panel. From Figure 2.2b we can see: A)Green-Red Status LED B)Red-Blue Status LED C)Power button D)I/O connector E)USB interface	10
2.3	ISO BOX electronic schematic	13
3.1	Polling functioning block scheme	18
3.2	Example of an event being missed by the processor that utilizes polling [4]	18
3.3	CPU activity allocation when each event is accompanied by an interrupt [4]	19
3.4	Interrupt block scheme	20
3.5	Interrupt nesting [4]	22
3.6	IVT [10]	23
3.7	I2C lines of communication [6]	24
3.8	Typical I2C Interconnection Block Diagram [5]	25
3.9	I2C communication between the MCU and a sensor, read with an oscilloscope	25
3.10	I2C data recognition [6]	26
3.11	I2C START and STOP signals [6]	27
3.12	Example of the writing of a single byte to a slave register [8]	28
3.13	Example of the reading of a single byte to a slave register [8]	29
3.14	Example of sequence of events, START case used [5]	31

3.15	Example of sequence of events, Master Read Transmission case used [5]	32
5.1	Old ISO BOX firmware logic flow	42
5.2	Current ISO BOX firmware logic flow, part 1	46
5.3	Current ISO BOX firmware logic flow, part 2	50
5.4	Functions performed by firmware at every main cycle	52
5.5	code evolution scheme with embedded	54
5.6	code evolution scheme without embedded	55
5.7	code error evolution scheme	56
5.8	LED time evolution	60
5.9	LED time evolution 2	60
5.10	LED time evolution 3	61
5.11	cold cranking, diesel engine [12]	62
5.12	Real case: voltage cranking, diesel car	63
5.13	Example of a simulated crank using the MATLAB code. Together are also shown the mean and variance used. On the right a detail is shown	67
5.14	Example of simulated Figure showing four different plots, each representing a signal resembling a cranking waveform but correctly identified by the algorithm as not being a cranking event	69
6.1	Setup - sensitive parts are blurred	72

List of Acronyms

MCU	MicroController Unit
MPU	MicroProcessors Unit
PSU	Power Supply Unit
CPU	Central Processing Unit
ALU	Arithmetic Logic Unit
IRQ	Interrupt ReQuest
IVT	Interrupt Vector Table
AIVT	Alternate Interrupt Vector Table
ISR	Interrupt Service Routine
PC	Program Counter
SDA	Serial Data Line
SCL	Serial Clock Line
FSM	Finite State Machine
OS	Operating System
IDE	Integrated Development Environment

Chapter 1

Introduction

1.1 The company and their main product

This project has been developed in collaboration with ITPhotonics.

ITPhotonics was founded in 2012 by its four current partners, who identified an opportunity for innovation in the market. Since then, they have continued to refine their products.

ITPhotonics is specialized in spectrophotometry and applied electronics [1], in its various forms.



Figure 1.1: ITPhotonics logo [1]

The main line of products of ITPhotonics is Polisphec (Portable and On Line SPECTrophotometer) a line of compact spectrophotometers, which varies in spectrum width, in branch, and in application [2].

It is used to obtain a data matrix from various products, which can come from different sectors. These include the industrial sector (e.g. cataphoresis processes, mashed potato production, wastewater analysis), the chemical sector, and the agri-tech sector. Currently, their strongest focus is in agri-tech (e.g. grains, silage, forage, raw materials), where ITPhotonics' instruments ensure the monitoring of the final product.

The functionality of Polisphec is designed to be both simple and effective. It has a

scan window, which must be in contact with the product flow during the process. The obtained data are stored and processed by a computer, which are then visible to the client either on the machinery panel board or on a separate screen, and can be downloaded via cable or Wi-Fi.

Polispec has two operative modes: handheld and online. In handheld mode, the device has an internal battery that makes it portable, and communicates the obtained data to an external computer, either through Wi-Fi or through a cabled communication, depending on the selected configuration. In online mode, the spectrophotometer is stably mounted on machinery that works close to the product, where it acquires data. The computer that then does the elaboration, can be internal to the Polispec or an external one.



Figure 1.2: Polispec [2]

In the online case, the Polispec is powered directly by the machinery, but some Polispec devices may not be equipped with power supply circuitry compatible with automotive voltage and disturbance ranges. However, machinery voltages, especially in automotive applications, can experience significant fluctuations.

To solve this issue, POWER BOX has been introduced, an optional accessory which ensures proper management of the spectrometer power supply.

If the final client does not own a personal external computer, it can be provided by ITPhotonics. The computer can be placed inside Polispec itself, or inside the POWER BOX.

When the computer is inside the POWER BOX, it is renamed ISO BOX. In this setup, the ISO BOX also regulates the power cycles for the computer, protecting it from significant power fluctuations which could potentially lead to hardware



Figure 1.3: Original ISO BOX Rendering [3]

damage from frequent powering on and off, as well as software malfunctions or filesystem corruption.

Sometimes the customer may want to install ISO BOX not for its power management tools, but just to separate the elaboration system from the measuring instrument, obtaining more flexibility in switching Polisppec between online mode and handheld mode.

1.2 ISO BOX

As the catalogue sais “ISO BOX is an intelligent control unit that, in addition to managing the power supply to the Polisppec sensors when installed on self-propelled or towing machinery, processes the signals it receives from the sensor and integrates sensor operation in the machine ISObus network. The same control unit can be used to integrate systems in proprietary CANbus network” [3].

ISO BOX main tasks are:

- powering on and off the computer in a clean and controlled manner
- continuous monitoring of ISO BOX environmental data via humidity and temperature sensors
- continuous monitoring of the battery and ignition voltage via voltage sensors
- respond to information requests from internal computer, which could ask for diagnostics information or setup data
- power on or power off the spectrometer, based on voltage levels or embedded PC requests
- save setup data changes

- in thresholds setup data changes, check that the chosen values are in valid ranges, and if not set the minimum/maximum value of the range
- signal via LED the working status of ISO BOX, including potential errors

1.3 Evolution of ISO BOX

When Polispec is in online mode, it can contain internally the computer, placing it in the slot otherwise occupied by the battery when in handheld mode.

But this option was not always available. Initially, Polispec could not house an internal computer, making the ISO BOX the only solution for power regulation and communication.

Born as just an accessory, it was electronically very different from the Polispec. They used different boards and the code developed was written for an atmega328 MicroController Unit (MCU). Communication between the computer and the MCU was handled via UART, using an internally developed communication protocol with 19 possible packets and their associated responses. As the design evolved to allow the computer to be housed directly inside the Polispec, a new board, named PAPBB (Powerboard), was developed.

PAPBB was more compact, and had on board a new MCU, specifically a Microchip controller from the PIC24F family. This was chosen for better control of libraries and devices. Additionally, it allowed I2C communication, and since it was already in use within the company, it was well-known to the engineers. This board's MCU, working closely with another board's (called "Supply board") MCU, managed the computer's power cycle, monitored the environment status, and signaled the Polispec status via LED.

Given the similarities between the PAPBB and the old ISO BOX, it was a natural to consider modernizing ISO BOX to work with the new board, improving its functionality and alignment with the updated design. The choice of modernizing the original ISO BOX offers significant economic advantages, as producing a single board instead of two, simplifies manufacturing and allows better agreements with suppliers. Furthermore, the new board is smaller than the previous one, allowing for more compact and cost-effective packaging.

1.4 The project

After explaining the context in which this thesis project was developed, we can now look at it in more detail.

The main objective of this project is the development of the firmware for this new version of ISO BOX.

The new code is supposed to be written in the same environment as the one written for Polispec, but using the basic structure of the previous ISO BOX algorithm. So the merging of these two previous works was the first objective. The second was to implement new novelties and improvements that weren't previously present.

Indeed, this modernization allowed to put together some qualities. For example by uniforming the board, it was also possible to electronically make indistinguishable ISO BOX and POWER BOX, making the change between the two easy and efficient. Therefore the code, assuming it has the knowledge via setup data of whether the embedded is present or not, has to be able to handle both the ISO BOX and the POWER BOX cases, since it can be uploaded on both.

Also, as the MCU allowed for I2C communication, the self-developed communication protocol was removed in favour of I2C, which is a wide spread, and secure protocol.



(a) ISO BOX front view



(b) ISO BOX rear view

Figure 1.4: Modernized ISO BOX rendering

The project was carried out in the following order, detailing each step taken throughout the process.

After an introductory study of ANSI C, with a focus on interrupts, the first step was to analyze the structure of the two previous codes. One of the key goals was to establish the main structure of the final code, and the branches it must take in case of errors or changes in the setup. While developing the logical flow, the practical elements have been also studied, such as the the functioning and the features of a MCU from the PIC24F family, and the I2C protocol, needed for communication with both the embedded computer and the peripherals (temperature sensors, voltage sensors and EEPROM).

Once this basis were set the development of the actual code started. Some libraries from the Polisppec code were reused, with adaptations made where necessary. Any modifications were carefully documented. Due to the similar environments and overlapping in functioning, the new code was largely based on the Polisppec code. A key similarity was in the communication between the MCU and the computer. In the Atmega328 setup, communication was UART-based, whereas in the PIC24F system, it relied on I2C, similar to the Polisppec implementation. The PIC MCU has two I2C ports: one configured as a master for communication with peripherals, and the other as a slave for interaction with the computer. Previously developed libraries were utilized for this, but with certain aspects revised, such as the slave registers, and improvements introduced. However, despite these similarities, there are substantial changes. As previously mentioned, in the Polisppec electronic system, PAPBB coordinates with the Supply board. In the new context of ISO BOX the relationship between the two is different. For instance, in the previous design the PAPBB was not equipped with an external memory, it relied on its flash memory to store permanent information. In this new setup the PAPBB has an EEPROM, which the PIC MCU can access via I2C.

Another difference is in the access to the LEDs. In Polisppec's case, only the Supply had direct access to the external status LEDs. The PAPBB had to communicate its status to the Supply, so that it could then change the LED signals. Now that the LEDs are directly controlled by the PAPBB, the associated code had to be modified to accomodate this necessity. The LEDs access introduced an additional problem. The type of LED used required two pins were to con-

trol them, however the pin available on the PIC24 were all utilized. To solve this problem an ingenious solution has been adopted, that allowed two pins to be used both as inputs and as outputs. This will be better explained in Chapter 5.

Great importance has been given to backward compatibility. In some case the reasoning is immediate, for example in the choice of the status LEDs signals. Those were kept as before, with new necessary additions. Backward compatibility is also one of the reasons why already developed library were preferred, even when substantial modifications were required. Libraries from both Supply and PAPBB in Polispec's system were used, with all changes made during the adaptation process documented. This allowed for a more trustworthy validation process, and easier debugging process.

New code setup conditions have been added, allowing the code to be used for both the POWER BOX and ISO BOX systems. A dedicated setup register has been introduced, to indicate the presence or absence of the embedded computer, causing the code to behave accordingly. Additionally, both methods of powering on of the embedded are now considered. To clarify, the computer utilized can be configured to turn on as soon as power is supplied or to wait for a power-up signal, similar to a standard personal PC. Before, it was assumed the computer would always be set on the first configuration. Now, the setup allows the user to specify the power-on mode, to assure a correct power cycle of the computer.

Lastly, two brand new features were added. The first is a safety check for I2C responses. The system now constantly verifies whether the voltage or temperature sensors, stop responding. For each scenario the proper set of action has been set and implemented.

The second feature relates to the front power button. While before its function was limited to the common function, it now recognizes both long and short presses. Depending on the duration of the press and the current section of the code, the system performs different actions.

1.4.1 Thesis structure

To assure that all previously listed elements are covered, but to allow a better flow of explanation, this document will be structured as follows. The document will begin with a quick overview of the hardware relevant components in Chapter

2. Chapter 3 will discuss the necessary theoretical framework, and finally the code and its validation will be thoroughly discussed in Chapter 5 and Chapter 6. Finally, we will conclude with potential future improvements and final remarks.

Chapter 2

Hardware

While the design or analysis of the associated hardware is beyond the scope of this project, mechanics and electronics are still described here to provide context for the firmware’s operation.

As previously mentioned, the firmware is uploaded on a device called ISO BOX, which works in an environment with two other elements: Polispec and a generator (machinery’s battery).

This document will focus exclusively on the latest version of the ISO BOX, which runs the new firmware, and will not cover the previous versions.

2.1 Mechanics

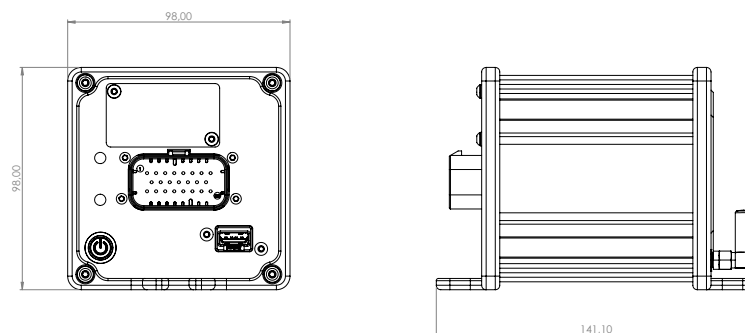


Figure 2.1: ISO BOX dimension scheme

ISO BOX is a $98 \times 98 \times 141.1$ mm aluminum box, with PA12 (nylon) frames. It is designed to withstand harsh environments and is rated IP65 for dust and water resistance. It features mounting support brackets, a heat sink on the back,

and a Wi-Fi antenna for remote access to the instrument, facilitating remote troubleshooting and repairs.

As shown in Figure 2.2, ISO BOX features a front panel that connects both to the machinery and the instrument, which will be explained in greater detail when the electronics are introduced. The panel includes two sets of status LEDs, a USB interface, a power button, and a removable plate that conceals configuration options.

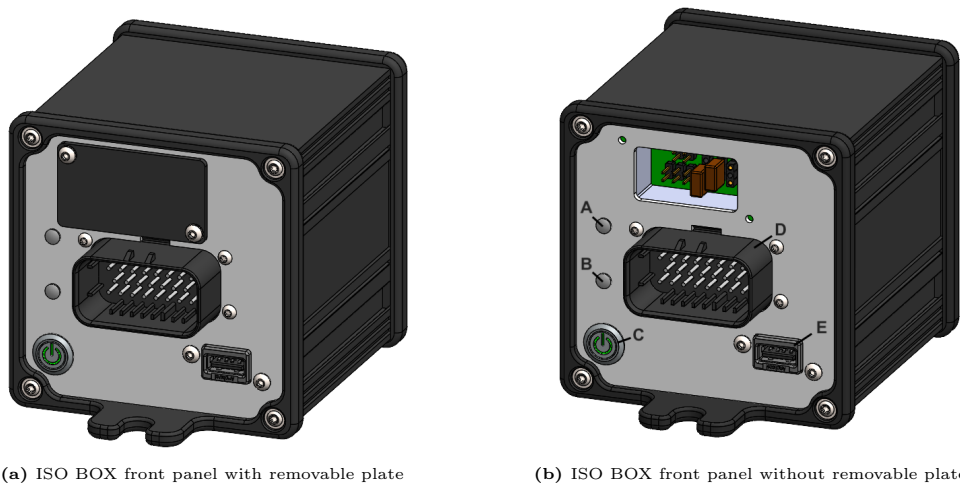


Figure 2.2: ISO BOX front panel. From Figure 2.2b we can see: A)Green-Red Status LED B)Red-Blue Status LED C)Power button D)I/O connector E)USB interface

The two sets of LEDs consist of one red and green pair dedicated to indicating the status of the ISO BOX code, and one yellow and blue pair providing information on the embedded power supply and overall power status. The functionality of the first set will be explained in greater detail in Chapter 5.

2.2 PIC24F MCU

A MCU is an integrated circuit that, just like MicroProcessors Unit (MPU)s, was introduced to replace antiquated multi-component Central Processing Unit (CPU)s. MCUs “combine all the necessary elements of a microcomputer system onto a single piece of hardware. MCUs do not require additional peripherals or complex operating systems to function” [9], contrary to MPUs.

The MCU used in this project is of the family PIC24F, hereafter also referred to as

PIC for brevity. The PIC Microcontroller, developed by Microchip Technology, “is the world’s smallest microcontroller, and it is found frequently in robotics, home and industrial automation and renewable energy systems” [9].

The PIC24 MCU offer many features, however, to avoid giving information of no use or just accessible by the datasheet, only the main features used in the project will be presented.

Core Features The MCU used in this project was a PIC24F.

Its CPU features a 16-bit data modified Harvard architecture, which means it feature separate memory spaces and busses for program and data, allowing parallel access and cosequent faster performances.

Key specifications include:

- **Memory:** it has 64k bytes of Program memory and 22k bytes of Program Memory instructions
- **Registers:** the PIC24F devices feature sixteen 16-bit working registers. These registers can act as data registers, address registers, or address offset registers
 - For example the CORCON Register (CPU Control Register) is a 16 bit register where each bit is a flag that gives information on one factor of CPU or Arithmetic Logic Unit (ALU)
- **Program Counter:** Program Counter (PC), which tracks the memory address of the next instruction to be executed, is 23 bits wide, allowing it to address up to 4 million instructions in the user program memory space

Peripherals and Tools The PIC24F MCU has a couple of useful tools that are worth mentioning.

- **Timers:** Five 16-bit timers were offered, each with its readable/writable registers and its interrupt. There are 3 types of timers A,B and C. The timer used was of Type A, because “the unique features of a Type A timer allow it to be used for timekeeping functions or as a secondary system clock source” [11]
- **I2C module:** it has two I2C communication ports, in which it can be set to act as a master, a slave, or a master in a multi master environment. It supports independent master and slave logic, 7-bit and 10-bit device

addresses, general call address, and clock stretching to provide delays for the processor to respond to a slave data request.

2.3 Electronics

On the inside ISO BOX contain three main electronic boards, that will be addressed as “Front panel Board”, “PCIE Can Bus Controller board” and “PAPBB”. For company privacy reasons, the full component numbers of the elements used in the electrical boards will not be mentioned. The key characteristics will be listed instead.

2.3.1 Front Panel Board

The Front Panel Board is directly connected to ISO BOX’s front panel, and houses the I/O connector where both the machinery (with its CANbus communication and battery voltage) and the Polispec are connected. Figure 2.3 illustrates all the signals passing through the I/O connector.

The first two signals (+13.8Vbatt and IGNITION) come from the machinery. When both signals are high, the MCU is powered on, initiating firmware execution. +13.8Vbatt is the battery voltage, and since the battery is always connected, it is the ignition signal that dictates when the system is starting, signaling ISO BOX to begin its operations as well. In the case of automotive applications (e.g. agricultural machinery in the field), the ignition is a digital signal that indicates the turning of the physical key, which is followed by cranking. The cranking process and its associated challenges will be presented in Chapter 5.

For control purposes, the MCU can activate a power feedback loop, allowing it to remain powered on even if the ignition signal goes low. This feature was introduced to allow the MCU to have time to safely power off the embedded computer when the key is turned off. It can be activated only by the PIC itself. In Chapter 5 we’ll explore the management of this feedback, to assure proper power off of the system, avoiding the error case of the PIC keeping the system powered on when not required.

The I/O connector includes 7 pins dedicated to CAN bus communication: two sets of 3 pins for connecting to two separate CAN buses, and an additional pin for handling changes in communication setup. These signals are utilized, via the PCIE CAN bus controller board (a commercial component), to transmit the computed data from the embedded computer to the machinery’s panel board.

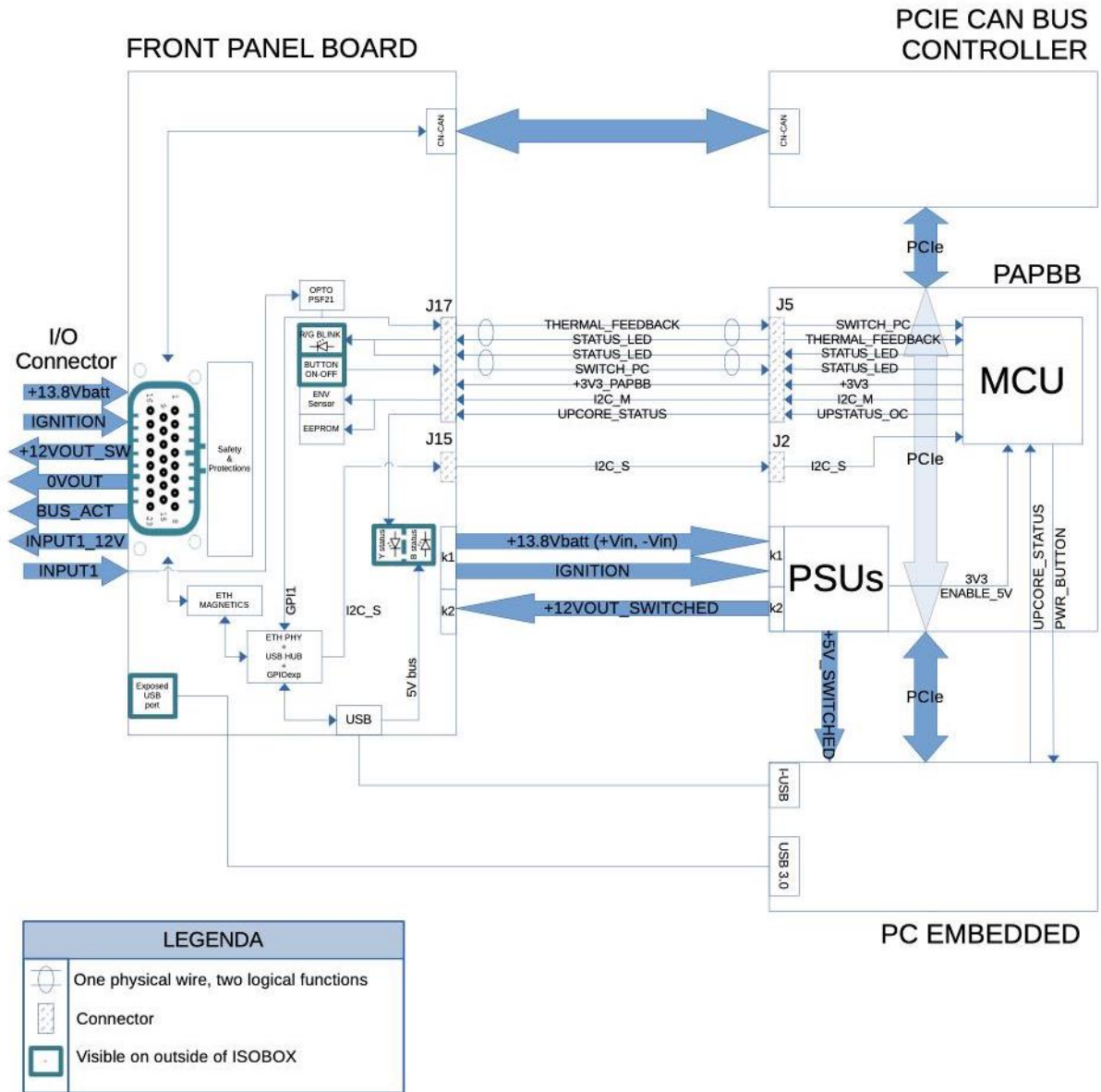


Figure 2.3: ISO BOX electronic schematic

The remaining pins on the I/O connector are dedicated to signals used or provided by the Polispec. The Polispec is powered by a 12V supply, which is enabled by the MCU when required, and regulated through the Power Supply Unit (PSU)

of the PAPBB.

The two signals INPUT1 and INPUT1_12V, form a loop that closes when the Polispec is powered and not in overtemperature condition. It is otherwise always open. By monitoring INPUT1 and knowing the power status of the Polispec, we are informed on its overtemperature status. This is achieved through an optoisolator: if the optoisolator in the Polispec is open, it indicates that the instrument is either not powered or is in overtemperature.

The BY (Blue and Yellow) LED depends on the PC's power status. The blue LED lights up when the PIC activates the 5V signal to power on the embedded computer, while the yellow LED turns on when the embedded computer sends an electronic feedback signal. When both LEDs are on, the resulting color is green, indicating that the system is functioning properly. The RG (Red and Green) LED is controlled directly by the PIC, and its signals will be detailed later in this thesis.

Lastly, the remaining elements on the Front Panel Board worth mentioning, are the environmental sensors (temperature, humidity, and voltage sensors) used to assure ISO BOX's safety, the EEPROM memory, and the magnetics used to ensure Ethernet signal integrity.

2.3.2 PCIE CAN bus controller board

As already stated the PCIE CAN BUS CONTROLLER board is a commercial board, dedicated to the CAN BUS communication with the panel board of the machinery and the embedded computer.

2.3.3 PAPBB

The final board is the PAPABB, the power and control board. This is where the MCU running our code is located, and where power electronics control takes place. The PSUs make sure that the signals are clean, and DCDC converters generate the 5V (to power on the computer), 3.3V, and 12V (to power on the Polispec) signals from the battery voltage.

From the MCU, which is powered by 3.3V, many signals are visible. It has 2 lines of I2C communication. In our setup it acts as a master on one line, where it communicates with sensors and EEPROM, and as slave in the other, where it

communicates with the computer.

In addition, we can see two I/O lines that are used simultaneously for the GR status LED (output) and to read two inputs: the Polispec thermal feedback and the power button. As previously stated, moving the LED control from the Supply board to the PAPBB board, implied the necessity of utilizing two PIC pins, but all available pins were already in use. To solve this, the decision was made to use two pins as both inputs and outputs.

To obtain this dual functionality, the lines are switched every 10 ms between input and output modes. When set as outputs, they pilot the status LED signaling, when set as inputs they read the two signals. A possible issue could be that the LED turns off in the input state. However, since the switching rate is 100 Hz, it would be difficult for the human eye to perceive any noticeable difference.

Another concern is the correct memorization of the power button signal. Unlike the thermal feedback, that doesn't require memory of its previous state, the power button signal must be tracked over time to determine the duration of the press. This distinction is necessary because the system differentiates between a long button press and a short one. Every 10 ms, the system checks if the button signal has changed (from high to low or vice versa), and based on how long the signal remains low (button pressed), it acts accordingly. How these two signals are distinguished and used is explained better in Chapter 5.

The concern is that during the switching of the LEDs, the MCU might miss a button press. This is prevented in two ways. First, there is an obvious physical limit, as it's unlikely that the client could press the button for less than 10 ms. For additional safety, a lower bound is placed on the duration of the button press. If the button is pressed for less than 500 ms, the signal is not considered. This approach helps prevent both missed signals by the MCU and accidental presses by the user.

The last signal, "UPCORE_STATUS" is the power status of the embedded computer. It is used in the code to make decisions on the action to take.

Chapter 3

Peripherals Communication

3.1 I/O communication

In this project, we work with various communicating devices, including both external peripherals and internal components of the PIC, such as timers. The communication between the PIC and these, can be implemented both asynchronously or synchronously. In asynchronous communication, unlike synchronous, the data stream is not accompanied by a clock signal that manages coordination. The data are sent when available, with the begin and end of the message signaled via additional data (start and stop bits).

This setup, combined with the fact that the speed of data elaboration of the CPU is usually way higher than the speed of transmission data of the peripherals, can lead to issues in relation to synchronization. The CPU could request data to the peripherals when they are not ready, or the peripherals could send data when the CPU is occupied with other operations. Consequently, various errors could occur, such as data not being read or sent, collisions, or the transmission of erroneous or inconsistent data. Additionally, due to the speed difference, if the CPU were to often wait for peripherals to respond or complete their operations, it would cause a significant waste of efficiency, as the CPU could otherwise carry out other tasks.

So a synchronous communication is preferable. To synchronize these two elements, multiple solutions have been proposed in time. We explain here two main ones: polling and interrupts.

3.1.1 Polling

In polling is the CPU that oversees communication.

It periodically inquires each peripherals about its status. If a state change is detected, the CPU starts the appropriate set of actions to handle the event.

Once the CPU has finished its operations, it resumes its cycle of peripherals interrogation, eventually checking each one.

This is can be implemented easily, even using basic sequences of `if` statements.

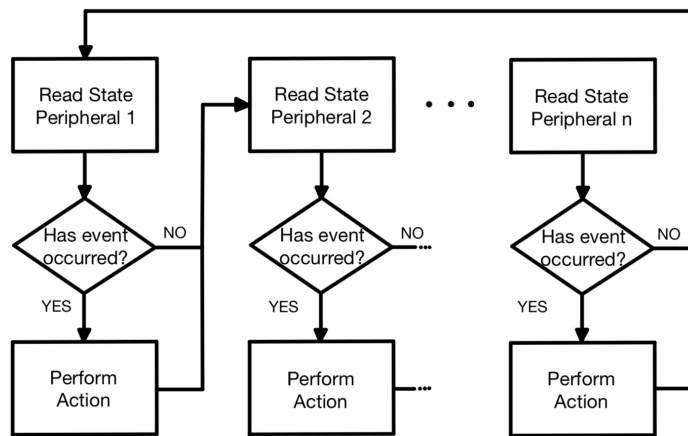


Figure 3.1: Polling functioning block scheme

This method resolves many synchronization issues, as only one component (the CPU) controls the process. Collisions risks are significantly reduced, and problems such as data not being sent or received are easier to manage. If appropriate protocols are used, control of correct and consistent data can also be assured.

However, polling has clear efficiency drawbacks. The CPU uses significant computational power to query the peripherals, and it cannot perform other tasks while waiting for I/O devices computation and response.

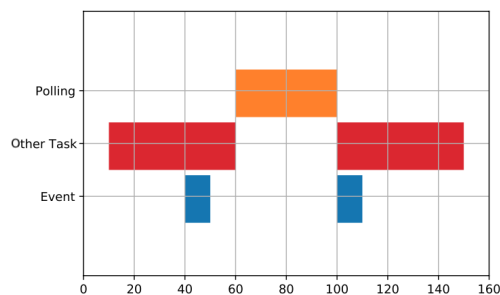


Figure 3.2: Example of an event being missed by the processor that utilizes polling [4]

If the polling interval is too long, events may be even missed. So if too many peripherals are connected, or if even one slow peripheral is in the loop, it can cause delays, increase the risk of missed events, and slow down the entire system.

3.1.2 Interrupts

The interrupt mechanism allows the CPU to do operations without periodically poll a device. It is now the external device that, when in need to communicate with the CPU, will send a signal to gain attention.

This signal is known as interrupt.

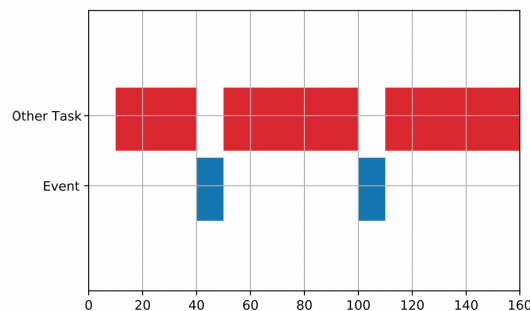


Figure 3.3: CPU activity allocation when each event is accompanied by an interrupt [4]

Usually CPUs will have one or more pins that can be triggered from the peripherals. There can be many ways to assert a pin, some via software, some via hardware.

For example, hardware triggers can be level-triggered or edge-triggered. An interrupt level-triggered is generated when the value read on the pin corresponds to a set value. An interrupt edge-triggered is started when the line changes logical state. An example of this is the timer interrupt. As the documentation says “A 16-bit timer has the ability to generate an interrupt on a period match or falling edge of the external gate signal, depending on the operating mode” [11].

Interrupts can also be software generated, meaning that is the execution of the code itself that triggers one. These can include both programmed ones, and ones not raised explicitly. In this second subcategory fall all traps, which can be considered error triggered interrupts. They will be better described later.

The signal conveyed by these pins is called Interrupt ReQuest (IRQ), and consequently these pins are often referred to as IRQ pins.

After the IRQ is detected, the program checks whether to that IRQ there’s a Interrupt Service Routine (ISR) associated, which is a special block of code. If

present, it is run by the CPU when it wants to serve the interrupt request raised by a peripheral, effectively momentarily blocking the normal run of the firmware.

The basic idea of interrupt work is divided in 4 steps:

1. CPU completes the current instruction
2. Context is saved
3. Execution of the ISR, if provided
4. Context is restored

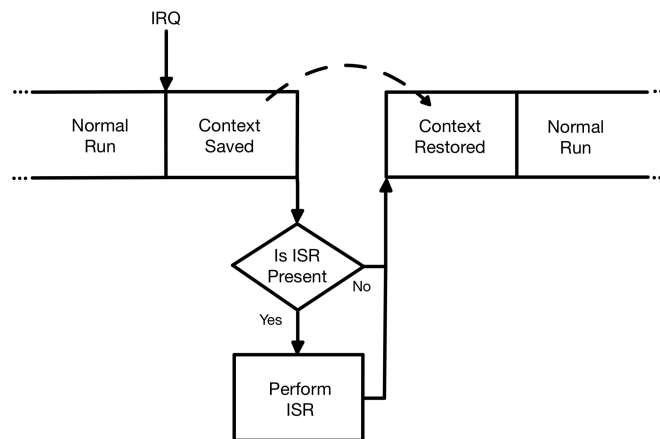


Figure 3.4: Interrupt block scheme

We can now see how the PIC24 implements these steps.

PIC24 Interrupts Management

To manage interrupts, PIC24 uses its registers. Three very important categories are: the IFSx (Interrupt Flag Status), the IECx (Interrupt Enable Control), and the IPCx (Interrupt Priority Control) registers, where x denotes the register number.

Each source of interrupt has an enable bit (one bit of IECx) which, as the name suggests, enables the associated interrupt. The status bit (one bit of IFSx) is set by the corresponding peripheral when the pin is correctly triggered, and must be cleared via software.

All interrupt event flags are checked at each instruction cycle, and if both these bits are set, then the IRQ will cause an interrupt to occur.

The CPU starts now to handle the interrupt received. As confirmed by the documentation, “No instruction will be aborted when the CPU responds to the IRQ.” [10]. The instruction will be completed, and the context will be saved. In particular the following information is saved on the software stack:

- the current PC value
- the low byte of the processor ALU STATUS register
- current CPU priority level: the IPL3 status bit (CORCON<3>)

With these information the program is able to correctly return to its previous state at the end of the ISR.

Some questions arise naturally: how does the CPU recognize which peripheral raised the interrupt, and how are simultaneous requests from different peripherals or interrupt request that occur during the ISR of another peripheral handled.

Lets see how the PIC24 tackles these problems.

Priority and Nesting To handle multiple interrupt requests the concept of priority and nesting are introduced. To each IRQ is associated a priority level. In PIC24 the CPU operates in one of sixteen priority levels, 0-15. To be served, an interrupt or trap must have priority level greater than the CPU current one, so an IRQ with priority level set to 0 is disabled. The CPU priority level is saved in the IPL3 status bit (CORCON<3>), and is set to the value of the ISR currently being served. Priority levels 8-15 are reserved for trap sources, which are interrupts dedicated to safety, and will be later introduced. So peripherals can be programmed on priority levels 0-7.

If two interrupts arrive at the same time, the one with higher priority level is served before. A ISR begin serves is said to be active.

Similarly, if an ISR is active, but an interrupt of higher priority arrives, then context is saved and the new one is served. When the new ISR has been served, then the CPU returns to the previous context, completing the lower priority level interrupt. This is called nesting.

But this is not always the case. Depends on the setup, because nesting can be both enabled or disabled. In the second case, all interrupts are set at priority level 7 (maximum priority), so any other interrupt cannot be served before the current routine is completed. If nesting is disabled it may happen that some peripherals are managed very slowly, risking dangerous delays, so normally nesting is enabled. In that case at each instruction cycle during the ISR, all priorities of

all pending interrupt requests are evaluated. If a new interrupt of higher priority arrives, while one is already in progress, the current one is stopped (context saved), and an interrupt will be presented to the processor. Of course, if a IRQ

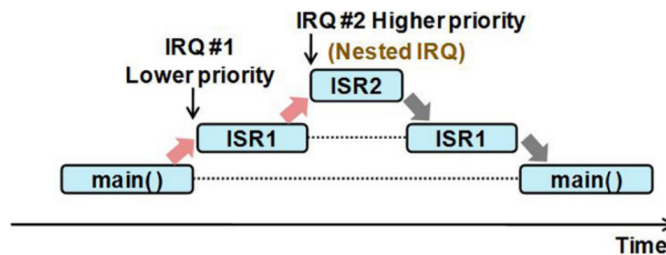


Figure 3.5: Interrupt nesting [4]

pin is triggered multiple times before it is served, the ISR will be run just once, as the corresponding IFSx bit is already set. It will be handled again at the first corresponding interrupt after the ISR is finished and IFSx has been cleared.

Given that there are significantly more interrupts than priority levels, it's possible for two interrupts with the same priority level to arrive simultaneously, or for two pending interrupts to exist at the same priority level when nesting is disabled. In such cases, the CPU is unable to determine which interrupt to address first. To resolve this, a “natural priority order” is present, as will be further explained in the next paragraph.

IVT The PIC24 interrupt controller module has a unique vector for each interrupt or exception source. It has a data structure residing in program memory called the Interrupt Vector Table (IVT).

The IVT contains 126 vectors, and each interrupt vector contains a 24-bit wide address. The value programmed into each interrupt vector location is the starting address of the associated ISR. If the vector is not programmed (empty) then if called causes a **RESET** action, restarting the entire code.

As we can see from Figure 3.6, the usage of IVT introduces a “natural priority order”, that is considered together with associated priority. If two interrupts have the same priority level, then their order in the IVT is considered, with interrupts that come before in the IVT have higher natural priority than those that come later on. PIC24 also provides an Alternate Interrupt Vector Table (AIVT), located in memory right after IVT. It is used for support in emulation and debugging, to be able to test without reprogramming the interrupt vectors.

Reset – GOTO Instruction	0x000000
Reset – GOTO Address	0x000002
Reserved	0x000004
Oscillator Fail Trap Vector	
Address Error Trap Vector	
Stack Error Trap Vector	
Math Error Trap Vector	
Reserved	
Reserved	
Reserved	
Interrupt Vector 0	0x000014
Interrupt Vector 1	
~	
~	
~	
Interrupt Vector 52	0x00007C
Interrupt Vector 53	0x00007E
Interrupt Vector 54	0x000080
~	
~	
~	
Interrupt Vector 116	0x0000FC
Interrupt Vector 117	0x0000FE
Reserved	
Reserved	
Reserved	
Oscillator Fail Trap Vector	
Address Error Trap Vector	
Stack Error Trap Vector	
Math Error Trap Vector	
Reserved	
Reserved	
Reserved	
Interrupt Vector 0	0x000114
Interrupt Vector 1	
~	
~	
~	
Interrupt Vector 52	0x00017C
Interrupt Vector 53	0x00017E
Interrupt Vector 54	0x000180
~	
~	
~	
Interrupt Vector 116	
Interrupt Vector 117	0x0001FE
Start of Code	0x000200

Decreasing Natural Order Priority

See Table 8-1 for Trap Vector Details

Figure 3.6: IVT [10]

Traps Of these 126 vectors, 8 are Non Maskable Traps, which function as “safety interrupts”. These interrupts are assigned maximum priority and activate in response to critical errors in the code, such as a stack error. The documentation tells us that “Traps are intended to provide the user a means to correct erroneous operation during debug and when operating within the application” [10].

Of these 8, 4 are reserved. The other 4 are:

- Oscillator Failure Trap
- Stack Error Trap
- Address Error Trap
- Arithmetic Error Trap

3.2 I2C communication

I2C, also known as Inter Integrated Circuit (IC), is a “a simple 2 bidirectional 2-wire bus for efficient inter-IC control” [6]. Developed in 1982 by Philips Semiconductor (now NXP Semiconductor), it has been license-free since 2006, leading it to become adopted by many semiconductor device companies, which have then introduced I2C-compatible devices. [7]

I2C is a low-speed communication protocol designed to connect controller devices, like MCUs and processors, with target devices such as data converters and various peripherals.

One of its main advantages is that only two bus lines are required, a Serial Data Line (SDA) and a Serial Clock Line (SCL).

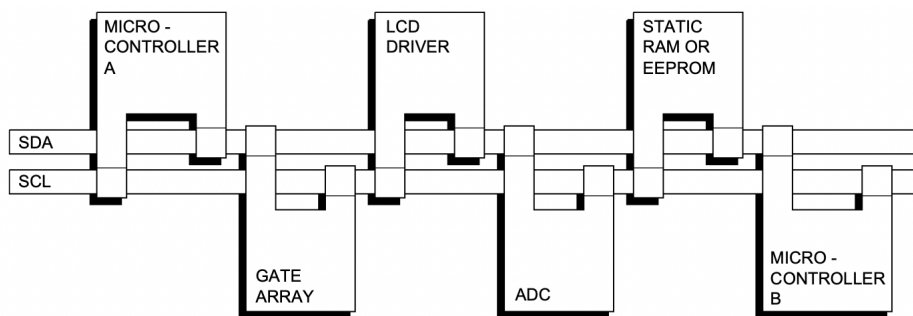


Figure 3.7: I2C lines of communication [6]

3.2.1 I2C Protocol

I2C is a serial communication protocol widely used in industrial environments. It works as follows: there are two communication lines, one called SDA and the other called SCL. Both are kept high (at the supply voltage) by a pull-up resistor, and can be pulled low (to ground) by the pins of the MCU (or any device connected to it).

The SDA line is where the data travels, always sent in 8-bit packets, while the SCL line carries the clock signal. All devices that want to communicate via I2C connect to these two lines. Although I2C lines can be multi-master, usually there is only one master on the line and one or more slaves. Each element on the line is assigned a 7-bit or 10-bit address (in our project, we use 7-bit addresses), and when the master wants to communicate with one of the slaves, it must first send a

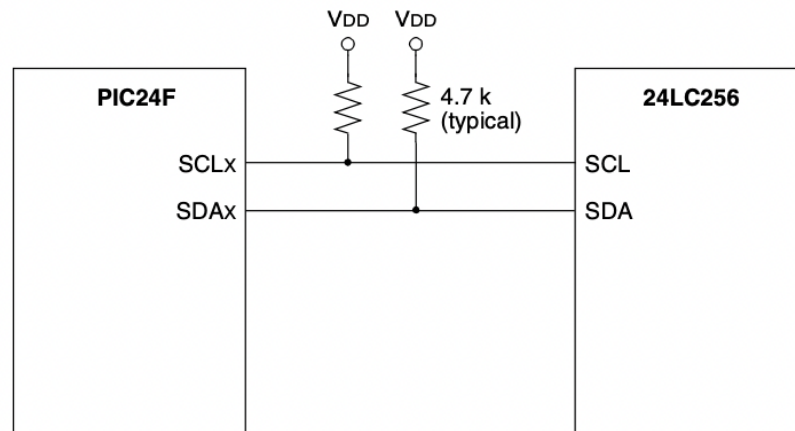


Figure 3.8: Typical I2C Interconnection Block Diagram [5]

packet with the address of interest. The slave that recognizes its own address on the line responds with an ACK (Acknowledge). If no slave responds, the master will read a NACK (Not Acknowledge) on the SDA. Now let's look at how this process occurs from a protocol perspective.

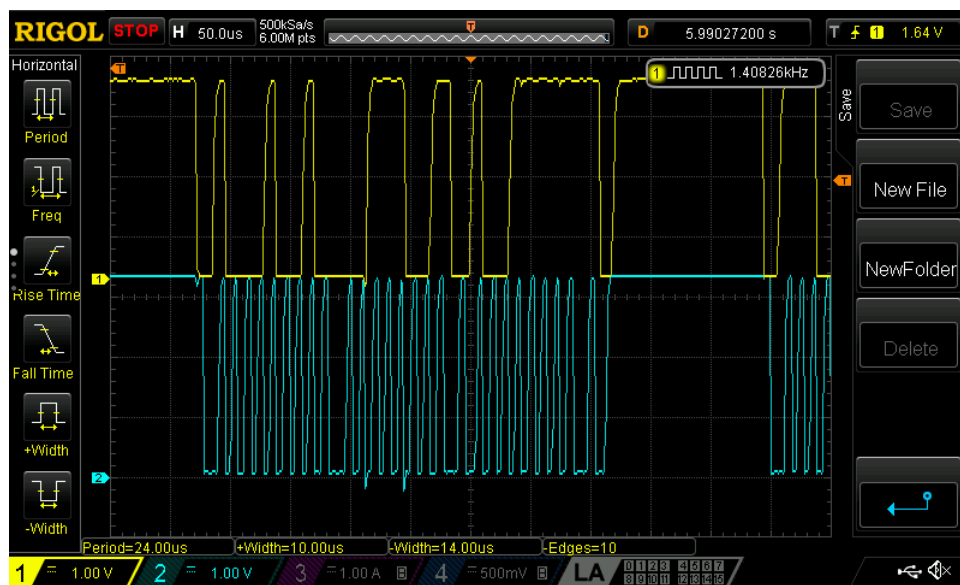


Figure 3.9: I2C communication between the MCU and a sensor, read with an oscilloscope

To describe when a line is pulled to ground (logical zero), we will use the term “lowering”, and for when the line is kept at Vcc (logical one), we will use the term “raising”.

To start communication, the master lowers the SDA. Lowering the SDA while the SCL is raised is interpreted as a START signal. At this point, the master takes control of the SCL and generates a square wave: the clock signal. Only that master will control the clock line for the entire duration of the communication. This is an extremely important detail, as one of the main issues in communication between electronic elements is synchronization. By having a single element manage synchronization, many problems are resolved.

Once the START signal is sent, the master sends the address of the slave it wants to communicate with. During each clock pulse of the SCL, one data bit is transferred, identified as follows: data is read on the SDA, which can only change when the SCL is low. The data is then read when the clock is high.

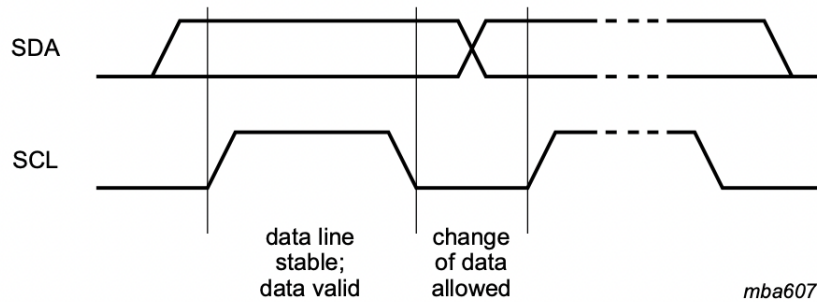


Figure 3.10: I2C data recognition [6]

In I2C, data is always sent in bytes, so after the 7-bit address one more is available. This is reserved to indicate whether the ongoing transaction is a Read or Write operation, namely whether the master is sending data to the slave, or requesting data from it.

Finally, there is one last bit that separates each packet, which is reserved for acknowledgment. The acknowledge allows the receiver to inform the transmitter that it has correctly received data, and that another byte may be sent. It takes place at the ninth bit of every packet transmitted. If the receiver desires to send an ACK (acknowledgment), it will lower the SDA line on the ninth clock pulse. Otherwise, it can leave the SDA line raised, which will be interpreted as a NACK (non-acknowledgment) by the transmitter. In the case of the address transmission, the slave that recognizes its address responds with an ACK. Data exchange then begins, either from the slave if it's a Read or from the master if it's a Write. This is the basic principle of communication. Data packets continue to be exchanged in this manner until the master decides to end the communication. At that point, it issues a stop signal, which occurs when the SDA line goes from low to high while the clock is high.

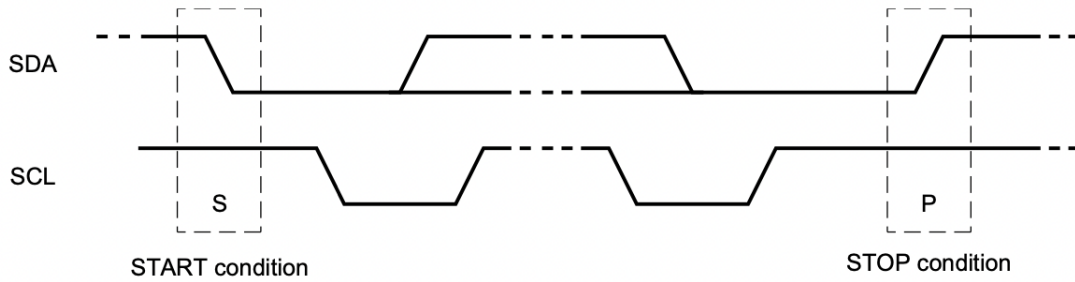


Figure 3.11: I2C START and STOP signals [6]

Collisions

In a multi-master environment, it is possible for multiple masters to attempt to initiate communication simultaneously, or one may start a new transmission while another is ongoing. These are called collisions. Fortunately, I2C includes “collision detection”.

PIC24 uses bus arbitration, that “ensures that if more than one node attempts a message transaction, one, and only one node, will be successful in completing the message” [5]. If a master tries to communicate while another device is already in control of the SCL, then it will lose arbitration and be left with a bus collision. When a collision is detected, an interrupt is raised, allowing the software developer to choose how to handle the situation.

A collision interrupt can be very useful also in hardware errors. Indeed if the SDA line, the SCL line or both, are in ground fault, then the master cannot communicate anymore, as all three cases are continuously read as collisions.

3.2.2 Read and Write operations as Master

Writing to a Slave

To write to a slave the MCU will send on the I2C line a START, followed by the slave’s address. It will complete the address with a 0 bit, to indicate that it is a Write operation. The slave will then respond with an ACK. In this ninth bit, the master may momentarily leave the clock control to the slave, doing a “clock stretch”. This can be done to allow more time to the slave to compute its answer. If the slave is a very basic device, the MCU can immediately start to transmit its data. But if the slave is more complex, it may have many registers containing different data. Then, the MCU will first send the address of the register of interest.

After the slave’s ACK, the MCU can send its data bytes, each acknowledged by

the slave. This process continues until either the slave responds with a NACK, or the master finishes the data to send. In the latter case the master sends a STOP signal to conclude the communication.

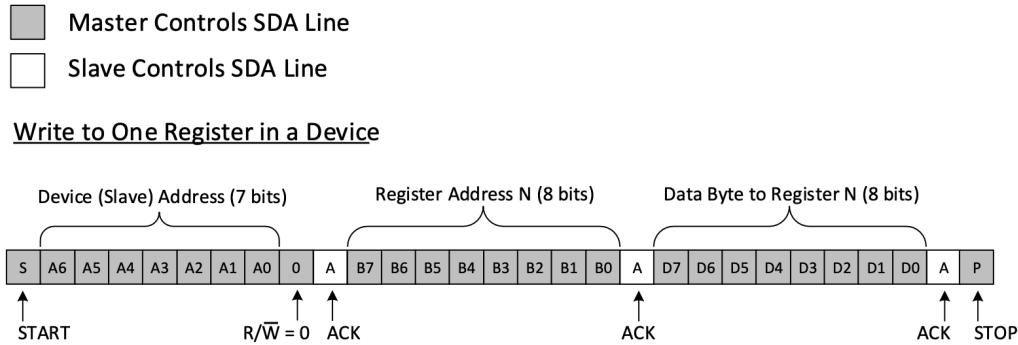


Figure 3.12: Example of the writing of a single byte to a slave register [8]

Reading from a Slave

To read data from a slave device, the MCU initiates communication by sending a START condition on the I2C line, followed by the slave's address. The eighth bit is set to 1 (Write flag), as the slave requires either the register from which to read, or a specific command on how the data must be sent. So, after the slave acknowledges (ACK) its address, the MCU sends the register or command to be read.

Now that the master has specified which data it requires, it issues a REPEATED START condition, which functions as a new START command issued before a STOP. Its purpose is to continue communication without releasing control. Otherwise, to restart the communication the master would have to issue a STOP, release the communication lines, and start a new one with a START, risking losing arbitration.

The master can now send the address with the eighth bit set to 0 (Read flag). With the read command given, the slave begins transmitting data on the SDA line, while the master remains in control of the clock signal, unless clock stretching is enabled. After each byte received, the master sends an ACK to continue data transfer. When the master has received the desired number of data bytes, it signals the end of the communication by sending a NACK, instructing the slave to stop transmitting. Finally, the master sends a STOP condition to complete the communication.

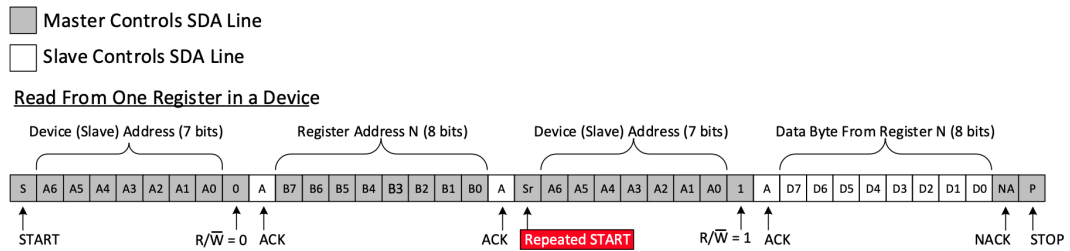


Figure 3.13: Example of the reading of a single byte to a slave register [8]

3.2.3 I2C on the MCU side

The MCU has two I2C outputs. For each one, there are two pins, the SCLx pin, dedicated to the clock line, and the SDAx pin, dedicated to the data line. “x” is replaced by the number of the communication port (SDA1-SCL1 and SDA2-SCL2). These outputs can be set as either master or slave outputs.

In our case, the output dedicated to communication with the embedded computer is set as a slave, meaning the MCU acts as a slave on the line. The other output, dedicated to communication with peripherals (such as sensors and memories), is set with the MCU acting as the master.

We’ve seen how the MCU functions as a master, and the process is largely symmetric when it acts as a slave. However, one challenge in slave mode is selecting the specific data requested by the master, as the MCU can supply a large amount of data and is missing dedicated physical registers for slave communication. To resolve this, software registers are implemented, allowing the master (the computer) to specify which register it wants data from.

The I2C module generates two types of interrupts for each line. One is assigned to Master-type events (MI2CxIF), and the other to Slave-type events (SI2CxIF). “The MI2CxIF interrupt is generated upon completion of these master message events:

- Start condition
- Stop condition
- Data transfer byte transmitted/received
- Acknowledge transmit
- Repeated Start
- Detection of a bus collision event

The I2CxIF interrupt is generated on detection of a message directed to the slave, including these events:

- Detection of a valid device address (including general call)
- Request to transmit data
- Reception of data

” [5].

These interrupts, aided by some registers, are our tools for the control of this communication.

Relevant Registers

The main MCU registers of interest are: Control Register (I2CxCON), Status Register (I2CxSTAT), Receive Buffer Register (I2CxRCV), and Transmit Register (I2CxTRN). The first is the one that controls I2C actions, changing its bits we can start hardware operations. The second “contains status flags indicating the module’s state during operation” [5]. It is very useful, as it is used to understand which part of the transmission has generated the current interrupt. The third is a read-only buffer, where the received data is stored until is read. The fourth is where the data is written when it is to be sent.

Physical example: START operation as Master

To begin a START operation, the first bit of I2CxCON, also called SEN (Start Enable) bit, must be set. This will cause the hardware operation to initiate. The I2C module contains a Baud Rate Generator, set at the beginning of the project. After that the MCU will pull down the SDA line (I2C START sequence), and the S and P bit will be changed. These two are two bits of the I2CxSTAT register, that signal the status of the communication. If the P bit is set then the last action detected was a STOP action (no communication is in progress), while if S is set then the last action recognized was a START or a REPEATED START. Both are hardware set/cleared, which means that it is the electronics that automatically do this, it cannot be done via software. So now that the S bit is set and P bit cleared, the master interrupt is called. In the master interrupt the software developer can decide how to manage the start operation, for example preparing the I2CxTRN register.

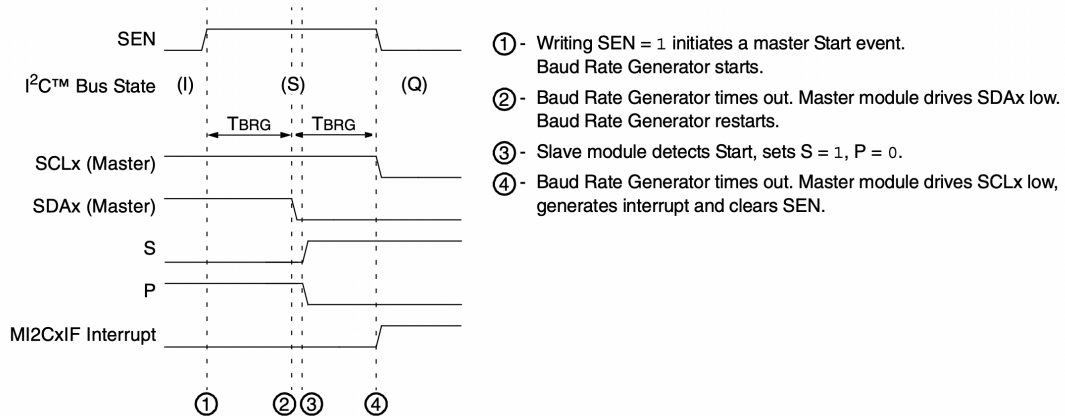


Figure 3.14: Example of sequence of events, START case used [5]

Software example: Master Transmission

To understand how to handle transmission using the interrupts, we consider the following example. Taking the case of Master transmission in a Read request, we observe Figure 3.15. Here, the interrupt is triggered multiple times during transmission, and as explained in Section 3.1, each interrupt has a single ISR associated with it. Therefore, the firmware developer must be able to identify the communication stage at each interrupt trigger to respond accordingly. This can be achieved by utilizing the previously introduced status register.

For instance, during a master transmission, the first interrupt generated is easily identifiable as it marks the initiation of the START sequence that we have triggered. However, identifying following interrupts, such as the ones at the ninth clock, can be harder. In some cases, for example in interrupts caused by a collision can be easy. The STAT bit can be checked. But how to tell whether to send the address or initiate a REPEATED START. We have to use the knowledge of previous operations. To do so set of internal variables are used, called structs, to maintain knowledge of where in the operation we are.

Similarly, in the case scenario of the MCU operating as a slave, we implement separated structs. In this mode, the MCU does not control the communication, making it more challenging to determine the current state.

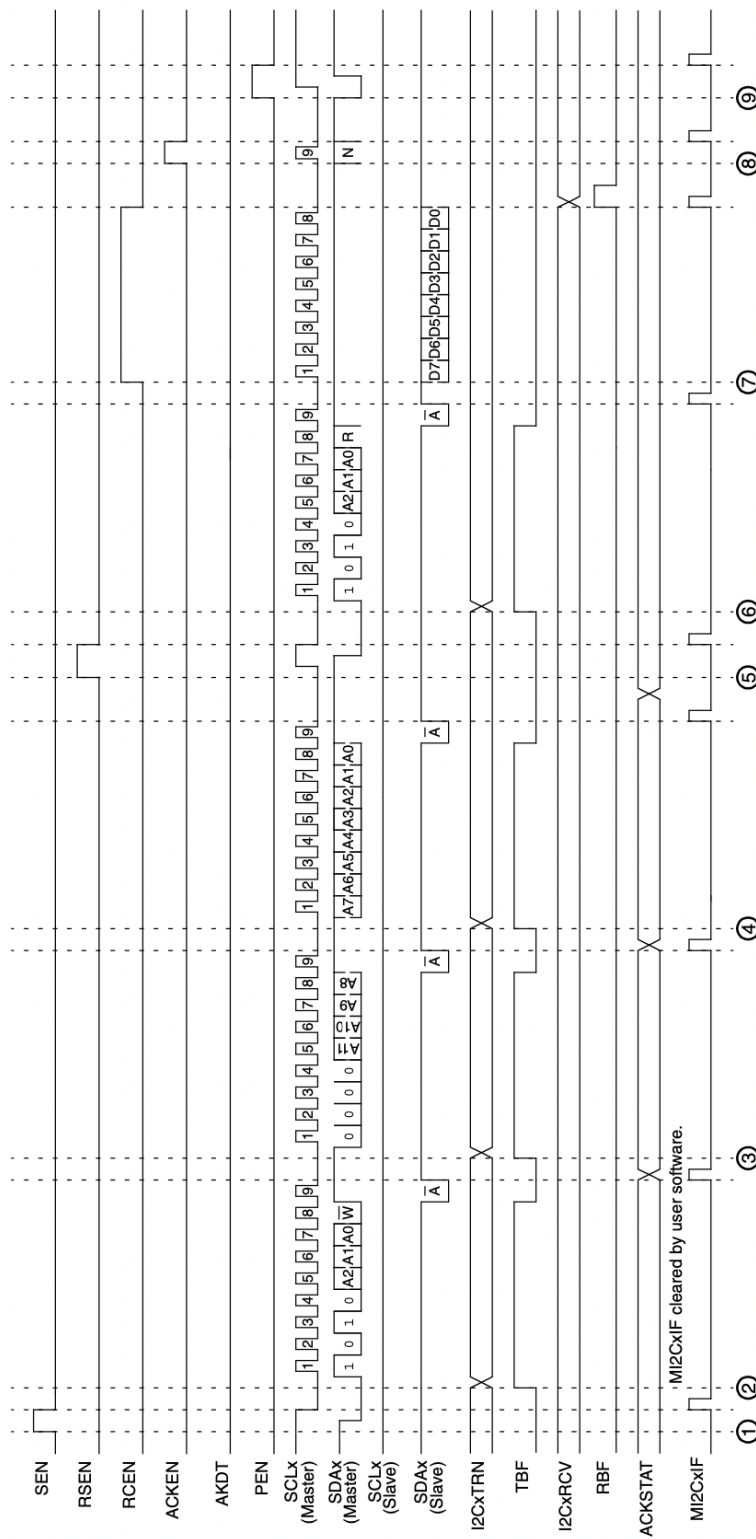


Figure 3.15: Example of sequence of events, Master Read Transmission case used [5]

Chapter 4

Objective of the project

In this chapter, the fundamental requirements that guided the development of the code are presented.

This section is dedicated to give a complete understanding of what the aim of this thesis was, while understanding the limitations given by the choices made throughout its implementation.

4.1 Objectives

The requirements that this code must satisfy are many, there are the main function ones, the error management ones, the communication ones and finally the special ones.

4.1.1 General Objectives

These are the objectives that the main flux of the code must satisfy.

The code must:

1. Power on the embedded computer safely when machine is started up. This implies powering on the embedded computer only if:
 - 1.1 The voltage is reliably over the safe voltage line
 - 1.2 The ignition signal is reliably high
 - 1.3 The safety requirement are met, if any are set (it can be possible to set a second minimum voltage threshold to exceed)
2. Power off the computer safely when the ignition signal becomes reliably low. This implies the following:

- 2.1 The computer is not powered off by removing the power supply, but sending a pulse signal
- 2.2 The power supply is not removed right after the pulse signal but only after the computer is reliably off
3. Power on the computer in accordance with its power-on modality, indicated at setup
4. Avoid turning the computer on, and completely bypass it altogether if the system is setup in POWER BOX modality
5. Accurately manage timing and delays between actions
6. Show its status via bicolor LED signaling
7. Continuously monitor ISO BOX environmental status via a temperature and humidity sensor
8. Be constantly ready for embedded computer's requests for data (setup values or diagnostics)
9. Continuously monitor battery voltage (`line voltage`) and key signal (`ignition voltage`)
10. Check correct functioning of I2C communication line at beginning

4.1.2 Communication with computer

These are the requirements that the firmware must satisfy in regard of the communication with the embedded computer.

The firmware must:

1. Respond to computer requests of setup info
2. Save setup changes in EEPROM if they contain valid data, where "valid" varies based on the specific setup parameter
3. Respond to computer requests of diagnostics data
4. Power on or power off Polispec if computer requires so
5. Power computer off if it asks so. In this case the user must be able to restart the device by pressing the power button for a long time

4.1.3 Error management

Possible errors must be managed too. Here's a list of all the error that the firmware takes action on. The action itself will be later discussed in Chapter 5.

The code must manage:

1. An overtemperature of ISO BOX: the temperature sensor placed in ISO BOX registers a stable value over the safety threshold. The overtemperature threshold is set at configuration
2. The recovery procedure once the overtemperture condition has been overcome
3. An undervoltage condition, so if the **line voltage**, which is the tension coming from external battery, is stably lower than the safety threshold
4. The recovery procedure after the undervoltage condition has been overcome
5. I2C communication errors. If one of the peripherals stably stops answering, it takes different actions based on when the error occurs and which element is not answering
6. The computer not powering up at all
7. The computer powering off by itself after it has been turned on
8. The computer being powered on prematurely, before the firmware has executed the necessary action to power it on
9. The computer not turning off after the power off pulse has been sent

4.1.4 Power button actions

The power button present on the front panel has been used to take several actions. Here we list all the actions that the firmware must take when registering a button pressure signal.

The code must:

1. Not take action if the signal is shorter than 500 ms
2. Differentiate between a "short press" and a "long press" signal. The length of the two will be shown in Chapter 5

3. During powering on, a long press must have no effect, and it must be “forgotten”, meaning it will not take future actions on the rest of the code
4. During powering on, as stated in point 1.3 of General Objectives, extra safety requirements can be set. If those are established, but they’re not met in the practical field, the user must be able to bypassed them via a short press. After being used the signal must be “forgotten”, meaning it will not take future actions on the subsequent code
5. During normal course of the code (what will be addressed as “Main State” in Chapter 5), if a short press is identified, then a power off pulse must be sent to the computer. This is used for testing, for example to simulate a random computer power off
6. During normal course of the code, if a long press is identified, it must safely power off the computer, just as if the ignition key were to be turned off. The device will then go in a still, waiting state. Here if the ignition signals goes stably low the code ends, otherwise if another long press is detected, the code restarts from the beginning

4.2 Preconditions

Here we summarize all the necessary characteristics that a customer or technician must follow to use this device correctly and ensure its proper functioning.

It is necessary that:

1. Configuration parameters are correctly set. In particular:
 - 1.1 Thresholds
 - 1.2 Embedded powering on modality
 - 1.3 POWER BOX mode or ISO BOX mode
 - 1.4 Polispec thermal feedback presence
2. Avoid tampering with ISO BOX electronics, such as modifying electrical connections or adding peripherals
3. Work within the safety thresholds of temperature and voltage

4. The developed firmware communicates with a properly updated computer, provided by ITPhotonics, as it must have the correct program to communicate as a slave via I2C.

4.3 Constraints

Here we show all the projects constraints.

1. This code cannot be modified on the fly once it is uploaded on the final instrument. For any bugs, updates, or operations requiring re-upload of the code, the instrument has to come back to the manufacturer (ITPhotonics) as it must be opened to gain direct access to the powerboard
2. Setup changes are accessible only from the embedded, therefore when in POWER BOX mode, where the embedded is bypassed (and usually not even present), changing them can be tricky. The instrument may have to come back to the manufacturer
3. Some setup changes cannot be made ‘on the fly’, as they would impact too much the code progression. For those it is necessary to set them, then turn off the entire ISO BOX, and at poweron they will be set. An example is the POWER BOX vs ISO BOX setup
4. If at the beginning one of the peripherals does not respond, the code blocks the entire process

Chapter 5

Software

This Chapter describes the firmware developed for the device in detail.

It begins by introducing the setup and diagnostics information accessible to the user, followed by an overview of the logical flow of the code, including an introduction to the one of the original ISO BOX code.

The Chapter then outlines the use of timers and delays in the code, alongside error management practices, showing how these elements satisfy the previously mentioned project's objectives. It then gives a list of LED signaling.

Finally, it concludes with a little digression in Section 5.4.3, explaining a study done for safety power-up taking into consideration the engine cranking, and the following code developed.

For company privacy reasons, the actual code is not shown.

5.1 Setup and Diagnostics

The firmware controls the powering on of the computer, and once it is on, the communication between the two starts. This communication, done via I2C, uses registers to distinguish which information the computer wants from the ISO BOX. Because the PIC MCU is the slave, is its registers that the computer questions. These registers are not physical hardware components but are instead implemented in software. The firmware uses an array to store all the registers information. When the computer needs to request data, it sends the number of the desired register. The MCU then interprets this request, identifies the corresponding memory address, and transmits the appropriate information back to the computer.

Using these, the computer can request information from the ISO BOX, change

firmware configuration, and even perform action. Lets see what information is exchanged.

Diagnostic Information

Via the embedded computer, some diagnostics information can be obtained, and eventually used by the computer to decide how to act. The associated registers give the following information:

- I2C error status. Tells if the communication between embedded and peripherals is working properly
- Local overtemperature status. Tells if the ISO BOX is in overtemperature condition or not
- Undervoltage status. Tells if ISO BOX is in undervoltage condition or not
- Local Environmental parameter. Tells ISO BOX's temperature and humidity
- Polispec power status. Tells if it is ON or OFF
- Computer boot status. Tells if the embedded computer, once it has turned on, has completed its booting
- Polispec thermal loop status. Reports passively the Polispec thermal status, so whether the Polispec is in overtemperature or not (value: TRUE/FALSE if enabled, else always FALSE)

Control Actions

Some control actions can be requested by the computer. By changing some I2C registers it triggers the firmware to act.

The action it can prompt are limited, and are the following:

- Power ON the Polispec
- Power OFF the Polispec
- Turn itself OFF (obviously, as it cannot communicate via I2C if it is turned off, it cannot do the opposite). If set, this register starts the power-off sequence and prevents automatic re-power-on sequence to start. The register clears automatically after the execution

Setup Information

The setup can be configured and checked through the embedded computer via I2C. The parameters that can be set are listed in Table 5.1.

Table 5.1: Setup Parameters

Parameter Description	Default Value	Range Of Value
ISO BOX / POWER BOX mode	ISO BOX mode	-
Power-on Safety Check	Disabled	-
Computer Power-on Mode	5V	-
Polispec's Thermal feedback expected	YES	-
Power-on Safety Check Threshold (if ON)	13.5V	[10 - 18]V
Under Voltage Threshold	7.2V	[6 - 9.5]V
Over Temperature Threshold	65°C	[40 - 100]°C
System Serial Number	0	-
Firmware Version	HARDCODED	-

5.2 Logic flow

To better understand the final code, first is explained its logical flow, starting from the one of the original ISO BOX.

5.2.1 Isoboard Firmware

The original ISO BOX firmware was uploaded on a MCU mounted on board named "Isoboard". Therefore, to distinguish the two codes, this will be referred to as "Isoboard Firmware".

It is a Finite State Machine (FSM), meaning that it is a State-oriented code. Its states are shown in Scheme 5.1.

It is mainly composed by 12 states, that can be divided in three categories: Power-on states, Main state and Power-off states.

Power-on states

This group of states manages the preliminary safety checks, the computer power-on sequence, and the Polispec power-on sequence.

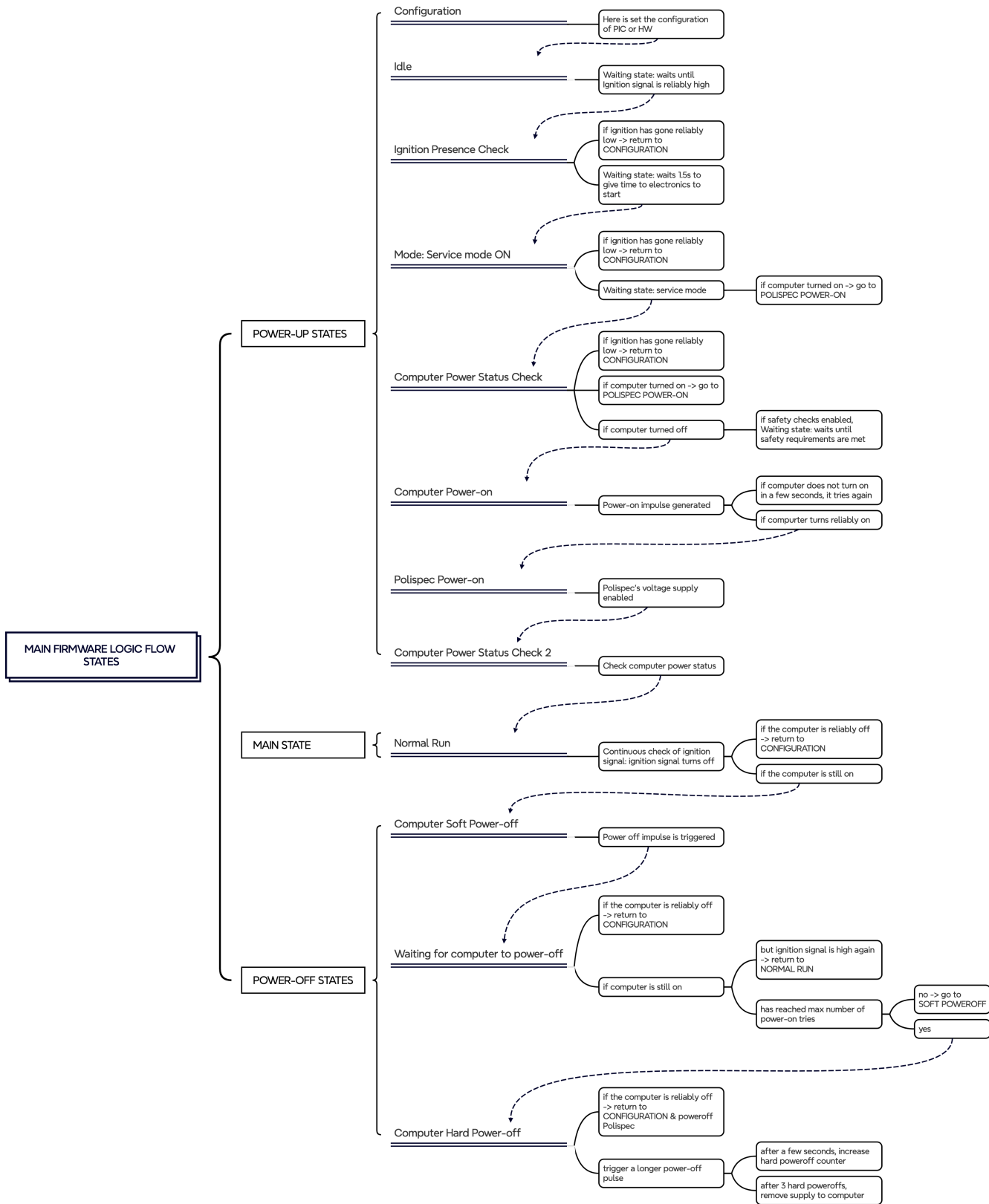


Figure 5.1: Old ISO BOX firmware logic flow

The firmware initiates when the Isoboard receives both line voltage and ignition voltage. After an initial configuration state, where hardware configurations are set, timers initialized, and I2C communication established, the safety controls begin.

The first set of states are “waiting” states, where code does not progress until specific conditions are met. The process begins in the IDLE state, where the firmware waits for the ignition key to be reliably “on”. Here, a reliable signal is defined as one that maintains a stable state (high/low) for a predefined duration. A threshold is set to determine whether the ignition signal is “on”, if the signal is below, the Isoboard remains powered, but the code does not progress further. Once this condition is satisfied, a second waiting state follows. In this state, the firmware waits for a timer to complete, allowing time for the electronics to perform necessary operations. After this timer completes, the firmware checks for “Service Mode”, a method to power-on the computer manually, that has now been removed.

Following these waiting states, the firmware performs a safety check on the computer power status. At this point, the computer should be powered off based on firmware control, however it is essential to verify whether it has been powered on manually. If it has, the firmware must skip some intermediate states (to avoid powering the computer off) and proceeds directly to the Polispcc power-on sequence. Otherwise, if at check the computer is off, one last assessment is done before starting its power-up routine.

This final check, which can be enabled or disabled, is relevant in scenarios involving cranking. As will be discussed in Section 5.4.3, the engine cranking can cause substantial battery voltage instability. If the electrical system of the machine is supplied by the same battery (**line voltage**), which is often the case, it could have repercussions on our instrument. For example, if an operator turns on the machinery electrical system but not the engine (such as when you do a half turn of your car key), the Isoboard will power on and initiate the firmware. However, if the operator subsequently starts the engine, initiating cranking, both the Isoboard and the computer could lose power abruptly, risking potential filesystem damage. To prevent this, we exploit a characteristic of cranking, visible in the Figure 5.12 in Section 5.4.3. Once cranking ceases, the alternator acts as a dynamo, charging the battery and raising the voltage beyond its initial level. This safety check requires that the **line voltage** (battery voltage) is reliably higher than a set threshold. The user can configure this threshold or disable it,

depending on machinery specifications, as described in Section 5.1.

Throughout each state, the firmware continuously monitors the ignition signal. If the it goes reliably off (under the chosen threshold), indicating the machinery has been turned off, the firmware takes appropriate action based on the computer's power status. If the computer has not yet powered on, the system safely goes back to the beginning of the code. Otherwise, the firmware must ensure its safe power-down sequence.

The process then proceeds to the power-on states. The embedded system powers on via a pulse signal, which simulates the manual press of the power-on button. If the computer does not power-on, the firmware repeats this pulse until successful. Following the computer power-on, the Polispec voltage supply is enabled, completing its power-on sequence. A final check then verifies the computer's power status, as the computer could inadvertently power off when the instrument is activated. This could occur due to a brief impulse noise caused by the powering of the 12V supply to the Polispec, which the computer may interpret as a power-off pulse. In this case, firmware returns to the begin without turning off the Polispec, so that the entire process my be done again without the risk of powering off the computer. Although it could be better to power on the Polispec first, this is not feasible given the current electrical board design, but its explanation is beyond the scope of this thesis.

Main state

The main state, known as the Normal Run State, operates for the majority of the time. Its primary objective is to continuously monitor the ignition signal and initiate the power-off sequence when the signal goes low. During this state, "normal" activities functions are executed at each cycle of the main loop, such as system monitoring, communication with the computer, and other routine operations.

Power-off states

The power-off states are encountered at the end of the code. First, the code initiates the power-off process by sending a power-off pulse to the computer. Following this, it enters a waiting state, allowing time for the embedded computer to power down. If the computer shuts down as intended, the Polispec is also turned off, and the code finishes.

However, if the embedded computer remains on, and the ignition key signal turns back on too during the waiting period, the code will resume operations in the

Normal Run State. If the embedded computer remains powered despite the ignition signal staying low, the code reattempts the power-off sequence by entering the soft power-off state again. This reattempt sequence is repeated for a predefined number of tries.

If the embedded computer still does not power down after these attempts, the code advances to the Hard Power-Off State. In this final state, the code sends a longer pulse signal, waiting for shutdown. Up to three such long pulses may be sent if necessary. Ultimately, if the computer still fails to turn off, the power is cut entirely.

5.2.2 New ISO BOX firmware

To avoid explaining again the states that are similar or identical to the previous code, only the differences and novelties will be presented.

The updated flow is divided into four sections: Power-on States, Normal Activity, Power-off States, and Error Management States. We also show important functions that are done at each code cycle. This code now accommodates different computer power-on methods and distinguishes between ISO BOX and POWER BOX modes, adding complexity.

Power-on States

The updated flow introduces two distinct initial states: Configuration and Re-Configuration. The process begins in the Configuration state as before, but now, when the code needs to restart, it transitions to the dedicated Re-Configuration state. This avoids reinitializing all configurations unnecessarily.

Following the Ignition Voltage Check, the previously included Service Mode state has been removed. Instead, a Line Voltage Check state has been added to ensure proper voltage levels at startup. After this, a specific function (detailed in Paragraph Functions) continuously monitors the `line voltage` to guarantee safety in undervoltage conditions.

The Computer Power Status check has been made more sophisticated. If the computer is found powered on before the firmware executes its designated power-on sequence, this is now classified as an error. In such cases, the code transitions to a dedicated error management state (explained in Paragraph Power-off States). If instead the computer is correctly off, the next step is to check the enabling of the safety check. If it is on, then the process proceeds to the next state, the Line Voltage Safety Check. If the safety check is disabled, the subsequent action

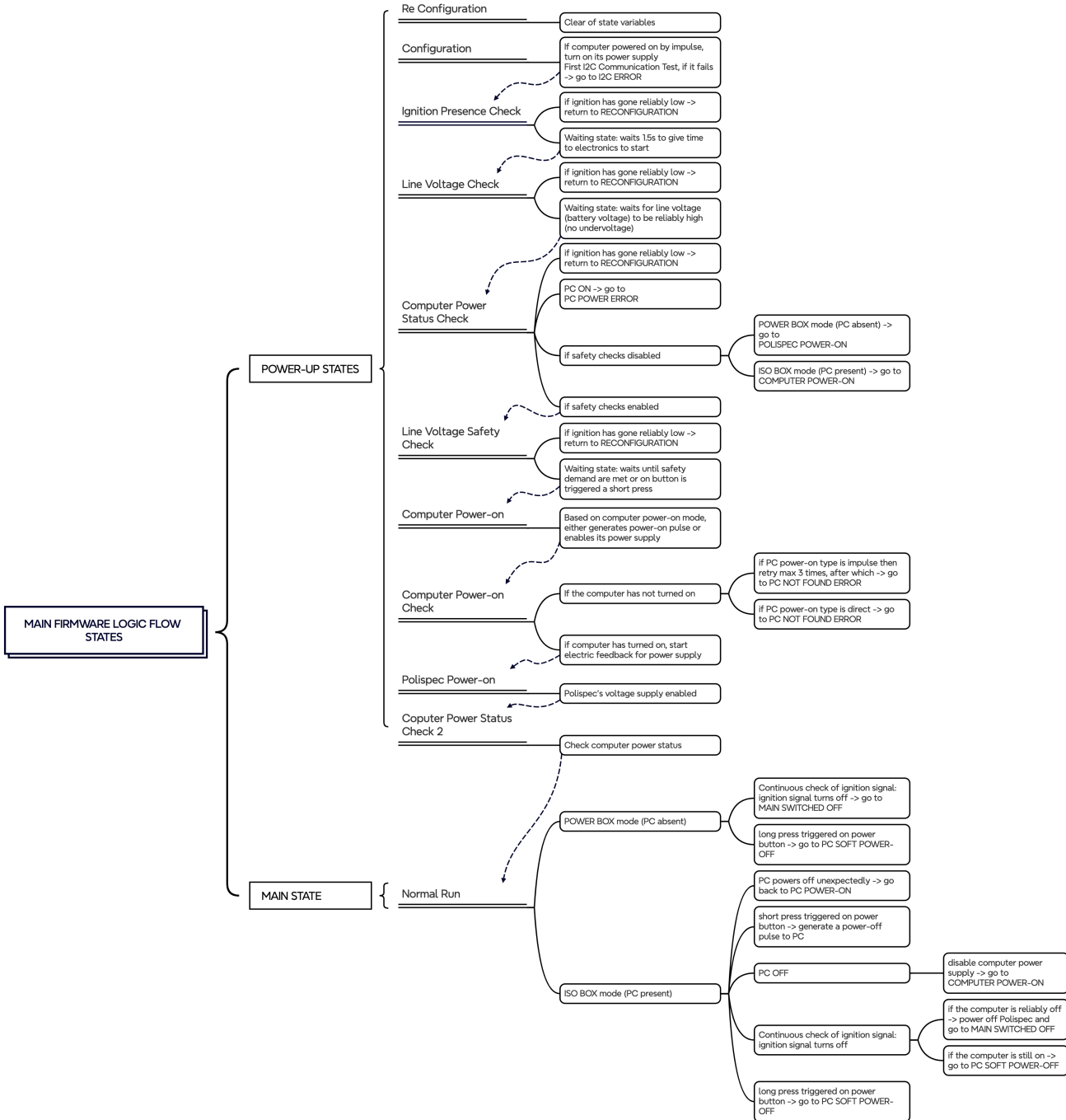


Figure 5.2: Current ISO BOX firmware logic flow, part 1

depends on the system mode (ISO BOX or POWER BOX). If the computer is present (ISO BOX mode), the firmware initiates the power-on sequence; if not (POWER BOX mode), it directly powers the Polispec.

Previously, cranking-related safety checks were conducted within the computer power state. In this updated version, these checks are now separated into the Line Voltage Safety Check State. If the user has enabled the safety check, but set a threshold that cannot be satisfied, the code could get stuck here. To avoid the risk, this state can be bypassed by briefly pressing the power button.

The computer power-on process remains mostly unchanged, but includes new error handling. If the computer fails to power on, the firmware transitions to an error state. For computers requiring an impulse signal to power on, the sequence is attempted three times to account for the possibility of the impulse not being correctly registered. For computers powered solely by supply voltage, repeated attempts to turn-on and subsequently off the power supply are unnecessary, as this clearly indicates an error.

The final two states are functionally identical to those in the previous code. However, in POWER BOX mode, the last state is skipped, as it is not applicable in this context.

Normal Activity

This state is largely similar to the previous version but incorporates different actions depending on the presence of the computer. In POWER BOX mode, the firmware can just monitor the ignition signal and the power button signal. If the ignition signal reliably goes low, the system transitions to the Main Switched Off state, effectively ending the process. If instead the flag associated to a long press on the power button is raised, then a special state is entered, which, while technically an end state as well, functions differently. This special state is discussed in greater detail in Paragraph Power-off States.

If the computer is present, additional scenarios are considered. As before, if the ignition signal goes low, the firmware initiates the power-off procedure. However, in this version, the procedure is also triggered by a long press of the power button. A short press of the power button, on the other hand, generates a power-off pulse directed at the computer. This feature is intended solely for debugging purposes, allowing users to restart the computer if it is behaving unexpectedly. Importantly, the system does not transition to a different state in the short press case, it stays in Normal Run. Within the Normal Run state, if the computer is

found to be reliably powered off (either due to a user action or because it turned off unexpectedly) the firmware tries to power it back on.

Power-off States

The three main states (soft power-off, waiting, and final shutdown) are maintained, with some refinements. The soft power-off and the waiting state are mostly unchanged, though the waiting state now accounts for additional scenarios. Specifically, the computer may power-off for various reasons. If the shutdown occurs due to an I2C communication issue (explained further in Paragraph Functions), the system transitions to a dedicated error state. If the computer powers off correctly, the process moves to the true end state: the Main Switched Off state. This final state ensures the firmware remains idle as the entire system shuts down. If the system is powered off using the power button, it enters the Button Power-Off State, which functions as a waiting state. In this state, the system waits for one of two events: another long press of the power button, which causes the code to restart, or the ignition signal going low, which transitions the system to the end state.

If the computer remains powered on despite shutdown attempts, the system behaves as before. If the ignition signal returns, the code restarts. Otherwise, the process escalates to the Hard Power-Off State, where the power supply to the computer is completely cut. Following this, the same checks performed in the soft power-off state are repeated.

A significant modification to the power-off logic has been introduced: previously, a set number of soft and hard power-off attempts were made. Now, the system proceeds directly to hard power-off after soft power-off fails. This change eliminates the risk of a feedback loop where a slow computer shutdown might be misinterpreted as a power-on request, causing the system to repeatedly power on and off.

Additionally, two error management states have been introduced. The first handles cases where the computer is found powered on before the firmware's power-on sequence. In this scenario, the Polispec is powered off, the system waits for a prolonged period (some minutes), and then performs a normal power-off sequence for the computer. The second error state addresses situations where the computer is not detected. In such cases, the system performs a hard power-off, powers down the Polispec, and waits for the overall system to shut down. This indicates a critical issue, such as an incorrect setup (e.g., the computer is configured as present

but not installed/the system should be in POWER BOX mode) or a hardware defect in the computer itself.

Error management states

The new category of states introduced in the firmware is Error Management States, comprising four distinct states: three dedicated to handling overtemperature conditions and one focused on managing I2C peripheral communication failures. These states are designed to address critical issues and ensure system safety and stability.

The overtemperature management states operate similarly to the power-off states but with a more immediate approach due to the urgency of the situation. In these states, the waiting period does not include checks on the computer's status before disconnecting its power supply. Once the computer is powered off, the final overtemperature state is dedicated to waiting for the overtemperature condition to resolve. The temperature sensor manager function is in charge of clearing the overtemperature flag when the situation stabilizes. For safety, even after the flag is cleared, the code remains in this state for at least 30 seconds after the computer's power-off. This ensures system stability before any state transitions occur.

The I2C error management state handles severe errors that can compromise the instrument's functionality. This state is triggered either at startup (if any peripheral fails to respond to I2C communication within a set timeframe), or during operation if critical voltage sensors lose communication. Entering this state implies a probable hardware error, even prompting a return to ITPhotonics for maintenance. So if the state is entered at beginning of code, it prevents the system from starting. The system signals this error by keeping the bicolor LED continuously red, indicating that repairs are necessary. For voltage sensor communication failures, the situation is particularly critical, as these sensors monitor the ignition and line signals. Without this data, the system cannot operate, as the firmware main objective is to monitor the system voltage. Upon entering the I2C error state, the system halts all operations and waits for communication to be restored. The communication is continuously test, the memory directly in the state, while dedicated functions monitor the status of humidity, temperature, and voltage sensors. The system can only exit this state if all sensors provide at least one response. An interesting case is the following. After resolving the error and re-entering the Reconfiguration State, the system must make sure that state

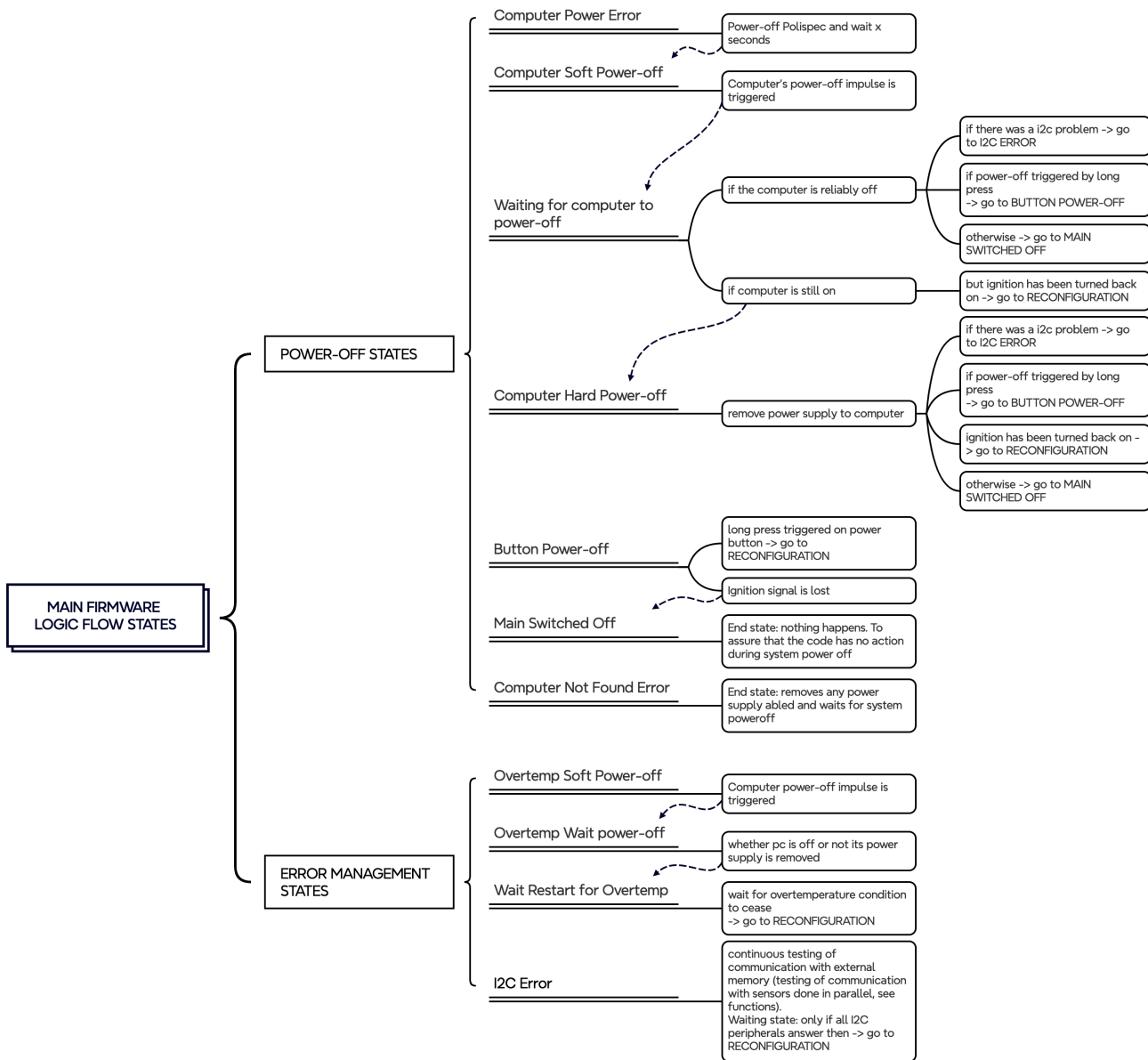


Figure 5.3: Current ISO BOX firmware logic flow, part 2

variables are correctly initialized, as if the I2C error occurred during the initial setup, these variables might remain unset. To fix this problem, a dedicated flag is used to track the successful execution of the Load Defaults function. When entering the Reconfiguration state, if it has not been set, the system loads the default settings.

Functions

At each main cycle, the firmware executes several key functions to ensure system monitoring, safety, and responsiveness. These functions include sensor management, I2C communication parsing, button press detection, thermal feedback updates, and error management.

The Sensor Managers are two functions responsible for interacting with the temperature and humidity sensor, and voltage sensors. They're responsible for requesting data, and setting flags for critical conditions such as overtemperature, undervoltage, and I2C errors. The last is set as the associated sensor does not provide any answer to requests of data for a set amount of time. These functions utilize a hysteresis for error handling to avoid rapid switching (or chattering) as thresholds are crossed. For example, the overtemperature flag is only cleared when the temperature stays reliably below a threshold set lower than the one used to set it. Similarly, undervoltage conditions are managed using the same principle.

The I2C Registers Parser ensures effective communication between the firmware and the computer. It cycles through the I2C registers, updating them based on changes to the system setup or responding to modifications made by the computer. Any changes in the setup are saved to external memory only if they fall within appropriate ranges.

The firmware also includes "passive" functions. The Power Button Check detects button presses and categorizes them as either short or long presses. A button press lasting more than 500 ms but less than 3 seconds raises a short press flag, while presses longer than 3 seconds raise a long press flag. Anything shorter than 500 ms is ignored. They're cleared in code. Flags are mutually exclusive: if the short flag is currently raised, but the button is pressed for a long time, then the long flag is set and the short one is cleared, and viceversa. The Thermal Feedback Update synchronizes the overtemperature signal with the computer by copying it to the appropriate register. The signal is visible in Section 2.3.3 of Chapter 2. Error Management Functions address the three main errors the system can en-

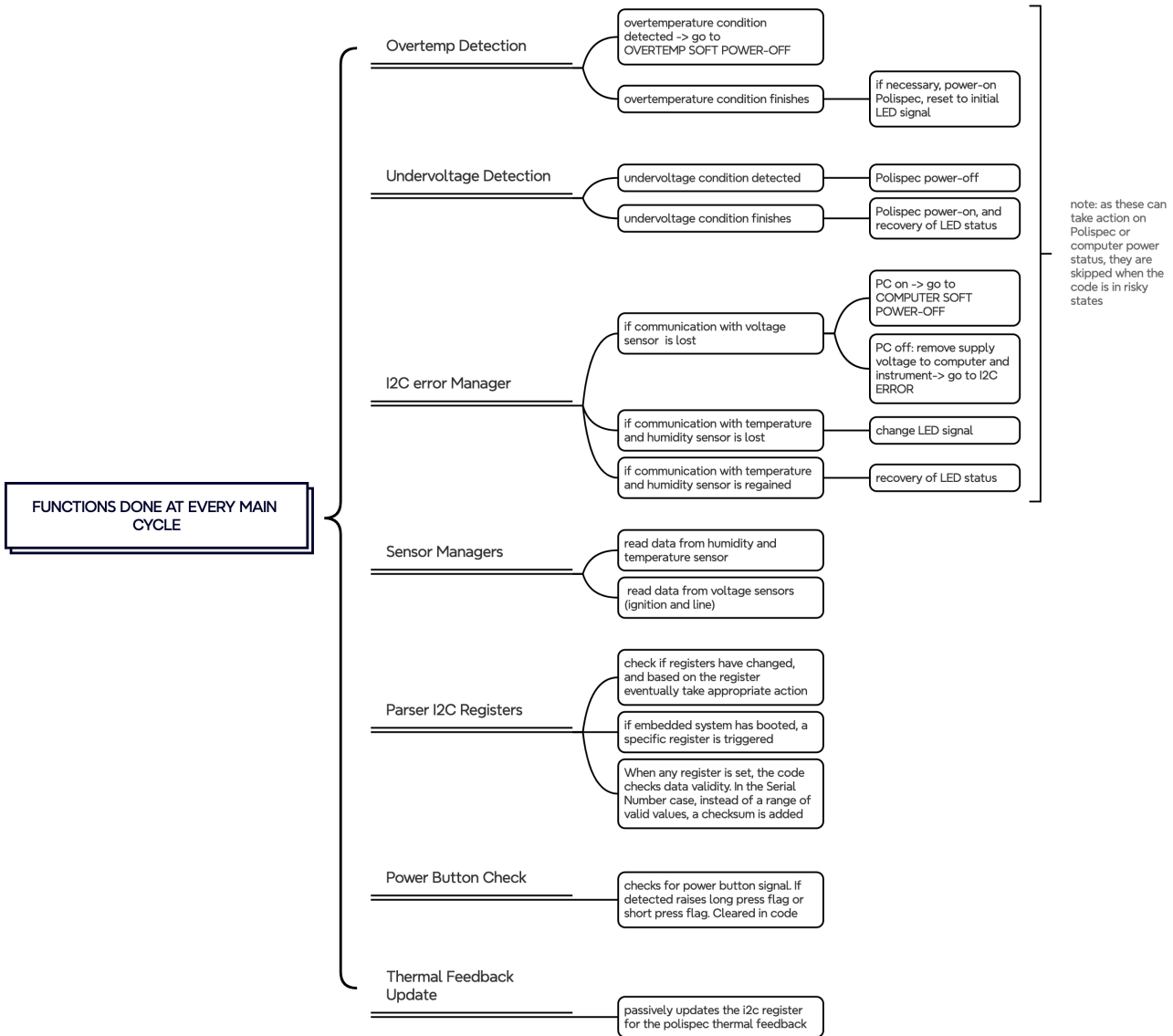


Figure 5.4: Functions performed by firmware at every main cycle

counter: undervoltage, overtemperature, and I2C communication errors. Since the ISO BOX is often mounted on machinery that operate continuously for extended periods, shutting it down creates risks of significant data loss. It is necessary to balance the system safety with these operational risks. Therefore, errors are classified based on their severity, so that critical issues are addressed immediately, while minor errors allow the system to continue functioning with minimal interference.

- Undervoltage errors are considered non-critical. When detected, the system powers off the Polispec, as the ISO BOX cannot maintain the 12V supply line. The error is signaled via LEDs, but the main code continues running uninterrupted. Once the undervoltage condition resolves, the LED signals are updated to show the current code's state, the Polispec is powered back on, and operations resume normally
- Overtemperature errors are more serious and initiate the previously described overtemperature management states. Typically, these states end in a system restart, meaning the cleared overtemperature condition happening is rare
- I2C communication errors addresses communication failures with peripheral devices. Loss of communication with external memory is considered non-critical, as it only affects the saving of setup changes, and the error will be noticed at next startup as the instrument will not start. For communication losses with the humidity and temperature sensors, the firmware flags the error via LED but continues its operations, as these failures are deemed lower risk. If the voltage sensors fail, it constitutes a critical error. These sensors are essential to monitoring the `ignition` and `line` signals. Without this data, the firmware cannot reliably control system power states, potentially leading to endless operation as the ignition goes low, but firmware is unable to detect it. If a voltage sensor communication error occurs while the computer is on, the system transitions into the power-off states. From there, as seen previously, the code will move to the I2C error state

5.3 Code Schemes

We show here an example of the typical progress of the code, in the specific cases of ISO BOX mode and POWER BOX mode.

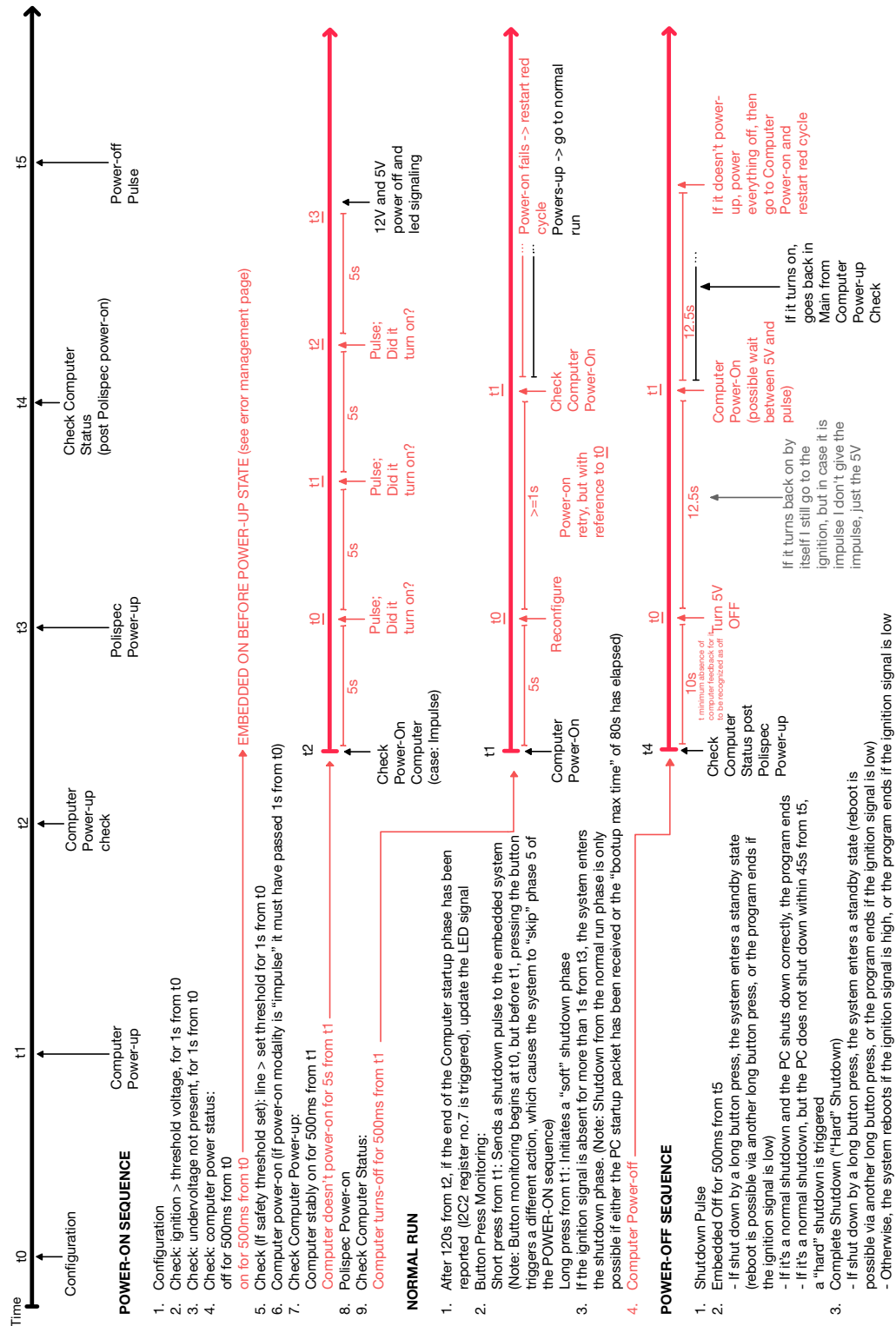


Figure 5.5: code evolution scheme with embedded

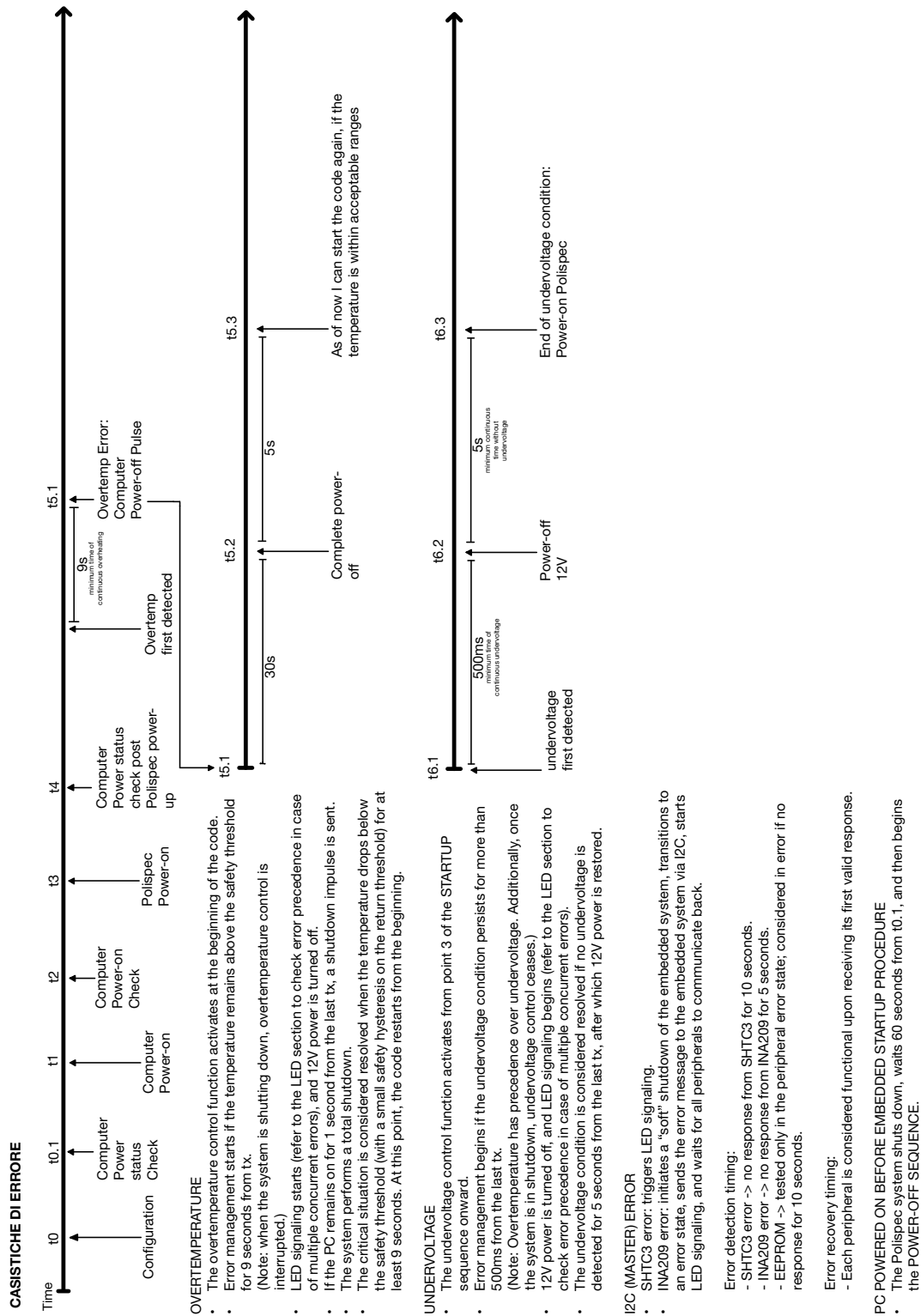


Figure 5.7: code error evolution scheme

LED signaling

This section outlines the LED signaling system implemented in the firmware.

The controlled LED is a bicolour LED capable of emitting both green and red light. The signals generated using this LED are listed in Tables 5.2, 5.3, and 5.4. To manage the conjunction of error cases, or determine which LED signal should take precedence, each signal is assigned a priority level. There are three priority levels: 1 (high), 2 (medium), and 3 (low). While the LED is displaying one signal, it can only be interrupted by a new signal with an equal or higher priority. For example, a signal with priority 2 cannot be overridden by one with priority 3. An exception to this rule is when a signal is assigned a “reset enabled” flag. In cases where a lower-priority signal needs to override a higher-priority one, the internal function can set this reset parameter. When enabled, this parameter allows the new signal to take precedence, regardless of its assigned priority.

In this implementation, I2C error signaling is assigned the highest priority (1) because it represents the most critical error.

Overtemperature and undervoltage errors are assigned medium priority (2), as they are important enough to override normal operational signaling but less critical than I2C errors. Even if they have the same priority, both errors share the same priority, the code structure ensures that overtemperature management takes precedence over undervoltage management, including their respective LED signals.

All normal activity signaling are assigned lowest priority.

During normal operation, the current LED signaling for standard activity is copied in a backup variable. This way, once the system recovers from an error, the normal activity signal can be restored.

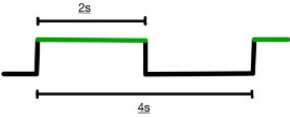
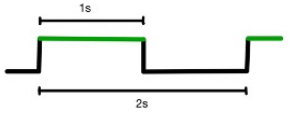
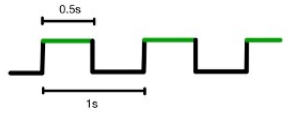
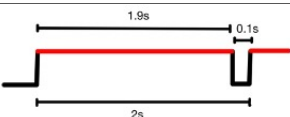
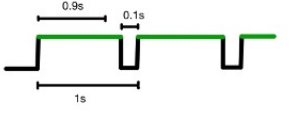
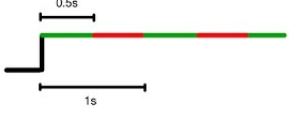
#	Blink: color, period, duty	Caption	Graph	Priority
-	off	No alimentation		-
0	Green, 4 s, 50%	Basics control (MCU) of the instrument are turned on: beginning of instrument powerup procedure	 <p>Note: slightly misleading graph</p>	3 (low)
1	Green, 2 s, 50%	Computer is being alimented: 5V supply turned on		3 (low)
2	Green, 1 s, 50%	Computer is returning an electronic signal: electronic connection OK		3 (low)
3.2	Red, 2 s, 95%	Computer not giving electronic feedback		3 (low)
3.3	Green, 1 s, 90%	Computer is giving back feedback: Software connection OK		3 (low)
4	Red and Green, 1 s, 50%	Computer is not giving any software feedback		3 (low)

Table 5.2: ISO BOX LED signaling

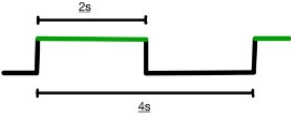
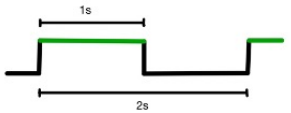

#	Blink: color, period, duty	Caption	Graph	Priority
-	off	No alimentation		-
0	Green, 4 s, 50%	Basics control (MCU) of the instrument are turned on: beginning of instrument powerup procedure	 <p>Note: slightly misleading graph</p>	3 (low)
1	Green, 2 s, 50%	Safety Checks, if enabled, are satisfied		3 (low)
3.1	Constant green light	Normal Run		3 (low)

Table 5.3: POWER BOX LED signaling

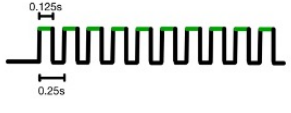
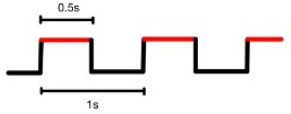

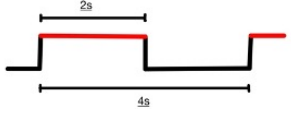

#	Blink: color, period, duty	Caption	Graph	Priority
5	Green, 0.25 s, 50%	The impulse of shut-down has been detected. Correct shutdown mode		3 (low)
6.1	Red, 1 s, 50%	ERROR: Computer is not giving electronic feedback anymore (NOTE: only ISO BOX mode)		3 (low)
6.2	Red, 0.25 s, 80%	ERROR: the instrument input voltage is too low		2 (mid)
6.3	Red, 4 s, 50%	ERROR: the instrument is overheating		2 (mid)
7	Red, constant	I2C communication with pheriperals has failed		1 (high)

Table 5.4: ISO BOX and POWER BOX shutdown and error signaling

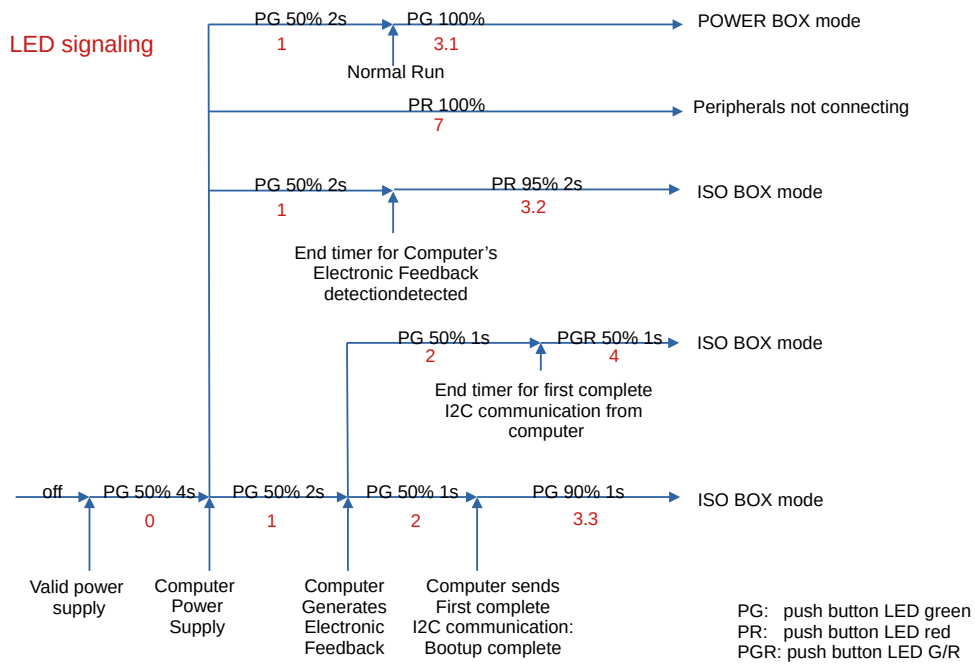


Figure 5.8: LED time evolution

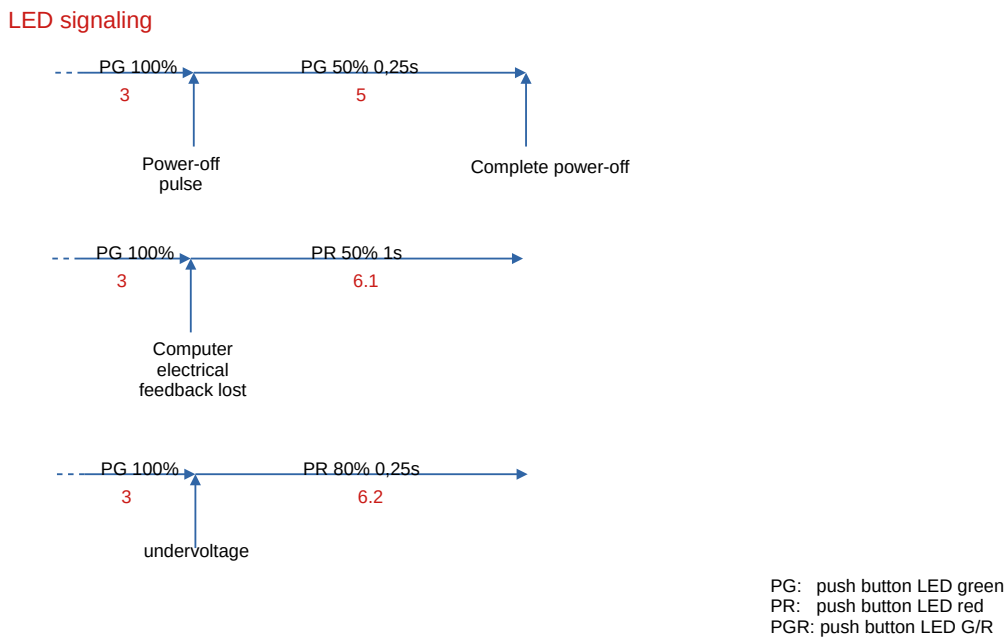


Figure 5.9: LED time evolution 2

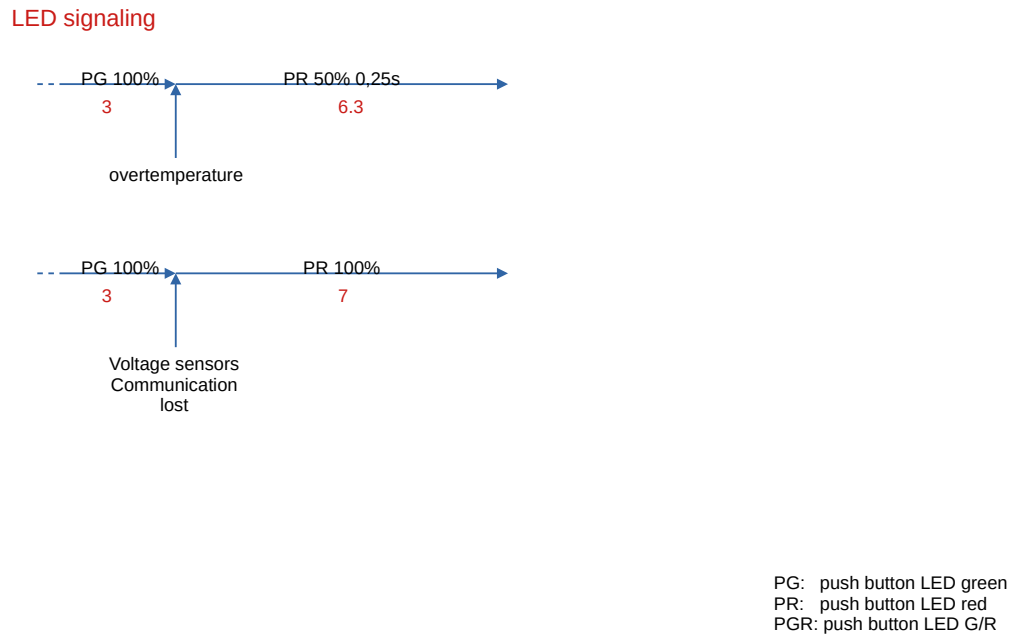


Figure 5.10: LED time evolution 3

5.4 Cranking

This Section is dedicated to explaining an extra study done for this project, the recognition of the cranking sequence from battery voltage. The objective was to add this check to the optional “safety checks” done at the beginning of firmware.

5.4.1 What is cranking

When examining the battery of a car or, in general, any automotive machinery equipped with an engine, we observe that its voltage is stable under normal conditions.

Before the engine starts, the battery powers essential systems such as the dashboard, interior and exterior lights, and locking mechanisms. These tasks require relatively small amounts of power, ensuring the voltage remains steady.

However, an interesting phenomenon is visible when the engine is started. Cranking voltage refers to the battery voltage during this initial phase of engine startup. Observing the voltage curve during cranking, we notice a significant drop, from 2 to 9 volts depending on the engine type.

This voltage drop occurs because of the engine starter. It is a smaller electrical

engine, that transforms electrical energy from the battery to mechanical energy. It starts moving the pistons, which will start rotating, starting the internal combustion.

As the engine begins to turn over, the engine starter continues to use voltage to maintain piston motion. So, during this phase, the voltage curve has a sinusoidal wave form, because the torque required to move the pistons changes based on their positions. This pattern continues briefly, until the engine reaches self-sustaining operation.

Once the engine is running independently, powered by the combustion in the cylinders rather than the battery, the alternator comes into play. At this point, the engine drives the alternator, creating a dynamo effect. This results in the battery voltage rising slightly above its initial level due to the alternator's charging function.

In Figure 5.11 we can see an example of a cold-crank waveform with a starting profile defined by the ISO standards, without the final raise of the alternator.

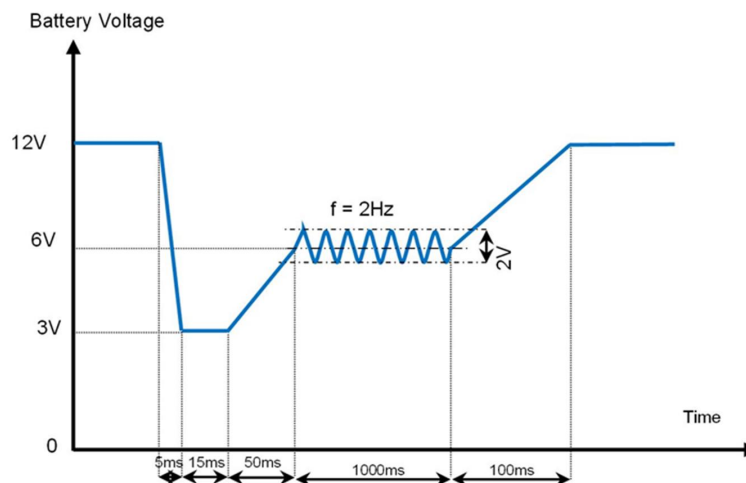


Figure 5.11: cold cranking, diesel engine [12]

Cold-crank is the starting of an engine at very cold temperatures, which is usually a worst case scenario.

We can see a real case scenario, of a normal temperature crank, in Figure 5.12. This is the crank of a diesel general use car.



Figure 5.12: Real case: voltage cranking, diesel car

5.4.2 Cranking problem

This study has been developed to ensure that the computer mounted in the ISO BOX powers on only after the engine cranking process is complete. To achieve this, we implemented an algorithm designed to detect the cranking signal waveform in the line voltage.

The approach is based on analyzing the mean and variance of the waveform to identify the characteristic shape of a cranking signal. The first consideration in this process is the frequency at which voltage can be sampled. Since we are working with a MCU with limited memory, storing a large number of data points is difficult. To resolve this, we opted for a method that uses two single variables to track the current mean and variance, which are updated incrementally with each new data point.

Choosing an appropriate sampling rate is very important. If the mean and variance are calculated over samples too close to each other, the variance may be too stable and not correctly register the initial drop in voltage. On the other hand, using samples too far apart may result in inaccurate data, and missing of the correct minimum voltage value. After analyzing the time required for sensor measurements and I2C communication, we determined that a sampling rate of 100 Hz would be effective. At this rate, we can compute the variance and mean at every data point without significant memory or processing constraints.

With the sampling rate decided, we focused on analyzing the cranking waveform to tell it apart from other voltage changes that could cause confusion. During this process, we identified a few key challenges:

- **Random Voltage Fluctuations:** Sudden drops or spikes in voltage could look like the first dip of the cranking signal, leading to false detections.
- **Failed Cranking Attempts:** Sometimes, the engine cranks, but it doesn't start. In these cases, the alternator doesn't kick in, so the voltage doesn't rise at the end.
- **Very Short Cranking Events:** If the cranking signal is too quick, it might be mistaken for noise since it doesn't show the typical pattern of a real cranking process.

The algorithm was built to handle these challenges, making it good at recognizing actual cranking signals while avoiding mistakes. This ensures the computer only powers on when the engine is fully cranked, preventing issues caused by unstable voltage.

5.4.3 Code for cranking

To develop and test the algorithm, we had to simulate a cranking signal. A typical cranking signal can be divided into five phases:

- **Initial Drop:** A sudden voltage drop of a few volts.
- **Slight Rise:** A small increase in voltage following the initial drop
- **Sinusoidal Oscillations:** A 5 Hz oscillation phase, reflecting the engine's cranking motion. (This frequency was based on recorded data from the car, but it can be adjusted to match specific cranking signals)
- **Linear Rise:** A gradual voltage increase
- **Final Stabilization:** The voltage settles slightly above the initial average value

The simulated signal was created in MATLAB using the following code.

```
1 sampling_time = 0.01; %seconds
2 sampling_rate = 1/(sampling_time);
3 signal_duration = 4; %seconds
4 total_data_points = sampling_rate*signal_duration;
5 unit_of_measure = 1; %use seconds
6 data_points_per_second = sampling_rate*unit_of_measure;
```

```
7 average_every_n = 1; %average at every data point (can be
  increased to reduce noise)
8 %average_every_n = 5; %example of increased average
9
10 % PHASE 0
11 mean_phase0 = 12;
12 variance_phase0 = 0.01;
13 duration_phase0 = 1;
14 samples_phase0 = data_points_per_second * duration_phase0;
15
16 %PHASE 1
17 frequency_phase1 = 0.5;
18 amplitude_phase1 = -3;
19 initial_value_phase1 = mean_phase0;
20 final_value_phase1 = 10;
21 variance_phase1 = 0.01;
22 duration_phase1 = 0.1;
23 samples_phase1 = data_points_per_second * duration_phase1;
24
25 %PHASE 2
26 %frequency_phase2 = 2; %theoretical
27 frequency_phase2 = 5; %found in diesel car
28 amplitude_phase2 = 0.5;
29 initial_value_phase2 = final_value_phase1;
30 final_value_phase2 = 10.5;
31 variance_phase2 = 0.01;
32 duration_phase2 = 0.6;
33 samples_phase2 = data_points_per_second * duration_phase2;
34
35 %PHASE 3
36 initial_value_phase3 = final_value_phase2;
37 final_value_phase3 = 12.5;
38 variance_phase3 = 0.01;
39 duration_phase3 = 0.3;
40 samples_phase3 = data_points_per_second * duration_phase3;
41
42 %PHASE 4
43 mean_phase4 = 12.5;
44 variance_phase4 = 0.001;
45 %to make sure total duration is correct
46 duration_phase4 = signal_duration - (duration_phase0 +
  duration_phase1 + duration_phase2 + duration_phase3);
47 samples_phase4 = data_points_per_second * duration_phase4;
48
```

```

49
50 % GENERATE CRANK SIGNAL PARTS
51 %phase0: constant noisy signal
52 signal_phase0 = sqrt(variance_phase0) * randn(samples_phase0, 1)
    + mean_phase0;
53 %phase1: sinusoidal wave and + ramp, with noise
54 time_phase1 = linspace(0, 2*pi, samples_phase1);
55 sine_wave_phase1 = amplitude_phase1 * sin(frequency_phase1 *
    time_phase1)';
56 ramp_phase1 = linspace(initial_value_phase1, final_value_phase1,
    samples_phase1)';
57 noise_phase1 = sqrt(variance_phase1) * randn(samples_phase1, 1);
58 signal_phase1 = sine_wave_phase1 + ramp_phase1 + noise_phase1;
59 %phase2: sinusoidal wave and + ramp, with noise
60 time_phase2 = linspace(0, 2*pi, samples_phase2);
61 sine_wave_phase2 = amplitude_phase2 * sin(frequency_phase2 *
    time_phase2)';
62 ramp_phase2 = linspace(initial_value_phase2, final_value_phase2,
    samples_phase2)';
63 noise_phase2 = sqrt(variance_phase2) * randn(samples_phase2, 1);
64 signal_phase2 = sine_wave_phase2 + ramp_phase2 + noise_phase2;
65 %phase3: ramp noisy signal
66 ramp_phase3 = linspace(initial_value_phase3, final_value_phase3,
    samples_phase3)';
67 noise_phase3 = sqrt(variance_phase3) * randn(samples_phase3, 1);
68 signal_phase3 = ramp_phase3 + noise_phase3;
69 %fphase4: constant noisy signal
70 signal_phase4 = sqrt(variance_phase4) * randn(samples_phase4, 1)
    + mean_phase4;
71
72 %FINAL CRANK SIGNAL
73 time_final_signal = linspace(0, signal_duration,
    total_data_points);
74 final_signal = cat(1, signal_phase0, signal_phase1, signal_phase2
    , signal_phase3, signal_phase4);

```

Listing 5.1: generation of crank signal

The idea was to analyze the signal starting with the first phase, which typically averages around 12 volts with very little variance. This is because, while there is some noise, the voltage remains relatively stable. When the cranking begins, the initial drop causes a noticeable spike in variance. By comparing the mean

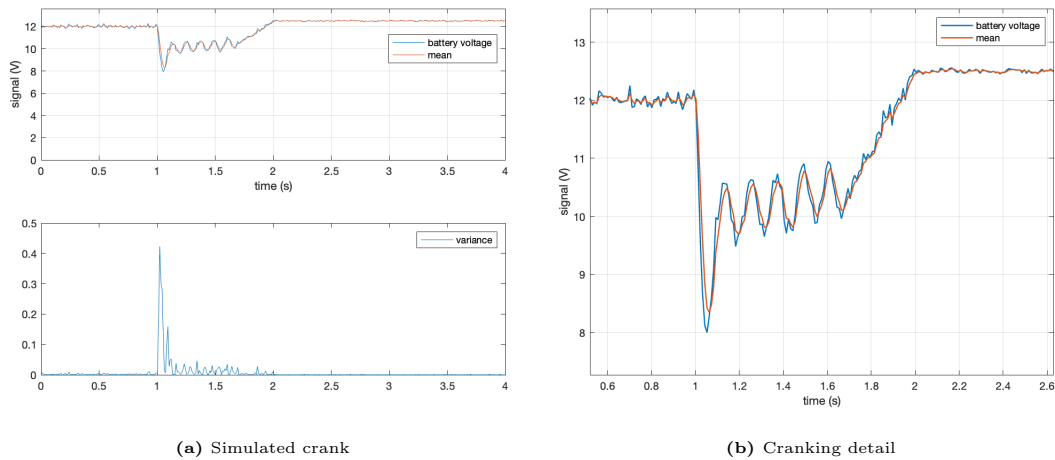


Figure 5.13: Example of a simulated crank using the MATLAB code. Together are also shown the mean and variance used. On the right a detail is shown

voltage before the drop with the minimum voltage found during this spike, we can calculate the drop's delta in voltage. If the drop is big enough, it signal the start of the cranking process. At this point, we begin counting the cranking samples, which allows us to calculate the cranking duration based on the sampling rate. We determined that it was sufficient to focus on the rising phase of the signal, as the fluctuations in the middle did not provide any additional useful information. By analyzing the final rise and the total duration of the cranking process, we could detect cranking. The following code was obtained.

```

1 for i = 2:1:(total_data_points/average_every_n)
2     current_average(i) = (current_average(i-1) + final_signal(i*
3     average_every_n))/2;
4     current_variance(i) = ( (final_signal(i*average_every_n) -
5     current_average(i))^2 )/2;
6     %find minimum value
7     if final_signal(i*average_every_n) < min_drop
8         min_drop = final_signal(i*average_every_n);
9     end
10    %begin cranking
11    if ((current_variance(i)>change_variance_min*current_variance
12    (i-1))&&(current_average(i)<current_average(i-1)-
13    change_mean_min))
14        average_fase0 = current_average(i-1);
15        variance_fase0 = current_variance(i-1);
16        cranking_samples_amount = 1;
17    end
18    %start counting cranking time (samples)
19    if ((cranking_samples_amount > 0)&&(cranking_done == false))

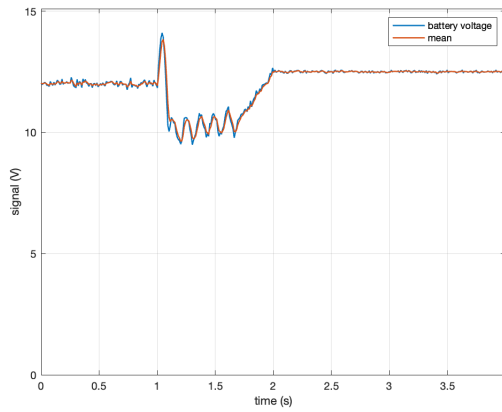
```

```
16     %I need to increment by 1, but since I'm averaging every
average_every_n samples, I'm actually considering n samples.
17     cranking_samples_amount = cranking_samples_amount +
average_every_n;
18     end
19     % if the drop was sufficiently low, and the average has risen
above the initial value by at least a certain threshold
20     if (((min_drop)<average_fase0-delta_init) && (current_average
(i)>average_fase0 + delta_fin))
21         %if it lasted long enough
22         if ( cranking_samples_amount * sampling_time <
durata_cranking_expected )
23             cranking_samples_amount = 0; %if cranking results
ended, but it was too short, then abort process, wait for new
crank
24         elseif ( cranking_samples_amount * sampling_time >
durata_cranking_expected )
25             cranking_done = true;
26         end
27     end
28 end
```

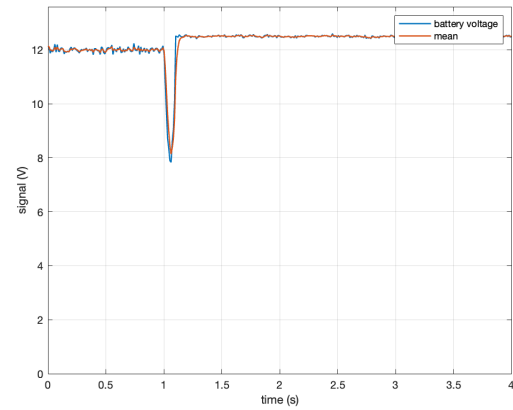
Listing 5.2: generation of crank signal

We used some constants that are specific to the characteristics of the waveform being analyzed, such as the intensity of the original drop, or the duration of crank. These constants cannot be determined automatically by the system, and require manual configuration by a user.

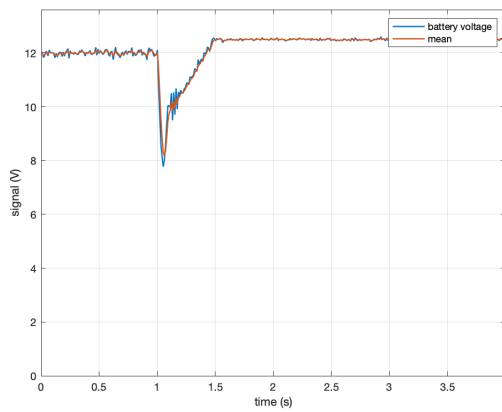
We show here some cases of generated waves that are correctly not recognized as cranking. Meaning that, after analysis, the flag “cranking_done” was FALSE.



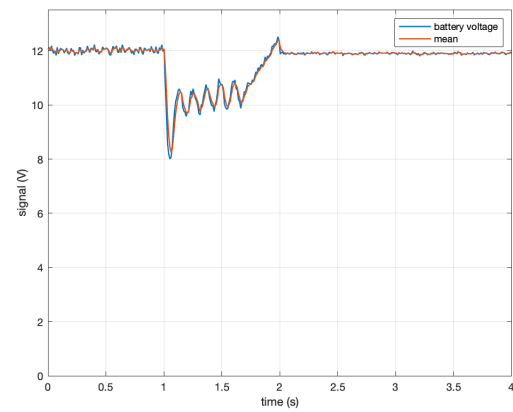
(a) Initial change in variance present, but the mean change is in the wrong direction



(b) Very strong noise



(c) Crank too short. Could be a cranking attempt gone wrongly



(d) Crank attempt but ends in the alternator not working. Final value too low

Figure 5.14: Example of simulated Figure showing four different plots, each representing a signal resembling a cranking waveform but correctly identified by the algorithm as not being a cranking event

Chapter 6

Validation

We introduce the testing process for the firmware developed, and the scenarios that were evaluated.

The cranking detection code was not validated. After discussions with customers, on whose machinery the device was to be mounted, it was discovered that in their specific case the machinery physically disconnects the voltage supply line during cranking. This makes detecting the cranking signal impossible. This code has therefore not been included in the final version of the code, and was not thoroughly tested due to both economic and time limits.

The testing setup was configured as follows:

The firmware was uploaded onto the PAPBB board, which was powered by a voltage generator. The generator provided two outputs: one simulated the `line` voltage, and the other simulated the `ignition` voltage.

Most tests were carried out on the “clean” version of the code. However, in some cases, small changes were made to simulate physical errors at specific times.

The PAPBB board was connected to the appropriate bicolor LED. However, it was not connected to the Polispec’s thermal feedback. A button was connected to the board to simulate the power button signal, simulating the operator’s input. To monitor the power status of the 5 V line (used to power the computer) and the 12 V line (used to power the Polispec) two indication LEDs were mounted on PAPBB. These LEDs illuminated when their respective signals were high.

The board was also connected to a computer for testing under two scenarios: immediate power-up and pulse-based power-up. In the immediate power-up case, the indicator LED was used to compare the activation of the supply voltage with the computer’s power status. In the pulse-based power-up case, the LED helped

diagnosing potential issues. It was used to identify whether eventual problems in power-on were caused by the supply line or from the pulse recognition mechanism. The computer was tested in two configurations: without an Operating System (OS) installed and with it (Windows) fully installed. This was done to evaluate differences in power-on timing under both conditions. In the field only the second case is used.

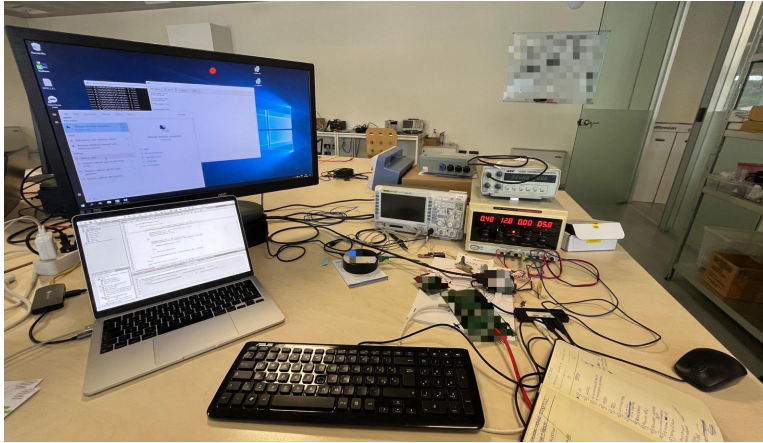


Figure 6.1: Setup - sensitive parts are blurred

Here is a list of all the tests performed, including their expected outcomes and the state of the code where the test was done.

Unless specified, test were done in all the following scenarios: POWER BOX mode, ISO BOX mode with embedded powered on at 5 V or using a pulse, and with the computer both with and without an OS installed.

6.1 Power-On Tests

Starting from the Idle state, the first test verified the ignition signal check.

Utilizing a wave form generator, a periodic square wave was sent on the ignition line. As expected, when the wave period was less than 1 second, the system did not power up, as the initial ignition check correctly prevented the code from progressing.

Next, the optional safety check was tested. It was enabled, but the line voltage, while high enough to not trigger the undervoltage error, was kept under the safety threshold. It was confirmed that the system did not start under these conditions. During this test, the brief press of the power button was also verified. When the safety check was enabled but not satisfied, pressing the button briefly allowed the

code to progress.

A second test on the button behavior was done at this safety check state. If the button was pressed for a long time, it was confirmed that this action had no consequences in this phase, as expected. Also, the system correctly “forgot” the action, meaning that even if the safety check was later satisfied, the long button flag was automatically cleared. This way it was assured that the system did not power off unexpectedly during normal operation.

6.1.1 ISO BOX mode

Starting from the Configuration State the following cases were tested:

First, the scenario where the computer was already powered. The expected behavior was that the system would transition to an error state. After a few seconds, the 5 V signal was deactivated, and the correct LED signal started (see Chapter 5).

Next, the case of the computer powering off right after the Polispec power-on. To test this, a small modification was introduced to the code: it was set to halt for a specific period, giving the programmer time to manually power off the computer. This test was conducted for both computer power-on methods (direct 5 V and impulse-based). In both cases, the expected behavior was observed: the computer was powered on back again, and the correct LED signal was displayed.

Computer power-on mode: 5 V supply

Starting from the Configuration state, we tested the case where the computer was missing. We physically removed the computer to test this. The expected result, explained in Chapter 5, was correct.

Computer power-on mode: power-on pulse

Starting from the Configuration state, we tested the case where the computer was missing. We physically removed the computer to test this. The expected result was that the system would try to power-on the computer three times, failing all three it would enter the associated error state. Correctly, for the first 20 s nothing was visible by the tester, then the LED signal became constant green, and after a few seconds the LED associated with the 5 V signal turned off, as that line was deactivated.

6.1.2 POWER BOX mode

We also tested the case of power-on without the computer. As expected, the system functioned correctly, and the code proceeded to power on the Polispec, ignoring the absence of the computer.

6.2 Normal Run Tests

First, the power-off threshold of ignition voltage was tested. It was set to the default value (5 V). It was expected that, if the ignition voltage fell below this threshold during normal operation the system would begin the power-off sequence. This was tested in both ISO BOX mode and POWER BOX mode, with the computer powered on in both modalities, and with the computer either having or not having an OS installed.

It was observed that since the ignition voltage needs to be reliably low to trigger the power-off sequence, we could apply a square wave to the `ignition` line without affecting the code. The system handled this correctly.

There is an additional safety check in place that ensures the computer cannot be powered off while it is still booting. Specifically, after receiving the signal to start the power-off sequence, the system has to check that either the computer has signaled that it has completed booting (via a special I2C register) or that a set time has passed, which we will refer to as `bootupMaxTime`. If the power-off signal (either ignition or a button pulse) is received before `bootupMaxTime`, the system waits for the time to pass. If the special register receives a message before this time ends, then the power-off sequence can start anyway. If the power-off signal arrives after the `bootupMaxTime`, then the system immediately powers-off. This behavior was tested and confirmed to work in all cases. In the POWER BOX mode, this check is not performed, and the system powers off as soon as the signal is received.

Next, we tested the power-off behavior during normal operation.

When the system was in Normal Run mode, the button was pressed for a long time, and as expected, the power-off sequence started. This was indicated by the LED signaling, and the computer powered off stably. We also tested the re-powering functionality. If the button was pressed briefly, nothing happened. However, when pressed for a long time, the system restarted, starting the initial

checks again (e.g., verifying the line voltage). The computer powered on again as expected.

If the button was pressed briefly during Normal Run, the system immediately shut down. As soon as the computer finished powering down (i.e., when it stopped providing electronic feedback to the MCU), the error management process started, as described in Section 6.4.

Finally the communication LED were checked. As seen in Paragraph LED signaling of Section 5.3, if the computer does not communicate with the MCU before the associated timer ends (120 s), the LED signaling differs from when the communication arrives. It worked properly on both tests.

6.3 Power-off Test

The only power-off test remaining was the Hard Power-Off modality. This test was performed by simulating the electronic feedback from the computer without actually having the computer present. Therefore the feedback wouldn't disappear after the power-off pulse. The system correctly waited for the specified amount of time before removing the line voltage.

6.4 Errors

The errors tested were many. First, we examined the case of the embedded system powering off during Normal Run. As explained in Figure 5.5, the system correctly triggered an LED signal when the computer powered off, and after a few seconds, the computer powered back on and the LED returned to green.

Then, the classical errors were tested.

- Undervoltage: the undervoltage threshold was tested. Set at default (7.2 V), we assured that in all modalities the system entered undervoltage error if the line voltage remained under this threshold for more than 500 ms. How the error is managed is explained in Chapter 5, and the system correctly followed the procedure when tested
- Overtemperature: a heat gun was used to simulate overtemperature, even though consequently the test was not perfectly precise. The overtemperature threshold was tested, but, to avoid damaging the system, for the test the threshold was lowered from the default value. The error was managed

as explained in Chapter 5, and the system correctly followed the procedure when tested. Overtemperature recovery was tested both when it occurred before the 30-second safety time threshold, in which case the system waited for the full 30 seconds before powering on, and when it happened after the safety time, in which case the system restarted immediately

- **I2C Errors:** at startup, we confirmed that missing communication from any of the peripherals stopped the system from starting. This was tested by changing the set addresses of the peripherals. To test I2C errors during normal operation, we needed to simulate the absence of a peripheral's response. Since it was not possible to physically disconnect the peripherals, we implemented a small modification to the code. A button was used to change the peripheral address on the MCU, simulating the missing responses. All peripheral errors were correctly identified, and their behavior followed the description in Chapter 5

Errors concomitance

We also needed to ensure that cases of error concomitances were handled correctly.

The two errors concomitances of interest were: The two error combinations of interest were: undervoltage and overtemperature, and both the voltage sensors and temperature sensor not responding. In the second case, as expected, the voltage sensor error took precedence because it is considered more critical.

For the undervoltage and overtemperature combination, we tested both possible sequences: undervoltage followed by overtemperature and overtemperature followed by undervoltage. If undervoltage occurred first, the system correctly powered off the 12 V line, signaling the error via the LED. After detecting the overtemperature error, its LED signaling took precedence, and the computer powered off. If overtemperature happened before undervoltage, we did not observe the undervoltage issue immediately. The overtemperature handling took precedence, and it was only after overtemperature recovery that the undervoltage condition showed. If undervoltage was still present when the system returned to the beginning of the code, the system would not continue, as the undervoltage check would not be satisfied.

6.5 Registers functionality

Lastly the registers were tested.

The register responsible for powering the Polispec on and off correctly responded to the computer's requests. The computer was able to trigger its own power-off sequence by changing the associated register. After power-off the system correctly waited for either a long button press or for the `ignition` signal to be lost. We also verified that changes in settings were properly saved. Some settings, such as thresholds, took effect immediately, while others, like the transition from ISO BOX to POWER BOX mode, were applied during the next power-up. This functionality was tested and worked as expected.

Chapter 7

Future Developments and Conclusions

This thesis was created with in collaboration with ITPhotonics. The main objective was the development of firmware for the updated version of their product, ISO BOX. ISO BOX works together with the Polisphec spectrophotometer, which is the company's main product, to manage its main computational unit. These three devices (ISO BOX, Polisphec, and the computer) are typically installed on machinery from which they draw their power. The power supply consists of two lines: a **line** voltage, connected to the machinery's batteries, and an **ignition** voltage, which signals when the machinery is powered on.

The machinery's batteries supply power not only to ITPhotonics' instruments but also to the machinery itself. Therefore, voltage levels can be very unstable, in particular at startup. Additionally, when the machinery is powered off, the sudden loss of the **ignition** voltage would result in a sudden cut to the computer's power, risking damage to its filesystem. ISO BOX was originally introduced to prevent these power-related risks to the computer.

A recent update to the Polisphec system resulted in the development of a new and less expensive electric board, that was also compatible with ISO BOX. This new board features a different MCU, which had to be programmed in a distinct Integrated Development Environment (IDE) from the previous MCU. A part of the firmware adaptation from one device to the other had already been introduced by the firm, as Polisphec's shares the same MCU and similar libraries.

The main objective of this thesis was the development of firmware for ISO BOX based on the original code's general flow, utilizing where possible the adapted libraries provided by ITPhotonics.

Additionally, new features and improvements were introduced, including: the integration of the front power button into the firmware flow, the LED signaling management system, error management for communication losses with the system's sensors, and the resolution of bugs found in the previous code or within the pre-adapted libraries.

The system was validated in a controlled laboratory environment, as the complete hardware setup was not available for field testing yet. The validation went through smoothly, with minor errors corrected along the way.

The final firmware met the defined objectives and performed as expected. Following some final adjustments to the overall product, the updated ISO BOX with the newly developed firmware, will soon be released.

References

- [1] ITPhotonics official website, [itphotonics.com, https://www.itphotonics.com/it/](https://www.itphotonics.com/it/), ultima consultazione: 11/8/2024
- [2] Polisphec official website, [polisphec.com, https://www.polisphec.com](https://www.polisphec.com), last consultation: 2/10/24
- [3] ITPhotonics Product Catalogue, “Reference Instruments For Spectrophotometric Analysis”
- [4] Alberto Morato, Mauro Tubiana, Embedded Real Time Control (ERTC) Course, Università degli studi di Padova, 2023, https://stem.elearning.unipd.it/pluginfile.php/451082/mod_resource/content/2/lecture6_Interrupt.pdf
- [5] Microchip Section 24. Inter-Integrated Circuit™ (I2C™), DS39702A, Copyright: © 2006 Microchip Technology Inc., [https://www.microchip.com.tw/RTC/RTC_DVD/Reference%20Manuals/16-Bits%20Family%20Reference%20Manual/PIC24F%20FRM%20Section%2024.%20Inter-Integrated%20Circuit%20\(I2C\)%20\(DS39702A\).pdf](https://www.microchip.com.tw/RTC/RTC_DVD/Reference%20Manuals/16-Bits%20Family%20Reference%20Manual/PIC24F%20FRM%20Section%2024.%20Inter-Integrated%20Circuit%20(I2C)%20(DS39702A).pdf)
- [6] I2C-bus specification and user manual: © NXP B.V. 2021, [www.nxp.com, https://www.nxp.com/docs/en/user-guide/UM10204.pdf](https://www.nxp.com/docs/en/user-guide/UM10204.pdf), ultima consultazione: 5/8/2024
- [7] Joseph Wu, Application Note, A Basic Guide to I2C, SBAA565, Texas Instruments, November 2022, 32 pages, <https://www.ti.com/lit/an/sbaa565/sbaa565.pdf?ts=1727119926586>
- [8] Jonathan Valdez, Jared Becker, Application Report, Understanding the I2C Bus, SLVA704, Texas Instruments, June 2015, 8 pages, <https://www.ti.com/lit/an/slva704/slva704.pdf?ts=1729278479466>

- [9] Josh Schneider, Microcontrollers vs. microprocessors: What's the difference?, IBM.com, June 13 2024, last consultation: 27/09/2024 <https://www.ibm.com/blog/microcontroller-vs-microprocessor/>
- [10] Microchip Section 8. Interrupts, DS39707A, Copyright: © 2006 Microchip Technology Inc., <https://ww1.microchip.com/downloads/en/DeviceDoc/39707a.pdf>
- [11] Microchip Section 14. Timers, DS39704A, Copyright: © 2006 Microchip Technology Inc., <https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/ReferenceManuals/39704a.pdf>
- [12] Kaashyap Kattamudi, Todd Toporski, "Design Considerations to Sustain Automotive Crank Conditions", monolithicpower.com, Copyright: © 2024 Monolithic Power Systems, <https://www.monolithicpower.com/learning/resources/design-considerations-to-sustain-automotive-crank-conditions>